FT-SR: A PROGRAMMING LANGUAGE FOR CONSTRUCTING FAULT-TOLERANT DISTRIBUTED SYSTEMS

(Ph.D. Dissertation)

Vicraj Timothy Thomas

TR 93-23

August 9, 1993

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This work was supported in part by the National Science Foundation under grant CCR-9003161 and the Office of Naval Research under grant N00014-91-J-1015.

FT-SR: A PROGRAMMING LANGUAGE FOR CONSTRUCTING FAULT-TOLERANT DISTRIBUTED SYSTEMS

by

Vicraj Timothy Thomas

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 3

FT-SR: A PROGRAMMING LANGUAGE FOR CONSTRUCTING FAULT-TOLERANT DISTRIBUTED SYSTEMS

Vicraj Timothy Thomas, Ph.D.
The University of Arizona, 1993

Director: Richard D. Schlichting

This dissertation focuses on the area of improving programming language support for constructing fault-tolerant systems. Specifically, the design and implementation of FT-SR, a programming language developed for building a wide variety of fault-tolerant systems, is described. FT-SR is based on the concurrent programming language SR and is designed as a set of extensions to SR.

A distinguishing feature of FT-SR is the flexibility it provides the programmer in structuring fault-tolerant software. It is flexible enough to be used for structuring systems according to any of the standard fault-tolerance structuring paradigms that have been developed for such systems, including the object/action model, the restartable action paradigm, and the state machine approach. This is especially important in systems building because different structuring paradigms are often appropriate for different parts of the system. This flexibility sets FT-SR apart from other fault-tolerant programming languages which provide language support for the one paradigm that is best suited for the class of applications they choose to support. FT-SR, on the other hand, is suitable for programming a variety of systems and applications.

FT-SR derives its flexibility from a programming model based on *fail-stop atomic objects*. These objects execute operations as atomic actions except when a failure or series of failures cause underlying implementation assumptions to be violated; in this case, notification is provided. This dissertation argues that fail-stop atomic objects are the fundamental building blocks for all fault-tolerant programs. FT-SR provides the programmer with simple fail-stop atomic objects, and mechanisms that allow these fail-stop atomic objects to be composed to form higher-level fail-stop atomic objects that can tolerate a greater number of faults. The mechanisms for composing fail-stop atomic objects are based on standard redundancy techniques. This ability to combine the basic building blocks in a variety of ways allows programmers to structure their programs in a manner best suited to the application at hand.

FT-SR has been implemented using version 3.1 of the *x*-kernel and runs standalone on Sun 3s. The implementation is interesting because of the novel algorithms and optimizations used within the language runtime system.

STATEMENT BY AUTHOR

ACKNOWLEDGMENTS

I am extremely grateful to my advisor Professor Richard Schlichting for truly being a friend, philosopher and guide to me. He has always been willing to take the time to discuss with me any problem, research related or not, and help me solve it. It is his words of encouragement that helped me persevere through the "dry spells" in my research. I am especially thankful for his efforts in preparing me for a career after graduate school by involving me in the grant writing process, teaching me the art of writing and presenting papers, and sending me to conferences to learn and network with other researchers. Rick taught me that research is about asking the right questions and searching for answers that embody general principles or abstractions. He also instilled in me the value of clearly expressing these questions and answers.

I thank Professors Gregory Andrews and Larry Peterson for their valuable comments about my research, which benefits greatly from products of their own research efforts. I thank Larry for his valuable feedback during the fault-tolerance research meetings and Greg for being my "surrogate" advisor during Rick's sabbatical. They have both been extremely supportive throughout my stay in Arizona.

I thank David Mosberger for his interest in FT-SR. He is responsible for designing the syntax of the "backups on" clause of the FT-SR create statement and for porting FT-SR to run on Unix machines.

I thank all the graduate students in the computer science department for making my stay in Tucson very enjoyable. I am especially thankful for the camaraderie of Mark Abbot, Nina Bhatti, Peter Bigot, Curtis Dyreson, Tyson Henry, Patrick Homer, Clint Jeffery, Nick Kline, Ed Menze, Shamim Mohamed, Bob Simms, Mike Soo and Ken Walker. Every one of them expanded my horizons in one way or another.

Of the many others who made my stay in Tucson so delightful, I thank Clint Jeffery and Nick Kline for being such great housemates. I thank Susie Wagner for her friendship and her innumerable thoughtful gestures. I thank Lesa Stern for the notes of encouragement she sent me while I was writing the dissertation and for the many delightful lunches we had together.

I thank my parents and sister whose prayers carried me through the thick and thin of graduate school. I thank them for their patience over the years, wondering if I'd ever get out of school.

This research was supported in part by the National Science Foundation under grant CCR-9003161 and the Office of Naval Research under grant N00014-91-J-1015.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This dissertation focuses on the area of improving programming language support for constructing fault-tolerant systems. Specifically, the design and implementation of FT-SR, a programming language developed for building a wide variety of fault-tolerant systems, is described. FT-SR is based on the concurrent programming language SR and is designed as a set of extensions to SR.

A distinguishing feature of FT-SR is the flexibility it provides the programmer in structuring fault-tolerant software. It is flexible enough to be used for structuring systems according to any of the standard fault-tolerance structuring paradigms that have been developed for such systems, including the object/action model, the restartable action paradigm, and the state machine approach. This is especially important in systems building because different structuring paradigms are often appropriate for different parts of the system. This flexibility sets FT-SR apart from other fault-tolerant programming languages which provide language support for the one paradigm that is best suited for the class of applications they choose to support. FT-SR, on the other hand, is suitable for programming a variety of systems and applications.

FT-SR derives its flexibility from a programming model based on *fail-stop atomic objects*. These objects execute operations as atomic actions except when a failure or series of failures cause underlying implementation assumptions to be violated; in this case, notification is provided. This dissertation argues that fail-stop atomic objects are the fundamental building blocks for all fault-tolerant programs. FT-SR provides the programmer with simple fail-stop atomic objects, and mechanisms that allow these fail-stop atomic objects to be composed to form higher-level fail-stop atomic objects that can tolerate a greater number of faults. The mechanisms for composing fail-stop atomic objects are based on standard redundancy techniques. This ability to combine the basic building blocks in a variety of ways allows programmers to structure their programs in a manner best suited to the application at hand.

FT-SR has been implemented using version 3.1 of the *x*-kernel and runs standalone on Sun 3s. The implementation is interesting because of the novel algorithms and optimizations used within the language runtime system.

CHAPTER 1

INTRODUCTION

With the use of computers permeating almost every aspect of everyday life, the dependability of computer systems is becoming an increasingly important issue. In today's world, the failure of a computer system can be inconvenient at best and life-threatening at worst. For example, the wide-spread inconvenience and loss of revenue that resulted from the failure of the AT&T long distance telephone network on January 15 1990, was caused by the inopportune failure of a single switching computer [Neu92]. In another incident, the failure of a computer proved fatal for two patients undergoing radiation therapy in Tyler, Texas, when the computer controlling the Therac-25 machine used in the therapy failed and subjected the patients to a massive overdose of radiation [Jac90]. Many more such examples of the deleterious effects of computer failures can be found in the *Inside Risks* column, published monthly in the Communications of the ACM. Building dependable computing systems, i.e. computers one can count on to provide correct service whenever service is needed, is therefore of utmost importance.

Developing the software for a computer system is often the most complex aspect of building the system. For reasons discussed later, this task becomes much more complex if the system must be dependable. Software developers and programmers of dependable systems therefore need design and development tools to simplify this complex task. Especially useful are tools in the form of good programming languages designed specifically for building dependable systems. Such a language would support the techniques commonly used to build dependable systems, be flexible enough for use in building a variety of systems, and be efficiently implementable. This dissertation introduces one such language, FT-SR, which has been designed expressly to support the development of software for dependable systems.

1.1 Dependable System Construction—Principles and Techniques

This section describes the general principles that underlie the construction of any dependable system or component; the application of these principles to building dependable computing systems is described in the following section. All definitions in this section are from [Lap91].

A system or component is defined as being *dependable* if reliance can justifiably be placed on the service it delivers. The description of the service a system is expected to provide is called its *specification*. If the delivered service deviates from the specification, the system is said to have *failed*. A failure results from an *error*, which is the part of the system state that is liable to cause a failure. The adjudged or hypothesized cause of an error is a *fault*. To summarize, a failure is caused by an error, which in turn is a manifestation of a fault. Note that an error does not necessarily lead to a failure. An error may go unnoticed until it affects the behaviour of the system.

A system may be viewed as being built out of components that are bound together and interact with each other. Each of these components may, in turn, be viewed as systems that are composed of other components, and so on. This layered decomposition may be continued until the components are so insignificant that decomposing them further is no longer possible or interesting. In this view, each layer sees the failure of an underlying component as an error, where the fault or cause of the error is the failure of the underlying component. If the error causes this layer to fail, in turn the failure will similarly manifest itself as a fault at the layer above it. This fault$\rightarrow$error$\rightarrow$failure sequence therefore repeats itself, with the failure of each layer being perceived as a fault by the next higher layer. The user of the system sees a failure only if this chain extends into the top-most layer causing it, and thereby the entire system, to fail.

In this layered view of dependable systems, each layer or component must be designed and built with the goal of eliminating all possible faults in that layer. The techniques used to eliminate faults may be classified into *fault-prevention* techniques and *fault-removal* techniques. Fault-prevention techniques aim at reducing the likelihood of a fault occurring. These techniques include the use of quality components and the adoption of good design practices. Fault-removal techniques detect and remove faults that have crept into the system despite the use of fault-prevention techniques, before the system is put into operation. Fault-removal techniques include formal verification and extensive testing of components before they are put into service.

Even the most carefully designed and engineered system will suffer from faults after being put into operation. Some faults are introduced into the system during its design, and despite the use of fault-prevention and fault-removal techniques, remain undetected until it is operational. These faults are called *design* faults. Faults may also be introduced into the system during operation. These are typically caused by adverse environmental conditions such as extreme temperatures, excessive vibration, electro-magnetic radiation, etc., and are therefore called *external* faults.

A layer l of an operational system may fail because of its inability to cope with a design fault within the layer or with an external fault. This failure manifests itself as an *operational* fault in a layer m above it. If m fails as a result of this fault, it induces an operational fault in a layer above it, which may in turn fail and induce an operational fault in another layer. This chain of operational faults therefore starts at a layer that failed because of its inability to cope with external or design faults and continues until the topmost layer of the system, in which case the entire system fails, or until it reaches a layer that can handle the fault. Each layer of a dependable system must therefore prevent the creation and propagation of operational faults by removing or compensating for the errors caused by design faults within the layer and operational faults induced by the failure of underlying layers, before these errors cause a failure. The techniques used to correct such errors in an operational system are called *fault-tolerance* techniques, and a system or component that can continue to function correctly despite design and operational faults is said to be *fault-tolerant*.

All fault-tolerance techniques depend on *redundancy* of one type or another. Redundancy is anything extra in the system that would not be required if it did not have to be fault-tolerant. There are three types of redundancy: *state redundancy*, *information redundancy*, and *time redundancy*. State redundancy is where multiple copies of the state of a component are maintained. If all the copies are active and identical, the redundancy is said to be *active*. If one copy is active and the others passive, the redundancy is said to be *passive*. The passive copies typically lag behind the active copy in time. For example, a bank that maintains copies of customer accounts at a branch office and its data processing center uses state redundancy. If the bank updates both copies of an account whenever there is a change in the account status, the redundancy is active. If the bank updates the copy at the branch office whenever there is a change but updates the data center's copy only at the end of each day, the redundancy is passive.

Information redundancy is where extra information is maintained in the system state to correct errors. For example, information redundancy is used in the banking industry, which requires that the amount on a cheque be written in both words and figures. Any errors in writing the amount in figures on the cheque are resolved in favor of the value written in words.

Time redundancy is the repeated execution of an operation. Time redundancy is typically used to tolerate faults that are *transient* in nature. For example, an external fault that goes away when the adverse environmental conditions subside is a transient fault. Time redundancy is commonly used by facsimile machines that repeatedly try to transmit a document until it is successfully received by the receiving machine.

Fault-tolerance techniques use redundancy to remove errors caused by faults before they lead to failures or to prevent faults from causing errors. The techniques that remove errors may be classified into *backward error recovery*, *forward error recovery*, and *error compensation* schemes, based on their use of redundancy. Backward error recovery schemes use passive state redundancy to maintain an error-free copy of the system state. These schemes, on detection of an error in the active system state, replace it with the error-free state. The system is then restarted from this state—an instance of time redundancy. Both forward error recovery and error compensation schemes use information redundancy. Forward error schemes use extra or redundant information in the system state to transform an erroneous state to a new state from which the system can continue to operate correctly. Error compensation schemes use redundant information to deduce the correct system state from the erroneous state and use the deduced state to deliver correct service. Unlike forward error recovery schemes, error compensation schemes do not attempt to transform the erroneous system state into a correct state.

The fault-tolerance techniques that prevent faults from causing errors may also be classified, based on their use of redundancy, into *fault masking* schemes and *fault-passivation* schemes. Fault masking schemes use active state redundancy to maintain multiple copies of components. The results produced by the copies are voted upon; should one of the copies fail and produce erroneous results, it is out-voted by the other copies and its failure masked. Fault-passivation schemes typically use passive state redundancy; these schemes detect layers or components that have failed and configure them out of the system by switching to a passive copy of the component.

1.2 Dependable Computing Systems

Computer systems are made up of two major layers, the hardware layer and the software layer, each of which may be further subdivided into layers. The fault-prevention, fault-removal and fault-tolerance techniques described earlier apply to each of these layers. Common fault-prevention techniques used in the hardware layers are the use of good logic design tools to design circuits, the use of quality components to build the hardware, and the use of air conditioning or heating to keep the hardware operating under optimal climatic conditions. Examples of hardware fault-removal techniques are the use of simulation to verify the circuits, rigorous testing of hardware components, and periodic preventive maintenance of the operational system in order to remove faults before they cause errors. Examples of hardware fault-tolerance techniques are the use of error detection and correction codes, and instruction retry [Pra86].

Fault-prevention techniques commonly used in the design and development of software layers are the use of top-down software design [Som92], structured programming [DDH72], program verification [Gri81], software walk-throughs [Zie83], and software development tools [Fis91]. All these techniques aim at systematizing the process of software development by encouraging the use of an engineering approach in the design and implementation of the software. The most common software fault-removal technique is debugging by exhaustive testing. This technique attempts to exercise every execution path in the system by testing it with diverse sets of inputs and by simulating a variety of operating conditions.

Different fault-tolerance techniques are used in software systems to tolerate design faults and operational faults. The techniques used to tolerate design faults, i.e. bugs in the software, are called *software fault-tolerance* techniques. These techniques include recovery block schemes [Ran75] and N-version programming [Avi85]. In the recovery block scheme, programs are structured as a sequence of *recovery* blocks, which are program structuring units like procedures, modules, blocks, etc.. Each recovery block has associated with it an *acceptance test* and zero or more alternate blocks. The acceptance test is a logical expression that is evaluated on exit from a block. If the program passes the acceptance test, the alternate blocks are ignored and control passes to the next recovery block. Otherwise, the process is restored to its state before it entered the block, and one of the alternate blocks executed. The acceptance test is then applied, and a failure results in yet another alternate block being tried. This continues until the program passes the acceptance test or until all alternate blocks have been tried, in which case the recovery block is regarded to have failed. The recovery block scheme is therefore based on passive redundancy.

The N-Version programming scheme uses multiple versions of critical software components developed independently to the same specification. All version are executed and their results voted upon to determine the result of the computation. This scheme therefore relies on active redundancy to tolerate design faults.

Software that can tolerate operational faults, i.e. failures of the underlying computing platform, is called *fault-tolerant software*. It is important to distinguish fault-tolerant software from software fault-tolerance; the former is the ability of a software layer to cope with operational faults caused by failures of underlying layers, while the latter is the ability of a software layer to cope with design faults in the layer itself. This dissertation focuses exclusively on fault-tolerant software and the techniques used to build it.

Like all fault-tolerance techniques, techniques used to build fault-tolerant software depend on redundancy of one form or another. There is, however, very little redundancy available with a single processor system. In such a system, if the processor fails, everything fails. On the other hand, *distributed* or *network computer* systems, which are collections of processors connected by a communication network, have a natural redundancy that is available for fault-tolerance. If a processor in such a system fails, other processors can compensate for its failure. It is this inherent redundancy that makes distributed systems obvious candidates for building dependable systems.

Writing software for distributed systems is, unfortunately, extremely difficult. Such software has no single locus of control, has no consistent view of the entire system, and must be able to handle completely asynchronous events. The problem becomes much more difficult if the software must also be fault-tolerant. With failures, a part of the executing program can suddenly fail and the rest of the software must be able to cope with this failure. To make matters worse, it is impossible to predict when and where these failures will occur. All these factors together make the development of fault-tolerant software extremely difficult.

Numerous techniques have been developed to help simplify the task of developing fault-tolerant software. These include the use of *failure models*, *abstractions*, and *software structuring paradigms*. A failure model precisely characterizes assumptions about the nature of failures suffered by a system or component. It simplifies the task of developing fault-tolerant software by limiting the kinds and number of failures with which the programmer must anticipate and handle. Examples of common failure models are the *fail-silent* or *crash* failure model [PVB$^+$88], where a failed component stops without making any erroneous state transitions, the *omission* failure model [ES86], where components may not respond to some inputs, the *timing* failure model [CASD85], where components may not respond to inputs in a timely manner, and the *Byzantine* failure model [LSP82], where components can suffer arbitrary failures.

An abstraction is a definition of the properties of a system or a component, independent of the details of its implementation. It is typically expressed in terms of the interface to the component and the functionality it provides. Complex software systems are implemented as layers of abstractions, where each abstraction builds on underlying abstractions. These abstractions are implemented by separate program modules, which allows programmers to tackle one small piece of a complex software system at a time, and allows different modules to be developed concurrently by different programmers. Moreover, since interactions between the modules are defined by the abstractions they implement, information flow within the system is readily apparent. This greatly simplifies the debugging of the system during its development and helps contain the effect of faults during its operation. Abstractions, and the resulting modularization, have also been used for building fault-tolerant *domain specific systems*, in which the systems can be configured to fit the needs of a given application [HS93].

Software structuring paradigms facilitate the task of writing fault-tolerant software by providing the programmer with standard ways of structuring this type of software. Three such paradigms have been developed: the *object/action* paradigm, the *restartable action* paradigm, and the *replicated state machine* paradigm. The object/action paradigm consists of objects, which encapsulate data, and actions, which are threads of execution that modify the data in one or more objects by invoking operations on objects. The paradigm ensures that the consistency of the data is maintained despite failures. The object/action paradigm is also called the transactional paradigm, and is useful for applications that must keep data consistent.

The restartable action paradigm consists of a thread of execution called an action, which can be restarted after a failure. The restarted action produces the same outputs that would have been produced had there been no failures. This paradigm is most commonly implemented by checkpointing schemes that allow processes to periodically save their state on a permanent storage device and, on recovery from a failure, restart the process from the last saved state. Even thought the restartable action paradigm is based on a single thread of execution or process, it is useful for structuring systems consisting of interacting processes.

The replicated state machine paradigm consists of state machines, which encapsulate state variables and modify them only in response to commands. The values of these variables are therefore uniquely determined by the sequence of commands received by a state machine. Fault-tolerance is achieved by replicating state machines on different machines and delivering all commands to all replicas in a consistent order. This paradigm is essentially a formalization of active replication. It is commonly used to mask failures in systems that must continue to deliver timely service despite failures.

Adequate programming language support can also greatly simplify the task of building any software system, and fault-tolerant systems are no exception. Numerous languages, extensions to existing languages, and language libraries have therefore been developed to provide the programmer with mechanisms that support commonly used fault-tolerance techniques. Examples of such languages include Argus [Lis85], Aeolus [LW85], and Plits [EFH82] that were designed from the very outset as languages for fault-tolerant programming; Arjuna [SDP91] and Avalon [HW87], designed as libraries to existing languages; and finally Fault-Tolerant Concurrent C [CGR88], HOPS [Mad86], and the languages described in [KU87], [KMBT92] and [SCP91], designed as fault-tolerance extensions to existing programming languages.

1.3 FT-SR

This dissertation focuses on the area of improving programming language support for constructing fault-tolerant systems. Specifically, the design and implementation of FT-SR, a programming language developed for building a wide variety of fault-tolerant systems is described. FT-SR is based on the concurrent programming language SR [AO93] and is designed as a set of extensions to SR.

A distinguishing feature of FT-SR is the flexibility it provides the programmer in structuring fault-tolerant software. It is flexible enough to be used for structuring systems according to any of the standard fault-tolerance structuring paradigms introduced earlier. This is especially important in systems building because different structuring paradigms are often appropriate for different parts of the same system. It is this flexibility that sets FT-SR apart from all the other fault-tolerant programming languages. Unlike FT-SR, other languages target a certain class of applications, pick the program structuring paradigm best suited to that class of applications, and provide language support for that structuring paradigm. The domain of applicability of such languages is therefore limited. FT-SR, on the other hand, is suitable for programming a variety of systems and applications.

FT-SR derives its flexibility from its programming model, which is based on *fail-stop atomic objects*. This dissertation argues that fail-stop atomic objects are the fundamental building blocks for all fault-tolerant programs. FT-SR provides the programmer with simple fail-stop atomic objects, and mechanisms that allow these fail-stop atomic objects to be composed to form higher-level fail-stop atomic objects that can tolerate a greater number of faults or different kinds of faults. The mechanisms for composing fail-stop atomic objects are based on the standard redundancy techniques discussed above. For example, FT-SR provides mechanisms that allow the programmer to use active redundancy to replicate a fail-stop atomic object to form a higher-level object that exports the same set of operations, but is more fault-tolerant. This ability to combine the basic building blocks in a variety of ways allows programmers to structure their programs in a manner best suited to the application at hand.

FT-SR has been completely implemented. The implementation consists of two main components: a compiler and a runtime system. Both the compiler and the runtime system are implemented in C and borrow from the existing implementation of SR. The compiler translates FT-SR code to C code, which is then linked with the runtime system. The runtime system provides primitives for creating, destroying and monitoring objects and object groups, handling failures of objects, restarting failed objects, invoking and servicing operations, and a variety of other miscellaneous functions. The runtime system is implemented using version 3.1 of the *x*-kernel [HP91]—an operating system kernel designed for experimenting with communication protocols—and runs stand-alone on Sun 3s. The implementation has also been ported to Unix.

1.4 Dissertation Outline

This dissertation is organized as follows. Chapter 2 describes in detail the different structuring paradigms for fault-tolerant programs. Programming languages that support each of these paradigms are also discussed and a case made for unifying these ideas into a single programming language.

Chapter 3 presents the programming model underlying FT-SR. In particular, it defines the notion of fail-stop atomic objects more precisely and shows how they can be used to structure programs. The way in which FT-SR has been designed to support fail-stop atomic objects is then described using simple example programs.

Chapter 4 shows how FT-SR can be used to build different kinds of systems. The examples illustrate the use of different language mechanisms in different contexts that result from the various fault-tolerance structuring paradigms.

Chapter 5 describes the implementation of FT-SR using the *x*-kernel. The compiler and runtime system are described, and the pros and cons of using the *x*-kernel discussed. The Unix port of this implementation is also described and finally some performance numbers presented.

Chapter 6 evaluates the design and implementation of FT-SR. The strengths of the programming model, language mechanisms and implementation are highlighted along with some of the observed deficiencies of the langauge.

Chapter 7 makes some concluding remarks and presents some future research ideas for FT-SR.

CHAPTER 2

FAULT-TOLERANT SOFTWARE

Software running on a dependable system must be fault-tolerant, i.e., it must continue to deliver correct service despite the failure of the underlying computing platform. This chapter describes the challenges faced by designers and programmers of fault-tolerant software, the techniques used to simplify this difficult task, and the programming languages that support these techniques.

Developing fault-tolerant software is an extremely difficult task for two main reasons. First, faults can occur in any part of the system at anytime. This problem is compounded by the fact that these faults can cause different system components to fail in different ways. As a result, the designer of fault-tolerant software must anticipate every possible failure and deal with its consequences. The second reason is that dependable systems are often built using distributed computer systems. Therefore, the designer must also deal with all the problems associated with distributed software, which include the difficulty of managing concurrently executing threads of execution and dealing with inconsistent views of the system at different sites.

To address these difficulties, techniques such as the use of failure models, abstractions, and standard system structuring paradigms have been developed. Each of these techniques is described in greater detail in the following sections. Numerous programming languages have also been developed that support these techniques. Examples of such languages and their support for the fault-tolerance techniques are also presented in this chapter.

2.1 Failure Models

A failure model specifies a designer's assumptions about the nature of failures a system, or component of a system, can suffer. It characterizes *how* a component will fail without making any statement about the actual causes of the failure. A failure model therefore limits the number and kinds of failures the system developer must anticipate and deal with to a manageable amount. The result is a simplification of the design process.

Failure models have been developed for different layers of a computer system ranging from the logic-gate layer to complex system components. The classical logic-gate level failure model is the *stuck-at* failure model, where failures manifest themselves as inputs or outputs of gates permanently "stuck" at logic 0 or logic 1 [AA86]. This fault-model is used extensively in designing tests for circuits. The failure models for logic circuits, which are composed of logic-gates, are formulated in terms of the values of the outputs produced by the circuit. A common failure model for logic circuits is the *fail-safe* failure model that assumes a circuit will never produce results that can cause failures in other parts of the system [Toh86].

