CONSUL: A COMMUNICATION SUBSTRATE FOR FAULT-TOLERANT DISTRIBUTED PROGRAMS

by

Shivakant Mishra

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

1992

Order Number 9225185

Consul: A communication substrate for fault-tolerant distributed programs

Mishra, Shivakant, Ph.D.

The University of Arizona, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

CONSUL: A COMMUNICATION SUBSTRATE FOR FAULT-TOLERANT DISTRIBUTED PROGRAMS

by

Shivakant Mishra

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

1992

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Shivakant Mishra

entitled CONSUL: A COMMUNICATION SUBSTRATE FOR FAULT-TOLERANT DISTRIBUTED PROGRAMS

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Richard D. Schlichting	10/24/91 Date
Larry L. Peterson	10/24/91 Date
Richard T. Snodgrass	10/24/91 Date
David Gay	10/24/91 Date
John Leonard	10/24/91 Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Richard D. Schlichting — 10/24/91 Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: B Mishra

ACKNOWLEDGMENTS

I want to express my thanks to my advisor, Rick Schlichting. He has been an unsurpassable advisor, mentor and friend throughout the course of this work. I also want to thank Larry Peterson for his advice and guidance, which helped to nurture this research. Through innumerable discussions with Rick and Larry, this thesis was improved in both content and presentation. Without their unselfish investment of time this dissertation would never have been completed.

I am also grateful to the other member of my committee, Rick Snodgrass, for his comments and suggestions on my work. I would like to thank the minor members of my committee, David Gay and John Leonard, for helping me with my graduate program.

I also want to thank my fellow graduate students, Vic Thomas, Mike Soo, Andrey Yeatts, Tyson Henry, Shamim Mohamed, Herman Rao, Patrick Homer, Nick Kline, Curtis Dyreson, Bob Simms, Jim Knight, and Mudita Jain, for their friendship.

Finally, I would like to thank my parents for their faith and support; my brother and sister for their encouragement; and the wonderful deserts, mountains, and canyons of Arizona for helping me keep my equilibrium.

This work was supported by The National Science Foundation under Grants CCR-8811423 and CCR-9003161, and by The Office of Naval Research under Grant N0001491J-1015.

TABLE OF CONTENTS

LIST OF FIGURES 8

LIST OF TABLES 9

ABSTRACT 10

CHAPTER 1: Introduction 12

1.1 Dependable Computing Systems 12

1.2 Implementing Fault Tolerance 15

1.3 Programming Fault-Tolerant Distributed Systems 16

1.4 Fault-Tolerant Services for the State Machine Approach 18

1.5 Dissertation Outline 19

CHAPTER 2: A Unified Framework 21

2.1 Fault-Tolerant System Model 21

2.1.1 Synchrony 22

2.1.2 Failure Models 23

2.2 System Organization 25

2.2.1 Services, Servers and the "Depends on" Relation 25

2.2.2 The State Machine Approach 26

2.3 Fault-Tolerant Services 27

2.3.1 Time Service 28

2.3.2 Broadcast Service 35

2.3.3 Membership Service 38

2.3.4 Recovery Service 42

2.3.5 Stable Store Service 44

2.4 Fault-Tolerant Systems 45
2.4.1 Isis 45
2.4.2 Advanced Automation System 46
2.4.3 MARS 46
2.4.4 DELTA-4 47
2.4.5 Arjuna 47
2.5 Conclusion 48

CHAPTER 3: System Architecture 49
3.1 Substrate Architecture 49
3.2 Interprocess Communication Support 53
3.2.1 Basic Operation of Psync 53
3.2.2 Fault-Tolerance Aspects 56

CHAPTER 4: Ordering Protocols 63
4.1 Semantic Dependent Order 64
4.1.1 Ordering the Operations 65
4.1.2 Overview 66
4.1.3 Algorithm 70
4.1.4 Correctness Arguments 74
4.1.5 Generalizing the Algorithm 78
4.1.6 Limitations 79
4.2 Related Work 79

CHAPTER 5: Failure Handling Protocols 81
5.1 Membership Service 81
5.1.1 Correctness Criteria 82
5.1.2 Single Failures 83
5.1.3 Multiple Failures 85
5.1.4 Correctness Arguments 91
5.1.5 Related Work 97

5.2 Recovery Service 98
5.2.1 Checkpointing and Message Logging 98
5.2.2 Recovery Stages 100

CHAPTER 6: Implementation and Performance 104
6.1 Overview of the x-Kernel 104
6.1.1 Communication Objects 104
6.1.2 Implementation techniques 105
6.1.3 Operations 106
6.2 Substrate Implementation 108
6.2.1 Message Structure 108
6.2.2 Establishing Connections 109
6.2.3 Restoring Connections 115
6.3 Performance 116

CHAPTER 7: Conclusion 121
7.1 Summary 121
7.2 Contributions and Limitations 123
7.3 Future Directions 124

REFERENCES 126

LIST OF FIGURES

1.1 Hierarchical System Organization 14

2.1 Failure model hierarchy 24

3.1 Overall System Architecture 50
3.2 Communication Substrate 51
3.3 Example Context Graph 54
3.4 Another Example Context Graph 55

4.1 Context Graph Representing Operations 67
4.2 Example Partial Ordering of Operation Invocations 69
4.3 Execution of update operations 73

5.1 Membership Protocol Assuming Single Failure 84
5.2 View Representing a Membership Check Period 90
5.3 Membership Protocol 91
5.4 Two Different Processes' View 92
5.5 Stages of Recovery Service 100
5.6 Context Graph at Recovery 101

6.1 Example x-Kernel Configuration 105
6.2 Relationship Between Protocols and Sessions 107
6.3 Operation Type Message 108
6.4 Monitoring Type Message 109
6.5 Protocol and Session Objects in the Communication Substrate 110
6.6 Message Flow Upwards in the Communication Substrate 112
6.7 Message Flow Downwards in the Communication Substrate 114
6.8 Response Time of the System 118

LIST OF TABLES

6.1 System Response Time (in msec) for a 2-replica system 118
6.2 System Response Time (in msec) for a 3-replica and 4-replica system 118
6.3 Response Time with Failure Handling Protocols (in msec) 119
6.4 Measure of Checkpointing Overheads 120

ABSTRACT

As human dependence on computing technology increases, so does the need for computer system dependability. This dissertation introduces Consul, a communication substrate designed to help improve system dependability by providing a platform for building fault-tolerant, distributed systems based on the replicated state machine approach. The key issues in this approach—ensuring replica consistency and reintegrating recovering replicas—are addressed in Consul by providing abstractions called fault-tolerant services. These include a broadcast service to deliver messages to a collection of processes reliably and in some consistent order, a membership service to maintain a consistent system-wide view of which processes are functioning and which have failed, and a recovery service to recover a failed process.

Fault-tolerant services are implemented in Consul by a unified collection of protocols that provide support for managing communication, redundancy, failures, and recovery in a distributed system. At the heart of Consul is Psync, a protocol that provides for multicast communication based on a context graph that explicitly records the partial (or causal) order of messages. This graph also serves as the basis for novel algorithms used in the ordering, membership, and recovery protocols. The ordering protocol combines the semantics of the operations encoded in messages with the partial order provided by Psync to increase the concurrency of the application. Similarly, the membership protocol exploits the partial ordering to allow different processes to conclude that a failure has occurred at different times relative to the sequence of messages received, thereby reducing the amount of synchronization required. The recovery protocol combines checkpointing with the replay of messages stored in the context graph to recover the state of a failed process. Moreover, this collection of protocols is implemented in a highly-configurable manner, thus allowing a system builder to easily tailor an instance of Consul from this collection of building-block protocols.

Consul is built in the *x*-Kernel and executes standalone on a collection of Sun 3 work-

stations. Initial testing and performance studies have been done using two applications: a replicated directory and a distributed wordgame. These studies show that the semantic based order is more efficient than a total order in many situations, and that the overhead imposed by the checkpointing, membership, and recovery protocols is insignificant.

CHAPTER 1

Introduction

Human dependence on computing technology has been increasing over the past 3 decades, a trend that is likely to continue. This dependence results from the diverse applications in which computers are currently being used. These include such things as aircraft control, space applications, medical applications, nuclear reactors, defense systems, banking systems, and telephones. It is clear that computer systems are becoming an integral part of everyday life.

The correct functioning of computing systems is vital to ensure the integrity of these applications. The consequences of a malfunction in these applications may range from mere inconvenience to economic disruption or even loss of life. Moreover, with the advancement in computing technology, modern computing systems have become extremely complex. This is especially true for software systems; they may contain millions of lines of code and hundreds of millions of possible states, making them virtually impossible to understand completely. The combination of increased reliance on computing technology and the intricacy of these systems implies that more attention needs to be paid to the dependability of these systems. This dissertation addresses this problem by proposing new techniques for enhancing the dependability of computing systems.

1.1 Dependable Computing Systems

While different applications have different consequences resulting from failures, *dependability* is a generic term used to express the need for a system to perform its intended task. Specifically, dependability is that property of a computing system that allows reliance to be justifiably placed on the service it delivers.[1] The *service* delivered by a system is its behavior as perceived by the *user*, another system or human with which it interacts.

[1] Definitions in this section are taken from [Lapr91].

Depending on the application, dependability may be viewed according to different, but complementary, properties. These properties include *availability*—readiness of the system to be used, *reliability*—continuity of the service, *safety*—avoidance of catastrophic consequences on the environment, and *security*—preservation of confidentiality. Investigation of each of these properties leads to a separate area of research.

While a dependable computing system must have one or more of these characteristics, its realization is complicated by the possibility of malfunction of one or more of the system components. A *failure* of a system occurs when the behavior of the system first deviates from that required by its *specification*, the latter being an agreed upon description of the expected service. It is important to note that this definition does not require a failure to be identified or even observed; all that is required is that a failure could be identified by a rigorous application of the specification. An *error* is that part of the system state—with respect to the computation process—that is liable to lead to failure. The cause of an error is a *fault*. An error is thus the manifestation of a fault in the system, while a failure is the effect of an error on the service.

In general, a complex computing system is not a monolithic entity. It consists of a set of interacting *components*, each of which may have faults and therefore may fail. To understand how failures in interacting components affect each other, we have to first discuss how a computing system is organized. While there are many different ways to organize a computing system, a typical approach is hierarchical [Dijk68]. In this approach, the system consists of a hierarchy of layers in which each layer is a component or a collection of hardware or software components. Typically, the layers at the bottom of this hierarchy are primarily hardware components, while layers higher up in the hierarchy are more likely to be composed of software components. Each such layer utilizes the functions provided by the layers below and, in turn, provides functions to the upper layers according to a certain specification. The internal details of how the functions are implemented is hidden from the higher layers. Operating system kernels, virtual memory, and file systems are examples of some of these abstract layers.

In this organization, where higher layers depend on lower layers, failures of lower layers may affect the higher layers. If a layer depends on some lower layer to provide a specific

function, a failure at the lower layer of abstraction may be propagated to layers at the higher levels. In particular, a failure at lower layers may be a fault from the perspective of the higher layers. This fault may cause an error which, in turn, may lead to the failure of the layer. This propagation of failures through these hierarchical layers is shown in Figure 1.1. This view of failure propagation through hierarchical layers simplifies the complexity associated with the failures of different components of the system and their effects on the other components of the system. In Chapter 2, we will refine this view further using a relation that models the specific dependence between components.

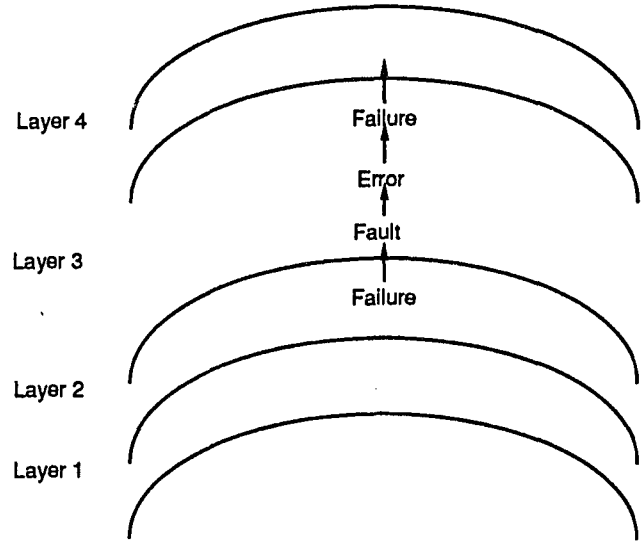

Figure 1.1: Hierarchical System Organization

Techniques for enhancing dependability can also be explained in relation to this organization. In particular, a computing system is made dependable by either preventing the original failures from occurring or by providing the hardware or software at a layer to mask the failure of the current or lower layers. These two possibilities lead to two approaches to constructing dependable computing systems: *fault prevention* and *fault tolerance*. Fault prevention attempts to ensure that a computing system is, and remains, free from faults that may be introduced in the system due to design problems or environmental factors. This requires that all the possible causes of faults be removed from the computing

system during its development, before the system is put into service and dependence is placed on its operation. *Fault avoidance* and *fault removal* are two general techniques of fault prevention. Fault avoidance is concerned with developing design methodologies so as to avoid the introduction of faults in the system during the design and implementation phases. Structured programming [Dahl72], top-down development [Somm89], information hiding [Parn72], separation of concerns [Dijk76], and abstract data types [Meye88] are all examples of the fault avoidance technique. Despite the use of fault avoidance techniques, some faults inevitably remain in the system. Fault removal is concerned with checking the implementation of a system and removing the faults that are thereby exposed. Various system testing techniques [Boeh73, Lala85, Myer76, Siew82] are the examples of fault removal.

In contrast, fault tolerance techniques assume that the system is not perfect, and thus provide ways to manage failures when they occur while the system is operational. The aim of fault tolerance is to prevent errors and faults from leading to system failures. There are several reasons why fault prevention techniques must increasingly be augmented by fault-tolerance techniques for the types of critical applications mentioned above. First, the application of fault prevention techniques have not, in general, proved sufficient to attain high levels of dependability. Second, the complexity of the system makes it difficult to get a perfect design. Third, the environment in which these systems operate may be harsh, which makes them more prone to failures. While it can be expected that the effectiveness of fault prevention techniques will continue to improve, it is highly unlikely that a complex system will ever be free from faults. Thus, to provide a high degree of reliability, measures to provide fault tolerance must be adopted.

1.2 Implementing Fault Tolerance

Fundamental to realizing fault tolerance is *redundancy*, that is, extra elements that would not be required to provide the service if there were no faults. A fault-tolerant system maintains certain redundant information that is used to continue providing the service when faults occur. This redundancy may be introduced into the system in many different ways: extra hardware components such as transistors, logic gates and memory units;

extra software components such as replicated processes; extra data such as replicated data; and extra time such as instruction retry and message retransmissions. Typically, a fault-tolerant system makes use of a combination of these techniques to ensure dependable operation.

Depending on the type of redundancy and the kinds of faults and failures that need to be tolerated, fault tolerance may be provided at a particular level of system abstraction in one of two ways. The first approach is called *design fault tolerance.* This approach deals with the faults introduced at the current level by attempting to mask them from the higher levels of abstraction using redundancy. Design fault tolerance, thus, deals with faults that may be introduced in the design of that level of the system, be it hardware or software. Techniques such as the recovery block scheme [Horn74, Ande76] and N-Version programming [Aviz85, Chen78] provide this type of fault tolerance. This technique is also sometimes called *software fault tolerance* [Lee90].

The second, called *operational fault tolerance*, deals with faults introduced by the failure of the next lower level of abstraction. In the hierarchy of abstract layers, operational fault tolerance attempts either to mask completely the failure of lower layers or to convert it to a more benign failure that can be handled with less difficulty at the higher layers. Component (hardware and software) redundancy is typically used to realize this type of fault tolerance. *Fault-tolerant programming* is a technique for implementing operational fault tolerance in software. In this technique, a program is written that uses redundancy to tolerate failures at lower layers. The specific problems that are addressed in this technique include detection of the failure of lower layers, maintaining consistency in the presence of failures and so on. A program written using these techniques is also sometimes called *fault-tolerant software.*

1.3 Programming Fault-Tolerant Distributed Systems

Distributed systems, in which many processing elements (possibly heterogeneous) are connected by a communication network, are especially relevant in the study of fault-tolerant systems. The relationship between distributed systems and fault tolerance is based on two factors. First, the assembling of many computers together in distributed systems gives

rise to a possibility of partial failure, that is, a situation where portions of the system continue to execute while other parts have failed. In fact, it is important to keep such a distributed system running because the probability of partial failures can be significant if there are a large number of processors. Thus, distributed systems need to be fault tolerant. Second, due to the multiplicity of the elements, distributed systems provide possibilities for redundancy and graceful degradation. Fault-tolerant systems can exploit this inherent redundancy in distributed systems to keep executing when failures occur. Thus, distributed systems and fault tolerance are two sides of the same coin: distributed systems need fault tolerance and fault-tolerant systems can make use of the redundancy provided by distributed systems.

From the perspective of a software designer, the problem of ensuring the dependability of a distributed system reduces to the problem of writing the system or application software as a fault-tolerant, distributed program able to continue correct execution despite failures in its underlying computing platform. Unfortunately, writing such programs has proven difficult due to factors such as the possibility of arbitrary partial failures, and random communication and processing delays. To deal with this complexity, various programming paradigms and techniques have been proposed to help the user design and reason about this type of software. While no one paradigm or technique is suitable for all kinds of fault-tolerant distributed programming, at least three different models have been proposed: the *object-action* model [Lamp81, Reed83, Gray87], the *conversation* model [Ande83], and the *replicated state machine* approach [Schn90]. We describe each in turn.

In the object-action model, the primary components are objects, which have a state and export certain operations to modify that state. The system consists of one or more objects that interact with one another by invoking various of these exported operations. The operations are executed as atomic actions in this model, *i.e.*, an action is either executed completely or not at all. Thus, the view provided by this model is that the system moves from one well-known state to another, with the guarantee that no failure can occur during state transition.

In the conversation model, processes and messages play a primary role. An application is structured out of a number of concurrent processes that communicate by exchanging

messages. Techniques for reliable communication between processes, consistent checkpointing of the states of the processes, and state recovery are required to implement this model. The object-action model and conversation model have been shown to be duals [Shri88].

The primary components of the state machine approach are state machines which consist of services, servers, and various programming language structures. The system consists of one or more state machines interacting with one another. Each state machine maintains some state variables that are modified in response to commands whose execution is deterministic and atomic with respect to other commands. The output of a state machine is completely determined by the sequence of requests for the command execution. Failures are masked by replicating these state machines. Issues such as maintaining replica consistency at all times and integrating repaired replicas are addressed by the state machine approach.

1.4 Fault-Tolerant Services for the State Machine Approach

Implementation of the state machine approach is simplified by using abstractions called *fault-tolerant services.* Examples of such services include *broadcast services*—to deliver messages to the cooperating processes reliably and in some consistent order, *membership services*—to maintain a consistent system-wide view of which processes are functioning and which have failed, and *recovery services*—to recover a failed process. Much work has been done on developing algorithms and implementation for these services [Birm87, Chan84, Cris88, Dole84, Koo87, Lamp78]. Chapter 2 elaborates on the characteristics of each of these.

In this dissertation, we describe the design and implementation of Consul, a unified collection of protocols that provide these services. This collection of protocols, which forms a communication substrate upon which fault-tolerant applications can be built, provides support to manage redundancy, failures, and recovery in a distributed system. Specifically, the fault-tolerance support includes process failure detection, restart of failed processes, and reliable communication between processes. Support for general distributed processing

is included in Consul as well. This support includes interprocess communication within a group of processes and different kinds of consistent orderings among messages exchanged in the system.

This dissertation contributes to both the theory and practice of providing fault tolerance in distributed systems. In particular, we introduce new algorithms for the protocols in Consul, as well as propose new system structuring techniques. The new algorithms are based on partial ordering among the messages exchanged in the system, a property provided by the Psync interprocess communication mechanism that forms the basis of the substrate [Pete89]. This results in more efficient algorithms than ones proposed in the literature. The new system structuring techniques make it easy to modify the system architecture and to add new protocols in the substrate without affecting the existing ones. This results in a system that is composable in the sense that a user can pick the right combination of protocols needed for the application and then easily build a system using that combination.

1.5 Dissertation Outline

This dissertation is organized as follows. In Chapter 2, we start by briefly describing a system model for fault-tolerant distributed systems and the details of the state machine approach. This is followed by a survey of various fault-tolerant services that have been proposed. In doing so, we attempt to provide a uniform framework for understanding the different approaches. A brief survey of a number of fault-tolerant systems based on these services concludes the chapter.

Chapter 3 outlines the major goals of the system we have designed and gives an overall view of its architecture. The requirements for the communication substrate are discussed and a brief description of its modules is given. This chapter also evaluates the flexibility of the proposed architecture and gives a brief overview of the Psync protocol that forms the basis of the substrate.

Chapter 4 describes the ordering service provided in Consul. We give a rationale for different kinds of orderings possible in a distributed system, followed by a description of a total ordering protocol and a semantic dependent ordering protocol that are provided by

the ordering service.

Chapter 5 describes the services that handle failures and recoveries. The two main protocols implementing these services—membership and recovery—possess novel attributes, including the use of a partial ordering of the messages rather than the more restrictive total order. The membership protocol maintains a consistent system-wide view of which processes are functioning at any given moment in time. Moreover, it also handles simultaneous failures and recoveries in a flexible and efficient manner. The recovery protocol takes advantage of the history of messages maintained by Psync to reconstruct the state of a recovering process. In doing so, it consistently orders the recovery of the processes with other activities of the system.