Failure models for more complex system components are often defined in terms of the errors they produce in the output [Pow92]. Such errors can be classified along two dimensions: *value* and *timing*. An output has a value error if the value or content of the output is outside a specified range of values, while it has a timing error if it is produced outside a specified range of times. Different failure models make assumptions of different strengths about the types of value and timing errors that can occur. For example, the strongest or most restrictive failure model assumes no value or timing errors, i.e. components never fail. At the other extreme, the weakest or least restrictive failure model assumes components can suffer from arbitrary value and timing errors. This failure model is called the *Byzantine* failure model [LSP82]; it even allows for failure scenarios where it might appear as if failed components are colluding with each other to create a worst-case situation. Byzantine failures are used to model the components of *ultra-dependable systems*, which must have an extremely low probability of failure.

Other useful failure models make assumptions about value and timing errors that lie somewhere between the no failure and arbitrary failure assumptions. The *fail-silent* or *crash* failure model [PVB$^+$88] assumes that components fail by producing timing errors where all outputs are infinitely delayed. Such components therefore behave correctly until they fail and produce no output thereafter. The fail-silent failure model is most commonly used to model processor failures. The *omission* [ES86] failure model also assumes no value failures but allows timing failures where an occasional output is infinitely delayed. These components therefore produce correct outputs on time or not at all. The omission failure model includes the more restrictive crash model because a crash failure may be viewed as an omission failure of all outputs after the time of failure. Omission failures are used to model components like a network that occasionally drops packets. The *performance* or *late* [CASD85] failure model also assumes no value failures but allows timing errors where some outputs might be produced later than the specified range of times. Outputs are therefore correct and on time, or correct but late. The performance failure model includes the omission failure model because an omission failure may be viewed as a timing failure where the output is infinitely late. Performance failures are often used to model real-time systems.

Numerous variations of the above failure models have also been developed. For example, the *Byzantine failures with authentication* failure model is a variant of the Byzantine failure model where it is possible to verify the source of all messages exchanged within the system [LSP82]. This added assumption greatly reduces the complexity of the algorithms used in systems that assume Byzantine failures. The *fail-stop* failure model is a variation of the crash failure model where the failure of a component is assumed to be detectable by other functioning components [SS83]. This assumption allows other components to differentiate between a failed component and one that is merely slow in responding to inputs.

Since a failure model is merely an assumption about how a system or component may fail, it can possibly be violated during system operation. The more restrictive the failure model, i.e. the more assumptions it makes about failures, the higher the probability that it will be violated. While this favours the use of less restrictive failure models, the complexity of the algorithms used and the degree of redundancy needed to ensure correct operation increases with less restrictive failure models. As a result, the choice of failure models is a trade-off between the degree of dependability required of the system and the complexity of the system. This trade-off is determined by the needs of the application: the greater the dependability required, the greater the justification for the increased complexity and cost of building the system. For example, the designer of a flight control computer for a commercial airplane cannot justifiably use any failure model other than Byzantine because of the lives that would be in jeopardy if the computer failed. On the other hand, the designer of a typical commercial transaction processing systems would have a difficult time justifying the use of anything less restrictive than the crash failure model. If fact, the crash failure model is the most commonly used failure model because of its simplicity and the empirical evidence that points towards a relatively low probability of it being violated [Gra86b].

2.2 Abstractions

An abstraction is a definition of the properties of a system or a component, independent of the details of its implementation. It is typically expressed in terms of the interface to the component and the functionality it provides.

Abstractions simplify the design and implementation of complex software systems. Such systems are designed as a hierarchy of abstractions, where each abstraction builds on other simpler abstractions. Abstraction are implemented by separate units or modules, which are developed independently. This is possible because inter-dependencies between modules are limited to the abstractions they implement and not their implementation.

The use of abstractions and the resulting lack of implementation dependencies between modules has proven useful in building *domain-specific software* systems that are configured to meet the needs of a given application [HS93]. This is done by configuring the system with just those modules that are needed to provide the functionality required by the application. This approach has also been used to dynamically re-configure systems to meet the needs of changing applications [OP92].

Abstractions are particularly useful for building fault-tolerant software. The use of appropriate abstractions to access the services provided by a module limits access to its internal state by other modules. This helps contain the effect of failures; since modules do not share state, errors in the state of a module cannot directly cause errors in the state of other modules. The use of such self-contained modules also allows the software to isolate failed modules and reconfigure the system accordingly. Failed modules can also be easily replaced by other modules implementing the same abstraction.

Some commonly used abstractions for fault-tolerance have been formalized and are often provided to programmers as standardized modules. An example of such an abstraction is that of *stable storage* [Lam81]. Stable storage is an idealized storage device that suffers no failures and is unaffected by failures of other components of the system. Reads and writes to a stable storage complete successfully or do not happen at all, and values written to a stable storage survive all failures. Examples of other standard fault-tolerance abstractions and the relationship between these abstractions can be found in [MS92].

2.3 Program Structuring Paradigms

All programs have a certain structure, which is determined by such factors as the application, the interactions between program modules and their functional dependencies, and the architecture of the underlying computing platform. Determining the structure that is most appropriate for a given program is a difficult task, and the more complex the program, the more difficult this task. Since fault-tolerant programs tend to be exceedingly complex, certain standard program structuring techniques have been developed for these programs. These techniques, called fault-tolerance program structuring paradigms, provide programmers with standard ways of structuring programs and abstractions that help understand the interactions between modules in the program.

The most important fault-tolerant programming structuring paradigms are the object/action model, the restartable action paradigm, and the replicated state machine approach. Since the vast majority of fault-tolerant programs use one or more of these paradigms, understanding these paradigms is fundamental to understanding fault-tolerant software. These paradigms and examples of systems supporting these paradigms are therefore described in greater detail in the rest of this section.

2.3.1 The Object/Action Model

The object/action paradigm consists of *objects* and *actions* [Gra86a]. An object is a passive entity that exports operations. Actions are threads of execution; they are active and invoke operations on objects to carry out tasks. Actions are atomic, which means they are *unitary* and *serializable*. The unitary property ensures that if an action completes successfully, it has been carried out completely. If a failure occurs before the action completes, then after recovery, the external effect of the action is as if it has successfully completed or not been started at all. The serializable property ensures that if several actions are carried out by concurrent processes, the result is always as if the individual actions were carried out one at a time in some serial order. These properties have also been called *totality* and *serializability* [Wei89], and *recoverability* and *indivisibility* [Lis85]. In the database literature, atomic actions are referred to as transactions [BHG87]. The atomic object/action paradigm is best suited for applications that manage persistent data whose consistency must be maintained despite failures.

The unitary property of actions is implemented by a *commit protocol* [BHG87]. Such a protocol ensures that when an action completes, either all objects affected by the action *commit* the action or all *abort* the action. A commit installs the effects of the action while an abort undoes its effects so it appears as if the action was never started. The *two-phase commit* protocol is the best known commit protocol [Gra79].

The serializability property of actions is implemented by *locking protocols*. Locking protocols prevent actions from interfering with one another by controlling access to shared resources. The most common locking protocol is called *two-phase locking* [EGLT76].

TABS

TABS is an example of a system supporting the object/action model [SDD+85]. It is a *distributed transaction facility* that supports the implementation of objects and actions in a distributed system, in addition to standard abstractions such as processes and inter-process communication. TABS supports user-specified objects that are implemented within server processes; each such process controls access to one class of objects. Servers implement locking, commit, and recovery protocols to ensure the atomicity of actions in the system.

The TABS programmer interface is implemented as a collection of three libraries: the *server* library, which provides routines for shared/exclusive locking and logging, a *transaction management* library, which provides routines for controlling the execution of transactions, and a *name server* library, which provides services that map object names to communication ports. Programmers use these library routines to implement classes of servers for the objects in the system. Since the support provided by TABS is fairly primitive, programmers must explicitly manage concurrency within servers by locking.

2.3.2 The Restartable Action Paradigm

A restartable action is a thread of execution that, when restarted after a failure, will produce the same outputs had there been no failure [SS83]. Some actions are naturally restartable and are called *idempotent* actions. For example, an action that copies a file from one directory to another is an idempotent action; if it fails during execution it can be restarted without any undesirable side-effects. Actions that are not naturally restartable can usually be made restartable so long as the duplication of any external effects of the action is not undesirable.

Restartable actions are used to program processors that suffer fail-stop failures. It is assumed that these actions have access to stable storage in addition to the usual volatile storage. They modify the contents of these storage devices and take them from an *initial* state to a *final* state. If the processor fails during execution, the action is restarted on a functioning processor. On restart, it executes *recovery code* that uses information on the stable storage to restore the system to a state from which the action can continue to execute to completion, taking it to the same final state it would have reached had there been no failures.

Restartable actions are most commonly implemented by checkpointing schemes. These schemes periodically save the entire state of an action on stable storage. If the action fails, it can be restarted by restoring its state from the last available checkpoint.

Checkpointing schemes therefore implement the backward recovery technique described in Chapter 1.

Even though restartable actions are described in terms of a single process executing on one processor, the paradigm can be extended to systems of interacting processes in a distributed system. The global state of such a system is a set of states, one from each process [SMR88]. This global state can be checkpointed by individually checkpointing each processes. However, the checkpointing and recovery of these processes must be coordinated with each other to avoid situations where the restart of a failed process p results in an inconsistent global system state and therefore requires the rollback of another process q to a previous checkpoint. Such a situation can arise if process p had sent a message to q between the time of its last checkpoint and point of failure. A rollback of p to this checkpoint results in a situation where q has a message that, according to p's new state, has not yet been sent. This requires a rollback of q to a checkpoint taken before it received the message. This may, in turn, require a third process r to be rolled back, setting off a cascade effect that results in all processes being rolled back. This effect is called the *domino effect*. A well-know technique used to prevent such a domino effect is called *conversations* [Ran75]. Alternatively, message logging and replay techniques can be used [JZ90, SW89, SY85].

UNICOS

UNICOS, an operating system for Cray machines derived from AT&T Unix System V, is an example of a system that supports restartable actions using checkpointing [KK89]. In addition to the standard Unix system calls, it provides two system calls for checkpointing and restart: `chkpnt` and `restart`. `chkpnt` is used to checkpoint a process; it creates a *restart* file containing the information needed to restore the process to execution at a later time. `restart` accepts a restart file and restores the process to the stored state. In addition to providing these system calls, UNICOS defines a new Unix signal called *SIGRECOVERY* that is sent to a restarted process. This signal may be fielded by the process and recovery code executed by the signal handler.

2.3.3 The Replicated State Machine Paradigm

In this paradigm, services are implemented by state machines. A state machine consists of *state variables*, which encode its state, and *commands* that transform its state [Sch90]. Each command is implemented by a deterministic program and executes atomically with respect to other state machine commands. As a result, the state of a machine and any outputs it produces are determined solely by the sequence in which it processes commands. The availability of a service is increased by replicating the state machine implementing the service and placing the replicas on different processors. To ensure that these replicas make the same transitions, commands to the state machine are processed by all replicas in the same order. The mechanism that delivers these commands therefore satisfies the *agreement* and *order* properties. The agreement property guarantees all functioning replicas receive every request and the order property guarantees all replicas receive commands in the same relative order.

Most fault-tolerance schemes that involve replicating data or processing are based on the replicated state machine paradigm. This paradigm allows systems to mask failures, thereby alleviating the need for expensive error processing. This is especially important in real-time systems that cannot afford the time needed for removing errors caused by faults. The replicated state machine paradigm is also used by systems based on the Byzantine failure model.

The replicated state machine paradigm is supported by systems like Circus [Coo85], Consul [Mis92], Delta-4 [Pow91], and Isis [BSS91]. Among other things, these systems provide the programmer with mechanisms that allow commands to be reliably delivered to groups of processes in a consistent order.

Consul

Consul is a platform for building fault-tolerant distributed systems that are based on the replicated state machine paradigm. It provides the services needed to implement the state machine paradigm. These services include *multicast* services that reliably deliver messages to a collection of processes, *membership* services that maintain a consistent system-wide view of which processes are functioning and which are failed, and *recovery* services that facilitate recovery of failed processes.

Consul is implemented as a collection of protocols, where each service is implemented by one or more protocols. At the heart of Consul is the *Psync* group-oriented communication protocol [PBS89]. Psync allows a group of processes to exchange messages in a way that explicitly preserves the consistent *partial order* of the messages. This partial order, which is a variation of the *happened before relation* [Lam78], is represented in Psync as a directed acyclic graph called the *context graph*. The other protocols within Consul use Psync to implement the services they provide. These protocols typically query Psync about the status of messages in the context graph and use the information to implement their functionality. Examples of protocols that are provided by Consul include total-order, semantic-dependent ordering, membership, and recovery.

2.4 Programming Language Support for Fault-Tolerance

Programming languages that support the failure models, abstractions, and structuring paradigms for fault-tolerance can greatly ease the task of writing fault-tolerant programs. Numerous such languages have been developed, some in the form of entirely new languages designed specifically for programming fault-tolerant software, some in the form of extensions to existing languages and systems, and some in the form of libraries to existing languages. Examples of languages that were specifically designed for fault-tolerant programming include Argus [Lis85], Aeolus [LW85] and Plits [Fel79, EFH82]. Examples of extensions to existing languages include Fault-Tolerant Concurrent C [CGR88], HOPS [Mad86], and languages described in [KU87], [KMBT92] and [SCP91] which extend Concurrent C [GR89], Modula-2, Ada [DoD83], Orca [BKT92] and SR respectively. Finally, fault-tolerance library support for existing languages is provided by Arjuna [SDP91] for C++, and Avalon [HW87] for C++, Common Lisp and Ada.

Different languages provide varying degrees of support for failure models, abstractions, and program structuring paradigms. They all assume the crash failure model and provide the support necessary to handle crash failures. This support is usually in the form of protocols and algorithms within the language runtime systems that are optimized for crash failures. Support for abstractions is usually provided in the form of an object-oriented programming model. In this model, abstractions are implemented by objects, which are the basic units of encapsulation and modularization. HOPS and the languages based on C++ are fully object oriented, while Aeolus, Argus, Plits, and the languages based on Ada and SR provide for abstract data types and encapsulation.

The support for the fault-tolerance structuring paradigms is what really distinguishes these languages. For example, the object/action model is supported by Aeolus, Argus, Avalon, HOPS, Plits, and Arjuna. Specifically, the Aeolus language has been designed to provide access to the synchronization and recovery features of Clouds [DLAR91], a fault-tolerant distributed operating system based on objects and actions. Avalon is a set of linguistic constructs designed to give programmers explicit control over the processing of atomic actions. It allows them to test transaction serialization orders at runtime and specify commit and abort operations. These capabilities allow programmers to exploit the semantics of the applications to enhance efficiency, concurrency, and fault-tolerance. HOPS allows the programmer to specify the concurrency control and recovery mechanisms used by the language runtime system to implement atomic actions. For example, the language allows the programmer to choose between the two-phase commit protocol and timestamp ordering for concurrency control, and between logs and shadows for recovery. Plits tags actions in the system with *activity* tags that are visible to the programmer. Programmers use the activity tag associated with an action to tag the data items it modifies. These tags, combined with language mechanisms that select data based on their tag values, greatly simplify the task of writing concurrency control and commit protocols. Finally, Argus allows the programmers to specify objects and the start and end of actions. It is described in more detail in Section 2.4.1.

The replicated state machine paradigm is supported by HOPS and Fault-Tolerant Concurrent C. HOPS allows the programmer to associate a replication attribute with the objects in the system. These attributes specify the type of replication desired: replication with quorum, primary-backup replication, or standard replication. Fault-Tolerant Concurrent C provides primitives that can be used to create and manage replicated processes. It is described in greater detail in Section 2.4.2.

Support for the restartable action paradigm is usually provided in the form of operating system calls for checkpointing and restarting processes. Checkpointing can be made transparent to the programmer by having the language runtime system automatically checkpoint programs at regular intervals. Such automatic checkpointing is however extremely complicated in a distributed program with concurrently executing threads because of the need for checkpoints to be mutually consistent [RLT78]. Orca is an example of a language that can automatically checkpoint executing programs. In particular, it provides automatic checkpointing that works for programs that have a long compute phase during which they do not interact with users or perform I/O. The mutual consistency of the checkpoints is ensured by coordinating the checkpointing using a reliable broadcast based algorithm.

Finally, the extensions to SR proposed in [SCP91] do not explicitly support any of the structuring paradigms. The extensions do, however, provide an ordered multicast mechanism that can be used to implement the replicated state machine paradigm. They also allow the programmer to define variables as being "stable"; such variables are stored on stable storage and therefore survive failures. This allows the programmer to implement fine grained checkpointing, and hence, the restartable action paradigm.

Two of the above languages and systems, Argus and Fault-Tolerant Concurrent C are described in detail in the remainder of this section. These languages support the object/action and replicated state machine fault-tolerance structuring paradigms respectively, and are representative of their class of languages.

2.4.1 Argus

Argus is an integrated programming language and system that supports the object/action paradigm. It allows the programmer to specify objects and ensures that programmer-defined actions are executed atomically. User specified objects in Argus are called *guardians*, which also serve as units of encapsulation. Operations exported by a guardian are called *handlers*. Actions may span multiple guardians and when an action commits, changes to resources modified by the action are made permanent by all guardians involved in the action. Actions can be aborted by the application program or by the system due to a failure. All changes made by an aborted action are automatically undone by the system. The programmer can associate exception handlers with actions; these exception handlers can take appropriate action based on the cause of an abort.

A guardian runs on a single node and is lost if the node crashes. However, the state of a guardian consists of both stable and volatile objects. Stable objects are stored on stable storage and on recovery from a crash, the language runtime system recreates the guardian and restores the stable objects. A programmer-defined recovery process is then started in the guardian to recreate the volatile objects. Once the volatile objects are restored, the guardian can resume background tasks and respond to new handler calls.

2.4.2 Fault-Tolerant Concurrent C

Fault-Tolerant Concurrent C (FTCC) is an extension to Concurrent C [GR89] that supports the replicated state machine paradigm. FTCC allows the programmer to build replicated state machines by replicating processes. Specifically, FTCC extends the Concurrent C process creation statement to create multiple copies of a process on one or more processors. The entire replicated process ensemble is identified by a single process identifier, with calls using this identifier being delivered by the FTCC runtime system to all members of the ensemble. A distributed consensus protocol in the runtime system ensures that all replicas process messages in the same total order.

FTCC provides the programmer with the ability to detect process failures, where a process is defined to have failed if all its replicas have been lost due to failures or explicitly destroyed by the program. The programmer can ask to be synchronously or asynchronously notified of such failures. Synchronous failure notification is provided by a *fault expression* associated with the call statement that is executed if the call fails. Asynchronous failure notification is provided by the built-in function `c_request_death_notice`, which takes two arguments: the identifier of a process to be monitored and a function to be called by the runtime system when the process fails. This monitoring can be terminated using the built-in function `c_cancel_death_notice`, which takes a process identifier as its argument.

2.5 Summary

This chapter described the difficulties associated with building fault-tolerant software and techniques such as failure models, abstractions, and structuring paradigms used to simplify this task. The programming languages that support these techniques were then presented. It was noted that while all these languages had very similar kinds of facilities to support failure models and abstractions, they differed considerably in their support for the structuring paradigms. Almost every language chose to support one paradigm and was therefore well suited for programming classes of applications that conformed to that paradigm. Even languages that attempted to support more than one structuring paradigm had extensive support for a paradigm and very rudimentary support for the others. Support for a single-paradigm has been shown to be constraining in many situations and is particularly inappropriate for constructing systems where different structuring paradigms are appropriate for different levels of abstraction[Bal91]. FT-SR, a language for constructing fault-tolerant systems that is introduced in the following chapters, avoids this shortcoming by supporting equally well any of the fault-tolerance structuring paradigms.

CHAPTER 3

FT-SR: PROGRAMMING MODEL AND LANGUAGE DESCRIPTION

FT-SR has been designed to be a programming language versatile enough for building a wide variety of fault-tolerant software systems. This versatility is derived from its programming model, which allows the programmer a great deal of flexibility in structuring systems. This chapter describes this programming model and the language mechanisms of FT-SR that support the model.

3.1 The FT-SR Programming Model

The FT-SR programming model assumes that all fault-tolerant programs are composed of fail-stop (FS) atomic objects. Such an object contains one or more threads of execution, which implement a collection of operations that are exported and made available for invocation by other FS atomic objects. These operations execute as atomic actions, i.e., they satisfy the unitary and serializability properties. However, as is always the case with fault-tolerance, these properties can only be *approximated* by an implementation, i.e., they can be only guaranteed *relative* to some set of assumptions concerning the number and type of failures. For example, algorithms to realize the unitary property often rely on stable storage in such a way that the failure of this abstraction can lead to unpredictable results. Or, a series of untimely failures might exhaust the redundancy of an implementation built using replication.

To account for cases such as these, the semantics of FS atomic objects include the concept of *failure notification*. Such a notification is generated for a particular object whenever a *catastrophic failure* occurs, where such a failure is defined to occur when an object's implementation assumptions are violated, or should the object be explicitly destroyed from within the program. The status of an operation being executed when such a failure notification occurs is indeterminate. Hence, the analogy to fail-stop processors implied by the term "fail-stop atomic objects" is strong: in both cases, either the abstraction is maintained (processor or atomic object) or notification is provided.

A fault-tolerant distributed system can be realized by a collection of FS atomic objects organized along the lines of functional dependencies. For example, an FS atomic object implementing the services of a transaction manager may use the operations exported by another FS atomic object implementing the abstraction of stable storage. These dependencies can be defined more formally using the *depends* relation given in [Cri91]. In particular, an FS atomic object u is said to *depend* on another object v if the correctness

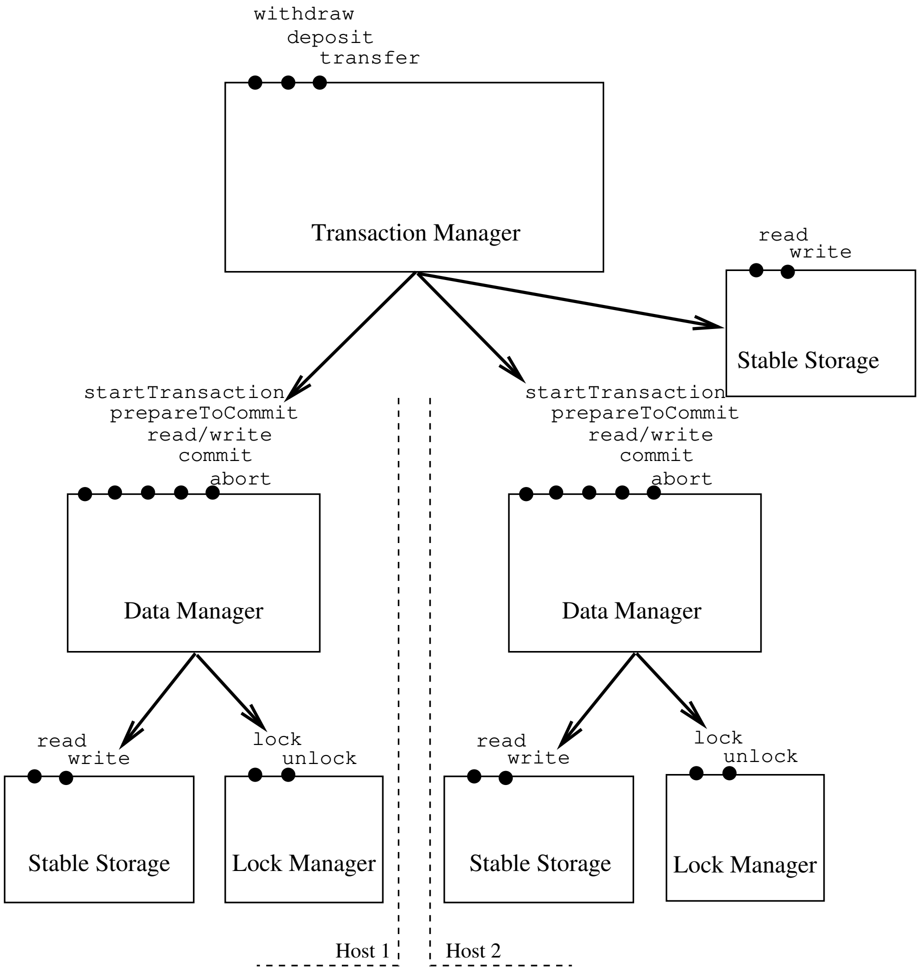

Figure 3.1: Fault-tolerant system structured using FS atomic objects

of u's behavior depends on the correctness of v's behavior. Thus, the failure of v may result in the failure of u, which in turn can lead to the failure of other objects that depend on u.

Increasing the dependability of a distributed system organized in this way is done by decreasing the probability of failure of its constituent FS atomic objects using fault-tolerance techniques based on the exploitation of redundancy. For example, an object can be replicated to create a new FS atomic object with greater resilience to failures. This replication can either be active, where the states of all replicas remain consistent, or passive, where one replica is a primary and others remain quiescent until a failure occurs. Or, an FS atomic object can contain a recovery protocol that would be executed upon restart following a failure to complete the state transformation that was in effect when the failure occurred. The applicability of each of these techniques depends on the details of the system or the application being implemented.

As an example of how a typical fault-tolerant system might be structured using FS atomic objects, consider the simple distributed banking system shown in Figure 3.1. Each box represents an FS atomic object, with the dependencies between objects represented by arrows. User accounts are assumed to be partitioned across two processors, with each data manager object managing the collection of accounts on its machine. The user interacts with the transaction manager, which in turn uses the data managers and a stable storage object to implement transactions. The transaction manager acts as the coordinator of the system; it decides if and when a transaction is to be committed and coordinates the two-phase commit protocol [Gra79] that is used to ensure that all data managers involved agree on the outcome of the transaction. Associated with the transaction manager is a stable storage object, which it uses to log the progress of transactions in the system. The data managers export operations that are used by the transaction manager to read and write user accounts, and to implement the two-phase commit protocol. The stable storage associated with each data manager is used to store the actual data corresponding to user accounts, and to maintain key values that can be used to restore the state of the data manager should a failure occur. The lock managers are used to control concurrent access.