Chapter 6 describes the implementation of Consul. The underlying infrastructure for this implementation is provided by *x*-Kernel, an operating system that aids in experimenting with the network protocol design [Hutc89, Pete90]. After outlining the relevant properties of the *x*-Kernel, we then turn to describing the important features of our system, with special attention to justifying certain design decisions. Performance measurements of the system as implemented on the *x*-Kernel are also presented.

Finally, the major contributions of this dissertation are summarized in Chapter 7 and the future course of this work is explored.

CHAPTER 2

A Unified Framework

Much of the work in fault tolerance has been driven by specific applications and operating environments, which has resulted in many diverse system models and techniques. In this chapter, we describe a framework that unifies these different models and techniques based on their fundamental characteristics. An important aspect of this is clarifying different terminologies that have been used in different ways by different researchers. We start by first describing a general system model for fault-tolerant systems and a system organization based on the concepts of service, server, and the "depends on" relation [Cris91]. This is followed by a brief description of the state machine approach and a survey of various of its supporting fault-tolerant services. An overview of some recent fault-tolerant systems is given at the end of the chapter.

2.1 Fault-Tolerant System Model

A fault-tolerant distributed system consists of a collection of components, such as a communication network, processors, and various software components. In general, it is assumed that there is no shared memory between the processors and that these processors communicate by exchanging messages. Each of these components interact with one another to provide the required service. In such a system, the way different components interact with one another and the mode in which each of these components may fail greatly affect the performance and dependability of the system.

Various components of the system interact with one another to perform a given task. This interaction may be periodic in nature or may be completely asynchronous. *Synchrony* defines the way in which a component interacts with other components of the system. Similarly, a component may fail in different ways. It may fail silently without taking any incorrect state transition or its behavior on failure may take arbitrary state transition.

Failure models define the way in which a component behaves when it fails. We discuss synchrony and failure models in the following.

2.1.1 Synchrony

A hardware or software system component is *synchronous* if it always performs its intended function within a finite time limit. This bound on the execution time of the synchronous component must hold whenever the component is correctly operating, and in particular, under all operation conditions within its specification. If the component does not meet this time limit, a failure has occurred. A common definition for synchronous components in the literature is that they interact with each other in a periodic fashion. This type of behavior follows from our definition since a bound on the maximum execution times of the individual components clearly makes it possible for them to interact with one another periodically.

Synchrony can be defined for communication channels, communication networks, processors, and protocols. In a *synchronous communication channel*, the transmission delay of a unit of data across the link is known and bounded. A *synchronous network* is one in which the transmission delay of a unit of data between any two nodes connected by the network is known and bounded. Similarly, a *synchronous processor* is a processor in which the time to execute a unit of work is known and bounded.

These definitions of synchronous components extend to a group of components as well. Synchronous channels are essential to build a synchronous communication network because the transmission of data through a network involves transmission of data across one or more channels and so any asynchrony in the communication channels can make the communication network asynchronous. Thus, on one extreme a distributed system is completely synchronous if it contains a synchronous network and synchronous processors, while on the other, a distributed system is completely asynchronous if it does not contain a synchronous network or synchronous processors. Examples of synchronous systems in the literature include [Cris90, Kope89a]; a completely asynchronous system is assumed in [Fisc85]. Some systems in the literature [Birm87, Chan84] have been described as asynchronous, but they employ restrictions to asynchrony [Dole83] and so are not completely

asynchronous. Examples of such restrictions include assumptions about failure detection or bounds on message transmission time.

The notion of synchrony can be applied to software components as well. A protocol is defined to be synchronous if the time to perform the protocol function is known and bounded. Typically, a protocol in a distributed system involves some processor activity and some communication with other processors in the system via the network. Hence, any asynchrony in the network or the processors may cause the protocol to execute indefinitely and not terminate. Thus, to implement a synchronous protocol, both the network and the processors involved in the protocol must be synchronous. Synchronous protocols have mostly been implemented in the presence of synchronized clocks [Cris85, Cris88, Kope89b]. In this approach, the local clocks of various processors are synchronized at regular intervals and various actions of the protocols are based on these clocks. As alluded to in [Verr90], synchronized clocks are not essential to implement synchronous protocols, so synchronous protocols may be *clockless* or *clock-driven* depending on whether or not they depend on synchronized clocks. If the sequence of events happening in a protocol is known and if the length of this sequence is bounded, its implementation in a completely synchronous distributed system yields a synchronous protocol.

2.1.2 Failure Models

When a specification of a component's acceptable behavior is available, it provides a standard against which the behavior of that component can be judged. The specification may prescribe both the component's response for any initial state and input sequence, and the real-time interval within which the response should occur. A component is *correct* if, in response to inputs, it behaves in a manner consistent with the specification [Cris91].

A *failure model* specifies the behavior of a component once it fails. Component failures may be classified into many types. In order to tolerate a failure, the actions to be taken upon detecting the failure depend on the characteristics of failures the component may suffer. Thus, the specification of a component behavior should not only include failure-free behavior, but also the behavior of the component when it fails.

A number of such failure models have been defined. In the *fail-stop* failure model,

it is assumed that the component does not make any inconsistent state transition after the failure and that this failure is detectable [Schl83]. In a *crash failure* model, a component is assumed to fail by stopping without undergoing any inconsistent state transition, but without the guarantee of detectability. This model has also been termed *fail-silent* [Powe88]. The *omission failure* model assumes that a component fails by omitting to respond to an input. Crash failures are a special case of omission failures where a component fails to respond to inputs after the first omission to produce the output. The *timing failure* model assumes that a component fails by giving an untimely response. Thus, the response is functionally correct but occurs outside the real-time interval specified. The timing failure can be *early* timing failure or a *late* timing failure; late timing failures are also sometimes called *performance* failures. A failure is classified as *arbitrary* or *Byzantine* if the component's failure behavior is completely unspecified. In particular, components may take unknown, inconsistent, or even malicious actions if they are assumed to suffer Byzantine failure.

Figure 2.1 shows the inclusion relationship between various failure models. Byzantine failures are the most general kind of failure that can occur and so that class includes the rest. Timing failures are a subclass of Byzantine failures. A component that suffers an omission failure can be understood as having infinite response time, meaning that omission failures are fully contained in timing failures. Crash failures are a proper subclass of omission failures where a component fails to respond to inputs after the first omission to produce the output. Finally fail-stop failures, where failure detection is assumed, are a subclass of the crash failures.

2.2 System Organization

As outlined in Chapter 1, a computing system is made fault tolerant by incorporating redundant components. The interactions and the dependencies among these components greatly affect the dependability and performance of the system. In this section, we describe the basic architectural building blocks of a fault-tolerant system and how they relate to one another. The state machine approach is one of the ways to implement this architecture, so a brief description of this approach follows at the end of this section.

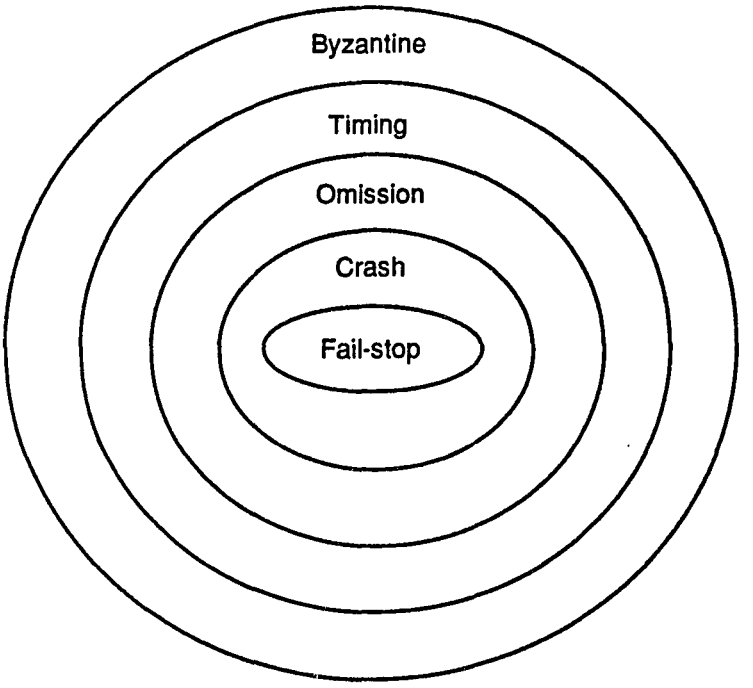

Figure 2.1: Failure model hierarchy

2.2.1 Services, Servers and the "Depends on" Relation

The organization of a computing system as a hierarchy of layers as described in Chapter 1 can be refined further by means of three concepts: service, server, and the "depends on" relation [Cris91]. A computing *service* specifies a collection of operations whose execution can be triggered by inputs from service users or by the passage of time. The execution of these operations may result in outputs to the users and a change in the state of the service. A service is implemented by a *server* that provides the operations without exposing to the users the internal service state representation and the implementation details. Operations defined by a service specification can be performed only by a server of that service, which can be implemented in either hardware or software. Examples of computing services include communication services, file services, and database services. These services are implemented by communication servers, file servers and database servers, respectively.

Servers typically implement services using other services. This relationship is specified

by using the "depends on" relation. Specifically, a server u "*depends on*" another server v if the correctness of u depends on the correctness of v. In this case, u is called the *user* of v and v is called the *resource* of u. The user/resource names are relative to the "depends on" relation; what is a user at one level of abstraction may be a resource at another level of abstraction.

Because of the dependency between servers for their correct behavior, a failure at one server may be propagated to the servers that depend on its service. This propagation of failures through different levels is a complex phenomenon. The failure of a certain type at a lower-level servers can result in a failure of a different type at the higher level server. In general, the failure is masked or converted to a more benign failure by a higher level server.

2.2.2 The State Machine Approach

A typical way to ensure that a service remains available despite server failures is by implementing the service as a group of redundant servers, so that if some of them fail, the remaining ones continue to provide the service. In such a case, the group of servers *mask* the failures of one or more servers from the users. The specific mechanisms needed to manage redundant server groups to make the group behavior functionally indistinguishable from that of single server depends critically on the failure semantics specified for group members and the communication services assumed.

The state machine approach is a general method for implementing fault-tolerant services by replicating servers and coordinating user interactions with server replicas [Schn90]. A state machine consists of *state variables*, which encode its state, and *commands*, which transform its state. A user of the state machine makes a request to execute a command by submitting it to the state machine. The execution of this command, which is atomic with respect to other commands, modifies the state variables and may produce an output. The semantic characterization of a state machine is such that the outputs are completely determined by the sequence of requests it processes, independent of time or any other activity in the system.

A fault-tolerant state machine is implemented by replicating a state machine and

executing each replica on a processor in a distributed system. If each replica being run by a nonfaulty processor starts in the same initial state and executes the same requests in the same order, then each will perform the same execution and produce the same output. The key, then, for implementing a fault-tolerant state machine is to ensure *replica coordination*, that is, that all replicas receive and process the same sequence of requests.