To increase the overall dependability of the system, the constituent FS atomic objects would typically be constructed using fault-tolerance techniques to increase their failure resilience. For example, the transaction and data managers might use recovery protocols to ensure that data in the system is restored to a consistent state following failure. Similarly, stable storage might be replicated to increase its failure resilience. The failure notification aspect of FS atomic objects is used to allow objects to react to the failures of objects upon which they depend. If such a failure cannot be tolerated, it may, in turn, cause subsequent failures to be propagated up the dependency graph. At the top level, this would be viewed by the user as the catastrophic failure of the transaction manager and hence, the system. Such a situation might occur, for example, should the redundancy being used to implement stable storage be exhausted by an untimely series of failures. In Chapter 4, we illustrate how the data manager and stable storage FS atomic objects from this example might be implemented using FT-SR.

Fail-stop atomic objects and the associated techniques for increasing failure resilience form a "lowest common denominator" that can be conveniently used to realize the seemingly disparate programming paradigms proposed for fault-tolerant programming. For example, consider the object/action model. A system built using the object/action paradigm may be implemented using FS atomic objects. Objects in the system correspond to FS atomic objects. An action corresponds to an abstract thread realized by the combination of concrete threads in the FS atomic objects. This abstract thread may span multiple FS atomic objects as a result of invocations made by concrete threads that are serviced by concrete threads in other objects. Standard locking and commit protocols are used to ensure the unitary and serializable nature of these actions across multiple objects. Viewed as a whole, this system appears to the user as one FS atomic object exporting the set of operations required by the application.

As a second example, consider the replicated state machine paradigm. State machines map directly to FS atomic objects. Commands to the state machine correspond to invocations on an FS atomic object, with locking techniques being used to ensure that the operation executions are atomic. A replicated state machine can be implemented by replicating FS atomic objects and ensuring that all commands to the state machine result in invocations on all replicas in a consistent order. Each ensemble of replicated FS atomic objects forms a higher-level FS atomic object representing the fault-tolerant version of a given state machine. The entire collection of such FS atomic objects can then be viewed as a single FS atomic object that implements the entire system.

Finally, consider the restartable action paradigm. Restartable actions can be easily implemented as threads executing in an FS atomic object that is restarted on a failure. The recovery protocol that is executed when the failed object is restarted can be used to restore the failed threads to a state from which they can execute to completion. The failure notification that is generated when the FS atomic object fails allows other objects in the system to learn of the failure and restart the failed object. Barring catastrophic failures that cause the recovery protocol to fail or the failed object to not be restarted at all, operations implemented by restartable threads will eventually complete successfully. In case of such catastrophic failures, the system comprising the object will fail and a failure notification generated. The entire system can therefore be viewed as a single FS atomic object.

The system shown in Figure 3.1 is an example of a system in which FS atomic objects are used to implement different programming paradigms in different parts of the system. Specifically, the transaction and data managers are built using the restartable action paradigm, while the stable storage objects are built using the replicated state machine approach. The user of the banking system sees the system as one implementing the atomic object/action paradigm and interacts with it accordingly.

3.2 The FT-SR Language Description

The goal of FT-SR is to support the building of systems based on the FS atomic object model and thus, by implication, the building of systems using any of the existing programming paradigms. Given the need for flexibility, we do not provide these objects directly in the language, but rather include features that allow them to be easily implemented. To this end, the language has provisions for encapsulation based on SR resources, resource replication, recovery protocols, synchronous failure notification when performing interprocess communication, and a mechanism for asynchronous failure notification based on a previous scheme for SR [SCP91]. Since our extensions are based on existing SR mechanisms, a short overview of the language is provided in Appendix A; for further details, see [AOC+88, AO93].

3.2.1 Simple FS Atomic Objects

Realizing much of the functionality of a simple FS atomic object—i.e., one not composed of other objects or using any other fault-tolerance techniques—in SR is straightforward since a resource instance is essentially an object in its own right. For example, it has the appropriate encapsulation properties and is populated by a varying number of processes that can function as threads in the FS atomic object model. SR operations are also very similar to the operations defined by the model; they are implemented by processes and can be exported for invocation by processes in other resource instances. Moreover, the execution semantics of SR operations are already close to those desired for FS atomic objects; the only additional property required is atomicity of operation execution in the absence of catastrophic failures, which occur for simple objects when the resource instance is destroyed due to failure or explicit termination. Ensuring atomicity therefore reduces to ensuring serializability, which can easily be programmed in SR by, for example, implementing each exported operation as a separate alternative in an input statement repeatedly executed by a single process. Standard locking-based solutions that allow more concurrency are also easy to implement in SR.

Figure 3.2 shows the outline of a simple FS atomic object that implements atomicity by use of an SR input statement. The object is a lock manager that controls access to a shared data structure implemented by some other resource. It exports two operations: a `get_lock` operation that is invoked by clients wishing to access the shared data structure and a `rel_lock` operation that clients invoke when they are done. If a client invokes the `get_lock` operation and the lock is available, a `lock_id` is returned. If the lock is unavailable, the client is blocked at the first guard of the input statement. The `get_lock` operation takes as its argument the capability of the invoking client. This capability is used as a means of identifying the client.

The one aspect of simple FS atomic objects that SR does not support directly—and hence the focus of our extensions in this area—is generation of a failure notification. As mentioned earlier, for simple objects this occurs when the processor executing the resource instance fails, or when the resource instance or its virtual machine is explicitly destroyed from within the program. In Chapter 5, we discuss how this failure is detected by the language runtime system, so here we concentrate on describing the mechanisms that are provided to field this notification in other resource instances. These facilities allow an abstract object to react to the failure of other objects on which it depends.

FT-SR provides the programmer with two different kinds of failure notification and consequently, two different ways of fielding a notification. The first is synchronous with respect to a call; it is fielded by an optional *backup operation* specified in the calling statement. The second kind of notification is asynchronous; the programmer specifies a resource to be monitored and an operation to be invoked should the monitored resource fail. To understand the need for these two kinds of failure notification, consider what might happen if the lock manager shown in Figure 3.2 or any of its clients fail. If the lock manager fails, all clients that are blocked on its input statement will remain blocked forever. Clients can use the FT-SR synchronous failure notification facility to unblock themselves from the call and take some recovery action in the event of such a failure. Figure 3.3 shows the outline of a client structured in this way. The statement of interest is where the client makes a call invocation using a capability to the lock manager's `get_lock` operation `lock_mgr_cap.get_lock`. Bracketed with this capability is the

```
resource lock_manager
  op get_lock(cap client) returns int
  op rel_lock(int; cap client)
body lock_manager
  var ...variable declarations...

  process lock_server
    do true ->
      in get_lock(client_cap) and lock_available() ->
        ...mark lock_id as being held by client_cap...
        return lock_id

      [] rel_lock(client_cap, lock_id) ->
        ...release lock...
        return
      ni
    od
  end lock_server
end lock_manager
```

Figure 3.2: Simple FS atomic object

```
resource client
  op ...
  op ...
body client()
  var lock_id:  int
  op mgr_failed(cap client) returns int
  ⋮
  lock_id := call {lock_mgr_cap.get_lock, mgr_failed} (myresource())
  ⋮

  proc mgr_failed(client_cap) returns lock_err
    return LOCK_ERR
  end mgr_failed
end client
```

Figure 3.3: Outline of Lock Manager client

capability to a backup operation `mgr_failed`. This backup operation is invoked should the call to `lock_mgr_cap.get_lock` fail, where the call is defined to have failed if the lock manager fails before it can reply to the call. In this example, the backup operation `mgr_failed` is implemented locally by the client, which we expect will be the most common usage; in general, however, the backup operation can be implemented by any resource. Note that the backup operation is called with the same arguments as the original operation and, hence, must be type compatible with the original operation. Backup operations can only be specified with call invocations; send invocations are non-blocking and no guarantees can be made about the success or failure of such an invocation if the resource implementing the operation fails. Both call and send invocations are guaranteed to succeed in the absence of failures. Execution is blocked if a call fails and there is no associated backup operation.

Consider now the inverse situation where a client of the lock manager fails. If the client fails while it holds a lock, all other clients will be prevented from accessing the shared data structure. The server can use the FT-SR asynchronous failure notification facility to detect such a failure and release the lock, as shown in Figure 3.4. This figure is identical to Figure 3.2 except for the **monitor** statement in the `get_lock` operation and the **monitorend** statement in the `rel_lock` operation. The server uses the **monitor** statement to enable monitoring of the client instance specified by the resource capability `client_cap`. If the client is down when the statement is executed or should it subsequently fail, the operation `rel_lock` will be implicitly invoked by the language runtime system with the `client_cap` and `lock_id` as arguments. Arguments to the operation specified in the **monitor** statement are evaluated at the time the **monitor** statement is executed and not when the failure occurs. Monitoring is terminated by the **monitorend** statement, which also takes a resource capability as its argument (as shown in Figure 3.4) or by another **monitor** statement that specifies the same resource. The ability to request asynchronous notification has proven to be convenient in a variety of contexts [CGR88, SCP91, BMZ92] and is in keeping with the inherently asynchronous nature of failures themselves.

3.2.2 Higher-Level FS Atomic Objects

FT-SR provides mechanisms for supporting the construction of more fault-tolerant, higher-level FS atomic objects using replication, and for increasing the resilience of objects to failures using recovery techniques. The replication facilities allow multiple copies of an FT-SR resource to be created, with the language and runtime providing the illusion that the collection is a single resource instance exporting the same set of operations. The SR create statement has been generalized to allow for the creation of such replicated resources, which we call a *resource group*. For example, the statement

```
lock_mgr_cap := create (i := 1 to N) lock_manager()
          on vm_caps[i]
```

```
resource lock_manager
  op get_lock(cap client) returns int
  op rel_lock(int)
body lock_manager
  var ...variable declarations...

  process lock_server
    do true ->
      in get_lock(client_cap) and lock_available() ->
        ...mark lock_id as being held by client_cap...
        monitor client_cap send rel_lock(client_cap, lock_id)
        return lock_id

      [] rel_lock(client_cap, lock_id) ->
        ...release lock if held by client_cap...
        monitorend client_cap
        return
      ni
    od
  end lock_server
end lock_manager
```

Figure 3.4: Lock Manager with client monitoring

creates a resource group with N identical instances of the resource `lock_manager` on the virtual machines specified by the array of virtual machine capabilities `vm_caps`. Both the quantifier `(i := 1 to N)` and **on** clauses are optional. If they are omitted the statement reverts to the semantics of the normal SR statement, which creates one instance of the named resource on the current virtual machine.

The value returned from executing the **create** statement is a resource capability that provides access to the operations implemented by the new resource(s). If a single resource instance is created, the capability allows the holder to invoke any of the exported operations in that instance as provided for in normal SR. If, on the other hand, multiple identical instances are created, the capability is a *resource group capability* that allows multicast invocation of any of the group's exported operations. In other words, using this capability in a **call** or a **send** statement causes the invocation to be multicast to each of the individual resource instances that make up the group. All such invocations are guaranteed to be delivered to the runtime of each instance in a consistent total order. This means, for example, that if two operations implemented by alternatives of an input statement are enabled simultaneously, the order in which they will be executed is consistent across all functioning replicas. Moreover, the multicast is also done atomically, so that either all replicas receive the invocation or none do. This property is guaranteed by the runtime system given no greater than *max_sf* simultaneous failures, where *max_sf* is a parameter set by the user at compile time. The combination of the atomicity and consistent ordering properties means that an invocation using a resource group capability is equivalent to an *atomic broadcast* [CASD85, MSMA90]. The results of a multicast call invocation are collected by the runtime system, with only a single result being returned to the caller; since FT-SR assumes processors with fail-silent semantics, returning the first result is sufficient in this case.

In addition to this facility for dealing with invocations coming into a resource group, provisions are also made for coordinating outgoing invocations generated within the group. There are two kinds of invocations that can be generated by a group member. In some cases, a group member may wish to communicate with a resource instance as an individual even though it happens to be in a group. For example, this would be the situation if each replica has its own set of private resources with which it communicates. At other times, the group members might want to cooperate to generate a single outgoing invocation on behalf of the entire group. To distinguish between these two kinds of communication, FT-SR allows a capability variable to be declared as being of type **private cap**. Invocations made using a private capability variable are considered private communication of the group member and not co-ordinated with other invocations from group members. Invocations using regular capability variables are, however, considered to be invocations from the entire group, so exactly one invocation is generated in this case. The invocation is actually transmitted when one of the group members reaches the statement, with later instances of the same invocation being suppressed by the language runtime system. This invocation could, in fact, be a multicast-type invocation as described above if the operation being invoked is within another resource group (i.e., if the capability used in the statement is a resource group capability). It should be noted that a private capability variable can be assigned to a regular capability of the same type and vice versa; whether an invocation is private or not is determined solely by the type of the variable used in making the invocation.

A resource group can also be configured to work according to a primary-backup scheme [AD76]. In this scenario, invocations to the group are delivered only to a replica designated as the primary by the language runtime, with the other replicas being passive. This type of configuration is achieved by placing the op restrictor {**primary**} on the declaration of operations in the group members that are to be invoked only if the replica is the primary.

FT-SR also provides the programmer with the ability to restart a failed resource instance on a functioning virtual machine. The recovery code to be executed upon restart is denoted by placing it between the keywords **recovery** and **end** in the resource text. This syntax is analogous to the provisions for initialization and finalization code in the standard version of SR. A resource instance may be restarted either explicitly or implicitly. Explicitly, it is done by the following statement:

```
restart lock_mgr_cap() on vm_cap
```

This restarts the resource indicated by the capability `lock_mgr_cap` and executes any recovery code that may be specified by the programmer. To restart an entire resource group,

```
restart (i:=1 to N) lock_mgr_cap() on vm_caps[i]
```

is used. The size of the reconstituted group can be different from the original. In both cases, it is important to note that the restarted resource instance is, in fact, a *re-creation* of the failed instance and not a new instance. This means, for example, that other resource instances can invoke its operations using any capability values obtained prior to the failure.

Implicit restart is indicated by specifying *backup virtual machines* when a resource or resource group is created. For example,

```
create lock_mgr() on vm_cap backups on vm_caps_array
```

creates an instance of the lock manager on the virtual machine specified by vm_cap and should this resource instance fail subsequently, it is restarted on one of the backup virtual machines specified in vm_caps_array. The **backups on** clause may also be used in conjunction with the group create statement; in this case, a group member is automatically restarted on a backup virtual machine should it fail. This facility allows a resource group to automatically regain its original level of redundancy following a failure.

Another issue concerning restart is determining when the runtime of the recovering resource instance begins accepting invocations from other instances. In general, the resource is in an indeterminate state while performing recovery, so we choose to begin accepting messages only after the recovery code has completed. The one exception to this is if the recovering instance itself initiates an invocation during recovery; in this case, invocations are accepted starting at the point that particular invocation terminates. This is to facilitate a system organization in which the recovering instance retrieves state variables from other resources during recovery.

Finally, we note that the failure notification facilities described in the previous section work with resource groups as one would expect. For such higher-level FS atomic objects, a catastrophic failure occurs when all the replicas have been destroyed by failure or explicit termination request(s), and there is no system guarantee of recreation. Thus, if a resource is not persistent, a notification is generated once all replicas have been destroyed, while for a persistent resource, a notification is generated once all replicas have been destroyed and the list of backup virtual machines exhausted. In either case, the way in which the notification is fielded is specified using backup operations or the **monitor** statement in the same way as before.

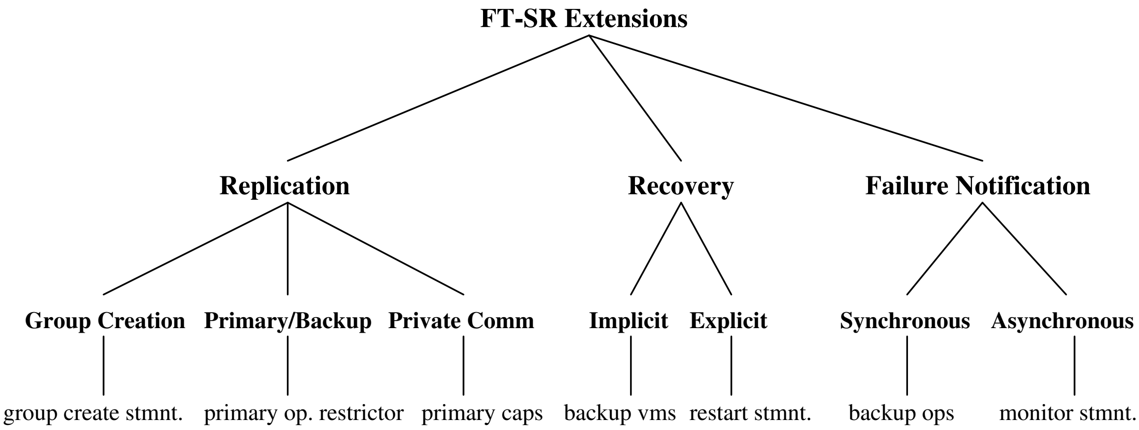

Figure 3.5: Summary of FT-SR extensions

3.3 Summary

The FT-SR programming model is based on FS atomic objects. Fault-tolerant programs are built by constructing simple FS atomic objects and composing them to form higher level FS atomic objects that are more fault-tolerant. The FT-SR programming language supports this model by providing mechanisms for constructing FS atomic objects and for composing them in a variety of different ways. These mechanisms are summarized in Figure 3.5. As shown in the figure, these mechanisms fall into 3 categories: mechanisms for replication, recovery, and failure notification. The mechanisms related to replication are the group creation statement, primary operation restrictor, and private capability variables. The mechanisms related to recovery include the specification of backups and the restart statement for implicit and explicit restarts. Finally, the mechanisms for failure notification are backup operations associated with call invocations and monitor statements.

Although FT-SR is basically SR with extensions for fault-tolerance, there are a few features of SR that are not available to the FT-SR programmer because of their incompatibility with the FT-SR programming model. One such feature is *shared operations*. SR allows global components to declare operations that can be implemented by any resource in the virtual machine. These operations are not associated with any particular resource and therefore do not fit in with the FT-SR programming model, where the association between resources and the operations they export is strong. In particular, the failure of a resource while servicing a call to a shared operation will not result in a backup operation being invoked because of the lack of a well-defined association between the operation and the failed resource.

Another SR feature not available in FT-SR is the **forward** statement. This statement allows an operation invocation to be forwarded to another operation of the same type for execution. The forward statement is not included in FT-SR because of the implementation difficulties caused by its interaction with the synchronous failure handling facility. Specifically, any implementation would have to keep the original invoker of the operation informed of the moves of the invocation, an expensive proposition. The cost and complexity involved in doing so does not seem worthwhile, especially since the forward statement is not commonly used in practice.

Finally, SR allows the programmer to associate *scheduling expressions* with the alternatives of an input statement to provide additional control over the order in which invocations are serviced. For example, a scheduling expression can be used by a resource allocator implementing a shortest-job-next allocation scheme to service invocations in increasing order of usage time requested [AO93]. Scheduling expressions override the FT-SR guarantee that invocations are processed in a consistent order by members of a replicated group. They must therefore be used with caution.

CHAPTER 4

PROGRAMMING WITH FT-SR

This chapter presents three examples that illustrate the versatility of the FT-SR programming language. The first example implements the data manager and stable storage of the distributed banking system described in Chapter 3. It illustrates the use of different fault-tolerance structuring paradigms in different parts of a system. In particular, the data manager uses the restartable action paradigm, the stable storage uses the replicated state machine approach, and the system as a whole implements the object/action model. The second example is a solution to the Dying Philosophers Problem, a fault-tolerant version of the well-known Dining Philosophers problem. The solution adapts an existing solution to the Dying Philosophers Problem and shows how FT-SR facilitates the modification of existing programs to add fault-tolerance. The final example is a distributed word-game, which also uses the replicated state machine paradigm for fault-tolerance, albeit in a somewhat different way.

4.1 A Distributed Banking System

This section describes the implementation of the data manager and stable storage of the distributed banking system of Chapter 3. As outlined there, the data manager implements a collection of operations that provide transactional access to data items located on a stable storage. The organization of the manager itself is based on the restartable action paradigm, with key items in the internal state being saved on stable storage for later recovery in the event of failure. The state machine approach is used to build stable storage.

The data manager controls concurrency and provides atomic access to data items on stable storage. For simplicity, we assume that all data items are of the same type and are referred to by a logical address. Stable storage is read by invoking its `read` operation, which takes as arguments the address of the block being read, the number of bytes to be read, and a buffer in which the values are to be returned. Data is written to stable storage by invoking an analogous `write` operation, which takes as arguments the address of the block being written, the number of bytes in the block, and a buffer containing the actual values.

Figure 4.1 shows the specification and an outline of the body of such a data manager. As can be seen in its specification, the data manager imports stable storage and lock manager resources, and exports six operations: `startTransaction`, `read`, `write`, `prepareToCommit`, `commit`, and `abort`. The operation `startTransaction` is invoked by the transaction manager to access data held by the data manager; its arguments

```
resource dataManager
  imports globalDefs, lockManager, stableStore
  op startTransaction(tid: int; dataAddrs: addrList;
      numDataItems: int)
  op read(tid: int; dataAddrs: addrList; data: dataList;
      numDataItems: int)
  op write(tid: int; dataAddrs: addressList; data: dataList;
      numDataItems: int)
  op prepareToCommit(tid: int), commit(tid: int), abort(tid: int)
body dataManager(dmId: int; lmcap: cap lockManager;
    ss: cap stableStore)
  type transInfoRec = rec(tid: int;
        transStatus: int;
        dataAddrs: addressList;
        currentPointers: intArray;
        memCopy: ptr dataArray;
        numItems: int)
  var statusTable[1:MAX_TRANS]: transInfoRec;
  var statusTableMutex: semaphore

  initial
    # initialize statusTable
      ...
    monitor(ss)send failHandler()
    monitor(lmcap)send failHandler()
  end initial

  ...code for startTransaction, prepareToCommit,
      commit, abort, read/write...

  proc failHandler()
    destroy myresource()
  end failHandler

  recovery
    ss.read(statusTable, sizeof(statusTable), statusTable);
    transManager.dmUp(dmId);
  end recovery
end dataManager
```

Figure 4.1: Outline of `dataManager` resource

are a transaction identifier `tid` and a list of addresses of the data items used during the transaction. `read` and `write` are used to access and modify objects. The two operations `prepareToCommit` and `commit` are invoked in succession upon completion to, first, commit any modifications made to the data items by the transaction, and, second, terminate the transaction. `abort` is used to abandon any modifications and terminate the transaction; it can be invoked at any time up to the time `commit` is first invoked. All these operations exported by the data manager are implemented as **procs**; thus, invocations result in the creation of a thread that executes concurrently with other threads. Finally, the data manager contains initial and recovery code, as well as a failure handler **proc** that deals with the failure of the `lockManager` and `stableStore` resources.

To implement the atomic update of the data items, the data manager uses the standard technique of maintaining two versions of each data item on stable storage together with an indicator of which is current [BHG87]. To simplify our implementation, we maintain this indicator and the two versions in contiguous stable storage locations, with the indicator being an offset and the address of the indicator used as the logical address of the item. Thus, the actual address of the current copy of the item is calculated by taking the address of the item and adding to it the indicator offset.

The data manager keeps track of all in-progress transactions in a *status table*. This table contains for each active transaction the transaction identifier (`tid`), the status (`transStatus`), the stable storage addresses of the data items being accessed by the transaction (`dataAddrs`), the value of the indicator offset of each item (`current-Pointers`), a pointer to an array in volatile memory containing a copy of the data items (`memCopy`), and the number of data items being used in the transaction (`numItems`). This table can be accessed concurrently by threads executing the **procs** in the body of the data manager, so the semaphore `statusTableMutex` is used to achieve mutual exclusion. New entries in this table also get saved on stable storage for recovery purposes. Reads and writes during execution of the transaction are actually performed by the data manager on versions of the items that it has cached in its local (volatile) storage.

The data manager depends on the stable storage and lock manager resources to implement its operations correctly. As a result, it needs to be informed when they fail catastrophically. The data manager does this by establishing an asynchronous failure handler `failHandler` for each of these events in the initial code using the **monitor** statement. When invoked, `failHandler` terminates the data manager resource, thereby causing the failure to be propagated to the transaction manager.

The failure of the data manager itself is handled by recovery code that retrieves the current contents of the status table from stable storage upon recovery. It is the responsibility of the transaction manager to deal with transactions that were in progress at the time of the failure; those for which `commit` had not yet been invoked are aborted, while `commit` is reissued for the others. To handle this, the recovery code sends a message to the transaction manager notifying it of the recovery.

The **procs** implementing the other data manager operations do not use any of the FT-SR primitives specifically designed for fault-tolerant programming and are therefore

```
persistent resource stableStore
  import globalDefs
  op read(address: int; numBytes: int; buffer: charArray)
  op write(address: int; numBytes: int; buffer: charArray)
  op sendState(sscap: cap stableStore)
  op recvState(objectStore: objList)
body stableStore
  var store[MEMSIZE]: char

  process ss
    do true ->
      in read(address, numBytes, buffer) ->
        buffer[1:numBytes] := store[address:address+numBytes-1]
      [] write(address, numBytes, buffer) ->
        store[address, address+numBytes-1] := buffer[1:numBytes]
      [] sendState(rescap) -> send rescap.recvState(store)
      ni
    od
  end ss

  recovery
    send mygroup().sendState(myresource())
    receive recvState(store); send ss
  end recovery
end stableStore
```

Figure 4.2: `StableStore` resource

not shown here. They can however be found in Appendix B, which contains all the code for the distributed banking system.

We now turn to implementing stable storage. One way of realizing this abstraction is by using the state machine approach, that is, by creating a storage resource and replicating it to increase failure resilience. Figure 4.2 shows such a resource; for simplicity, we assume that storage is managed as an array of bytes.