Replica coordination can be decomposed into two requirements: *agreement* and *order*. Agreement requires that every nonfaulty state machine replica receive every request. This can be satisfied by using any protocol that allows a designated processor, called the *transmitter*, to disseminate a value to other processors in such a way that two properties are maintained: that nonfaulty processors agree on the same value and that all nonfaulty processors use the transmitter's value unless it is faulty.

The second part of replica coordination is order. This requires that every nonfaulty state machine replica process requests in the same relative order. This can be satisfied, for example, by assigning unique identifiers to requests and having state machine replicas process requests according to a total ordering relation on these unique identifiers. This process becomes non-trivial when requests can be submitted independently to different replicas and failures can occur.

Protocols to implement agreement and order have received considerable attention in the literature [Birm87, Chan84, Cris85, Cris88, Garc82, Garc88, Kope87, Lamp78, Marz84, Stro87]. In the next section, we survey these and other related protocols.

2.3 Fault-Tolerant Services

While different systems have implemented agreement and order in different ways, there are some general fault-tolerant services that facilitate implementing these requirements in a distributed system. These include a *time service*, *broadcast service*, *membership service*, *recovery service*, and *stable store service*. In this section, we give a survey of various implementation approaches for these services. An attempt has been made to unify these different approaches according to their basic functional behavior.

2.3.1 Time Service

One of the fundamental properties of a distributed system is the simultaneity or near-simultaneity of events caused by concurrent execution. While this property improves the efficiency of the system, it also complicates distributed processing. This complexity arises due to the fact that many distributed applications need to know the causal relationship among various events and the real-time at which these events occur. However, determining such relationships is complicated by two factors inherent in a distributed system: the variable and unknown delay in the communication network, and the variable and unknown drift of the clocks on different processors. The only way a processor can learn about an event at another processor is by message passing. Because of the first factor, the causal relationship among the events cannot be determined from the order in which the corresponding messages notifying the event are received. In fact, multiple events may be seen at different times and in different order at different processors. Similarly, due to the second factor, the local clock time of an event at one processor cannot be compared with the local clock time of another event at another processor complicating the determination of the real-time at which an event occurs and the causal relationship among various events. The *time service* in a distributed system provides the basis by which causal ordering among events and the real-time at which an event occurs can be determined.

A time service can be thought of as an abstract common clock. It may be built in one of two ways. In the first approach, local processor clocks are synchronized to implement a function that maps real-time t to clock time $\hat{C}_p(t)$ at every process p. This synchronization is performed at regular intervals in such a way that the clocks do not drift too far apart from each other. The time of an event at a process executing in processor P is then defined to be the value of P's clock at the point when the event occurs. The timing of events at different processors may be compared by allowing for the maximum difference by which the local clocks may differ before they are synchronized. The second approach derives the temporal order in which different events occur in the system without direct association to a hardware clock value. To do this, a logical clock is constructed that causally orders different events of the system. For any two events (say a and b), each timed by the logical

clock, exactly one of the following three relationships holds—event a occurred before event b, event a occurred after event b, or events a and b occurred at the same logical time.

Both of these approaches have certain advantages and disvantages. For example, logical clocks do not provide a mapping from the timing of an event to real time, whereas synchronized clocks may provide this mapping by synchronizing with an external time source. On the other hand, logical clocks provide causality among different events depending on what events have been seen by the processes when an event occurs. In particular, two events happen at the same time at different processes if neither process is aware of the other event. A synchronized clock coerces an order that depends on their local times; thus, the causality relation is lost in a distributed clock.

In the following, we discuss the details of how synchronized clocks and logical clocks are constructed in a distributed system.

2.3.1.1 Synchronized Clocks

Synchronized clocks are implemented by a technique called *clock synchronization*, in which the local clocks on different processors are synchronized periodically before they drift too far apart. There are two ways in which processor clocks may be synchronized. In the first, termed *internal clock synchronization*, the processor clocks are always kept within a certain maximum drift of one another. In the second, termed *external clock synchronization*, the processor clocks are always kept within certain maximum deviation from an external time reference. By definition, externally synchronized clocks are also internally synchronized. On the other hand, internally synchronized clocks may deviate arbitrarily from the external time reference. In the following, we introduce the salient features and algorithms of clock synchronization. In the following discussion, $C_p(t)$ denotes the local clock time at process p at real time t. This discussion mainly deals with the internal clock synchronization.

Properties of synchronized clocks

Synchronized clocks satisfy the following three properties.

1. *Monotonicity*: The clock is a monotonically increasing counter, that is

$$C_i(t+\tau) \geq C_i(t) \qquad \tau \geq 0$$

In general, since a clock increases by discrete values, it is possible that in a small real-time interval, τ, $C_i(t+\tau) = C_i(t)$. However, the granularity of most clocks is very small and for all practical reasons it is correct to assume that $C_i(t)$ is a strictly increasing function of t.

2. *Precision*: The synchronized clocks are always within some maximum deviation of each other. That is,

$$| C_i(t) - C_j(t) | < \beta$$

where β is the specified synchronization precision.

3. *Interval Preservation*: Also known as the *linear envelope*, this property states that any interval measured by the synchronized clocks is within some linear function of the real time interval:

$$(1-\rho)\tau \leq C_i(t+\tau) - C_i(t) \leq (1+\rho)\tau$$

Here, ρ is called the clock drift rate.

While β specifies the maximum allowed drift between any two clocks, ρ specifies the maximum allowed drift of a clock from real time. Thus, together ρ and β specify the interval in which the local clocks must resynchronize. Multiple local clocks that are synchronized so as to satisfy the above three properties can be thought of as a common abstract clock $\hat{C}$ that has the following property for all pairs of i and j

$$(1-R) < \frac{\hat{C}_i(t+\tau) - \hat{C}_j(t)}{\tau} < (1+R)$$

where R is the maximum allowed drift between any two synchronized clocks per unit time. This value is called the drift rate of the synchronized clock.

A synchronized clock may be used to measure intervals and to order various events in the system. One way to measure time intervals by is using a function *get_time_elapsed(t : time)* that returns the time elapsed since the clock showed time t. This function typically compensates for the changes in the clock value due to synchronization. In another approach [Halp84], the notion of a clock is not bound to specific hardware and a processor

may possess any number of clocks. In particular, every instance of clock synchronization logically gives rise to a new version of the clock. Here, the version of clock used to time an event is the most recent version at the time the event occured. Various events in the system can be ordered using the local clock time of the processor at which they occur.

Complexities of clock synchronization

One of the basic functions needed to synchronize clocks is the ability to read the value of a remote clock. This is done either through the exchange of messages (using some underlying communication network) or through special hardware that generates clock signals and propagates them to other processors. In either case, there is a random propagation delay introduced before a process receives the message or the signal. Thus, the time it takes to read a local clock or to set a local clock is not deterministic. This variation, along with the variable processing time for various messages received, introduces a random processing delay in the process of clock synchronization. The random propagation delays and the random processing delays limit the extent to which the clocks may be synchronized. The need to consider failures also complicates the algorithms, especially when the failures may be arbitrary.

A few results are known that put a limit on the closeness with which clocks may be synchronized. In [Lund84], the authors show that n clocks cannot be synchronized with certainty closer than $(1 - 1/n)(max - min)$ even in the absence of any failures. Here, max and min represent the maximum and minimum delay in message communication. Another result states that N clocks cannot be synchronized in the presence of more than $N/3$ Byzantine failures when no authentication scheme is used. However, clocks may be synchronized in presence of any number of Byzantine failures if an authentication scheme is used [Dole84]. Optimal algorithms for clock synchronization under different failure scenarios are also known [Srik87].

Algorithms for clock synchronization

The problem of clock synchronization has been studied extensively, and a large number of algorithms have appeared in the literature [Cris89, Halp84, Kope87, Lamp84, Lund84,

Srik87]. A survey of some of these algorithms appeared in [Rama90]. These algorithms differ from each other in their assumptions about the clock, and the network topology, as well as their failure hypothesis. The mechanics of clock synchronization involves exchanging messages containing local clock values and then computing a correction factor and applying it to the local clock. Since clocks drift apart from each other as time progresses, the whole process is repeated periodically.

Both hardware and software approaches have been taken to address the problem of clock synchronization. Software approaches tend to be more flexible but suffer from large clock skews. The lower limit on clock skews in such cases is the difference between the minimum and the maximum message transit time. In the hardware approach, special hardware is used to propagate signals between network nodes and to achieve synchronization. The hardware approaches provide a smaller clock skew, but are expensive and inflexible. Some of the algorithms use a combination of hardware and software, where the smaller skew of the hardware algorithms is sacrificed for a lower software cost.

All of the software approaches use a convergence function that guarantees the properties of monotonicity, precision and interval preservation. One class of algorithms consists of first exchanging local clock values and then applying a fault-tolerant averaging function to these values to compute a new clock value [Lamp84, Lund84]. Among fault-tolerant averaging functions used are egocentric average, fast convergence algorithm, fault-tolerant midpoint, and fault-tolerant average [Schn87]. These algorithms require a fully connected network, a known upper bound on message transit delay, and initial synchronization of the clocks.

In another class of algorithms, the clock values of various processors are first obtained through an agreement protocol that guarantees an agreement among all correct processors on a vector of values, one from each clock [Lamp84]. Each process then applies the same averaging function to compute a new clock value. The agreement process manages processor faults and ensures that all the processes apply the averaging function on the same set of values. In general, these algorithms do not require a fully connected network or initial synchronization of clocks. However, they do require a bound on message transit delays and a limit on the maximum number of processes that may fail.

A third class of algorithms use a synchronizer process to synchronize clocks [Halp84, Srik87]. To avoid problems caused by a single point of failure (*i.e.*, failure of the synchronizer), every process in the system attempts to become the synchronizer at roughly the same time, and at least one of them succeeds. To ensure that this happens at "roughly the same time", a protocol that guarantees an agreement on the expected time of next synchronization is used. These algorithms require a bound on the message time delays and initial synchronization of the clocks. The interconnection network need not be fully connected.

A probabilistic approach has been used in [Cris89], where a process reads a clock of another process with a given precision with probability as close to one as desired. When a process succeeds in reading the clock, it knows the actual reading precision achieved. This method of reading a remote clock can also be used to improve most of the algorithms described above. A master-slave arrangement, in which one clock acts as master and others as slaves, is used to synchronize the clocks here, where the slave clocks adjust their value according to the value of the master clock. In general, the algorithms to elect a new master are fairly complex.

All the algorithms described above make the assumption that the message transit times are bounded. In [Marz84], the author addresses unbounded message delays and use timeouts to detect the communication failures. Since message delays are unbounded, there is always a chance that false communication failures are detected.

2.3.1.2 Logical clocks

In [Lamp78], Lamport defines a *happened before* relation that can be used to define a clock in terms of the ordering of events in a distributed system. Specifically, given that a and b are events, a "happened before" b (denoted $a \rightarrow b$) if any of the following are satisfied.

1. a and b are events in the same process and a comes before b, or
2. a corresponds to the sending of a message and b corresponds to the receipt of the same message.

Furthermore, this relation is transitive, so that if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. If events a and b are such that $a \not\rightarrow b$ and $b \not\rightarrow a$, then they are said to be *concurrent.*

This relation is used to construct a logical clock C by assigning a value $C(a)$ to every event a in the distributed system. This value can be thought of as the logical time at which the event occurred. This assignment is done in such a way that the happened before relation is preserved, so that for any two events a and b, if $a \rightarrow b$ then $C(a) < C(b)$. This logical clock can then be used to order the various events in the system..

The algorithms proposed in the literature to implement logical clocks differ in the notations they use and the amount of information they convey through the clock values. In the solution proposed by Lamport [Lamp78], the system-wide clock C is implemented by a collection of individual clocks C_i for each process P_i; here, C_i is a function that assigns an integer $C_i(a)$ to every event a that happens in process P_i. The logical clock assigns an integer $C(a)$ to event a by using C_i, *i.e.*, $C(a) = C_i(a)$. Each process P_i implements C_i by maintaining a counter K_i which is incremented between successive events. Also, on receipt of a message, m, P_i sets K_i to the larger of the current value of K_i and a value greater than the logical clock time of the event corresponding to the sending of m. In this solution $C(a) < C(b)$ if $a \rightarrow b$. However, the converse is not true. As a result, given any two events, it is not always possible to determine if they are concurrent from their logical clock values using this approach.

This approach has been extended in [Fidg88, Matt89] to identify such concurrent events from their logical clock values. In this approach, the clock value is a vector of size n (sometimes called *version vector*), where n is the total number of processes in the system. Each entry i in this vector keeps a count of the messages received from process P_i. The update of the vector follows a similar procedure to that described above. Two vectors V_1 and V_2 can then be compared as follows :

$$V_1 < V_2 \quad if \quad \forall i,\ 0 \leq i < n,\ V_1[i] \leq V_2[i] \quad and \quad \exists j,\ 0 \leq j < n,\ V_1[j] < V_2[j]$$

Using this, two events a and b are concurrent if the corresponding logical times (say, vectors V_a and V_b respectively) satisfy the following :

$$V_a \not< V_b \quad and \quad V_b \not< V_a$$

A logical clock is also constructed in Psync [Pete89]. Here, the complete temporal order of events in the system is represented in the form of a graph called the *context graph*. A node in the graph represents an event and an edge represents the happened before relationship. For any two events a and b, there is a path from a to b in the graph if $a \rightarrow b$. The absence of a path between a and b implies that a and b are concurrent events.

2.3.2 Broadcast Service

To implement replica coordination in a replicated state machine, processes in the distributed system need to be able to communicate with one another. A broadcast service provides a mechanism by which a process in a group sends a message to every other process in the group. In general, such a service is useful in many kinds of distributed applications other than just replicated state machines. Examples include distributed database update and commit protocols, managing replicated data, distributed synchronization, and distributed transaction logging.

Properties of Broadcast Services

Many different broadcast services have been designed, with features depending on the requirements of the applications. In general, there are five different and independent properties that may be provided through a broadcast service.

1. *Multicast* : The message is disseminated to all the processes in the group. In a point-to-point network, this is achieved by sending a copy of the message to every process in the group separately. Local area networks such as ethernets and token rings provide a multicast primitive to send a message to multiple destinations. In such a case, this primitive may be used to implement this property.

2. *Atomicity* : The message is delivered either to all the correctly functioning processes in the group or to none. This property ensures that the information received (through broadcast) by every correct process in the group is identical.

3. *Reliability* : The message is delivered to every process in the group. If some process has failed, a mechanism is provided to deliver this message following recovery.

4. *Order* : Messages sent by different processes are delivered in some consistent order at all the group members. Possible consistent orders include:

a. *Partial order*: The messages are delivered in an order that preserves the happened before relation. Processes may receive concurrent messages in different orders, but a message is always delivered after all the messages that proceed this message in the happened before relation have been delivered. This is sometimes also called *causal ordering* [Birm87].

b. *Semantic dependent order*: Messages are delivered such that the application semantics are not violated. Typically, this ordering is a combination of other kinds of ordering depending on the semantics of the information carried in a message.

c. *Total order*: Messages are delivered in the same order to all the processes. If a message m_1 is delivered before m_2 at one process, m_1 is delivered before m_2 at every process.

d. *Total order preserving causality*: Messages are delivered in the same order at all the processes and this order preserves the happened before relation.

These orderings become more and more restrictive as we go down this list and, in general, more expensive to implement. As a result, the ordering used by an application should be the least restrictive that is sufficient to preserve the correctness of the application.

5. *Termination*: Every message is delivered to all correct processes in the group within a known time interval. This property can be satisfied only if the communication network is synchronous.

Examples of Broadcast Services

The various broadcast services that have been developed differ in which of the above properties they provide. A large number of broadcast services, typically called an *atomic broadcast* service, provide atomicity and total order. Examples include [Cris85, Birm87, Pete89, Chan84, Mell89, Nava88, Kaas89, Veri89]. The total order provided by [Pete89, Veri89] also preserve causality while the services provided by [Cris85, Veri89] also include

the termination property. An atomic broadcast service is useful in many distributed agreement applications such as propagating updates to manage replicated data and committing distributed transactions.

The broadcast service proposed in [Garc91] preserves both atomicity and reliability, but not necessarily order. This service, sometimes called a *reliable broadcast service*, is useful in applications that need fast delivery of messages where the order of delivery is not critical. Examples include managing highly available replicated databases and some real-time applications.

The broadcast services proposed in [Birm87, Pete89] provide atomicity and partial order. These services are useful in cases where concurrent events may be executed in different order at different processes. Moreover, using these services, it is possible to construct more restrictive broadcast services. An example of this is found in Chapter 4, where a broadcast is described that provides atomicity and a semantic dependent ordering based on the commutativity of the operations.

Algorithms

The algorithms used to implement broadcast services are typically complex due to the uncertain nature of the communication network and the possibility of processor failures. In particular, messages may be lost or corrupted on the communication channel or may be received in different order at different processors, while processors may fail in different modes.[1] As a result, the two main problems that are encountered in designing such algorithms—how to order messages and how to make the broadcast atomic—must deal with these situations. The way in which this is done is also influenced by the assumptions made about the topology of the network and the types of failures the network and processors can undergo.

In [Cris85], synchronized clocks are used to order different messages. Each message includes the clock time at which it was sent and the messages are ordered according to this time. The clocks may be synchronized by one of the methods mentioned in the

[1] In fact, it has been shown that it is impossible to reach agreement in a completely asynchronous distributed system in the presence of a single processor failure [Fisc85]. Despite appearances, the algorithms proposed in the literature are consistent with this result since they restrict the asynchrony of the network, typically by putting a bound on the message transmission delay.

previous section. The message is delivered at time $t + \Delta$, where t is the time when the message was sent and Δ is a constant that depends on the network properties. Atomicity is achieved by diffusing every incoming message onto every outgoing link and treating non-receipt of a message at time $t + \Delta$ as a failure. With this approach, a family of broadcast protocols that tolerate increasingly general fault classes—omission, timing and Byzantine—is constructed. All these protocols assume a point-to-point communication network.

Algorithms proposed in [Lamp78, Mell89, Pete89, Birm87] use logical clocks to implement order. The atomicity is achieved by either positive acknowledgement, where every receiver sends an acknowledgement for every message received [Birm87], or by negative acknowledgement, where a retransmission is requested by the receiver on detecting a missing message [Pete89, Mell89]. All these algorithms assume a point-to-point communication network and a crash failure model.

Another approach employs a single process to order messages [Chan84, Garc91, Nava88, Kaas89]. In this approach, every broadcast message is first sent to one process, called the *funnel process*, that puts a sequence number on the message and then resends it to all the processes in the group. The messages are then delivered in an order corresponding to the sequence number. This approach provides only for a total ordering among the messages exchanged in the system. There are also two other disadvantages to this approach. First, the funnel process is a single point of failure and the protocols must provide a way to recover from this failure, something that can be very complicated. Second, the funnel process is potentially a performance bottleneck since it must process every broadcast message. The atomicity in this approach is achieved by positive acknowledgement [Nava88], negative acknowledgement [Kaas89], or a combination of positive and negative acknowledgement [Chan84]. Once again, crash failures and a point-to-point communication network are assumed.

2.3.3 Membership Service

To ensure consistent action, a group of cooperating processes typically needs to have an agreement on the set of functioning members at any moment in time. Changes in group

membership may occur due to the failure of processes, the recovery of previously failed processes, new processes joining the group, or a process voluntarily leaving the group. A membership service is used to maintain such a consistent, system-wide view of which processes are functioning at any given moment. This service has proved to be one of the most fundamental services in fault-tolerant distributed systems, simplifying many problems.

There are actually two types of membership services, each serving a different purpose [Veri90]. The first can be viewed as a user-level service that typically translates the failure or recovery confirmation into an event that is then ordered with respect to other events in the system. This ordering is then made available to the application to use in making decisions. Examples of this kind of service include [Birm87, Chan84, Cris88, Kope89b]. In this case, the application program is explicitly notified of the changes in the group membership.

The other type of membership service is sometimes called a *monitor service* [Veri90]. In contrast to the user-level orientation of the first type, the monitor service is used by the system itself to maintain a consistent view of which processes are functioning and hence participating in system decisions. For example, such information is used in reliable multicast protocols to determine when a message has been received and acknowledged by every functioning process so that it can be committed to the application. The processor failure or recovery event must again be consistently ordered with respect to other events such as interprocess communication to guarantee that messages are committed consistently, but the failure notification is not necessarily passed on to the application. Examples of this kind of protocol include [Veri90, Mish91].

Correctness

Intuitively, an algorithm solves the membership problem if it ensures that the replicated processes using this service remain consistent in the presence of failures and recoveries. Although this implies that the solution to the membership problem is application-dependent, there are solutions that are general enough to ensure the correctness of any distributed application. Typically, such a solution enforces agreement among all the processors on a

unique sequence of process joins and departures, and the precise points at which these membership changes occur. A large number of membership services proposed in the literature satisfy this condition [Chan84, Birm87, Cris88, Kope89b, Ricc91]. However, such a condition may actually be overly restrictive for many applications. The membership protocol described in Chapter 5 is much less restrictive in this regard. Here, an *sf-group* at process P is defined to be the set of all the processes that have failed simultaneously as perceived by the process P. The proposed solution ensures that all the processes in an sf-group are removed simultaneously and the order of removal of these sf-groups is same at all the processes, but the points at which at which these changes occur need not be same at all processes.

There are some critical applications, such as process control, in which the membership service must also satisfy the *timeliness* property. This property states that, once initiated, the membership service is guaranteed to terminate in a known real time interval. This property is typically satisfied by membership services built on top of synchronous systems [Cris88, Kope89b]. The membership protocols in asynchronous systems do not satisfy the timeliness property.