Replica failures are dealt with by restarting the resource on another machine; this is done automatically since `stableStore` is declared to be a persistent resource. The recovery code that gets executed in this scenario starts by requesting the current state of the store from the other group members. All replicas respond to this request by sending a copy of their storage state; the first response is received, while the other responses remain queued at the `recvState` operation until the replica is either destroyed or fails. The newly restarted replica begins processing queued messages when it is finished with recovery. Since messages are queued from the point that its `sendState` message was sent to the group, the replica can apply these subsequent messages to the state it receives to reestablish consistency with the state of the other replicas.

```
resource main
  imports transManager, dataManager, stableStore, lockManager
body main
  var virtMachines[3]:  cap vm # array of virtual machine capabilities
  var dataSS[2], tmSS: cap stableStore # capabilities to stable stores
  lm:  cap lockManager; # capability to lock manager
  dm[2]:  cap dataManager # capabilities data managers

  virtMachines[1] := create vm() on HERSHEY
  virtMachines[2] := create vm() on LUCIDA
  virtMachines[3] := create vm() on BODONI # backup machine

  # create stable storage for use by the data managers and
  # transaction manager
  dataSS[1] := create (i := 1 to 2) stableStore() on virtMachines
      backups on virtMachines[3]
  dataSS[2] := create(i := 1 to 2) stableStore() on virtMachines
      backups on virtMachines[3]
  tmSS := create (i := 1 to 2) stableStore() on virtMachines
      backups on virtMachines[3]

  # create lock manager, data managers, and transaction manager
  lm := create lockManager() on virtMachines[2]
  fa i := 1 to 2 ->
    dm[i] = create dataManager(i, lm, dataSS[i]) on virtMachines[i]
  af
  tm = create transManager(dm[1], dm[2], tmSS) on virtMachines[1]
end main
```

Figure 4.3: System Startup in Resource `main`

Stable storage could also be implemented as a primary-backup group by adding a {**primary**} restriction to the `read` and `write` operations. The process `ss` would then send the updated state to the rest of the group at the end of each operation by invoking a `recvState` operation on the group. This operation would be implemented by extending the input statement in `ss` to include this operation as an additional alternative.

The main resource that starts up the entire system is shown in Figure 4.3. Resource `main` creates a virtual machine on each of the three physical machines available in the system. Three stable storage objects are then created, where each such object has two replicas and uses the virtual machine on "bodoni" as a backup machine. The two data managers are then created followed by the transaction manager. Notice how the system is created "bottom up," with the objects at the bottom of the dependency graph being created before the objects on which they depend. This way, each object can be given capabilities to the objects on which it depends upon creation.

To summarize, the distributed banking system provides its users with transactional access to their accounts. These transactions are implemented by transaction and data managers, which are implemented using the restartable action paradigm. Finally, these managers maintain permanent copies of data and logs on a stable storage constructed using the replicated state machine approach. This example therefore demonstrates the ease with which FT-SR can be used to program systems that use different structuring paradigms at different levels.

4.2 The Dying Philosophers Problem

The dying philosophers problem is a fault-tolerant version of the dining philosophers problem [Dij68]. In this scenario, philosophers sit around a table and alternate between thinking and eating spaghetti. Each philosopher needs two forks to eat, but there are only as many forks available as there are philosophers, with one fork between every two philosophers. When a philosopher decides to eat, he acquires the forks to his immediate left and right. If one or both are unavailable, the philosopher waits until the neighbor is done using the fork. When a philosopher is done eating, both forks are released and the philosopher goes back to thinking.

The dying philosophers problem is an extension of the dining philosophers problem where philosophers may die at any time. When a philosopher dies, any forks in his possession are released and made available to other philosophers. Like the dining philosophers problem, which models the resource allocation problem in distributed systems, the dying philosophers problem models resource allocation in fault-tolerant distributed systems.

Solutions to the dying philosophers problem must ensure that resources are shared fairly and that contenders for resources do not deadlock or starve. The solution presented here is based on a solution to the dining philosophers problem proposed in [CM84a]. The solution associates with each philosopher a personal servant who manages the forks. The servant negotiates for forks with the neighboring servants and when both forks are acquired, passes them on to the philosopher. The servants are actually not necessary for the solution but rather are useful as a way of separating the act of acquiring forks from their actual use. For simplicity, we assume that the deaths of philosophers and servants are linked so that the death of one implies the death of the other. Dead philosophers and servants do not recover.

The FT-SR solution to the dying philosophers problem is derived from an SR solution to the dining philosophers problem described in [AO93]. Since the SR solution is not designed to be fault-tolerant, neither it nor the FT-SR adaptation are structured according to any of the standard fault-tolerance paradigms. What this example illustrates then is the ease with which some existing programs can be modified to make them fault-tolerant.

The FT-SR program consists of three resources: a `philosopher` resource, a `servant` resource, and a `main` resource. Philosopher resources and their associated servant resources are uniquely identified by an integer between one and the number of philosophers in the problem. This number also represents the position of the corresponding

```
resource main()
  import philosopher, servant
  var n:  int
  writes("how many philosophers?  "); read(n)
  var s[1:n]:  cap servant
  var forks[1:n]:  int

  # create servants and philosophers
  fa i := 1 to n →
    s[i] := create servant(i, n)
  af
  create (i := 1 to n) philosopher(s[i], i) on machine[i]

  # give each servant capabilities to all other servants
  fa i := 1 to n →
    send s[i].links(s)
  af

  # initialize each servant's forks
  forks[1] := 2; forks[n] := 0
  forks[2 :  n-1] := ([n-2] 1)
  fa i := 2 to n-1 -> send s[i].forks(forks) af
end main
```

Figure 4.4: Resource `main` of the Dying Philosophers Problem

philosopher around the table. As shown in Figure 4.4, the main resource creates the servant resources first and passes them their identity and the total number of philosophers. The philosopher resources are then created as a replicated group. Each resource is passed its unique identifier and a capability to its servant. Operation `links` of the servants is then invoked with an array containing capabilities to all servants. Finally, array `forks` is initialized with the number of forks assigned to each servant; that is `forks[i]` contains the number of forks assigned to servant i. Notice that servant 1 gets two forks, servant n gets none, and all others get one fork. This asymmetric distribution of forks ensures that servants do not deadlock trying to acquire forks. Array `forks` is passed to the servant using operation `forks`.

Figure 4.5 shows the `philosopher` resource. The actions of the philosopher are implemented by the process `phil`. As in the general description above, philosophers alternate between eating and thinking. When a philosopher is ready to eat, it acquires forks from its servant by invoking the servant's `getforks` operation. When the philosopher is done eating, it returns the forks by invoking the servant's `relforks` operation.

Of particular interest in the philosopher resource is the `monitor` statement, which is used by the philosophers to monitor each other for failure. A naive implementation

```
resource philosopher
  import servant
body philosopher(s: private cap servant; id: int)

  monitor myresource() send mygroup().philDied(id)

  process phil
    do true →
      s.getforks()
      write("Philosopher", id, "is eating")
      s.relforks()
      write("Philosopher", id, "is thinking")
    od
  end

  proc philDied(id)
    send s.philDied(id)
  end
end
```

Figure 4.5: The `philosopher` resource.

would have each of the n philosophers execute $n-1$ monitor statements to monitor each of the other philosophers. This effect can be achieved more easily, however, by using a single monitor statement as shown. This statement, when executed by a group member m, tells the system to monitor m and invoke the `philDied` operation of the group when m fails. The failure of any philosopher therefore results in the `philDied` operation of every other philosopher being invoked. Moreover, since invocations to the group are consistently ordered, all philosophers learn of all failures in a consistent order. This consistent ordering would be impossible to achieve using multiple monitor statements. When a philosopher is notified of such a failure, it in turn informs its servant.

Figure 4.6 shows the specification of the servant resource. As described earlier, operations `getforks` and `relforks` are invoked by the associated philosopher to acquire and release forks. Operations `needL` and `passL` are invoked by the servant on the left when it needs a fork or is passing one, respectively. `needR` and `passR` are analogous operations invoked by the servant on the right. As seen earlier, the main resource uses operations `links` and `forks` to give the servants capabilities to all other servants and the initial distribution of forks, respectively.

Figure 4.7 shows process `server` of the servant resource, which implements most of the functionality of the servant. It keeps track of the forks held by the servant and negotiates for forks with server processes of other servants. The forks held by a servant are tracked using variables `numOwned`, `numHave`, `ownL`, `ownR`, `haveL`, and `haveR`. `numOwned` and `numHave` are integers that record, respectively, the number of forks owned by the

```
resource servant
  # operations invoked by associated philosopher
  op getforks() {call}, relforks() {call}
  op philDied(id:  int) {send}

  # operations invoked by neighboring servants
  op needL() {send}, needR() {send},
    passL() {send}, passR() {send}

  # initialization operations invoked by main
  op links(s[1:*]:  cap servant),
    forks(f[1:*]:  int)
body servant(myid:  int; n:  int) separate
```

Figure 4.6: Specification of resource `servant`

servant and the number of forks currently held by the servant; the number of forks owned are based on the original allocation in array `forks` but changes as philosophers fail and their forks redistributed. `ownL` and `ownR` are booleans that are true if the servant owns a left fork or a right fork, while `haveL` and `haveR` are booleans that are true if the servant actually has possession of a left fork or a right fork. For example, a servant may own a left fork, but not have it if the fork has been borrowed by its left neighbor. In this case, the left neighbor would have a right fork but not own one.

Initially, servant 1 is assigned two forks and therefore has its variables `numOwned` and `numHave` set to two, and both `ownL` and `ownR` set to true. Servant n is not assigned any forks and therefore has `numOwned` and `numHave` set to zero, and both `ownL` and `ownR` set to false. All other servants have `numOwned` and `numHave` set to one, `haveL` set to false, and `haveR` set to true.

The main loop of process `server` contains an input statement that handles invocations from its philosopher and neighboring servants. Operation `hungry` acquires forks for its philosopher; it determines the forks the servant is missing, and requests them by invoking the `needL` or `needR` operations of the appropriate neighbors. When the requisite forks have been obtained, the philosopher is allowed to eat. The servant then waits until the philosopher is done eating. Boolean variables `dirtyL` and `dirtyR` are used to ensure fairness. A fork is marked as being dirty as soon as its philosopher uses it. Servants relinquish dirty forks to neighboring servants that request them, even if their philosopher has also requested forks.

When a philosopher and its servant fail, the `philDied` operation of the other servants is invoked by their respective philosophers. This operation calls a local proc `redistribForks` with the identity of the failed philosopher and a boolean flag indicating whether its philosopher is hungry or not. The following actions are performed in proc

```
process server
  receive links(servants); receive forks(forkDist)
  l := servants[((myid-2) mod n) + 1] # determine left neighbor
  r := servants[(myid mod n) + 1] # determine right neighbor
  numHave := numOwned := forkDist[myid]
  if numOwned = 2 ->
    ownR := true; haveR := true; ownL := true; haveL := true
  [] numOwned = 1 ->
    ownR := true; haveR := true; ownL := false; haveL := false
  [] else ->
    ownR := ownL := false; haveR := haveL := false
  fi
  dirtyL := dirtyR := false
  do true ->
    in hungry() ->
      # ask for forks I don't have
      if ~haveR -> send r.needL() fi; if ~haveL -> send l.needR() fi
      do ~(haveL and haveR) ->
        in passR() ->
          haveR := true; dirtyR := false; numHave++
        [] passL() ->
          haveL := true; dirtyL := false; numHave++
        [] needR() st dirtyR ->
          haveR := false; dirtyR := false
          send r.passL(); send r.needL(); numHave--
        [] needL() st dirtyL ->
          haveL := false; dirtyL := false
          send l.passR(); send l.needR(); numHave--
        [] philDied(id) ->
          redistribForks(id, true)
        ni
      od
      # let my philosopher eat; wait for it to finish
      send eat(); dirtyL := dirtyR := true; receive relforks()
    [] needR() -> # neighbor needs my right fork (its left)
      if numHave <= 2 -> haveR := false; dirtyR := false fi
      send r.passL(); numHave--
    [] needL() -> # neighbor needs my left fork (its right)
      if numHave <= 2 -> haveL := false; dirtyL := false fi
      send l.passR(); numHave--
    [] philDied(id) ->
      redistribForks(id, false)
    ni
  od
end server
```

Figure 4.7: Process `server` of resource `servant`

```
proc redistribForks(id, philHungry)
  var forksXferred:  int

  # set cap of failed servant to null
  servants[id] := null
  # transfer forks to servant on the right
  fa i := 0 to n-1 st servants[((id +i) mod n) + 1] != null ->
    forksXferred := forkDist[id]
    forkDist[((id - 2 -i) mod n) + 1] +:= forksXferred
    forkDist[id] := 0
    exit
  af

  # was it a neighbor of mine that died?
  if id = ((myid - 2) mod n) + 1 ->
    # my left neighbor died:  find new left neighbor
    fa i := 1 to n-1 st servants[((myid - 2 - i) mod n) + 1] != null ->
      l := servants[((myid - 2 - i) mod n) + 1]
      exit
    af
    numOwned +:= forksXferred
    numHave +:= forksXferred
    if ownL and  haveL -> haveL := true; numHave++ fi
    if  ownL and haveL -> haveL := false; numHave-- fi
    if forksXferred = 1 ->
      ownL := true; haveL := true; dirtyL := false
    [] forksXferred >= 2 ->
      if  ownR and haveR -> send r.passL(); numHave-- fi
      ownL := true; haveL := true; dirtyL := false
      ownR := true; haveR := true; dirtyR := false
    fi
  [] id = (myid mod n) + 1 ->
    # my right neighbor died:  find new right neighbor
    fa i := 1 to n-1 st servants[((myid + i) mod n) + 1] != null ->
      r := servants[((myid + i) mod n) + 1]
      exit
    af
    if ownR and  haveR -> haveR := true; numHave++ fi
    if  ownR and haveR -> haveR := false; numHave-- fi
    if philHungry and  haveR -> send r.needL() fi
  fi
end redistribForks
```

Figure 4.8: Proc `redistribForks` of resource `servant`

`redistribForks`. First, the servant to the right of the failed philosopher is given the forks held by the failed philosopher and its servant; it is now considered the owner of these forks. Next, the servants to the left and right of the failed philosopher determine their new right and left neighbors respectively, and all servants update their records to reflect the change in system state.

Figure 4.8 shows the actual code for proc `redistribForks`. Since all servants know how many forks are owned by each servant, the forks owned by the failed servant are simply transferred to the servant to its right. However, some additional bookkeeping is needed for the two neighbors of the failed servant. First, each needs to determine its new neighbor. This is easily accomplished because the position of the servant around the table is encoded by its unique identifier. Secondly, the `numOwned` and `numHave` counts of the right neighbor are incremented by the number of forks transferred. The `numHave` count of both neighbors is then adjusted to reflect forks they may have borrowed from or lent to the failed resource. For instance, the right neighbor increments its `numHave` count if it owns a left fork but does not have one, since this fork must have been borrowed by the failed resource. Similar adjustments are made for forks borrowed from the failed resource. The variables `ownL`, `haveL`, `ownR`, and `haveR` of the right neighbor are also updated, depending on the number of newly acquired forks. If the philosopher on the left is hungry and a right fork is needed, the servant resends it request for a right fork to its new right neighbor.

If a servant ends up with more than two forks, the extra forks are lent to any neighbors that need them. Since this servant no longer needs to borrow forks, the extra forks remain with the neighbors. This ensures that forks claimed from dead servants are shared by all servants.

A complete solution to the dying philosophers problem can be found in Appendix C.

4.3 A Distributed Word Game

The word game described here is a multi-player game that consists of a square grid of letters and a list of words. Hidden in the grid are words from the list, oriented vertically or horizontally. The object of the game is to find all the words from the list in the grid.

To solve the problem, each player picks a different word from the list and searches the grid for that word. When it finds the word, it highlights the letters of the grid that form the word, and goes on to search for the next word in the list that is not already claimed by another player.

The fault-tolerant version of the distributed word-game assumes that players may die. Since players search for words independently, other players need take no special action when a player dies other than ensuring that any word held by the dead player is returned to the list of available words. So long as at least one player is available, the game will complete.

The solution presented here attempts to maximize the chances of completing the game by creating different players on different machines. It also restarts failed players if a

```
resource main
  import player
body main()
  var playerCap:  cap player
  var vmCap[3]:  cap vm

  vmCap[1] := create vm() on HERSHEY
  vmCap[2] := create vm() on LUCIDA
  vmCap[3] := myvm()

  playerCap := create (i := 1 to 2) player(i) on vmCap[i]
      backups on vmCap[3]
  send playerCap.start()
end main
```

Figure 4.9: Main resource for the distributed word-game problem

backup machine is available. The replicated state machine approach is used to ensure that all players have a consistent view of the words that have been assigned to or found by other players, so they do not duplicate each others efforts. This example differs from the other examples of replicated state machines presented above because the replicas in this case do not all perform identical tasks—different players search the grid for different words. They do, however, maintain identical state information, which is updated only in response to commands sent to the group.

The FT-SR program that solves the problem consists of two resources: a `main` resource that creates the players and initiates the game, and a `player` resource that plays the game. Figure 4.9 shows the main resource. It creates a two player group on two different virtual machines and specifies a third virtual machine to be used as a backup for restarting failed players. A player is assigned a unique identifier upon creation. Once the players are created, the main resource starts the game by invoking the `start` operation exported by the players.

Figure 4.10 shows the specification of the player resource. The `start` operation initializes the screen used to display the progress of the game and the data structures used by the players. The game is then started by creating a process that executes proc `play`.

The proc `play` implements the state machine that is used to ensure that all players maintain a consistent view of the game. The state of the game is maintained in a table that records the status of each word in the list. This status table records two kinds of information. The first is the words that have been found, together with their position, orientation, and identity of the player that found the word. The second is the identity of the player searching for the word for each word under consideration. The other data structures maintained by a player are the grid of letters and list of words, both of which remain unchanged and therefore do not contain any state information.

```
resource player
  const NUMWORDS := 11
  const GRIDSIZE := 15
  const MAXWORDLEN := 13
  type statusType = [NUMWORDS][4] int
  type gridType = [GRIDSIZE][GRIDSIZE] char
  type wordsType = [NUMWORDS][MAXWORDLEN] char

  op start()
  op play(wordsType; gridType; statusType)
  op getWord(int); # player wants a word
  op foundWord(int; int; int; int; char); # player found a word
  op playerDied(int); # player died
  op sendState(cap player) # recovering player wants state
  op getState(statusType) # receive state
body player(myid:  int) separate;
```

Figure 4.10: Specification of resource `player` of the word game

Figure 4.11 shows the outline of proc `play`. Its main loop consists of an input statement that is executed repeatedly until the game is completed. When a player needs a new word, it invokes the `getWord` operation of the group with its identifier as an argument. Every player, on receiving this message, searches its status table to find a word that has neither been found in the grid nor is currently being searched for by another player. When such a word is found, the status table is updated to mark the word as being assigned to the player that invoked the operation. This player recognizes the invocation as being its own, and proceeds to search the grid for the word. Note that this algorithm is critically dependent on all players processing messages in a consistent order, as provided for by the FT-SR group invocation mechanism.

When a player finds a word, it invokes the group's `foundWord` operation with its identity, the position of the word in the word-list, the co-ordinates in the grid where the word was found, and the orientation of the word as arguments. All players, on receipt of this message, update their status table to reflect the new information. They also highlight the word on the grid being displayed on the screen. The player then invokes the `getWord` operation to obtain a new word. The problem is solved when all words in the list have been found.

Since players can fail during execution, all players monitor each other using a monitor statement similar to the one described in the dying philosophers problem. When a player fails, the system invokes the group `playerDied`. On receipt of this invocation, any word held by the failed player is marked in the word status table as being available to other players.

Failed players are automatically restarted if a backup virtual machine is available.

```
proc play(words, grid, wordStatus)
  monitor myresource() send playerDied(me)
  send mygroup().getWord(me)
  do true ->
    in getWord(id) ->
      ...Search array word status for available word...
      ...if word is found, mark word as being searched by player...
      ...if no word found, exit loop...
      if id = me ->
        ...search for word...
        send mygroup().foundWord(me, wnum, x, y, orient)
        send mygroup().getWord(me)
      fi

    [] foundWord(id, wnum, x, y, orient) ->
      ...mark word wnum as found by player id at position x, y...
      ...highlight word on display...

    [] playerDied(id) ->
      ...search wordStatus for word being searched by dead player...
      ...mark word as being available...

    [] sendState(newPlayer) ->
      send newPlayer.getState(wordStatus)
    ni
  od
end play
```

Figure 4.11: Outline of process `play`

```
recovery
  send mygroup().sendState(myresource())
  ...initialize screen...
  ...initialize data structures ...
  receive getState(words, wordstatus)
  ...display current status of game ...
  send play(words, grid, wordstatus)
end recovery
```

Figure 4.12: Recovery code executed by resource `player`

Figure 4.12 shows the recovery code executed by a restarted player. It sends a message to the group `sendState` operation, requesting a copy of the current state. It then re-initializes the screen and local data structures, and waits for a copy of the state. The other players, on receiving a request for state, send the recovering player the word status table by invoking its `getStatus` operation. The recovering player then creates process `play` to resume playing. Process `play` then obtains a new word and proceeds to search the grid.

The complete FT-SR program that solves the word-game problem can be found in Appendix D. This program consist of approximately 260 lines of code. A Consul based solution to the same problem consists of approximately 1850 lines of C code. This striking difference in program size demonstrates the advantage of using a system based on a high-level concurrent programming language over one based on a low-level sequential language.

4.4 Summary

The examples presented in this chapter demonstrate the flexibility and expressiveness of the FT-SR language. The distributed banking system shows that FT-SR is indeed flexible enough for programming any of the standard fault-tolerance structuring paradigms. It illustrates the use of the replicated state machine paradigm to build a stable storage device, and the restartable action paradigm to build the data manager, which in turn implements the object/action paradigm.

The dying philosophers problem shows that the FT-SR language mechanisms can also be used to adapt existing programs for fault-tolerance. In this example, the flexibility of the group operations simplify the implementation of the philosophers even though they are not structured according to the replicated state machine paradigm. The expressiveness of the monitor statement allows the use of a single monitor statement to detect the failure of any of the philosophers in the system.

The word game is an example of a system that exploits the redundancy available in a distributed system for increased performance and fault-tolerance. Such a system continues to provide correct service even in the presence of failures, albeit with degraded performance.

CHAPTER 5

IMPLEMENTATION AND PERFORMANCE

FT-SR has been implemented to run stand-alone on a network of Sun 3s. The implementation consists of two major components: a compiler and a runtime system, both of which are written in C. The compiler generates C code, which is in turn compiled by a C compiler and linked with the FT-SR runtime system. The runtime system provides primitives for creating, destroying and monitoring resources and resource groups, handling failures, restarting failed resources, invoking and servicing operations, and a variety of other miscellaneous functions.

This chapter describes the details of the FT-SR implementation. Also described briefly is the implementation of FT-SR/Unix, a Unix implementation of FT-SR.

5.1 The FT-SR Compiler

The FT-SR compiler is very similar to an existing SR compiler, which is to be expected since FT-SR is syntactically close to SR. The compiler uses lex for lexical analysis and yacc for parsing [Com86], and consists of about 16,000 lines of code. This section describes the parts of the FT-SR compiler that implement each of the FT-SR extensions to SR.

Monitor statement. The syntax of the monitor statement is described by the following yacc specification:

```
monitor_stmt:  TK_MONITOR expr TK_SEND invocation;
```

where TK_MONITOR and TK_SEND are tokens that represent the keywords **monitor** and **send**, respectively, `expr` describes an expression that evaluates to a resource or operation capability, and `invocation` describes an operation invocation.

The C code generated for the monitor statement is very similar to that generated for a send invocation. In both cases an *invocation block*, which encodes the identity of the operation being invoked and its actual parameters, is generated and initialized. In addition, the code generated for the monitor statement initializes the fields of the invocation block that specify the resource or resource group being monitored. The generated code then invokes the runtime system primitive `sr_monitor` with the invocation block as its argument.

```
create_expr:
  TK_CREATE quantifiers_opt rsrc_cap_call location_opt backups_opt;

restart_stmt:
  TK_RESTART quantifiers_opt rsrc_cap_call location_opt backups_opt;

rsrc_cap_call:
  TK_ID paren_list;

location_opt:
  /* null */
  | TK_ON expr;

backups_opt:
  /* null */
  | TK_BACKUPS location_opt;
```

Figure 5.1: Yacc specification for the create statement

Create and restart statements. The resource create and restart statements are very similar, both in terms of their syntax and the code generated by the compiler. Figure 5.1 shows the yacc specification of these statements. In this specification, `TK_CREATE`, `TK_RESTART`, `TK_ON`, and `TK_BACKUPS` are tokens that represent the keywords **create**, **restart**, **on** and **backups** respectively, and TK_ID is an identifier. The non-terminals `quantifiers_opt`, `rsrc_cap_call`, `location_opt`, and `backups_opt` describe an optional quantifier, a resource name or resource capability variable with parameters to be passed to the resource, an optional **on** clause, and an optional **backups on** clause, respectively.

We describe here the code that is generated for the most general form of the create statement, a group create statement with location and backup clauses. Figure 5.2 shows a section of the parse tree for such a statement. When node `BCREATE` is encountered during code generation, the subtree rooted at `backup_opt` is visited and code generated to collect all the backup virtual machine capabilities into an array. Node `GCREATE`, which is the right subtree of node `BCREATE`, is then visited. The code generated at this point is a loop that allocates and initializes *resource creation blocks* for the resources in the group and collects these creation blocks in an array. Resource creation blocks identify the resource to be created, the virtual machine it is to be created on and the parameters to be passed to the resource. The parameters that control execution of this loop come from the quantifiers in the `quantifiers_opt` subtree, while the body of the loop comes from the `create` subtree. Finally, a call to the runtime system primitive `sr_create_group` is generated with the resource identifier, size of the group, array of resource creation blocks, size of the resource capability, and number of backup virtual machines as arguments.

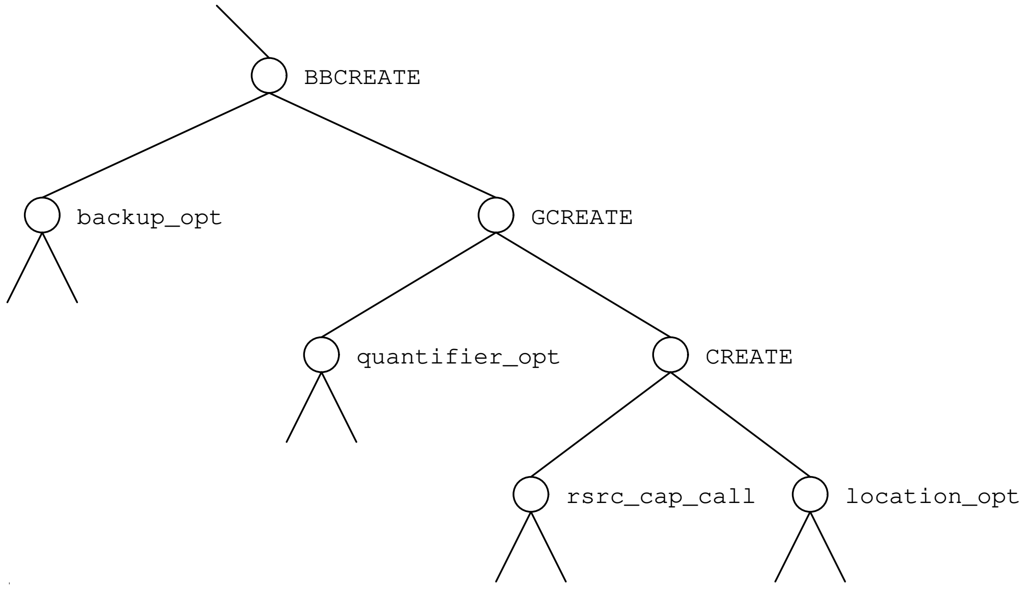

Figure 5.2: Parse tree for group create statement with backups

```
capability_def:
  private_opt TK_CAP cap_for;

private_opt:
  /* null */
  | TK_PRIVATE;
```

Figure 5.3: Yacc specification for private capability variables

Backup calls. The syntax of an FT-SR call invocation with backup operation is described by the following yacc specification:

```
invocation:
  TK_LBRACE expr TK_COMMA expr TK_RBRACE paren_list;
```

where `TK_LBRACE`, `TK_COMMA`, and `TK_RBRACE` are tokens that represent a left-brace, comma, and right-brace, respectively. Non-terminals `expr` and `paren_list` describe an expression that evaluates to an operation capability and a list of parameters.

The code generated is very similar to that generated for a call statement. The only difference is that two invocation request blocks are generated to be passed as arguments to the runtime system primitive `sr_invoke` rather than one as with a normal call.

Private capability variables. Figure 5.3 shows the yacc specification for private capability variables. In the figure, `TK_CAP` and `TK_PRIVATE` are the keywords **cap** and **private** respectively. The non-terminal `cap_for` expands to a resource or operation name or an operation type name.

When the FT-SR compiler encounters a capability variable declaration, it records in the symbol table whether the variable is private or not. When the code for an invocation is generated, the symbol table is checked to determine the kind of capability being used and the invocation block tagged appropriately.

The primary operation restrictor and the **monitorend** statement have not been implemented and are therefore not described here.

5.2 The FT-SR Runtime System

The FT-SR runtime system is implemented using version 3.1 of the *x*-kernel, a micro-kernel framework for implementing communication protocols. The runtime system, not counting the *x*-kernel, consists of 9600 lines of C code. The runtime system is linked with the code generated by the FT-SR compiler to form an executable, which is then used to boot all the processors in the system. At boot time, a processor is designated as being the one on which program execution begins.

Figure 5.4 shows the organization of the FT-SR runtime system on a single processor. As shown in the figure, each FT-SR virtual machine exists in a separate *x*-kernel user address space. In addition to the user program, a virtual machine contains those parts of the runtime system that create and destroy resources, route invocations to operations, and manage intra-virtual machine communication. This user resident part accounts for about 85 percent of the runtime system. The remaining 15 percent resides inside the kernel and is responsible for the creation and destruction of virtual machines and inter-virtual machine communication.

Figure 5.4 also shows some of the important modules in both the kernel and user resident parts of the runtime system, and the communication paths between them. The kernel resident modules shown are the Communication Manager, the Virtual Machine (VM) Manager, and the Processor Failure Detector (PFD). The communication manager consists of multiple communication protocols that provide point-to-point and broadcast communication services between processors. The VM Manager is responsible for creating and destroying virtual machines, and for providing communication services between virtual machines. The PFD is a failure detector protocol; it monitors processors and notifies the VM manager when a failure occurs. The user space resident modules shown in Figure 5.4 are the Resource Manager, the Group Manager, the Invocation Manager, and the Resource Failure Detector (RFD). The Resource Manager is responsible for the creation, destruction and restart of failed resources. The Group Manager is responsible for the creation, destruction and restart of groups, restart of failed group members, and all communication to and from groups. The RFD detects the failure of resources and group members.

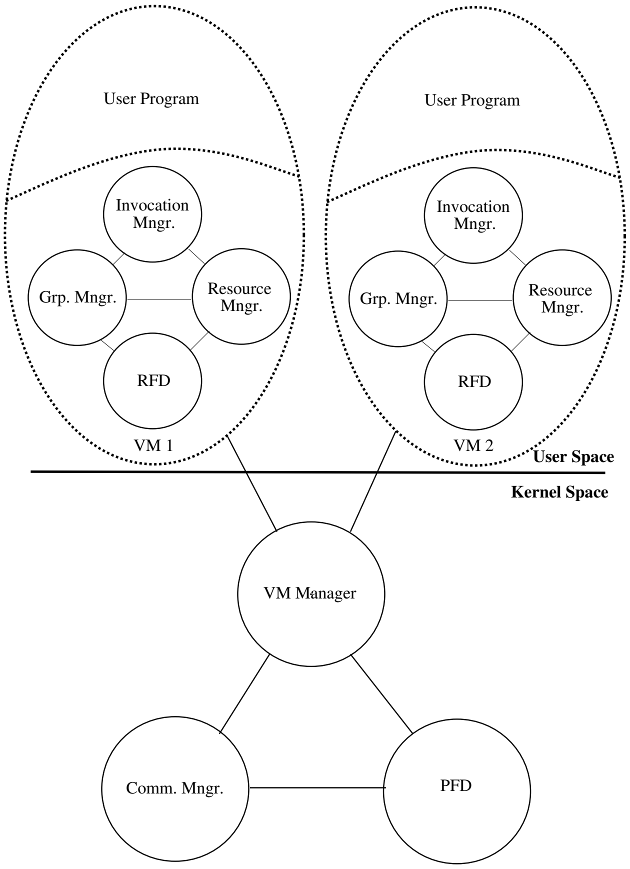

Figure 5.4: Organization of the FT-SR runtime system

The FT-SR runtime system, while similar in structure to that of standard SR, differs significantly in terms of how its different constituent modules implement their functionality. The reasons for this difference are threefold. First, unlike SR, which implements its own threads package within a Unix process, FT-SR uses threads provided by the *x*-kernel. Therefore, while SR has complete control over its threads and can schedule them when it deems safe, the FT-SR runtime has no control over when a thread is preempted and another thread scheduled. This gives rise to numerous concurrency problems that have to be dealt with within the FT-SR implementation. Second, the need to deal with failures required large modifications to the SR runtime system, since it uses centralized control in places and is written under the assumption that all parts of the distributed program will always be available. The FT-SR runtime system cannot make such an assumption, and this need to anticipate and deal with failures affects the design of almost every runtime system module. Finally, in addition to the standard language features, the FT-SR runtime system has to, of course, provide support for the fault-tolerance extensions. The implementation of some of these extensions required new modules within the runtime system and at times the re-design of other modules.

From the programmer's viewpoint, there are two major aspects in which FT-SR differs from SR: the support for using resources as FS atomic objects, and the support for various fault-tolerance techniques. Accordingly, we focus in the remainder of this section on describing those parts of the FT-SR language runtime system that provide this support. It is worth keeping in mind that, since FT-SR is designed to support construction of fault-tolerant systems, every effort has been made to keep the implementation as efficient as possible. For example, whenever possible, the implementation takes advantage of the fact that the processors can only suffer from fail-silent failures and that the maximum number of simultaneous failures *max_sf* is known *a priori*.

Failure detection and notification. Failure detection and notification is the single most important difference between FS atomic objects and SR resources. Failure detection is initiated in one of two ways: when a resource explicitly asks to be notified of a failure using the **monitor** statement, or when the communication module of the runtime system cannot complete an invocation and suspects a failure. Depending on how the failure detection was initiated, the runtime system either notifies the user program of the failure by generating an implicit invocation of the operation specified by the **monitor** statement, or, if a backup operation has been supplied with an invocation, by forwarding the invocation to the backup operation.

Failure detection in FT-SR is done at three levels: at the processor level by the PFD, at the virtual machine level by the VM Manager, and at the resource level by the RFD. The PFD at each processor monitors the other processors in the system and notifies the local VM manager of any failures. The VM manager then maps these processor failures to failures of virtual machines and notifies the RFD of these failures. The RFD in turn maps virtual machine failures to resource failures and passes this information on to any of the other runtime system modules that might have asked to be notified of the failure. To detect the termination of a resource that is explicitly destroyed, the RFD sends a message to its peer on the appropriate virtual machine asking to be notified when the resource is destroyed. Similarly, a VM Manager can ask another VM Manager to send a failure notification when a virtual machine is explicitly destroyed.

We now describe in detail the parts of the PFD, VM Manager, and RFD that are involved in failure detection. The PFDs in the system constantly monitor all processors for failures. To facilitate this, every PFD periodically broadcasts a *heartbeat* message, which serves to inform the other PFDs that the sending processor is functioning. If no heartbeats are received from a processor within a certain interval, the processor is assumed to have failed. Two bitmaps are also used: a `membership` bitmap that records the local view of which processors are functioning and which have failed, and a `heartbeat` bitmap that records the receipt of heartbeat messages. Whenever a heartbeat is received from a processor a corresponding bit in the heartbeat bitmap is set.

Changes in the status of a processor are detected by the PFD function `checkHeart-Beats`, outlined in Figure 5.5. This function is invoked at regular intervals; the current

```
void checkHeartbeats(pstate)
PSTATE *pstate;
{
  for (i = 0; i < NUM_MACHINES; i++) {
    if (!pstate->heartBeat[i])
      /* haven't heard a heartbeat from machine i in a while */
      if (pstate->membership[i]) {
        /* this machine was previously up and running */
        pstate->membership[i] = FALSE; /* reset membership bit */
        ...notify VM Manager of failure...
      }
    else
      if (!pstate->membership[i]) {
        /* got a heartbeat from machine that was declared dead */
        pstate->membership[i] = TRUE; /* set membership bit */
      }

    /* clear heartbeat bit */
    pstate->heartBeat[i] = FALSE;
  }
}
```

Figure 5.5: Outline of function `checkHeartBeats`

implementation uses an interval of about 3 heartbeat periods. For each processor in the system, the function examines the corresponding bit of the heartbeat bitmap. If no heartbeat has been recorded from that processor, and if the membership bitmap records the processor as being functional, it is assumed to have failed. If the heartbeat bitmap records a heartbeat from a processor and if the membership bitmap records the processor as having failed, the processor is assumed to have recovered from that failure. The membership bitmap is then updated to reflect the change in status of processors that have failed or restarted. Finally, all bits of the heartbeat bitmap are cleared.

The heartbeat messages contain a copy of the membership bitmap of the sending processor. A PFD, on receipt of the message, checks the state of its membership bit in the message. If the bit has the processor marked as failed, the PFD aborts the system and halts the processor. This is done in order to remedy situations where one or more processors decide that a functioning processor has failed. For example, such a situation can be caused by the failure of outgoing communication links from a processor.

The PFD also maintains a list of processors whose failure must be communicated to the VM Manager on its machine. When a processor in this list fails, the VM Manager is informed of the failure by means of an *x*-kernel *control operation*, which is a standard *x*-kernel operation used to communicate information to a protocol.

The VM Manager maintains two mappings: a mapping from processors to the virtual machines that would be lost if a processor fails, and a mapping from virtual machines to the RFDs to be notified if a virtual machine is lost. On being informed of a processor failure, the virtual machines lost due to the failure are determined from the first mapping. For each virtual machine, the RFDs interested in its failure are determined and notified of the failure by means of a control operation.

Each RFD maintains a mapping from virtual machines to the resources that would be lost due to the failure of a virtual machine. It also maintains a monitor table that lists the actions to be taken when a resource fails. Possible actions include the invocation of a failure handler operation. On being notified of the failure of a virtual machine, the resources lost due to the failure are determined. For each such resource, the corresponding monitor table entry is examined to determine the action to be taken.

The entries in the monitor and mapping tables maintained by the RFD, VM Manager, and PFD are created and initialized when an FT-SR monitor statement is executed. The runtime primitive `sr_monitor` is invoked with an invocation block for the failure handler operation as its argument. Among other things, this invocation block contains the identity of the resource to be monitored.

Monitoring is coordinated by the RFD on the processor that has the failure handler and is known as the *coordinating RFD*. As a result, the monitor request is forwarded to function `sr_reg_monitor` of the coordinating RFD, where a local function `sr_local_register` is invoked to register the request in the monitor table. A new table entry is created and the invocation block saved in the table. The local VM Manager is then asked to monitor the virtual machine that has the resource to be monitored and the resource is added to the table that maps virtual machine failures to resource failures.

The VM Manager determines the processor that has the virtual machine to be monitored and asks the PFD for a failure notification should that processor fail. The monitored virtual machine is added to the table that maps processor failures to virtual machine failures. The RFD making the monitor request is then added to the table that maps virtual machines to RFDs to be notified when the virtual machine fails.

The failure of a resource because of an explicit termination request rather than a processor crash is detected co-operatively by the coordinating RFD and the RFD on the virtual machine that has the resource, referred to as the *remote RFD*. The notification of such a failure is requested by the coordinating RFD by invoking function `sr_req_notification` of the remote RFD. In response to this request, a monitor table entry containing the address of the coordinating RFD is created at the remote RFD. Similarly, the failure of a resource because of its virtual machine being explicitly destroyed by the program is co-operatively detected by the VM Managers on the processor containing the failure handler and the processor containing the resource.

Figure 5.6 shows an outline of the RFD functions that register monitor requests.

Replication and Recovery. FT-SR provides two mechanisms for increasing failure resilience: replication and recovery. For replication, the most interesting aspect of the

```
void sr_reg_monitor(ibp)
invb ibp; /* invocation block */
{
  ...
  if ((handlervm = sr_find_vm(ibp->opc.rid)) != sr_my_vm) {
    /* the failure handler is on a different vm--let it
    take care of the monitoring.
    /
    ...invoke sr_reg_monitor on vm that has handler...
    return;
  }

  if (destvm != sr_my_vm) {
    /* resource to be monitored is on a different VM. Ask RFD of that
    VM to send a failure notification if the resource is destroyed.
    /
    ...invoke sr_req_notification on vm that has resource...
  }

  /* failure handler is local--register monitor request locally */
  sr_local_register(ibp);
}

void sr_local_register(ibp)
invb ibp; /* invocation block */
{
  ...
  ...ask VM manager for notification if vm containing resource fails...
  ...add resource to list of resources affected by failure of that vm...

  /* make an entry in local monitor table */
  ...allocate table entry, set action code to INVOKE...
  ...save copy of invocation block ...
}

void sr_req_notification(ibp)
invb ibp; /* invocation block */
{
  ...
  /* make an entry in local monitor table */
  ...allocate table entry, set action code to NOTIFY...
  ...save vm id.  to notify if resource is destroyed...
}
```

Figure 5.6: Outline of RFD functions that register monitor requests

implementation is managing group communication, since messages sent to a resource group as a result of invocations have to be multicast and delivered to all replicas in a consistent total order. The technique we use is similar to [CM84b, KTHB89, GMS89], where one of the replicas is a *primary* through which all messages are funneled. Another *max_sf* replicas are designated as *primary-group members*, with the remaining being considered *ordinary members*. Upon receiving a message, the primary adds a sequence number and multicasts it to all replicas of the group. Only the replicas that belong to the primary-group acknowledge receipt of the message. As soon as the primary gets these *max_sf* acknowledgements, it sends an acknowledgement to the original sender of the message; this action is appropriate since the receipt of *max_sf* acknowledgements guarantees that at least one replica will have the message even with *max_sf* failures. The primary is also involved when messages are sent by the group as a whole, that is, when group members use a capability that is not a private capability when making a invocation. The runtime system suppresses such an invocation from all group members except the primary. When the primary receives an acknowledgement that its invocation has been received, it multicasts that acknowledgement to the other group members.

The Group Managers at each site are responsible for determining the primary and the members of the primary-group set. Specifically, they maintain a list of all the group members and keep track of which member is the primary, which belong the primary-group set, and which are ordinary members. This list is ordered consistently at all sites based on the order in which the replicas were specified in the group create statement. The first replica of the list is then designated the primary and the next *max_sf* as members of the primary-group; the remaining replicas are ordinary members. The consistent ordering of replicas ensures that all Group Managers will independently pick the same primary and assign the same set of replicas to the primary-group set.

Figure 5.7 shows an outline of the runtime system primitive `sr_create_group`, which is invoked by the code generated for a group create statement. It is analogous to the resource creation primitive `sr_create_resource` and takes as arguments the identity of the resource to be replicated, the number of replicas, an array of creation blocks for each replica and backup, the size of the group capability variable returned by this primitive, and the number of backups. A unique identifier is generated for the group and the creation blocks initialized using the arguments to the resource creation primitive and the group identifier. The first creation block, which is the creation block for the primary, is assigned extra storage to hold a group capability. Runtime system function `sr_create_replica` is then invoked repeatedly, once for each replica and backup to be created. It takes as its argument the array of creation blocks. Every time it is invoked, a different creation block is marked as being current to identify the replica or backup being created.

Figure 5.8 shows an outline of function `sr_create_replica`. It is analogous to the resource creation function `sr_create_res` and takes an array of creation blocks as its argument. This array is searched for the creation block marked current, which corresponds to the replica to be created. The virtual machine on which the resource is to be created is determined from its creation block, and the creation request forwarded to

```
Ptr sr_create_group(rpatid, grpsize, crblist, rcapsize, numbackups)
int rpatid; /* resource pattern */
int grpsize; /* group size */
struct crb_st crblist[]; /* array of creation blocks */
int rcapsize; /* size of group capability */
int numbackups; /* number of backups */
{
  ...determine group id....

  for (i = 0; i < grpsize + numbackups; i++) {
    ...fill in the rest of the creation block from the parameters...
    if (i == 0){
      ...init creation block of primary so it returns group cap...
    }

    if (i < grpsize)
      /* creation block is for active group member */
      crbp->type = CREATE_CRB;
    else
      /* creation block is for a backup */
      crbp->type = BACKUP_CRB;
    (Ptr)crbp = (Ptr)crbp + crbp->crb_size; /* next creation block */
  }
  crbp->type = NULL_CRB; /* null terminate list */

  crbp = crblist;
  while (crbp->type != NULL_CRB) {
    /* create replica using creation block marked current */
    crbp->current = TRUE;
    sr_create_replica(crblist);
    crbp->current = FALSE;
    (Ptr)crbp = (Ptr)crbp + crbp->crb_size;
  }
  return grpCap;
```

Figure 5.7: Outline of runtime system primitive `sr_create_group`

that virtual machine. In the case of the primary replica, the remote virtual machine returns a capability to the group after the primary is created.

The creation of a local replica is very similar to the creation of a local resource. In addition to the actual creation of a resource instance, the group primary, primary-group members, and ordinary group members are determined and array `grpStatus` initialized with this information. Flag `grpMember` in the resource descriptor of the local replica is also set to the status of the replica. Finally, the RFD function `sr_callbck_vmfail` is invoked, once for each group member, asking for the Group Manager function `sr_replica_failed` to be invoked if any of them fail.

The Group Managers are responsible for dealing with the failure of group members. The action taken when a failure occurs varies, depending on whether the failed member was the primary, a primary-group member, or an ordinary member. If the primary fails, the first member of the primary-group is designated as the new primary. The designation of a new primary or the failure of a primary-group member will cause the size of the primary-group to fall below *max_sf*. When this happens, ordinary members are added to the primary-group to bring it up to *max_sf* members. No special action is needed when an ordinary member fails. If the resource from which the failed member was created is declared as being persistent and backup virtual machines were specified in the create statement, failed replicas are restarted on these backups. Restarted replicas join the group as ordinary members.

Figure 5.9 shows an outline of function `sr_replica_failed` of the Group Manager, which is invoked when a group member fails. Whether the failed member was the primary, primary-group member, ordinary member, or a backup is determined from array `grpStatus`, which maintains the status of all group members. Based on this status, control is transferred to the appropriate arm of a switch statement. If the failed replica was a primary, a new primary is selected from the primary-group set. Control then flows into the next arm of the switch, which handles the failure of primary-group members. In this code, the next available ordinary group member or backup is promoted to the primary group, and control flows to the next arm of the switch which handles the failure of an ordinary group member. If a backup is available, it is made an ordinary group member. A backup is activated by its Group Manager by invoking the runtime system function `sr_activate_bckup`. This function simply creates a process to execute the recovery code associated with the resource.

Supporting recovery involves: (1) restarting the resource instance, either as a result of an explicit request or due to an automatic restart on a backup virtual machine and (2) ensuring that the recovery code is executed. Resource restarts are handled by the runtime system primitive `sr_restart_resource`. This is very similar to the runtime system primitive `sr_create_resource`, which is used to create a resource. In fact, both these primitives initialize a resource creation block and use the resource manager function `sr_create_res` to actually create the resource. A flag in the resource creation block indicates that the recovery code of the resource is to be executed on its creation.

```
void sr_create_replica(crblist)
struct crb_st crblist[];
{
  ...
  /* find creation block for replica to be created */
  while (crbp->current != TRUE) {
    replicaNum++;
    (Ptr)crbp = (Ptr)crbp + crbSize;
  }

  /* compute the active group size and the total group size */
  grpsize = crbp->grp_size; totsize = grpsize + numBackups;

  ...if replica is to be created on a different VM,
    invoke sr_create_replica on that vm.  & wait for call to return.
    For primary replica, copy group capability that is returned...

  /* replica to be created locally */
  ...Allocate new resource instance descriptor and initialize it...

  /* determine status of group member */
  if (replicaNum == 0)
    res->grpMember = RES_GRP_PRIMARY;
  else if (replicaNum < grpsize)
    if (replicaNum <= sr_max_sf)
      res->grpMember = RES_GRP_PRIMGRP;
    else
      res->grpMember = RES_GRP_MEMBER;
  else
    res->grpMember = RES_GRP_BACKUP;

  ...allocate and intialize array grpStatus...
  ...create new process to execute initial code of resource...

  /* ask to be informed when the other replicas fail */
  crbp = (crb)crblist;
  for (i = 0; crbp->type != NULL_CRB; i++) {
    ...request sr_replica_failed be called if replica fails...
  }
}
```

Figure 5.8: Outline of function `sr_create_replica`

```
int sr_replica_failed(callBackReg)
ReplicaCallbck *callBackReg;
{
  ...
  /* take appropriate action based on what kind of replica failed */
  switch (replicaStatus) {
    case RES_GRP_PRIMARY:
      /* primary replica failed */
      ...search for first primary group member...
      ...if no primary group member found, abort...

      /* found a new primary */
      res->grpStatus[i] = RES_GRP_PRIMARY;

      if (res->replicaNum == i)
        /* I am the new primary!  Update my status flag */
        res->grpMember = RES_GRP_PRIMARY;
    case RES_GRP_PRIMGRP:
      ...search for first ordinary member to promote to primary-grp...
      ...if ordinary member found, promote it...
      res->grpStatus[i] = RES_GRP_PRIMGRP; /* promote to primary-grp */

      ...if no ordinary member found, search for backups...
      res->grpStatus[i] = RES_GRP_PRIMGRP; /* promote to primary-grp */
      if (res->replicaNum == i) {
        /* I am the new primary grp.  member!  */
        if (res->grpMember == RES_GRP_BACKUP)
          /* I used to be backup--activate myself */
          sr_activate_bckup(res, replicaNum);
        res->grpMember = RES_GRP_PRIMGRP; /* Update my status flag */
      }
    case RES_GRP_MEMBER:
      ...check if any backup can be made group members...
      ...if backup is found, make it a group member...
      res->grpStatus[i] = RES_GRP_MEMBER; /* promote to group member */

      if (res->replicaNum == i) {
        /* I am the new group member!  I was backup--activate myself */
        sr_activate_bckup(res, replicaNum);
        res->grpMember = RES_GRP_MEMBER; /* Update my status flag */
      }
      ...if no backups were created decrement active group size by 1...
  }

  ...primary replica invokes any failure handler for failed replica...
```

Figure 5.9: Outline of Group Manager function `sr_replica_failed`

5.2.1 Pros and Cons of using the *x*-kernel

FT-SR is implemented using the *x*-kernel because a bare machine implementation allows the language to be used to build realistic fault-tolerant systems. A bare machine implementation is especially important for experimental purposes since systems can be tested by actually crashing and restarting processors. Such an implementation also allows for a better evaluation of the cost incurred by the different fault-tolerance mechanisms because performance is not affected by operating system pecularities.