Protocol Invocation

As mentioned above, changes in membership occur when a process fails or recovers. Thus, the membership protocol is initiated when a process is suspected to have failed or when a functioning process learns about the recovery of a previously failed processor. The technique used to detect the failure varies from system to system depending on the system model used. Typically, a failure of a process P is suspected when no messages from P arrive in a given interval of time. This *failure detection* mechanism is typically implemented by a *heartbeat* protocol where every functioning membership of the group periodically sends "I am alive" messages. Examples of protocols using such a mechanism include [Cris88, Birm87, Kope89b]. The failure detection protocol can also be application dependent, where the application messages being exchanged are monitored and a failure is suspected when a message expected by the application fails to arrive within certain interval of time [Mish91]. For recovery, notification is typically asynchronous: the recovering process

informs the other members of the group about its recovery and then the membership protocol is initiated. These mechanisms may also be used to detect failures or recoveries while the membership protocol itself is in progress, thus allowing simultaneous failures and recoveries to be handled.

Network Partitions

A network partition occurs when a subset of processes in the group cannot communicate with another subset due to a failure. In such a case, processes in each subset may conclude that all the processes in the other subset have failed. Some of the membership protocols proposed for asynchronous systems can tolerate network partitions by allowing a subset with a clear majority of processes to continue functioning [Chan84, Ricc91]. However, the protocols proposed for synchronous systems cannot tolerate a network partition, since this may lead to divergent views among different processors. There are known techniques to reconcile divergent views [Stro87], but inconsistent actions may be taken while the reconciliation protocol is in progress.

Algorithms

Few algorithms have been proposed in the literature to solve the membership problem. In [Cris88, Kope89b, Ezhi90], the authors have proposed solutions to the membership problem in synchronous systems with a broadcast communication network. The algorithm proposed in [Cris88] relies on an atomic broadcast service and a message diffusion service; periodically, each process affirms its existence by sending a *present* message. In [Kope89b], global time is used to control the access to the communication channel by a synchronous TDMA strategy; with every message broadcast, a process includes certain membership information that is used by all the processes to compute group membership.

Membership algorithms for asynchronous systems are inherently more complex since it is impossible to distinguish a failed process from one that is merely slow. These protocols essentially assume that a process that does not respond for a given time interval has failed [Birm87, Chan84, Ricc91]. Typically, these protocols make use of acknowledgements and message retransmission. A completely connected network with FIFO channels is required

in the algorithm proposed in [Ricc91]. A distinct manager process is used to coordinate updates to the other processes' local views. A two phase protocol is used by the manager to coordinate updates and a three phase protocol is used to select a new coordinator when the manager is thought to have failed. The protocol proposed in [Chan84] also makes use of a distinct manager process. In this approach, all the normal traffic is suspended while the protocol is in progress. The protocol is three phase for the manager process and two phase for other processes.

2.3.4 Recovery Service

In a distributed system where a group of processes cooperate to accomplish a task, every process maintains some private state and communicates with other processes in the group by exchanging messages. The state of such a process is characterized by a sequence of events—an event being a computation that does not require any message exchange, or the sending or receipt of a message. The state of a process after receiving a message, say m, becomes dependent on the state the sender had just before it sent m. Thus, as a result of the message exchanges in the system, states of all the processes become dependent on one another. The *system state* is then a history of events that constitute the set of all the process states. A system state is consistent if for every event corresponding to the receipt of a message in the state, the event corresponding to the sending of that message is also included [John90].

In a system liable to failures, processes may fail and recover as the system progresses. When a process fails, some or all of its state is usually lost. As a result, when it recovers, its state must be reconstructed and the states of the other processes potentially modified so that the system state remains consistent. For example, consider a scenario in which a process fails after sending a message that is received by other processes. Due to this failure, the event corresponding to the sending of this message may be lost and hence, may not be in the process state at the time of recovery. If the event corresponding to the receipt of that message remains in the other processes' states, the resulting system state will be inconsistent. A recovery service is used to deal with this type of problem by ensuring that the state of the system remains consistent following recovery of one or more

processes.

Checkpointing and Rollback Recovery

Checkpointing and rollback recovery is one way to restore a consistent system state after a failed process recovers. This technique makes use of *stable storage*, which is storage that survives crashes in such a way that the data previously saved there can be accessed on recovery. In this approach, the processes periodically save their states as a *checkpoint* on the stable store during their execution. Recovery involves rolling back the processes to the most recent combination of saved states such that the system state remains consistent.

There are two approaches to creating these checkpoints. In the first approach, each process periodically takes a checkpoint independent of the other processes. Upon recovery, the processes must find a set of checkpoints, one from each process, such that the system state constructed out of these checkpoints is consistent. In this approach, no coordination between the processes is required while taking a checkpoint but processes must coordinate during recovery. One of the drawbacks of this approach is that the rollback of a process may result in a cascade of rollbacks that, in the worst case, can push all processes back to their starting states. This is called the *domino effect* [Rand75, Russ80]. Moreover, since cascading rollbacks may require any of the previously stored checkpoints, the processes must retain all of their checkpoints indefinitely.

This independent checkpointing approach is used in a variety of contexts [Bhar88, Hadz82, Kim78, Kim86, Ng88, Rama88, Stro85]. The scheme proposed in [Hadz82] is limited to a centralized database, while the ones proposed in [Kim78, Kim86] rely on an intelligent underlying processor system to automatically establish checkpoints of the coordinating processes. In the scheme proposed in [Bhar88], a recovering process computes the set of globally consistent checkpoints by invoking a two phase rollback algorithm. In the first phase it collects the information about relevant message exchanges in the system and uses it in the second phase to determine both the set of processes that must roll back and the set of checkpoints up to which rollback must occur. In [Ng88], the authors propose a commit protocol for checkpointing distributed transactions. Although the domino effect is possible here, it is shown that the lost work can be reduced by reusing

portions of completed computations. In [Rama88], synchronized clocks have been used for checkpointing and rollback recovery. These clocks coupled with the idea of a pseudo-recovery block approach [Shin84] are used to develop a checkpointing algorithm.

In the other main approach, processes coordinate with each other to take a checkpoint [Bari83, Koo87, Leu89, Tami84]. Typically, the processes use a two-phase commit protocol to take a checkpoint, thus ensuring that the set of checkpoints stored is consistent. In this scheme, two checkpoints need to be stored at any time: a permanent checkpoint that cannot be undone and a tentative checkpoint that can be undone or changed to a permanent checkpoint. Note that even with the coordinated checkpointing, there is a need for some synchronization. In the absence of such synchronization, processes cannot all restore their checkpoints simultaneously and *livelocks* can be introduced [Koo87]. To avoid this, the recovery is again done in two phases. In the first phase, a request to restart from a checkpoint is sent. In the second phase, a decision to restart is propagated.

Message Logging

Independent checkpointing can further be enhanced by the use of message logging in a technique sometimes called *optimistic recovery* [John90, Sist89, Stro85]. In these schemes, processes take checkpoints independently and log input messages along with some dependency information in stable storage. Recovery then consists of (a) restoring an earlier possible state of the failed process using a checkpoint from the stable store plus potentially replaying the logged messages, (b) recognizing the set of processes whose states depend on lost states using the dependency information and rolling them back, and (c) committing messages to the outside when it is known that the states that generated the messages will never need to be undone. The logging of messages can also be done on volatile storage as has been shown in [John87, Pete89, Stro85]. In this case, messages are logged on the volatile storage of other processes and then replayed to the recovering process at the time of recovery.

2.3.5 Stable Store Service

As described above, a stable store is an abstraction of storage that survives processor failures. The operations provided by a stable store vary depending on its complexity. The basic operations, applicable on variables, are *read* and *write*. An important property of these operations is *atomicity*, *i.e.*, once invoked, they execute completely or not at all [Lamp81]. The granularity of the variable on which these operations are applied can be a single bit, a simple variable like an integer, or a more complicated structure. Another operation that is typically provided is a mapping from the logical address of a variable to its physical address. With this operation, an application can use only logical addresses of variables stored and need not worry about their actual physical location. Some stable storage also maintains the size or number of variables stored. This is useful to the application when the size or number of variables stored changes frequently.

The abstraction of stable storage can be implemented in a variety of ways depending on the needs of the application. It can be as simple as a non-volatile storage, such as a disk, that supports read and write operations. Or, it can be implemented by replicating the information on multiple processors with independent failure behavior [Cris85]. In this way, the hope is that one replica will always be available despite failures. Such an implementation of the stable storage follows the state machine approach, and hence, the consistency of replication is maintained by using the services described earlier in this section. Indeed, the state machine approach can be viewed as an implementation of stable storage.

2.4 Fault-Tolerant Systems

2.4.1 Isis

Isis is a distributed programming environment developed at Cornell University that provides tools for building fault-tolerant applications [Birm91b]. This support includes two key aspects: *virtually synchronous process groups* and *group communication*. In a virtually synchronous environment, routines can be programmed as if distributed actions were performed instantaneously and in lock-step even though the physical realization is

concurrent.

Support for group communication includes a collection of reliable multicast protocols such as ABCAST (total order multicast) and CBCAST (partial order multicast). CBCAST is the basis of all the multicast protocols in the system; it also provides for the partial ordering of messages even among multiple groups that have overlapping memberships. Reliability in Isis encompasses failure atomicity, delivery ordering guarantees, and a form of group addressing atomicity in which membership changes are synchronized with group communication.

Four different kinds of groups are supported by Isis: peer groups, client/server groups, diffusion groups and hierarchical groups [Birm91a]. In a peer group, processes cooperate as equals in order to get a task done. In a client/server group, a peer group of processes act as servers on behalf of a potentially large set of clients, where the clients interact with the servers in a request/reply style. Note that in this type of group, clients do not receive multicast messages from the servers. A diffusion group is a client/server group in which a broadcast message from a server is received by all the servers and the clients. Hierarchical groups are tree-structured sets of groups that arise when large server groups are needed in a system.

2.4.2 Advanced Automation System

The Advanced Automation System (AAS) is a distributed, real-time system under development by IBM to replace the present en-route and terminal approach U.S. air traffic control computer systems [Cris90]. High availability of the air traffic control services is an essential requirement of the system. To achieve this requirement, design techniques that enable the system to automatically tolerate multiple concurrent failures are being used.

AAS is based on the server/service and depends on relation model defined in Section 2.2.1. Services at each layer mask a certain number of failures and propagate others to the higher layers. Redundancy is employed at the hardware level as well as software level by replicating individual components. Local processor clocks are synchronized to within a certain precision by a fault-tolerant clock synchronization algorithm. All of the protocols in the system, including atomic multicast and membership, are synchronous in nature.

The dependency structure of the fault-tolerant services starts with the clock synchronization service at the lowest layer. The atomic multicast service then uses the clock synchronization service, which is in turn used by the membership service.