The use of the *x*-kernel greatly simplified the development of the many communication protocols that went into the FT-SR runtime system. The protocol development and composition tools provided by the *x*-kernel allowed for the easy decomposition of the runtime system into small modules or micro-protocols. Each of these modules could be developed independently and then composed together to form the runtime system. This modularization also allows the path of a message through the protocol stack to be determined dynamically on a per-message basis. This is used, for example, by the VM Manager shown in Figure 5.4 to choose dynamically between a point-to-point communication protocol or a broadcast protocol within the CM, depending on whether the communication is meant for all VM Managers or one particular VM Manager.

However, the use of the *x*-kernel to implement the FT-SR runtime system presented some unique challenges as well. As mentioned earlier, the runtime system has no control over the *x*-kernel process scheduler. This, combined with the fact that user level processes in the *x*-kernel are pre-emptive, gives rise to a number of concurrency problems in the runtime system. These concurrency problems are solved by using semaphores to protect shared data structures. Unfortunately, semaphores operations in *x*-kernel user space are expensive since they involve crossing into the kernel. For example, a P and V operation on a semaphore together cost 120 microseconds. (For comparison, a procedure call on a Sun 3 costs 3 microseconds.) Three such P and V operation pairs are needed for an RPC, thereby adding 360 microseconds to the overall cost.

Another problem that arises from the lack of control over process scheduling is the difficulty involved in determining the identity of the currently executing process. This identity is used to access process specific information such as the resource to which it belongs, and is maintained in an FT-SR process descriptor. This FT-SR process descriptor is distinct from the process descriptor maintained by the *x*-kernel. However, the identity of a process, and hence its FT-SR process descriptor, is maintained in the kernel process descriptor. The identity of an executing process can therefore be determined only by crossing into the kernel and examining the kernel process descriptor. Unfortunately, this is an expensive operation that costs about 300 microseconds. An RPC involves 4 such operations.

Finally, while the *x*-kernel proved to be an excellent platform for developing the kernel-resident part of the runtime system, its support for user level programming is woefully lacking. FT-SR was the first major user level software developed on the *x*-kernel and therefore many bugs in the *x*-kernel support for user tasks were encountered. This,

Operation	Time (msec)
RPC (same VM)	0.83
RPC (diff. VM, same processor)	6.36
RPC (diff. VM, diff. processors)	6.41

Table 5.1: Times (in msec) for RPC between resources

coupled with the fact that even primitive debugging tools like adb or gdb do not work with user level code, made debugging of the system extremely difficult. A Unix based version of FT-SR described in Section 5.4 solves this problem to a large extent.

5.3 Performance of FT-SR

In this section we report on the performance of some of the key features of FT-SR. Three kinds of performance are reported: the cost of an RPC from one resource to another, the cost of an RPC to and from a group and between groups, and the cost of resource and resource group creation. Wherever appropriate, these timings are compared with the corresponding timings for a standalone version of SR, Consul, and ISIS. The comparisons with SR and Consul are particularly meaningful because, like FT-SR, these systems are implemented using the *x*-kernel on bare Sun 3s.

Table 5.1 summarizes the times to do an RPC between two resources. The three rows show, respectively, the time to do an RPC between two resources on the same virtual machine, different virtual machines but the same processor, and different virtual machines on different processors. As is to be expected, inter-VM RPCs are much more expensive than an intra-VM RPC. A similar RPC in SR between two VMs on the same processor costs 5.4 msecs. The overhead incurred due to the fault-tolerance built into the FT-SR runtime system is therefore about 0.9 msecs per RPC, or about 17 percent. The corresponding number reported for Psync, the group communication protocol that forms the basis of Consul, is a round-trip time of 2.9 msecs for a 1 byte message. It must be noted, however, that even an RPC without any parameters requires messages that are much longer than a byte and involve much more processing. Moreover, FT-SR is much more dynamic than Consul in terms of the lifetimes of operations that can be invoked and the locations of resources exporting these operations, both of which greatly add to the complexity and cost of an RPC.

Table 5.2 shows the cost of RPCs to and from resource groups. In the table, Column A shows the cost of an operation when the invoker of the operation is on the same virtual machine as the group primary. Column B shows the corresponding cost when the invoker and the primary are on different processors, and therefore, different virtual machines. In the group to group RPC, the time in Column A corresponds to the case where the primaries of the two groups are on the same virtual machine and Column B to the primaries on virtual

From	To	Group Size	A (msec)	B (msec)
resource	group	1	3.24	9.23
resource	group	2	6.84	12.76
resource	group	3	6.84	12.76
resource	group ($max\_sf = 2$)	3	8.35	13.95
group	resource	3	7.19	7.90
group	group	3	14	16

Table 5.2: Times (in msec) for RPC involving groups

machines on different processors. Except for the case noted in the table, the maximum number of simultaneous failures, *max_sf*, is set to 1.

Two interesting observations can be made from Table 5.2. First, the location of the resource making the invocation with respect to the location of the group's primary resource has a tremendous impact on performance. Second, for groups of size larger than *max_sf* + 1 the cost of a group invocation appears to be independent of the size of the group. The explanation for both these observations lies in the the group communication algorithms used by the runtime system. An RPC to a group can be decomposed into two parts: an RPC from the caller to the group primary and the dissemination of the call to the rest of the group. The cost of the dissemination is independent of the location of the caller. However, as can be seen from Table 5.1, the cost of the RPC from the caller to the primary depends heavily on whether they are on the same virtual machine or not. In fact, in all four cases of resource to group communication shown in rows 1 through 4 of Table 5.2, the difference between the times in Column B and Column A is about 5.5 msecs, which is almost identical to the difference between the times for an RPC between resources on two different processors and an intra-VM RPC.

For groups larger than *max_sf* + 1, the cost of an invocation to the group is independent of the size of the group. This is because only the primary replica and the *max_sf* primary group members are involved in an invocation to a group. The other members of the group do not actively participate in the group invocation protocol and therefore do not have an impact on its performance.

A comparison of Tables 5.1 and 5.2 shows that the cost of an RPC to a single member group is about 41 percent larger than the cost of an RPC to a single resource. This represents the overhead incurred due to group related operations such as message sequencing. However, this seemingly large overhead is deceptive because the cost of an RPC to a group of 2 or more members is less than the cost of two RPCs to each of them.

Invocations on groups are also supported by Consul and ISIS. In Consul, the difference between the time an operation is invoked and its execution in a 3 replica system is 4.0

Operation	Time (msecs)
Resource Creation:	
same VM	2.6
diff. VM, same processor	8.8
diff. VM, diff. processors	8.4
Group Creation:	
1 replica	10.0
2 replicas	20.0
3 replicas	30.0

Table 5.3: Times (in msecs) for resource and group creation

msecs. Note that unlike an RPC, this does not include a response to the invoker of the operation. ISIS implements a causal broadcast which costs 9.66 msecs on Sun 4s for a group of size three. Again, like Consul, this time does not include a response to the invoker.

Table 5.3 shows the cost of the group and resource create statements. These times are interesting because in addition to the cost of a resource creation, they reflect the cost of restarting a resource since the actions taken on a restart are very similar to those taken on a create.

Two interesting observations can be made from Table 5.3. First, the time to create a resource on a virtual machine on a different processor is less than the time to create it on a different virtual machine on the same processor. This is because of the added concurrency available when processing is distributed over two processor. Secondly, the cost of creating a resource group increases linearly with the size of the group. This is because the runtime system create replicas in sequence and not in parallel.

The numbers presented in this section show that the performance of FT-SR is comparable with other systems that have been developed for building fault-tolerant systems. We believe the small performance penalty incurred by FT-SR is greatly outweighed by the benefits of using a higher-level distributed programming language to construct these type of systems.

5.4 Implementation of FT-SR/Unix

This section describes the implementation of FT-SR/Unix, a version of FT-SR that runs on Unix. Since the compiler for FT-SR/Unix is almost identical to that of FT-SR, we limit the discussion to the runtime system of FT-SR/Unix. In the rest of this section, FT-SR refers to the standalone implementation of FT-SR and FT-SR/Unix refers to the Unix implementation.

The organization of the FT-SR/Unix runtime system is a blend of the runtime systems of FT-SR and an existing Unix implementation of SR. Like the FT-SR runtime system, the functionality of the runtime system is divided into two parts, one of which resides within the virtual machines and the other in a separate execution-time manager called FT-SRX. The virtual machines and FT-SRX are implemented as separate Unix processes.

The structure of a virtual machine is very similar to that of FT-SR. However, unlike FT-SR, each virtual machine contains a thread-manager which creates, schedules, and destroys threads within the virtual machine independent of the Unix scheduler. This approach is similar to that used by SR.

The execution manager is based on the SR execution manager SRX. It is responsible for creating and destroying virtual machines, keeping track of the location of virtual machines, detecting the failure of nodes, and handling broadcast communication between virtual machines. Unlike SRX, which is centralized, FT-SRX is fully distributed and an instance of FT-SRX resides on every node in the system. This is essential because the FT-SR/Unix runtime system must itself be fault-tolerant and therefore cannot have a single point of failure.

The virtual machines in FT-SR/Unix communicate directly with each other using UDP, and with their local instance of FT-SRX using TCP. The instances of FT-SRX on the different nodes communicate with each other using UDP sockets. In each case these choice of UDP or TCP was made on the basis of the guarantees made by these protocols, and the corresponding guarantees made by the communication mechanisms in the standalone version. This allows the protocols used by FT-SR to be used in FT-SR/Unix with little or no modifications. For example, TCP guarantees ordered and reliable message delivery, which is similar to the guarantees made by the kernel-call interface between virtual machines and the VM Manager of FT-SR. UDP, on the other hand, makes no guarantees about order and reliability and is therefore similar to the message delivery service provided by the network used by FT-SR. Therefore, FT-SR protocols that expect reliable message delivery use TCP in the FT-SR/Unix implementation, while those that are designed to handled message losses use UDP.

When a virtual machine needs to communicate with another virtual machine, a message is sent to the local FT-SRX asking for the port and host address of the destination virtual machine. This information is then used to open a direct UDP connection between the two virtual machines. If a virtual machine needs to broadcast a message, the message is sent to its FT-SRX, which then broadcasts it to the instances of FT-SRX on the other nodes and the other virtual machines on that node.

FT-SRX instances reside at well-know ports. This allows virtual machines and other instances of FT-SRX to establish a connection easily with an instance of FT-SRX. This also implies that only one instance of an FT-SRX can reside on a node, irrespective of the number of FT-SR programs running on that node. Messages to and from FT-SRX are therefore tagged with identifiers that identify the program sending the message.

The failure detection protocol implemented by the FT-SRX is very similar to the PFD of FT-SR. In addition the functions of the PFD, it detects the explicit destruction of virtual machines and notifies any virtual machines that might have asked to monitor the one that was destroyed.

5.5 Summary

This chapter described details of the implementation of FT-SR on a network of standalone Sun 3s. The implementation consists of two major components: a compiler and a runtime system. The compiler is very similar to an existing SR compiler and therefore only the parts that implemented the FT-SR extensions to SR were described.

The FT-SR runtime differs significantly from the SR runtime system for three reasons. First, the *x*-kernel is used to manage threads instead of a custom threads package, second, the runtime system itself has to be fault-tolerant, and finally, extra support is needed for the FT-SR extensions to SR. The effect of these on the design of the runtime system was described.

The two most interesting aspects of the runtime system—failure detection and notification, and support for replication—were described in detail. The failure detection scheme is interesting because it involves runtime system modules at different levels of the system. The support for replication is interesting because of its novel use of *max_sf*, the maximum number of simultaneous failures allowed, to optimize a primary replica based group invocation scheme.

The performance of the implementation was measured and the observed timings presented. It was shown that the performance of FT-SR compares well with other systems developed for building fault-tolerant systems. Moreover, the slight performance penalty incurred by using FT-SR is outweighed by the benefits of using a high-level distributed programming language.

Finally, FT-SR/Unix, a Unix based implementation of FT-SR was described. The FT-SR/Unix implementation is a blend FT-SR and SR. The overall structure of the system is similar to that of FT-SR but like SR, it implements its own threads package and exists entirely within Unix user processes.

CHAPTER 6
AN EVALUATION OF FT-SR

This chapter is an evaluation of the design and implementation of the FT-SR programming model and language. The strengths of the design and implementation are highlighted along with some of the observed deficiencies of the language.

6.1 Novelty and Universality of the Programming Model

Many different programming models have been developed for fault-tolerant programming. A distinguishing feature of these models is their varying degrees of support for system structuring in general, and the standard fault-tolerance structuring paradigms in particular. In this section we argue that the FT-SR programming model is novel in its approach to system structuring and is at the same time universal in that it encompasses all the other programming models.

The FT-SR programming model is novel for two reasons. First, it is based on fail-stop atomic objects, which we believe to be the fundamental building blocks of fault-tolerant systems. The use of FS atomic objects therefore greatly simplifies the design and implementation of such systems. The second reason is that the model describes the mechanisms used to structure fault-tolerant systems rather than dictates policies. These mechanisms may then be used by programmers to implement a policy of their choosing.

The FT-SR programming model captures the fundamental principles of fault-tolerant programming in the form of fail-stop atomic objects. Recall that a fail-stop atomic object implements a collection of operations that are exported and made available for invocation by other FS atomic objects. These operations execute as atomic actions unless a catastrophic failure occurs, in which case a failure notification is generated. Ideally, a fault-tolerant system as a whole behaves as an FS atomic object: commands to the system are executed atomically or a failure notification is generated. This assures users of the system that, unless a failure notification is generated, their commands have been correctly processed and any results produced can be relied upon. Such a system is much easier to design and implement if each of its components are in turn implemented by FS atomic objects. Since the failure of any component is detectable, other components do not have to implement complicated failure detection schemes or deal with the possibility of erroneous results produced by failed components. These components may in turn be implemented by other FS atomic objects, and this process continued until the simplest components of the system are implemented by simple FS atomic objects. At each level, the guarantees made by FS atomic objects simplify their composition to form other more complex FS atomic objects.

The FT-SR programming model is also unique in that it specifies the mechanisms used to structure FS atomic objects but does not set any policies that limit the programmer to any one program structure. This allows the programmer to decide on the program structure best suited for the application and structure the program accordingly. This approach is very different from those taken by other systems that dictate what the program structure ought to be and restrict the programmer to that structure. For example, Argus supports only the object/action paradigm and therefore all Argus programs must be structured according to this paradigm. Even the HOPS programming model, which like the FT-SR model attempts to provide the programmer with greater flexibility, provides programmers a fixed set of policies to choose from. For instance, the programmer can choose between the two-phase commit protocol and timestamp ordering for concurrency control, and between logs and shadows for recovery.

The fundamental nature of the FT-SR programming model is evidenced by the observation that all three standard program structuring paradigms, the object/action model, the restartable action paradigm, and the replicated state machine approach, really implement the atomicity property of FS atomic actions. In other words, systems structured according to any of these paradigms satisfy the unitary and serializability properties. This is obviously the case with a system structured according to the object/action model. A system structured according to the restartable action paradigm satisfies the unitary property because commands to such a system always complete. A failure may delay the execution of a command, but execution will eventually be restarted and completed. The serializability property must be ensured by the application for any execution to be meaningful. A system structured according to the replicated state machine approach also satisfies the unitary property by ensuring that command execution always completes. Command execution is not affected by failures because of the active redundancy used by these systems. The serializability property is ensured by ensuring that commands are executed in the same sequence at each processor.

The universality of the FT-SR programming model is demonstrated by the fact that systems constructed using the standard program structuring paradigms can also be constructed using FS atomic objects. For example, in a system using the object/action model, objects map directly to FS atomic objects and actions to abstract threads realized by a combination of concrete threads in the FS atomic objects. In a system using the replicated state machine approach, state machines map directly to FS atomic objects and commands to invocations on these objects. Finally, restartable actions map to threads executing in an FS atomic object that is restarted on a failure.

The FT-SR programming model goes a step beyond other programming models that support one or more of the above structuring paradigms by making provisions for catastrophic failures. For example, the object/action model and the restartable action paradigm assume the existence of stable storage that never fails. Similarly, the replicated state machine approach assumes that the number of failures suffered by the system will not exceed the number of replicas. The FT-SR programming model recognizes that these assumptions can be violated, leading to a catastrophic failure and the generation of a failure notification. This allows other components of the system to react to the failure and take appropriate recovery action, or at the very least, shut down the system gracefully.

The usability of the FT-SR programming model is greatly enhanced by virtue of it being supported by a high-level concurrent programming language. This simplifies the construction of fault-tolerant distributed systems by allowing for the seamless integration of the distribution and fault-tolerance aspects of these systems. This is in contrast to systems like ISIS whose fault-tolerance model is supported by C, which is fundamentally a sequential programming language and is therefore not conducive to distributed programming.

6.2 Suitability of Language for Systems Building

The experience gained from building a variety of fault-tolerant systems leads us to believe that FT-SR does indeed provide the right level of abstraction needed in a language designed for building systems. We argue this point further by comparing the mechanisms and abstractions provided by FT-SR with the various abstractions commonly used for fault-tolerance, and showing that the FT-SR abstractions form a "lowest common denominator" that can be used to implement any of the other abstractions.

The various structuring paradigms that have been developed for fault-tolerant distributed systems provide the programmer with techniques and abstractions for the problem being solved. The relationship between these abstractions and the mechanisms provided by FT-SR can be illustrated by arranging them in a hierarchy based on the dependency relationship [MS92]. A version of this hierarchy is shown in Figure 6.1, where circles represent abstractions, rectangles represent mechanisms provided by FT-SR, and the labeled boxes at the bottom represent the portions of the FT-SR runtime system that implement these language mechanisms. At the top of the hierarchy are objects and actions. These depend directly on restartable actions, which are needed to implement protocols like the two-phase commit protocol in which failed processes must recover to complete successfully. These restartable actions, in turn, depend on two other abstractions: idempotent actions and replicated state machines. Idempotent actions are used to implement schemes like *intentions lists* to install a set of changes to data on a stable storage device [Lam81]. The stable storage itself may be implemented using the replicated state machine paradigm, as was shown in Chapter 4. Finally, the mechanisms and runtime modules in FT-SR are at the base of the hierarchy.

The horizontal line in Figure 6.1 divides functionality implemented within the FT-SR runtime system from that implemented by the FT-SR programmer. Different languages have chosen to draw this line at different levels, representing different trade-offs between the power and flexibility of the abstraction implemented by the langauge—the higher the line, the greater the power of the abstraction but lower its flexibility. Languages like Argus, draw this line at the very top of the hierarchy, providing programmers with the power of the atomic/action model but limiting them to that one model. Languages that support process checkpointing draw the line at restartable actions but programmers still

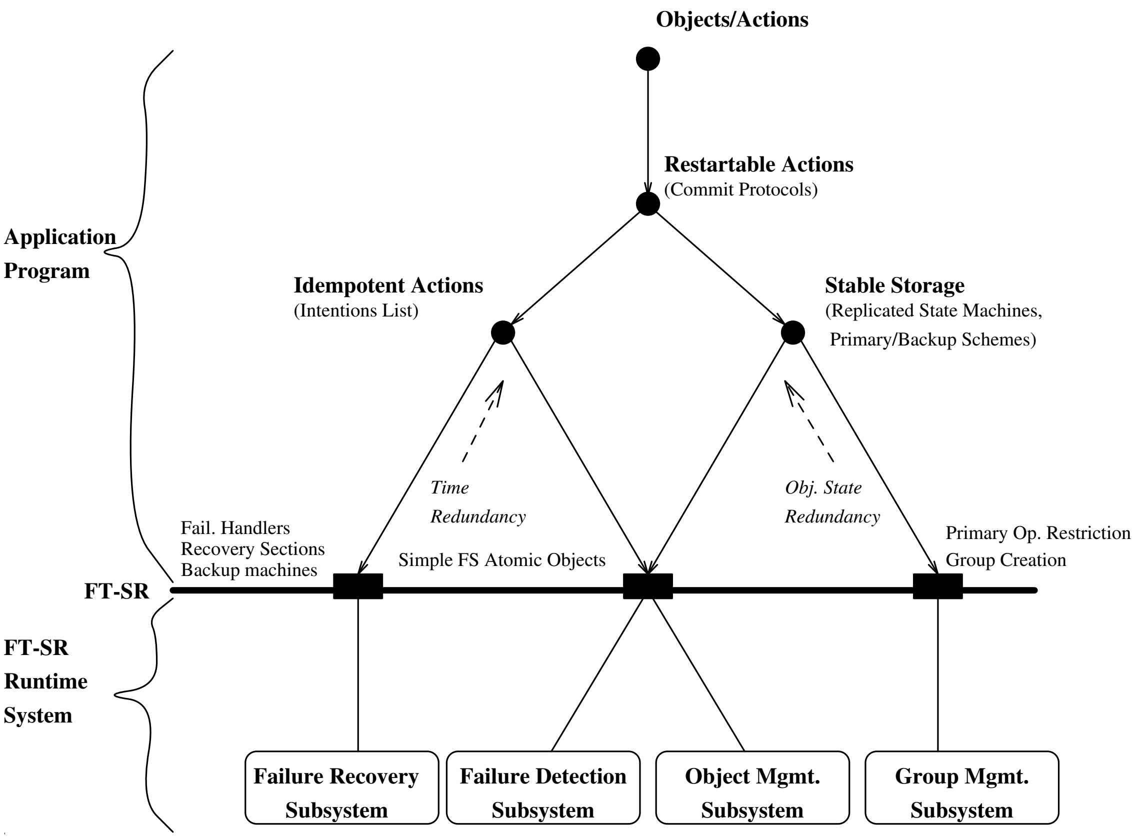

Figure 6.1: Hierarchy of Fault-Tolerance Abstractions

have the option of implementing the more powerful object/action model if required by the application. Languages like FTCC, which provide support for replication, draw this line even lower. They do not, however, usually provide support for restarting failed operation and cannot, therefore, be used to build anything but replicated objects. By drawing this line even lower, FT-SR allows programmers to implement any of the abstractions provided by other langauges.

The level of abstraction provided by a language has implications on the ease with which it can be used to build systems. A higher level of abstraction greatly simplifies the construction of systems that conform to that abstraction but can result in inefficient solutions when used to build systems that must be forced to conform to the abstraction [Bal91]. On the other hand, a low level abstraction can be used to efficiently build a wider range of systems because it can be used to implement the high-level abstraction best suited for the system. Of course, this means the programmer is saddled with the task of implementing one or more levels of abstraction.

We argue that FT-SR provides an appropriate level of abstraction for building systems. Its mechanisms are primitive enough to give the programmer the ability to build all other abstractions, yet powerful enough to be able to do so with relative ease. Such flexibility allows FT-SR to be used for a variety of different applications and system architectures. The primitive nature of the mechanisms also allows them to be efficiently implemented, an important consideration for a systems programming language.

6.3 Coherence of Language Design

The fault-tolerance mechanisms of FT-SR are designed with two important considerations in mind. The first is that the mechanisms be orthogonal so that any interplay between these mechanisms not result in unexpected behavior. The second consideration is that, whenever possible, these mechanisms use or form natural extensions to existing SR mechanisms. These considerations preserve the semantic integrity of the language and at the same time keep it relatively simple and therefore, easy to understand and use. We illustrate this point with several examples.

Recall that FT-SR provides mechanisms for failure detection, failure handling, restarts, and replication, all of which can be meaningfully combined to achieve different effects. For example, the failure detection and handling operations work uniformly well with groups and individual resources. Both the monitor statement and the invocation statement with backup work with groups just as they do with resources. In either case a failure notification is generated when no resource or resource group member is available to handle invocations.

Similarly, restart operations can be used to restart entire groups, group members, or individual resources. In every case, programmer-specified recovery code that is associated with the resource is executed, and identical rules are used to determine when the restarted resource starts accepting new invocations. Moreover, the explicit restart of a resource or resource group behaves just like an automatic restart except that it gives the programmer greater control on the placement of the restarted resource and the arguments with which it is restarted.

In addition, since group capabilities are identical to resource capabilities, all invocations that work with resources also work with resource groups. Therefore, in addition to the normal invocation operations, group operations may be specified as failure handlers in monitor statements and backup operations in call invocations. The indistinguishability of resource groups and resources extends to invocations from a group—it is impossible to tell if an invocation originated from a group or an individual resource.

Finally, a group consisting of one member is identical to an individual resource. For example, private capability variables, when used by an individual resource, are identical to regular capability variables because in this situation all capability variables are private by default. Also, the primary restrictor has no effect when used by an individual resource since it can be considered as being the primary member of a group of size one.

The orthogonality of the FT-SR fault-tolerance mechanisms keeps the language small and therefore easy to learn and use. Orthogonality allows a small set of mechanisms to be combined in different ways to achieve different effects. The lack of restrictions or special cases governing this combination of mechanisms eliminates any programming pitfalls that can snare a novice programmer.

The second aspect of language coherence is that wherever possible, the fault-tolerance mechanisms of FT-SR are integrated into existing SR mechanisms. For example, the group create statement is a natural extension of the SR resource create statement, both in terms of its syntax and semantics. Furthermore, the primary restrictor is similar to the existing send and call operation restrictors, while the specification of a resource recovery section is similar to that of its final section. Finally, a failure handler is essentially an operation that is invoked as a result of a failure and is therefore expressed using existing language mechanisms.

The integration of the FT-SR mechanisms into existing SR mechanisms serves two purposes: one, it keeps the language small, and two, the fault-tolerance aspects of the language blend in with its concurrency aspects, leading to a logically and aesthetically integrated design.

6.4 Salient features of the Implementation

The most noteworthy feature of the FT-SR implementation is that it runs on stand-alone machines. This allows the language to be used to build realistic fault-tolerant systems, which can then be tested by actually crashing processors. Similar systems such as Rajdoot and ISIS are implemented on Unix, where crashes can only be simulated by explicitly killing processes. A standalone implementation also allows for a better evaluation of the cost incurred by the different fault-tolerance mechanisms because performance is not affected by operating system peculiarities.

Another novel feature of FT-SR is the optimization of the algorithms within the runtime system based on *max_sf*, the maximum number of simultaneous failures the system can suffer. It is because of this optimization that the cost of an invocation to a group is dependent only on *max_sf* and not the size of the group. This is especially significant in light of the observation that a *max_sf* value of one is sufficient for most systems [Gra86b]. This gives FT-SR a considerable advantage over systems such as ISIS where the cost of an invocation grows linearly with the size of the group.

FT-SR also uses an interesting variation of the primary replica approach to sequence invocations to a group. Other systems like Orca that use similar schemes also implement expensive distributed election protocols to elect a new primary when the current primary fails. With these protocols, the time to elect a new primary increases with the number of replicas. This is generally undesirable because replication is often used by systems that cannot afford the time necessary to deal with failures. FT-SR solves this problem by using a scheme that does not involve a distributed election. Moreover, the new primary is selected from a group of *max_sf* replicas that maintain the same state information as the primary and can therefore take over the task of a primary on short notice.

6.5 Language Design Alternatives

The design of FT-SR has evolved over the course of this research. The current design is the result of having considered and tried many different alternatives. This section describes some of these design alternatives considered and the reasons that led to the current design.

The language feature that underwent the greatest number of changes is the ability to differentiate between group invocations, which are invocations made by a group member on behalf of the group, from individual invocations, which are invocations made by a group member as an individual. The current version of FT-SR uses, of course, private capability variables to distinguish between the two. However, an early version introduced three new statements, **gcall**, **gsend**, and **gcreate**, which were group versions of the **call**, **send**, and **create** statements. The group versions of the statements were used for group invocations or to create resources as a group while the regular versions were used to make individual invocations and to create resources as an individual. This approach had the advantage of being extremely flexible but suffered from two disadvantages: (1) the common case of group members making group invocations required the use of the special group statements and (2) the design required the addition of three new keywords to the language.

The disadvantages of the above approach led to a redesign of the group operations. The special group versions of invocation and create statements were abandoned and instead a new keyword **owns** introduced. This was akin to the keyword **imports** and was used to import resources that group members communicated with using individual invocations instead of group invocations. The name of the resource rather than the actual resource instance being communicated with was therefore used to decide on the kind of invocation generated. This design had the advantage of having one uniform set of invocation and create statements that could be used in all situations, but suffered from a serious lack of flexibility. Specifically, if a resource was declared as being owned, it was impossible for group members to make a group invocation to any instance of that resource. For example, it was impossible to build a system where group members made group invocations to a stable storage used by the entire group, as well as individual invocations to a private stable storage.

The current design—where programmers can specify private capability variables—combines the flexibility of the first design and the better integration with existing language mechanisms offered by the second. In addition, the common case where invocations made by a group member are group invocations is the default action.

The other language mechanism that underwent significant change was the ability to specify the automatic restart of a failed resource on backup virtual machines. In the current implementation this is specified by means of a **backups on** clause associated with the create statement. Earlier versions of the language used a keyword **persistent** to specify the automatic restart of a resource. That is, such a resource was declared as being a **persistent resource** in the specification of the resource. Backup virtual machines were then separately specified by listing extra virtual machines in the resource create statement.

For example, the statement

```
create persRes() on vmArray
```

was used to create a persistent resource `persRes` on the virtual machine specified by the first element of the array `vmArray`. The virtual machines specified by the remaining elements of the array were then used as backups. The group create statement was similar, with any extra elements in the array of virtual machine capabilities used as backups.

This design had the disadvantage that the automatic restart was specified in a completely different place from the specification of backup virtual machines. In other words, the create statement alone did not specify if the resource being created was to be automatically restarted; rather this information had to be pieced together from the resource specification and the create statement. The number of backup virtual machines specified was also not obvious from the create statement. We believe that the current design, where a **backup on** clause is used to specify automatic restarts and backup virtual machines, overcomes the problems with the first design and at the same time obviates the need for keyword **persistent**.

Another alternative considered for the group create statement was the use of *implicit iterators* to specify replication and to pass different arguments to different replicas. In this approach, if a group is created and an actual argument is an array whose dimensionality is one higher than the dimensionality of the corresponding formal, this array is implicitly iterated. For example, consider a resource `res` that takes a single integer as a parameter. A group consisting of n instances of `res` would be created by

```
create res(intArray) on vmArray
```

where `intArray` is an array of n integers and `vmArray` an array of n virtual machine capabilities. Since the dimensionality of the argument `intArray` is one higher than that expected for resource `res`, a group creation is implied. Array `intArray` is then implicitly iterated with replica i being created with element `intArray[i]` as its argument. As with the current design, backups were specified by means of a **backups on** clause. This design was abandoned because of the complexity of implicit iteration. It also suffered from the disadvantage that both the create statement and the resource specification had to examined to determine if a single instance or a replicated group was being created. Moreover, it could not be used to create groups of resources that take no arguments.

6.6 Observed Deficiencies of FT-SR

In this section we present some of the deficiencies of FT-SR based on our experience and on the experience of graduate students who used the language in an advanced course on fault-tolerance.

Of all the features of FT-SR, the group create statement seemed to present the most problems for users. They were caught unawares by a subtlety in its implementation semantics, which caused programs to fail when the statement was encountered. Two reasons underlie this problem: one, replicas in a group are created one at a time; and two, like resource creation in SR, further execution of the creating process blocks until the initial thread of the resource returns. Hence, if the initial thread of the first replica never returns, subsequent replicas never get created. When this happens, the first or primary replica starts executing and finds the system in an inconsistent state where all other replicas are missing even though there have been no failures. This inconsistency causes group operations to fail, often resulting in the failure of the entire program.

This problem can be solved by changing the implementation to create replicas in parallel. Clearly, such an implementation would be more in keeping with users intuition about the statement. It would also be fairly easy to realize because of the multi-threaded nature of the implementation.

A second problem that was frequently encountered was the need to duplicate resource initialization code in the recovery section. This problem arises because programs often need to re-initialize local variables on recovery from a failure to the same values they had originally. One such example can be see in the distributed word game program shown in Appendix D. An obvious solution to this problem would be to have the compiler generate code to re-initialize constants on recovery, with variables that need such re-initialization being declared as constants. Another possible solution is the use of a single initial/recovery block with the ability to determine at runtime if the block was being executed as a result of a create or restart.

We also noticed a strong temptation to write fault-tolerant programs by implementing a centralized solution and then replicating critical resources. While this made program development much easier, it often resulted in solutions that could have been implemented much more efficiently by a decentralized program. This is because what might have been relatively inexpensive local invocations in a program that was structured to be decentralized from the start were instead implemented by much more expensive group invocations.

Another problem was that the strong typing of FT-SR made the development of general-purpose resources extremely difficult and often led to inelegant solutions. For example, the stable storage resource shown in Appendix B exports two kinds of read and write operations, each of which operates on data of different types. A much more elegant solution would have read and write operations that operated on arrays of bytes, with all other data types coerced into arrays of bytes. Such a solution would also allow the stable storage resource to be re-used by other programs. The strong typing of FT-SR also makes it difficult to develop resource libraries.

The failure notification mechanisms of FT-SR, while adequate for most purposes, would be more useful if integrated with a general exception handling mechanism. This is based on the observation that operations can fail due to one of two reasons: (1) the failure of the resource implementing the operation, or (2), an application specific exceptional condition that prevents the operation from completing. In the current version of the language, the mechanisms used to signal and handle these two kinds of failures are completely disjoint. The first is signaled by the FT-SR failure notification mechanisms and handled by backup operations or failure handlers, while the second is signaled using error codes returned by operations and handled by standard conditional execution. The use of a general exception handling mechanism would allow programmers to use one uniform mechanism to handle the failure of an operation, irrespective of the cause of the failure, thereby resulting in programs that are easier to write and understand.

6.7 Summary

In this chapter the distinguishing features of the FT-SR programming model were highlighted. It was argued that this model is novel for two reasons: one, it is based on FS atomic objects, the fundamental building blocks of fault-tolerant systems, and two, it provides the programmer with a great deal of flexibility in structuring systems. It was shown that the use of FS atomic objects can simplify the design and implementation of fault-tolerant systems. It was also shown that the structuring flexibility provided by the model allows the programmer to structure systems in a manner best suited for the application.

The principles that were used to design the FT-SR language features that implement the programming model were then described. It was shown that these mechanisms are orthogonal in terms of functionality and can be meaningfully combined to achieve different effects. This orthogonality, together with the lack of restrictions on combining different mechanisms, keeps the language small and easy to learn and use.

The suitability of these mechanisms for systems building was then evaluated. It was shown that the abstractions implemented by these mechanisms form a lowest common denominator that could be used to implement any of the other abstractions developed for fault-tolerance.

The evolution of these language mechanisms was then described. The various design alternatives that were considered and the reasons the eventually led to the current design were presented.

Finally, some of the observed deficiencies of the language were described and wherever possible, solutions to these problems presented.

CHAPTER 7
CONCLUSIONS

7.1 Summary

This dissertation has described the design and implementation of FT-SR, a programming language for building fault-tolerant distributed systems. The distinguishing feature of FT-SR is the flexibility it provides the programmer in structuring fault-tolerant systems, allowing it to be used to build a variety of different types of systems.

A case for such a language was made in Chapter 2. It was shown that the main difference between the languages developed for fault-tolerant programming lay in their support for the different fault-tolerance structuring paradigms. Most of these languages support one of the structuring paradigms and even languages that attempt to support more typically have extensive support for a paradigm and very rudimentary support for the others. Support for a single paradigm is however inappropriate in a language for constructing systems where different structuring paradigms are appropriate for different levels of abstraction. FT-SR was therefore designed to be able to support equally well any of the fault-tolerance structuring paradigms. The development of such a multi-paradigm language for fault-tolerant programming can be viewed as analogous to the evolution of standard distributed programming languages, which have progressed from languages such as CSP [Hoa78] and Concurrent Pascal [BH75] that support only a single synchronization paradigm to those such as SR, Dislang [LL81], Pascal-FC [BD88], and StarMod [Coo80] that support multiple approaches.

FT-SR derives its flexibility from its programming model, which was described in Chapter 3. The model is based on FS atomic objects, which form the basic building block of all fault-tolerant systems. In this model, programs are realized as collections of FS atomic objects, organized along the line of functional dependencies. FS atomic objects are constructed by composing together other FS atomic objects. This composition is based on redundancy techniques and results in FS atomic objects with a greater resilience to failures. The actual redundancy techniques used, and hence the resulting program structure, depends on the details of the system or the application being implemented.

The FT-SR language mechanisms that implement this programming model were also described in Chapter 3. These mechanisms fall into three categories: mechanisms for replication, recovery, and failure notification. The use of these mechanisms was illustrated using several simple examples.

The ease with which FT-SR can be used to implement different kinds of fault-tolerant systems was demonstrated in Chapter 4. The distributed banking system illustrated the use of FT-SR in building systems that employ different structuring paradigms in different parts of the system. The dying philosophers problem showed how FT-SR facilitates the modification of existing programs to add fault-tolerance. The distributed word game showed the use of redundancy for increased performance and fault-tolerance.

The implementation of FT-SR was described in Chapter 5. The most interesting aspect of the implementation is its use of *max_sf*, the maximum number of simultaneous failures the system can suffer, to optimize runtime system algorithms. The benefits of this optimization are apparent in the performance of the FT-SR group operations. Unlike other systems, the cost of these operations is dependent only on the value of *max_sf* and not the size of the group. Typically, the value of *max_sf* is much smaller than the size of the group, giving FT-SR a considerable advantage over other systems.

The timings measured for some FT-SR operations were also presented in Chapter 5. A comparison of some of these timings with corresponding numbers from a standalone implementation of SR shows the execution overhead of fault-tolerance to be about 17 percent when no failures occur. However, FT-SR compares well with systems like Consul and Isis, also developed for building fault-tolerant systems. The slight performance penalty incurred by using FT-SR instead of these systems is outweighed, in our view, by the benefits of using a high-level distributed programming language.

Finally, the design and implementation of FT-SR was evaluated in Chapter 6. It was argued that the FT-SR programming model was novel in its use of FS atomic objects. These objects capture the fundamental principles of fault-tolerant programming and hence their use as building blocks of a fault-tolerant system can simplify its design and implementation. The model was also shown to be universal in that it encompasses the models supported by the other languages developed for fault-tolerant programming.

The design of the FT-SR language mechanisms was also revisited. These mechanisms were shown to form an orthogonal set that can be can be freely combined in different ways to achieve different effects. It is because of this orthogonality that FT-SR consists of a relatively small number of extensions to SR and yet is expressive enough to implement a wide variety of systems. The evolution of the design of some of these mechanisms was also described.

Finally, some of the observed deficiencies of the language, based on the experience gained from using the language, were described along with possible solutions.

7.2 Future Work

This research can be extended in many different directions. These include refining the design and implementation of FT-SR itself, adding support for different failure models, and exploring programming language support for real time fault-tolerant systems.

The design of FT-SR is by no means complete. Some of the deficiencies of the language that were discovered by experience gained from using the language were described in Chapter 6. Much more such experience is needed to be able to really validate the design and implementation of FT-SR.

In order to gain more experience with FT-SR, it must be made more accessible to users by being available on many different commonly used platforms. The current implementation is based on version 3.1 of the *x*-kernel, which only runs on Sun 3s. FT-SR/Unix was a first step in making FT-SR available on a widely used platform. It also served to demonstrate the relative ease with which FT-SR can be implemented on other platforms. Particularly useful would be an implementation based on version 3.2 of the *x*-kernel. Such an implementation would run unchanged both standalone on the Mach micro-kernel and as a Unix application. It would have the added advantage of having only one version of the software to maintain instead of the two versions that are currently maintained.

Another area of future research is the addition of support for different failure models. FT-SR currently assumes that processors suffer only from crash failures and algorithms used within the language runtime are optimized for this failure model. Support for other failure models model would allow FT-SR to be used to build a wider variety of systems. For example, support for the Byzantine failure model will allow FT-SR to be used to build ultra-dependable systems. Ideally, the semantics of the language mechanisms would be independent of the underlying failure model, allowing programmers to develop programs without consideration of the underlying failure model. The failure model would then be specified during compilation, at which time the runtime system modules optimized for that failure model would be linked in. These failure model based optimizations would complement the existing optimizations based on the maximum number of simultaneous failures the system can suffer.

Finally, programming language support for real-time fault-tolerant systems is an area that is ripe with research opportunities. Both real-time systems and fault-tolerant systems are in themselves extremely difficult to program and the problem is much worse with real-time fault-tolerant systems. An interesting idea to explore would be the extension of the FT-SR notion of a catastrophic failure to the time domain, with an eye towards designing an integrated set of mechanisms to deal with timing and processor failures. It would also be interesting to study the timing guarantees that can be made by the different fault-tolerance structuring paradigms and their suitability in real-time systems.

APPENDIX A

THE SR DISTRIBUTED PROGRAMMING LANGUAGE

An SR program consists of one or more *resources*. These resources can be thought of as patterns from which resource instances are created dynamically. Each resource is composed of two parts: an interface portion which specifies the interface of the resource and a *body*, which contains the code to implement the abstract object. The specification portion contains descriptions of objects that are to be exported from this resource—made available for use within other resources—as well as the names of resources whose objects are to be imported. Of primary importance are the declaration of *operations*—actions implemented by sequences of statements that can be invoked. These declarations specify the interface of those operations that are available for invocation from other resources. For example,

```
op example1(var x:  int; val y:  bool)
```

declares an operation, `example1`, that takes as arguments an integer `x` that is passed with copy-in/copy-out (**var**) semantics and a boolean `y` that is copy-in only (**val**). Result parameters (**res**) are also supported, as are operations with return values.

The declaration section in the resource body together with its specification define the objects that are global to the resource, i.e., accessible to any process within the resource. All of the usual types and constructors are provided. In addition, there are *capability variables*. Such a variable functions either as a pointer to all operations in a resource instance (a *resource capability*), or as a pointer to a specific operation within an instance (an *operation capability*). A variable declared as a resource capability is given a value when a resource instance is created, while an operation capability is given a value by assigning it the name of an operation or from another capability variable. Once it has a value, such variables can be used to invoke referenced operation(s), as described later.

The resource instances comprising a given program may be distributed over multiple *virtual machines*, which are abstract processors that are mapped to physical machines in the network. A resource instance is created and placed on a virtual machine using the following:

```
res_cap := create res_name(arguments) on virtual_machine_cap
```

Execution of this statement creates an instance of the resource `res_name` on the virtual machine specified by the virtual machine capability `virtual_machine_cap` and assigns a capability to the newly created resource to the capability variable `res_cap`.

An operation is an entry into a resource. An SR operation has a name, and can have parameters and return a result. There are two different ways to implement an operation: as a *proc* or as an alternative in an *input* statement. A proc is a section of code whose format resembles that of a conventional procedure:

```
proc opname(parameters) returns result
  op_body
end
```

The operation body `op_body` consists of declarations and statements. Like a procedure, the declarations define objects that are local to the operation `opname`. Unlike a procedure, though, a new process is created, at least conceptually, each time `opname` is invoked. It is possible to get standard procedure-like semantics, however, depending on how the proc is invoked (see below). The process terminates when (if) either its statement list terminates or a **return** is executed.

An operation can also be implemented as an alternative of an input statement. An input statement implementing a collection of operations $opname_1$, $opname_2$, . . ., $opname_n$ has the following form:

```
in opname1(parameters) -> op_body1
□ opname2(parameters) -> op_body2
  ...
□ opnamen(parameters) -> op_bodyn
ni
```

A process executing an input statement is delayed until there is at least one alternative $opname_i$ for which there is a pending invocation. When this occurs, one such alternative is selected non-deterministically, the oldest pending invocation for the chosen alternative is selected, and the corresponding statement list is executed. The input statement terminates when the chosen alternative terminates.

An operation is invoked explicitly using a **call** or **send** statement, or is implicitly called by its appearance in an expression. The explicit invocation statements are written as

```
call op_denotation(arguments)
send op_denotation(arguments)
```

where the operation is denoted by a capability variable or by the operation name if the statement is in the operation's scope. An operation can be restricted to being invoked only by a call or a send by appending a {**call**} or {**send**} operation restrictor to the declaration of the operation.

Execution of a **call** terminates once the operation has been executed and a result, if any, returned. Its execution is thus synchronous with respect to the operation execution. Execution of a **send** statement is, on the other hand, asynchronous: a **send** terminates when the target process has been created (if a proc), or when the arguments have been queued for the process implementing the operation (if an input statement). Thus, the effects of executing the various combinations of **send/call** and **proc/in** are described by the following table.

Invocation	*Implementation*	*Effect*
call	**proc**	procedure call
send	**proc**	process creation
call	**in**	rendezvous
send	**in**	asynchronous message passing

To illustrate how the individual pieces of the language fit together, consider the implementation of a bounded buffer shown in Figure A.1. Two operations are exported from this resource: `deposit` and `fetch`; `deposit` places a value in the next available slot if one exists, while `fetch` returns the oldest value from the buffer. A depositing process is delayed should the buffer be full. Similarly, a fetching process is delayed whenever the buffer is empty. Note also that the resource has a parameter `size`; its value determines the number of slots in the buffer. The use of resource parameters in this way allows instances to be created from the same pattern, yet still vary to a certain degree. Finally, note the single input statement to implement both the deposit and fetch operations, and the use of a send statement in the initialization code to initiate the main (parameterless) proc `buff_loop`. Creating a process in this manner is so common that the keyword **process** can be used instead of **proc** as an abbreviation for the **send** in the resource initialization code and corresponding **op** declaration.

```
resource buffer
  op fetch() returns value: int
  op deposit(val newvalue: int)
body buffer(size: int)
  var first, last: int := 0, 0
  var slot[0:size - 1]: int

  initial
    send buff_loop()
  end

  proc buff_loop()
    do true ->
      in deposit(newvalue) and first != (last + 1) % size ->
        slot[last] := newvalue
        last := (last + 1) % size
      □ fetch() returns value and first != last ->
        value := slot[first]
        first := (first + 1) % size
      ni
    od
  end
end
```

Figure A.1: Bounded buffer resource

APPENDIX B

A DISTRIBUTED BANKING SYSTEM

The following is a complete FT-SR program implementing the distributed banking system described in Chapter 3. It consists of four components: global component `globalDefs` and resources `dm`, `statbleStore`, and `main`, all of which are described in Chapter 4.

```
global globalDefs
   const MEMSIZE := 50
   const MAX_TRANS := 6
   const MAX_TRANS_DATA := 5

   # list of machines
   const BODONI := 22
   const HERSHEY := 28
   const LUCIDA := 48

   type transInfoRec = rec (
          tid: int
          transStatus: char
          dataAddrs[1:MAX_TRANS_DATA]: int
          currentPtrs[1:MAX_TRANS_DATA]: int
          memCpy[1:MAX_TRANS_DATA]: int
          numItems: int)
end globalDefs
```

```
resource dm
   import globalDefs, stableStore
   op startTrans(tid: int; dataAddrs[1:*]: int; numItems: int)
   op dmRead(tid: int; dataAddrs[1:*]: int; var data[1:*]: int; numItems:
int)
   op dmWrite(tid: int; dataAddrs[1:*]: int; data[1:*]: int; numItems: int)
   op prepareToCommit(tid: int)
   op commit(tid: int)
   op abort(tid: int)
body dm(dmId: int; ss: cap stableStore)
   var statusTable[1:MAX_TRANS+1]: transInfoRec
   var currPtr: int
   var buffer[1:2]: int
   sem statusTableMutex := 1
   op failHandler()

   # initialize stable store
   fa i := 1 to 10 by 3 ->
      buffer[1] := 1
      buffer[2] := i * 10
      ss.dataWrite(i, 2, buffer)
   af
   monitor (ss) send failHandler()

   # initialize statusTable
   fa i := 1 to MAX_TRANS ->
      statusTable[i].transStatus := 'E'
      ss.logWrite(i, statusTable[i])
   af

   proc startTrans(tid, dataAddrs, numItems)
      var t := MAX_TRANS + 1

      P(statusTableMutex)
      fa i := 1 to MAX_TRANS st statusTable[i].transStatus = 'E' ->
         t := i; exit
      af
      if t > MAX_TRANS ->
           write("out of slots in transaction table")
           V(statusTableMutex)
           return
      fi

      # ...acquire locks here...

      statusTable[t].transStatus := 'A'
      statusTable[t].tid := tid
      statusTable[t].numItems := numItems
```

```
   fa i := 1 to numItems →
      statusTable[t].dataAddrs[i] := dataAddrs[i]
      ss.dataRead(dataAddrs[i], 1, buffer)
      currPtr := buffer[1]
      statusTable[t].currentPtrs[i] := currPtr
      ss.dataRead(dataAddrs[i]+currPtr, 1, buffer)
      statusTable[t].memCpy[i] := buffer[1]
   af
   V(statusTableMutex)

   # write status table entry onto stable store
   ss.logWrite(t, statusTable[t])
end startTrans


proc dmRead(tid, dataAddrs, data, numItems)
   var t := MAX_TRANS + 1

   # find transaction in status table
   fa i := 1 to MAX_TRANS st statusTable[i].tid = tid →
      t := i; exit
   af
   if t > MAX_TRANS →
        write("dmRead: cannot find transaction in transaction table")
        return
   fi

   fa i := 1 to numItems, j := 1 to statusTable[t].numItems
           st dataAddrs[i] = statusTable[t].dataAddrs[j] →
      data[i] := statusTable[t].memCpy[j]
      next
   af
end dmRead


proc dmWrite(tid, dataAddrs, data, numItems)
   var t := MAX_TRANS + 1

   # find transaction in status table
   fa i := 1 to MAX_TRANS st statusTable[i].tid = tid →
      t := i; exit
   af
   if t ≥ MAX_TRANS →
        write("dmWrite: cannot find transaction")
        return
   fi

   fa i := 1 to numItems, j := 1 to statusTable[t].numItems
           st dataAddrs[i] = statusTable[t].dataAddrs[j] →
      statusTable[t].memCpy[j] := data[i]
```

```
            next
         af
      end dmWrite

      proc prepareToCommit(tid)
         var t := MAX_TRANS + 1
         var nonCurrCopy: int

         # find transaction in status table
         fa i := 1 to MAX_TRANS st statusTable[i].tid = tid →
            t := i; exit
         af
         if t ≥ MAX_TRANS →
              write("prepareToCommit: cannot find transaction")
              return
         fi

        # write modified objects to the "non-current" copy
        fa i := 1 to statusTable[t].numItems →
           nonCurrCopy := statusTable[t].dataAddrs[i] +
                 statusTable[t].currentPtrs[i] mod 2 + 1
           buffer[1] := statusTable[t].memCpy[i]
           ss.dataWrite(nonCurrCopy, 1, buffer)
        af
     end prepareToCommit

      proc commit(tid)
         var t := MAX_TRANS + 1

         # find transaction in status table
         fa i := 1 to MAX_TRANS st statusTable[i].tid = tid →
            t := i; exit
         af
         if t ≥ MAX_TRANS →
              write("commit: cannot find transaction in transaction table")
              write("commit: transaction must have committed")
              return
         fi

         if statusTable[t].transStatus = 'A' →  # not yet committed
              # set current pointers to point to other copy of data
              fa i := 1 to statusTable[t].numItems →
                 buffer[1] := statusTable[t].currentPtrs[i] mod 2 + 1
                 ss.dataWrite(statusTable[t].dataAddrs[i], 1, buffer)

              af
              statusTable[t].transStatus := 'D'
```

```
      fi
      if statusTable[t].transStatus = 'D' → # cleanup
           ss.logWrite(t, statusTable[t])
           # unlock objects....

           statusTable[t].transStatus := 'E'
           ss.logWrite(t, statusTable[t])
      fi
   end commit


   proc abort(tid)
      var t := MAX_TRANS + 1

      # find transaction in status table
      fa i := 1 to MAX_TRANS st statusTable[i].tid = tid →
         t := i; exit
      af
      if t ≥ MAX_TRANS →
           write("abort: cannot find transaction in transaction table")
           return
      fi

      statusTable[t].transStatus := 'E'     # abandon transaction
   end abort

   proc failHandler()
      destroy myresource()
   end failHandler

   recovery
      write("dm restarting")
      fa t := 1 to MAX_TRANS →
         ss.logRead(t, statusTable[t])
      af
   end recovery
end dm
```

```
resource stableStore
   import globalDefs
   op dataRead(address: int; numwords: int; var buffer[1:*]: int)
   op dataWrite(address: int; nuwords: int; buffer[1:*]: int)
   op logRead(logEntryNum: int; var logEntry: transInfoRec)
   op logWrite(logEntryNum: int; logEntry: transInfoRec)
   op sendState(sscap: cap stableStore)
   op recvState(dataStore[1:*]: int; logStore[1:*]: transInfoRec);
body stableStore(memsize:int)
   var dataStore[1:memsize]: int
   var logStore[1:MAX_TRANS]: transInfoRec
   op ss()

   proc ss()
      do true →
           in dataRead(addr, numwords, buff) →
                buff[1:numwords] := dataStore[addr:addr+numwords−1]

           [] dataWrite(addr, numwords, buff) →
                dataStore[addr:addr+numwords−1] := buff[1:numwords]

           [] logRead(entryNum, logEntry) →
                logEntry := logStore[entryNum]

           [] logWrite(entryNum, logEntry) →
                logStore[entryNum] := logEntry

           [] sendState(rescap) →
                send rescap.recvState(dataStore, logStore)
           ni
      od
   end ss

   recovery
      send mygroup().sendState(myresource())
      receive recvState(dataStore, logStore); send ss()
   end recovery
end stableStore
```

```
resource main
   import globalDefs, stableStore, dm
body main()
   var vmcap[1:3]: cap vm
   var sscap: cap stableStore

   write("res. main starting")
   vmcap[1] := create vm() on HERSHEY
   vmcap[2] := create vm() on LUCIDA
   vmcap[3] := create vm() on BODONI

   sscap := create(i:= 1 to 2) stableStore(MEMSIZE) on vmcap[i] backups on
vmcap[3]
   create dm(1, sscap) on vmcap[1] backups on vmcap[2:3]

end main
```