2.4.3 MARS

MARS (MAintainable Real-time System) is a system being developed at Institut für Technische Informatik, Austria that is designed for distributed real-time process control applications [Kope85, Kope89a]. Its primary application area is industrial systems (*e.g.*, a rolling mill, railway control systems) where hard deadlines are imposed by the controlled environment. The design goal is not only high performance, but also system behavior that is deterministic and predictable even under peak load. The operating system ensures that the timing constraints specified during the design are met at run-time.

MARS uses a transaction model to describe the activities of a real-time system. The clocks of different processors of the distributed system are synchronized and all the protocols in the system are clock-driven. Active redundancy is used to provide for fault tolerance, *i.e.*, every component in the system is a self-checking component implemented by two components with a comparison of results. These components communicate via (unreliable) messages, with a higher level protocol providing for reliable broadcast of messages to groups of components. These components are connected using a synchronous real-time bus to form a cluster. A TDMA strategy is used to control the access to the bus ensuring collision-free access. A MARS system is typically configured as a set of clusters with a high degree of interconnectivity.

The dependency structure of the fault-tolerant services starts with the basic clock synchronization service at the lowest level. A membership service is then implemented based on the availability of synchronized clocks. An atomic multicast service, a remote action monitoring service, and an improved clock synchronization service form the higher layers.

2.4.4 DELTA-4

The Delta-4 project seeks to define a dependable distributed, real-time operating system that allows integration of heterogeneous computing elements [Powe88]. In Delta-4, the basic units of fault tolerance are the nodes or host computers. Replication of individual software components on different host computers provides the redundancy needed for fault tolerance. The fault-tolerance techniques are user-transparent.

The host computers of the distributed systems communicate through a dependable communication system called MCS (Multicast Communication System), which provides atomic multicast capabilities. Intelligent network controllers (NAC), which are fail-silent, are used to connect the host to the token bus, giving the illusion of synchronous channels. An interesting feature of this system is that the communication protocols are synchronous in nature even though the clocks are not synchronized.

2.4.5 Arjuna

Arjuna is an object-oriented programming system, being developed at the University of Newcastle upon Tyne, that provides a set of tools for the construction of fault-tolerant, distributed applications [Shri89]. The objects in Arjuna are persistent, *i.e.*, they are long-lived entities surviving failures, and are manipulated using atomic actions (also known as atomic transactions). Arjuna provides *nested atomic actions* that maintains the integrity of the objects (and hence the integrity of the system) in the presence of failures such as node crashes and message loss.

In Arjuna, objects are abstract data types. In a quiescent state, an object is *passive*. An object becomes *active* when one of its operations is invoked from within an atomic action. The object then remains active until the atomic action commits or aborts. In the case of nested atomic actions, the object remains active until the outermost action either commits or aborts, or any of the enclosing actions abort. Operations on remote objects are invoked using a remote procedure call mechanism called *Rajdoot* [Panz88].

Arjuna provides a number of integrated mechanisms such as naming, locating and invoking operations on objects, concurrency control, recovery control, and managing object states for long term as well as short term storage. A prototype version has been imple-

mented in C++ to run on a collection of Unix [2] workstations connected by a local area network [Lipp72].

2.5 Conclusion

From this survey of implementation techniques for fault-tolerant systems, we observe that the performance and dependability of these systems depends critically on the performance of the protocols that implement various fault-tolerant services. In this dissertation, we provide techniques to improve the performance of the protocols and to satisfy diverse needs of various applications with little overhead. The protocols proposed in Consul are more efficient than those previously proposed because they are based on the partial order of messages exchanged as opposed to the total order. Consul also satisfies the diverse needs of various applications by providing a flexible architecture in which a user can build an optimal system by selecting various protocols. We describe these techniques in the following chapters.

[2] Unix is a trademark of AT&T Bell Laboratories

CHAPTER 3

System Architecture

In this dissertation, we describe Consul, a communication substrate that provides fault-tolerant services for distributed programs written using the state machine approach. The general aim of this substrate is to ensure *replica coordination* in the presence of failures. To do this in a convenient manner, Consul is designed to realize three important objectives. The first is to provide support for interprocess communication and for different kinds of consistent orderings among the messages exchanged in the system. The former is needed to allow coordination among replicas, while the latter simplifies maintaining consistency of the application. The second objective is to provide support for recovering from failures and for continued processing in the presence of failures. This includes support for replication, making message delivery reliable, failure detection capability, state restoration capability, and stable storage. The third objective is to have an architecture flexible enough to satisfy the diverse requirements of many different applications. The architecture should be such that an application pays for only the functionality it needs, yet is still suitable for many different kinds of applications.

This chapter describes our system model and the overall architecture of Consul. In doing the latter, we give a brief description of each of the modules that constitute the communication substrate. We also provide an overview of Psync [Pete89], the interprocess communication protocol that forms the basis of the substrate.

3.1 Substrate Architecture

We assume a distributed system in which multiple processors are connected by a communication network. There is no shared memory or common physical clock. Processes communicate by exchanging messages through the communication network, which is assumed to be asynchronous, *i.e.*, there is no bound on the transmission delay for a message

between any two machines. Processors in this system fail by crashing, *i.e.*, they fail silently without making any incorrect state transition; a process is said to fail if the processor on which it is executing fails. Messages may be lost, but it is assumed that they are never corrupted. We do not assume any broadcast capability in the communication network, but this capability is exploited if available.

The overall system architecture is shown in Figure 3.1. A copy of the communication substrate on each processor provides the interface between the application processes and the communication network. These copies interact with each other to provide the relevant fault-tolerant services required for the state machine approach.

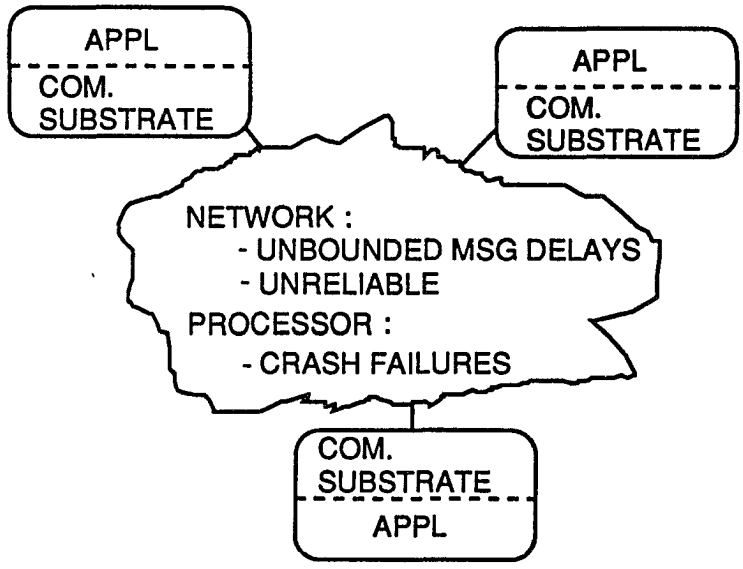

Figure 3.1: Overall System Architecture

Each fault-tolerant service is implemented in Consul by one or more actual protocols. Each of these protocols implements one function, such as reliable multicast with different kinds of message ordering, stable storage, failure detection, and so on. Besides these, two configuration protocols, called the divider and the (re)start protocols, are also included

in Consul; these are necessary to configure the system according to the application's requirements.

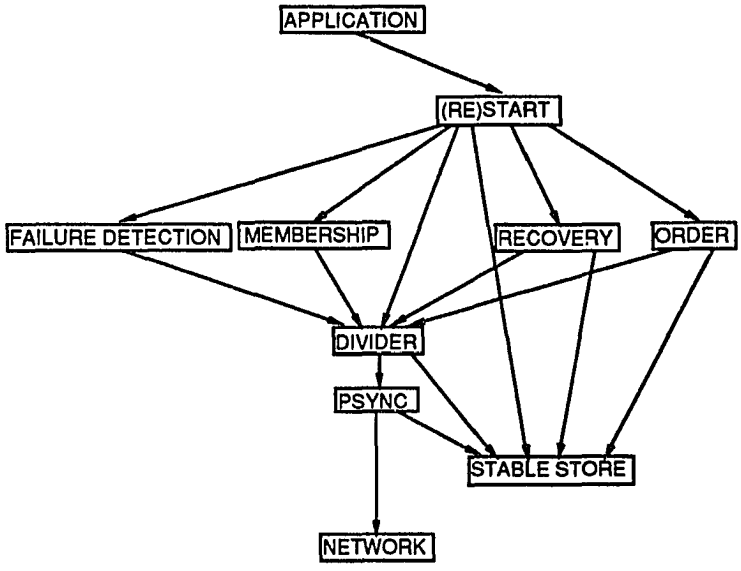

Figure 3.2: Communication Substrate

Figure 3.2 illustrates the detailed architecture of the protocols in Consul. In this figure, there is an arrow from protocol u to protocol v if u depends on v in the sense of Section 2.2.1. At the base of the substrate is the stable store protocol. This protocol provides a storage facility that survives processor crashes that is used by all other protocols of the substrate to periodically checkpoint their states. This protocol can also be used by the application to maintain its state, if desired. The stable store protocol provides operations to read and write variables in the stable store, as well as operations to write a group of variables.

As mentioned above, Psync is the main communication mechanism in Consul [Pete89]. It provides a group-oriented interprocess communication mechanism in the form of a multicast facility that maintains the partial order among the messages exchanged in the system. A detailed description of Psync can be found in Section 3.2.

The order protocol is chosen from a suite of different and independent protocols, each

providing a different kind of message ordering. Psync itself provides partial ordering among the messages. Based on this partial order, two other kinds of orderings have been constructed: a total order preserving causality and a semantic dependent order. The semantic dependent order takes advantage of the commutativity of operations to provide an ordering that is less restrictive than total ordering, while still preserving the correctness of the application.

The failure detection and membership protocols deal with process failures. The failure detection protocol is used to monitor processes for failures. It does this based on message traffic, *i.e.*, if no message is received from some process in a given interval of time, its failure is suspected. The membership protocol maintains a consistent system-wide view of which processes are functioning and which have failed at any point in time. It does this by establishing an agreement among correct processes concerning the failure of a process that is suspected to be down. Similarly, when a previously failed process recovers, this protocol consistently incorporates the process into the system.

The recovery protocol comes into play when a previously failed process recovers. Specifically, it deals with restoring the state of the recovering process to the current state, and consistently incorporating the process into the group. The recovery protocol makes use of the checkpoints stored by different protocols as well as the message retransmission mechanism of Psync.

The (re)start and divider protocols are configuration protocols, *i.e.*, they aid the user in building a system according to the requirements of the application. The (re)start protocol establishes a connection among various protocols needed by an application for proper communication. Once these connections are established, the (re)start protocol remains quiescent, and is not involved in the normal functioning of the system thereafter. It becomes active again at the time of recovery and reestablishes the connections that were lost when a failure occurred. The divider protocol is a demultiplexing protocol that directs messages in the system to the appropriate protocols. The rationale for these protocols is explained further in Chapter 6.