APPENDIX C

THE DYING PHILOSOPHERS PROBLEM

The following is a complete FT-SR program implementing the dying philosophers problem described in Chapter 4. It consists of three resources: a `main`, `philopher`, and `servant`. A detailed description of these resources can also be found in Chapter 4.

```
resource main()
   import philosopher, servant
   var n, t: int
   writes("how many philosophers? "); read(n)
   writes("how many sessions per philosophers? "); read(t)
   var s[1:n]: cap servant
   var forks[1:n]: int
   var machine[1:n]: cap vm

   # create servants and philosophers
   fa i := 1 to n →
      s[i] := create servant(i, n)
   af
   create (i := 1 to n) philosopher(s[i], i, t) on machine[i]

   # give each servant capabilities to all other servants
   fa i := 1 to n →
      send s[i].links(s)
   af

   #initialize each servant's forks
   forks[1] := 2; forks[n] := 0
   forks[2 : n−1] := ([n−2] 1)

   fa i := 2 to n−1 →
      send s[i].forks(forks)
   af
end main
```

```
resource philosopher
   import servant
body philosopher(s: private cap servant; id, t: int)
   op philDied(id: int)

   monitor myresource() send philDied(id)
   process phil
      fa i := 1 to t →
         s.getforks()
         write("Philosopher", id, "is eating")     # eat
         s.relforks()
         write("Philosopher", id, "is thinking") # think
      af
   end

   proc philDied(id)
      send s.philDied(id)
   end

end
```

```
resource servant
   # operations invoked by associated philosopher
   op getforks() {call}, relforks() {call};
   op philDied(id: int) {send};

   # operations invoked by neighboring servants
   op needL() {send}, needR() {send},
      passL() {send}, passR() {send};

   # initialization operations invoked by main
   op links(s[1:*]: cap servant),
      forks(f[1:*]: int)
body servant(myid: int; n: int)
   var ownL, ownR, haveL, haveR, dirtyL, dirtyR: bool
   var l, r: cap servant
   var servants[1:n]: cap servant
   var forkDist[1:n]: int
   var numOwned, numHave: int
   op hungry() {send}, eat() {send};
   op redistribForks(id: int; hungry: bool) {call};

   proc getforks()
      send hungry()  # tell server philosopher is hungry
      receive eat()  # wait for permission to eat
   end getforks

   process server
      receive links(servants)
      receive forks(forkDist)
      l := servants[((myid−2) mod n) + 1]
      r := servants[(myid mod n) + 1]
      numHave := numOwned := forkDist[myid]
      if numOwned = 2 →
           ownR := true; haveR := true
           ownL := true; haveL := true
      [] numOwned = 1 →
           ownR := true;  haveR := true
           ownL := false; haveL := false
      [] else →
           ownR := ownL := false; haveR := haveL := false
      fi
      dirtyL := dirtyR := false
      do true →
           in hungry() →
                # ask for forks I don't have:
                # ask right neighbour for its left fork
                # ask left neighbour for its right fork
                if ~haveR → send r.needL() fi
```

```
            if ~haveL → send l.needR() fi
            do ~ (haveL and haveR) →
                 in passR() →
                      haveR := true; dirtyR := false; numHave++
                 [] passL() →
                      haveL := true; dirtyL := false; numHave++
                 [] needR() st dirtyR →
                      haveR := false; dirtyR := false
                      send r.passL(); send r.needL(); numHave−−
                 [] needL() st dirtyL →
                      haveL := false; dirtyL := false
                      send l.passR(); send l.needR(); numHave−−
                 [] philDied(id) →
                      redistribForks(id, true)
                 ni
            od
            # let my philosopher eat; wait for it to finish
            send eat(); dirtyL := true; dirtyR := true
            receive relforks()

       [] needR() →
            # neighbour needs my right fork (its left)
            if numHave ≤ 2 → haveR := false; dirtyR := false fi
            send r.passL(); numHave−−

       [] needL() →
            # neighbour needs my left fork (its right)
            if numHave ≤ 2 → haveL := false; dirtyL := false fi
            send l.passR(); numHave−−

       [] philDied(id) →
            redistribForks(id, false)
       ni
   od
end server

proc redistribForks(id, philHungry)
   var forksXferred: int

   # set cap of failed servant to null
   servants[id] := null
   # transfer forks to servant on the right
   fa i := 0 to n−1 st servants[((id +i) mod n) + 1] ≠ null →
      forksXferred := forkDist[id]
      forkDist[((id − 2 −i) mod n) + 1] +:= forksXferred
      forkDist[id] := 0
      exit
   af

   # was it a neighbor of mine that died?
```

```
      if id = ((myid − 2) mod n) + 1 →
           # my left neighbor died: find new left neighbor
           fa i := 1 to n−1
                    st servants[((myid − 2 − i) mod n) + 1] ≠ null →
              l := servants[((myid − 2 − i) mod n) + 1]
              exit
           af
           numOwned +:= forksXferred
           numHave += forksXferred
           if ownL and ~haveL → haveL := true; numHave++ fi
           if ~ownL and haveL → haveL := false; numHave−− fi
           if forksXferred = 1 →
                ownL := true; haveL := true; dirtyL := false
           [] forksXferred ≥ 2 →
                if ~ownR and haveR → send r.passL(); numHave−− fi
                ownL := true; haveL := true; dirtyL := false
                ownR := true; haveR := true; dirtyR := false
           fi

      [] id = (myid mod n) + 1 →
           # my right neighbor died: find new right neighbor
           fa i := 1 to n−1 st servants[((myid + i) mod n) + 1] ≠ null →
              r := servants[((myid + i) mod n) + 1]
              exit
           af
           if ownR and ~haveR → haveR := true; numHave++ fi
           if ~ownR and haveR → haveR := false; numHave−− fi
           if philHungry and ~haveR → send r.needL() fi
      fi
   end redistribForks
end servant
```

APPENDIX D
THE WORD GAME PROBLEM

The following is a complete FT-SR program implementing the word game problem described in Chapter 4. It consists of two resources: a `main` resource and a `player` resource, both of which are described in detail in Chapter 4.

```
resource player
   const NUMWORDS := 11
   const GRIDSIZE := 15
   const MAXWORDLEN := 13
   type statusType = [NUMWORDS][4] int
   type gridType = [GRIDSIZE][GRIDSIZE] char
   type wordsType = [NUMWORDS][MAXWORDLEN] char
   op play(wordsType; gridType; statusType)
   op start()                # start game
   op getWord(int);          # player id wants word
   op foundWord(id: int; wordNum: int; xcoord: int; ycoord: int; orient:
char);
   op playerDied(id: int);  # player id died
   op sendState(cap player) # recovering player wants state
   op getState(statusType)  # get state

body player(myid: int)

    var grid: gridType := (
    ('K','E','R','N','E','L','R','R','E','C','O','V','E','R','Y'),
    ('B','C','S','Z','F','R','E','L','A','T','I','V','I','T','Y'),
    ('Y','I','P','E','F','G','H','I','J','K','L','M','N','O','P'),
    ('P','E','R','F','O','R','M','A','N','C','E','M','N','O','C'),
    ('B','C','O','E','F','G','H','I','J','K','L','M','N','O','O'),
    ('B','C','T','E','F','N','T','G','V','Q','O','B','D','A','M'),
    ('B','C','O','E','F','M','E','M','B','E','R','S','H','I','P'),
    ('B','C','C','E','F','G','H','O','J','K','L','M','N','O','U'),
    ('D','X','O','T','P','S','Y','N','C','M','A','V','K','I','T'),
    ('B','C','L','E','R','G','H','V','J','F','R','M','C','A','E'),
    ('B','C','E','E','I','G','H','I','J','U','I','P','M','X','R'),
    ('B','C','R','E','D','G','H','I','J','S','Z','M','N','U','S'),
    ('C','O','N','V','E','R','S','A','T','I','O','N','W','Y','P'),
    ('B','C','E','E','F','G','H','I','J','A','N','M','N','E','Q'),
    ('B','C','L','E','W','O','R','K','S','T','A','T','I','O','N'))

    var words: wordsType := (
```

```
    ('P','R','O','T','O','C','O','L','\n','\n','\n','\n','\n'),
    ('P','R','I','D','E','\n','\n','\n','\n','\n','\n','\n','\n'),
    ('R','E','L','A','T','I','V','I','T','Y','\n','\n','\n'),
    ('P','E','R','F','O','R','M','A','N','C','E','\n','\n'),
    ('C','O','M','P','U','T','E','R','S','\n','\n','\n','\n'),
    ('M','E','M','B','E','R','S','H','I','P','\n','\n','\n'),
    ('C','O','N','V','E','R','S','A','T','I','O','N','\n'),
    ('A','R','I','Z','O','N','A','\n','\n','\n','\n','\n','\n'),
    ('K','E','R','N','E','L','\n','\n','\n','\n','\n','\n','\n'),
    ('W','O','R','K','S','T','A','T','I','O','N','\n','\n'),
    ('R','E','C','O','V','E','R','Y','\n','\n','\n','\n','\n'))

    # wordStatus: array of size NUMWORDS X 4.
    # Col 1. records id. of player that found or is working on the word
    #                             (0 = word not assigned to any player),
    # cols. 2 and 3 record the x and y co-ords of the word in the grid,
    # col. 4 records the orientation of the word (1 = vert.l, 2 = hor.)
    var wordStatus: statusType := ([ NUMWORDS] ([4] 0))

    var i, x, y: int
    var wnum, wordlen: int
    var found: bool
    var orient: char
    var me: int

  # screen initalization functions
  external init_screen() returns int            # initialize screen
  external passGridElem(int; int; char) returns int # init grid
  external passWordElem(int; int; char) returns int # init wordlist

  # display functions
  external displayGrid() returns int            # display grid
  external displayWordList() returns int        # display wordlist
  external displayWord(int; int) returns int  # highlight word in list
  external clearWord(int; int) returns int      # unhighlight word
  external displayFind(int; int; char; int) returns int #highlight grid

  proc start()
     me := myid
     init_screen()
     fa row := 1 to GRIDSIZE, col := 1 to GRIDSIZE →
        passGridElem(row, col, grid[row][col])
     af
     displayGrid()
     fa row := 1 to NUMWORDS, col := 1 to MAXWORDLEN →
        passWordElem(row, col, words[row][col])
     af
     displayWordList()
     monitor myresource() send mygroup().playerDied(me)
     send play(words, grid, wordStatus)
```

```
end start

proc play(words, grid, wordStatus)
   send mygroup().getWord(me)
   do true →
        in getWord(id) →
              i := 1;
              do i ≤ NUMWORDS and wordStatus[i][1] ≠ 0 →
                    i++
              od
              if i > NUMWORDS →
                    next
              [] else →
                    wordStatus[i][1] := id
              fi
              wnum := i
              displayWord(id, wnum)
              if id = me →
                    # search for word
                    wordlen := 0;
                    do words[wnum][wordlen + 1] ≠ '\n' →
                          wordlen++
                    od

                    # check rows
                    found := false
                    fa row := 1 to GRIDSIZE,
                              col := 1 to GRIDSIZE−wordlen
                              st not found →
                       found := true
                       fa letter := 1 to wordlen
                                 st grid[row][col+letter]
                                 ≠ words[wnum][letter] →
                          found := false
                       af
                       if found →
                             x := row
                             y := col + 1
                             orient := 'h'
                       fi
                    af

                    # check columns
                    if not found →
                         fa col := 1 to GRIDSIZE,
                                   row := 0 to GRIDSIZE−wordlen
                                   st not found →
                             found := true
                             fa letter := 1 to wordlen
                                       st grid[row+letter][col]
```

```
                                ≠ words[wnum][letter] →
                            found := false
                        af
                        if found →
                            x := row + 1
                            y := col
                            orient := 'v'
                        fi
                    af
                fi

                if not found →
                    write("*panic* word ", wnum, " not in grid!")
                [] else →
                    # found word
                    send mygroup().foundWord(me, wnum, x, y, orient)
                    send mygroup().getWord(me)
                fi
            fi

        [] foundWord(id, wnum, x, y, orient) →
            if wordStatus[wnum][1] = id and wordStatus[wnum][4] = 0 →
                wordStatus[wnum][1] := id
                if orient = 'v' →
                    wordStatus[wnum][4] := 1
                [] orient = 'h' →
                    wordStatus[wnum][4] := 2
                fi
                wordStatus[wnum][2] := x
                wordStatus[wnum][3] := y
                displayFind(x, y, orient, wnum)
            fi

        [] playerDied(id) →
            fa i := 1 to NUMWORDS
                    st wordStatus[i][1] =id and wordStatus[i][4] = 0 →
               wordStatus[i][1] := 0
               clearWord(id, i)
            af

        [] sendState(playercap) →
            send playercap.getState(wordStatus)
        ni
    od
end play

recovery
  var grid: gridType :=  (
  ('K','E','R','N','E','L','R','R','E','C','O','V','E','R','Y'),
  ('B','C','S','Z','F','R','E','L','A','T','I','V','I','T','Y'),
```

```
('Y','I','P','E','F','G','H','I','J','K','L','M','N','O','P'),
('P','E','R','F','O','R','M','A','N','C','E','M','N','O','C'),
('B','C','O','E','F','G','H','I','J','K','L','M','N','O','O'),
('B','C','T','E','F','N','T','G','V','Q','O','B','D','A','M'),
('B','C','O','E','F','M','E','M','B','E','R','S','H','I','P'),
('B','C','C','E','F','G','H','O','J','K','L','M','N','O','U'),
('D','X','O','T','P','S','Y','N','C','M','A','V','K','I','T'),
('B','C','L','E','R','G','H','V','J','F','R','M','C','A','E'),
('B','C','E','E','I','G','H','I','J','U','I','P','M','X','R'),
('B','C','R','E','D','G','H','I','J','S','Z','M','N','U','S'),
('C','O','N','V','E','R','S','A','T','I','O','N','W','Y','P'),
('B','C','E','E','F','G','H','I','J','A','N','M','N','E','Q'),
('B','C','L','E','W','O','R','K','S','T','A','T','I','O','N'))

var words: wordsType := (
('P','R','O','T','O','C','O','L','\n','\n','\n','\n','\n'),
('P','R','I','D','E','\n','\n','\n','\n','\n','\n','\n','\n'),
('R','E','L','A','T','I','V','I','T','Y','\n','\n','\n'),
('P','E','R','F','O','R','M','A','N','C','E','\n','\n'),
('C','O','M','P','U','T','E','R','S','\n','\n','\n','\n'),
('M','E','M','B','E','R','S','H','I','P','\n','\n','\n'),
('C','O','N','V','E','R','S','A','T','I','O','N','\n'),
('A','R','I','Z','O','N','A','\n','\n','\n','\n','\n','\n'),
('K','E','R','N','E','L','\n','\n','\n','\n','\n','\n','\n'),
('W','O','R','K','S','T','A','T','I','O','N','\n','\n'),
('R','E','C','O','V','E','R','Y','\n','\n','\n','\n','\n'))

var wordstatus: statusType

 me := myid
 send mygroup().sendState(myresource())

 init_screen()
 fa row := 1 to GRIDSIZE, col := 1 to GRIDSIZE →
    passGridElem(row, col, grid[row][col])
 af
 displayGrid()
 fa row := 1 to NUMWORDS, col := 1 to MAXWORDLEN →
    passWordElem(row, col, words[row][col])
 af
 displayWordList()
 receive getState(wordstatus)
 fa i := 1 to NUMWORDS
         st wordstatus[i][1] ≠ 0 and wordstatus[i][4] ≠ 0 →
    # word has been found
    displayWord(wordstatus[i][1], i)
    if wordStatus[i][4] = 1 →
         displayFind(wordstatus[i][2], wordstatus[i][3], 'v', i)
    [] else →
         displayFind(wordstatus[i][2], wordstatus[i][3], 'h', i)
```

```
         fi
      af
      send play(words, grid, wordstatus)
   end recovery
end player
```

```
resource main
   import player
body main()
   const HERSHEY := 28
   const LUCIDA := 48

   var playerCap: cap player
   var vmCap[4]: cap vm
   vmCap[1] := create vm() on HERSHEY
   vmCap[2] := create vm() on LUCIDA
   vmCap[3] := myvm()

   playerCap := create (i := 1 to 2) player(i) on vmCap[i] backups on
vmCap[3]

   send playerCap.start()
end main
```

REFERENCES

[AA86] Jacob A. Abraham and Vinod K. Agarwal. Test generation for digital systems. In Dhiraj K. Pradhan, editor, *Fault-Tolerant Computing: Theory and Techniques*, chapter 1, pages 1–94. Prentice-Hall, 1986.

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of Second International Conference on Software Engineering*, pages 562–570, October 1976.

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Company, 1993.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael A. Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[Avi85] Algirdas Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[Bal91] Henri E. Bal. A comparative study of five parallel programming languages. In *Proceedings of EurOpen Conference on Open Distributed Systems*, May 1991.

[BD88] A. Burns and G. Davies. Pascal-FC: A language for teaching concurrent programming. *ACM SIGPLAN Notices*, 23(1):58–66, January 1988.

[BH75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–206, June 1975.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1987.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[BMZ92] Peter A. Buhr, Hamish I. MacDonald, and C. Robert Zarnke. Synchronous and asynchronous handling of abnormal events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, September 1992.

[BSS91] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CASD85] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Digest of Papers, The Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206. IEEE Computer Society, June 1985.

[CGR88] R.F. Cmelik, N.H. Gehani, and W. D. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 55–61. IEEE Computer Society, IEEE Computer Society Press, June 1988.

[CM84a] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CM84b] Jo-Mei Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Com86] Computer Systems Research Group, Computer Science Division, University of California, Berkeley. *Unix Programmer's Manual: Supplementary Documents 1*, April 1986.

[Coo80] Robert P. Cook. *MOD—a language for distributed programming. *IEEE Transactions on Software Engineering*, SE-6(6):563–571, November 1980.

[Coo85] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78. ACM SIGOPS, December 1985.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[DDH72] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured Programming*. A.P.I.C. studies in data processing; no. 8. Academic Press, 1972.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[DLAR91] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *Computer*, 24(11):34–44, November 1991.

[DoD83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. Washington D.C., 1983.

[EFH82] C.S. Ellis, J.A. Feldman, and J.E. Heliotis. Language constructs and support systems for distributed computing. In *ACM Symposium on Principles of Distributed Computing*, pages 1–9. ACM SIGACT-SIGOPS, August 1982.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[ES86] Paul D. Ezhilchelvan and Santosh K. Shrivastava. A characterisation of faults in systems. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 215–222, January 1986.

[Fel79] Jerome A. Feldman. High level programming for distributed computing. *Communications of the ACM*, pages 353–368, June 1979.

[Fis91] Alan S. Fisher. *CASE: Using Software Development Tools*. Wiley professional computing. Wiley, New York, 2 edition, 1991.

[GMS89] Hector Garcia-Molina and Annemarie Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 354–361, June 1989.

[GR89] N. H. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.

[Gra79] James N. Gray. Notes on data base operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems, An Advanced Course*, chapter 3.F, pages 393–481. Springer-Verlag, 1979.

[Gra86a] James N. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, June 1986.

[Gra86b] Jim Gray. Why do computers stop and what can be done about it. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

[Gri81] David Gries. *The Science of Programming*. Texts and monographs in computer science. Springer-Verlag, New York, 1981.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HS93] Matti A. Hiltunen and Richard D. Schlichting. An approach to constructing modular fault-tolerant protocols. Technical Report TR 93-10, Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, 1993.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language support for reliable distributed systems. In *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, pages 89–94. IEEE Computer Society, IEEE Computer Society Press, July 1987.

[Jac90] Jonathan Jacky. Inside risks: Risks in medical electronics. *Communications of the ACM*, 33(12):138, December 1990.

[JZ90] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, pages 462–491, 1990.

[KK89] Brent A. Kingsbury and John T. Kline. Job and process recovery in a UNIX-based operating system. In *Proceedings of the 1989 Winter USENIX Technical Conference*, pages 355–364, 1989.

[KMBT92] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pages 297–311, March 1992.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[KU87] John C. Knight and John I. A. Urquhart. On the implementation and use of Ada on fault-tolerant distributed systems. *IEEE Transactions on Software Engineering*, SE-13(5):553–563, May 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam81] Butler W. Lampson. Atomic transactions. In B.W. Lampson, M. Paul, and H.J. Seigert, editors, *Distributed Systems—Architecture and Implementation*, chapter 11, pages 246–265. Springer-Verlag, 1981. Originally vol. 105 of Lecture Notes in Computer Science.

[Lap91] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1991.

[Lis85] Barbara Liskov. The Argus language and system. In M. Paul and H.J. Siegert, editors, *Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science, Volume 190*, chapter 7, pages 343–430. Springer-Verlag, Berlin, 1985.

[LL81] C.-M. Li and M.T. Liu. Dislang: A distributed programming language/system. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 162–172, 1981.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LW85] Richard J. LeBlanc and C. Thomas Wilkes. Systems programming with objects and actions. In *The 5th International Conference on Distributed Computing Systems*, pages 132–139, Denver, Colorado, May 1985. IEEE Computer Society.

[Mad86] Hari Madduri. Fault-tolerant distributed computing. *Scientific Honeyweller*, Winter 1986-87:1–10, 1986.

[Mis92] Shivakant Mishra. *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs*. PhD thesis, Department of Computer Science, University of Arizona, Tucson, Arizona, 1992. Also available as techical report 92-06.

[MS92] Shivakant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, Department of Computer Science, University of Arizona, Tucson, AZ 85721, 1992.

[MSMA90] P.M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[Neu92] Peter G. Neumann. Inside risks: Avoiding weak links. *Communications of the ACM*, 35(12):146, December 1992.

[OP92] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[PBS89] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[Pow91] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[Pow92] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers, The Twenty Second International Symposium on Fault-Tolerant Computing*, pages 386–395, July 1992.

[Pra86] D. K. Pradhan, editor. *Fault-Tolerant Computing: Theory and Techniques*, volume 1 and 2. Prentice-Hall, 1986.

[PVB+88] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton. The Delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 246–251, June 1988.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[RLT78] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–166, 1978.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SCP91] Richard D. Schlichting, Flaviu Cristian, and Titus D. M. Purdin. A linguistic approach to failure-handling in distributed systems. In Algirdas Avižienis and Jean-Claude Laprie, editors, *Dependable Computing and Fault-Tolerant Systems, Vol. 4: Dependable Computing for Critical Applications*, pages 387–409. Springer-Verlag, Wien and New York, 1991.

[SDD+85] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed transactions for reliable systems. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 127–146, 1985.

[SDP91] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, January 1991.

[SMR88] S.K. Shrivastava, L.V. Mancini, and B. Randell. On the duality of fault tolerant system structures. In G. Goos and J. Hartmanis, editors, *Experiences with Distributed Systems: Lecture Notes in Computer Science, Volume 309*, pages 19–37. Springer-Verlag, Berlin, 1988.

[Som92] Ian Sommerville. *Software Engineering*. International computer science series. Addison-Wesley, Reading, Massachussetts, fourth edition, 1992.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *IEEE Transactions on Computing Systems*, 1(3):222–238, August 1983.

[SW89] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[SY85] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Toh86] Yoshihiro Tohma. Coding techniques in fault-tolerant, self-checking, and fail-safe circuits. In Dhiraj K. Pradhan, editor, *Fault-Tolerant Computing: Theory and Techniques*, chapter 5, pages 336–415. Prentice-Hall, 1986.

[Wei89] William E. Weihl. Using transactions in distributed applications. In Sape Mullender, editor, *Distributed Systems*, pages 215–235. Addison-Wesley Publishing Company, ACM Press, New York, New York, 1989.

[Zie83] Carol A. Ziegler. *Programming System Methodologies*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.