3.2 Interprocess Communication Support

General support for interprocess communication in Consul is provided by a protocol called Psync [Pete89]. This protocol provides a *conversation* abstraction through which a collection of processes exchange messages. The general form of the conversation is defined by a directed acyclic graph that preserves the partial order of the exchanged messages. This section gives an operational overview of Psync. It first defines the basic operations for sending and receiving messages, and then it outlines how Psync operates in the presence of network and processor failures.

3.2.1 Basic Operation of Psync

Processes begin a conversation with one of these two operations.

```
conv = active_open(participant_set)
conv = passive_open()
```

The first operation actively begins a conversation with the specified set of participants. This set is also called the *membership list.* The second operation passively begins a conversation; the invoking process is blocked until some active process starts a conversation that contains the invoking process in its participant set. The conv returned by the two operations serves as the process' handle on the conversation.

Once a process possesses a conv handle, it can send and receive messages using the operations

```
node = send(msg, conv)
node, msg = receive(conv)
```

where msg is an actual message—an untyped block of data—and node is a handle or capability for that message. Each participant is able to receive all the messages sent by the other participants in the conversation but it does not receive the messages it has sent. Fundamentally, each process sends a message in the *context* of those messages it has already sent or received. Informally, "in the context of" defines a relation among the messages exchanged through the conversation. This relation is represented in the form of

a direct acyclic graph, called a *context graph*. Each participant has a *view* of the context graph that corresponds to those messages it has sent or received. The semantics of `send` and `receive` are defined in terms of the context graph and a participant's view.

Figure 3.3 gives a sample context graph for a conversation in which m_1 was the initial message of the conversation; m_2 and m_3 were sent by processes that had received m_1, but independent of each other; and m_4 was sent by a process that had received m_1 and m_3, but not m_2. Note that messages, neither of which are in the context of the other are said to have been sent at the *same logical time*. For example, m_2 and m_3 were sent at the same logical time.

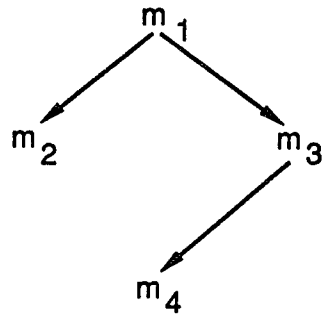

Figure 3.3: Example Context Graph

The context graph contains information about which processes have received what messages. In particular, receipt of a message implies that the sender has seen all its predecessor messages. Thus, if some message m is followed in the context graph by a message from all the participants except for m's sender, then m has necessarily been seen by all participants. Formally, message m_p sent by process p is said to be *stable* if for each participant $q \neq p$, there exists vertex m_q sent by q in the context graph, such that m_p proceeds m_q. Because for a message to be stable implies that all processes other than the sender have received it, it follows that all future messages sent to the conversation must be in the context of the stable message; *i.e.*, they cannot precede or be at the same logical time as the stable message. Note that for a node to be stable is analogous to the message being *fully acknowledged* [Schn82].

For example, suppose the context graph depicted in Figure 3.4 is associated with a conversation that has three participants, denoted a, b and c, where $a_1, a_2, \ldots$ denotes the sequence of messages sent by process a, and so on. Messages a_1, b_1, and c_1 are the only stable messages. Also, participant a has sent two unstable messages: a_2 and a_3. a_2 is unstable because it is not followed by any message from process c.

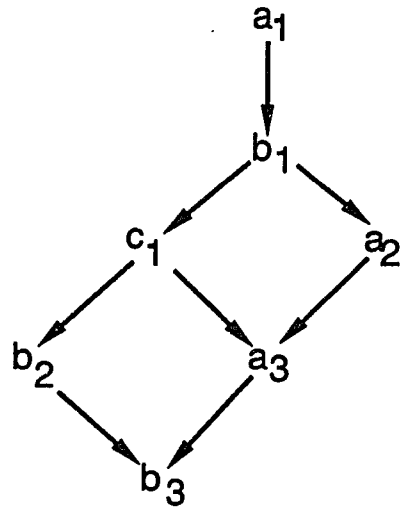

Figure 3.4: Another Example Context Graph

Because the context graph provides such useful information, the conversation abstraction provides the following set of operations for traversing the context graph and querying the state of various nodes in the context graph.

`node = root(conv)`: return the root node.

`node_set = leaves(conv)`: return the set of leaf nodes.

`process_id = sender(node)`: return the node's sender.

`node_set = next(node)`: return the set of nodes to which there is an edge from node.

`node_set = prev(node)`: return the set of nodes from which there is an edge to node.

`outstanding(conv)`: return true if all messages have been received.

`precedes(node1, node2, conv)`: return true if there is a path from $node_1$ to $node_2$.

`stable(node, conv)`: return true if node is stable.

`num = unstable(conv)`: return the number of unstable messages sent by the process.

`participant_set = participant(conv)`: return the set of participating processes.

Psync maintains a copy of a conversation's context graph at each processor on which a participant in the conversation resides. Each process receives messages from this local copy of the context graph, which is called the *image*. Each time a process at one processor sends a message, Psync propagates a copy of the message to each of these processors. This propagated message contains the ids of all the messages upon which the new message depends, *i.e.*, it identifies the nodes to which the new message is to be attached to the context graph.

3.2.2 Fault-Tolerance Aspects

Implementing conversations in a distributed environment is in practice complicated by three factors: the underlying network fails to deliver messages, processors fail, and processor failures are indistinguishable from both network partitions and processors that are slow in responding [Schl90]. This section describes aspects of the basic protocol that are included to account for these factors. It also describes facilities basic fault-tolerant primitives that are designed to support Psync applications such as the other protocols in the communication substrate.

3.2.2.1 Transient Network Failures

Consider the possibility of transient network failures. Such failures imply that for a given message sent from one processor to another, zero or more copies of the message are delivered to the destination processor. For the purpose of this discussion, assume processors do not fail.

Psync places any message received out-of-order in a holding queue until all messages upon which it depends arrive. Let m be a message sent by a participant on processor h in the context of m', and let h' be a processor that receives m but has not yet received m'; *i.e.*, m is placed in the holding queue on h'. Psync associates a timer with each message in the holding queue. When the timer for message m expires, a request to retransmit m' is

sent to h. That processor is guaranteed to have m' in its image because a local participant just sent a message in the context of m'. This is true even if the participant that originally sent m' does not reside on h. Because it is possible that the predecessors' predecessors are also missing, the retransmission request identifies the subgraph of G that needs to be retransmitted, not just the message(s) known to be missing. When a processor receives a retransmission request, it responds by resending all messages in the subgraph.

3.2.2.2 Last ACK Problem

Although Psync automatically recovers from missing messages upon which some other message depends, it is possible for the last message sent—*i.e.*, a message upon which no messages depend—to be lost. We characterize this as an instance of a general "last ACK problem" faced by many protocols. To help applications accommodate this possibility, Psync is augmented to allow its blocking operations—**passive_open** and **receive**—to include a timeout argument. The return code then indicates whether the operation was successful or the timeout expired. Processes use a timeout larger than the maximum communication delay to and from all participating processors.

In addition, Psync provides a **resend(node)** operation. Applying this operation to a node causes an exact duplicate of the corresponding message to be sent to all processors maintaining an image of G. The resent version of the message is identical to the original copy of the message except that it is flagged as having been resent. Should a processor that receives a resent message already have a copy of the message, it (1) discards the duplicate copy, and (2) resends all the messages in its image that are immediate successors of the duplicate message. Finally, should a participant apply **resend** to a stable message, Psync does nothing; *i.e.*, it does *not* resend the message as instructed. This is because resending a stable message is unnecessary: by definition, a stable message has been delivered to all participants and a reply has been received from all participants.

3.2.2.3 Processor Failures

Psync guarantees two things about the context graph in the presence of processor failures:

- All running processes are able to continue exchanging messages.
- A message contained in any running processor's image will eventually be incorporated into every running processor's image if processor failures are infrequent.

The first condition is easy to guarantee because each process depends only on the local state of the conversation. Thus, a participant can successfully invoke **send** because being able to send a message depends only on the leaves of the participant's view. Also, a participant's ability to successfully **receive** messages sent by another running process depends only on the processor's ability to incorporate new messages into the local image. The processor, in turn, can always incorporate messages received from another running processor into its image because the only prerequisite for doing so is that all the predecessor messages be present. Should some of the predecessor messages not be present, the receiving processor can retrieve them from the sending processor. The sending processor is guaranteed to have all the preceding messages because it just sent a message that depends on them.

The key to satisfying the second condition is to correctly deal with a processor failing after it has sent a message. Psync addresses this problem with the following extension to the retransmission request strategy defined above: When a processor does not receive a response to a retransmission request message that it sent to a particular processor, it broadcasts the message to all the processors. Should the broadcast message fail to yield the missing message, the message that triggered the retransmission request is discarded. Given this extension to the protocol, consider how the second condition is satisfied for two different quantifications of "infrequent".

First, assume a single processor failure. Without loss of generality, suppose processor h fails immediately after sending message m in the context of message m'. There are three cases to consider.

- *Case 1:* No other processor receives m. Message m does not appear in any running processor's image.
- *Case 2:* All processors receive m.

  - *Subcase a:* No processor has m' in its image; thus, the broadcast retransmission request fails. Neither messages m' nor m appear in any processor's image. Note that m' must have been sent from processor h, otherwise at least one running processor (the sending processor) would have a copy of it.
  - *Subcase b:* All processors have m' in their image. Message m can be successfully incorporated in each processor's image.
  - *Subcase c:* Some processors have m' in their image. Broadcasting the retransmission request will retrieve m' and both m and m' will be incorporated into each processor's image.

- *Case 3:* Some processors receive m. A processor that receives m incorporates it into its image as in case 2. A processor that does not receive m will at some future time receive message m'' in the context of m, causing the processor to retrieve m from the processor that sent m''.

Thus, the same set of messages are incorporated into all images when a single processor fails.

Second, suppose there are multiple processor failures. Psync continues to incorporate messages into all images unless there are "too many" failures, where "too many" is quantified as follows. A message m is defined to be *n-stable* if n-1 processes other than the sender of m have sent a message in the context of m. For a message to be n-stable implies that a copy of m is contained in at least n images, assuming a one-to-one correspondence between images and processes. For a message that is n-stable, n processor failures are "too many" failures. Thus, a copy of m can be retrieved from some image in the presence of up to n-1 processor failures. A message that is stable is contained in all images.

Note that the preceding discussion does not assume perfect knowledge of when a particular processor has failed, *i.e.*, it can be implemented using a simple timeout and retry strategy. In the worst case, a given processor might decide that another processor is down when it is not, but this does not affect the correctness of the protocol. For example, suppose a processor that receives m incorrectly decides that h is down. Broadcasting the retransmission request is wasteful but not incorrect. As another example, suppose a

processor that receives m decides to ignore m' and all the messages that depend on it (case 2b), but some processor that has a copy of m' is still running. A new message will eventually arrive that directly or indirectly depends on m' and the recovery procedure outlined in the previous section will be exercised.

3.2.2.4 Application Support

Several Psync operations are provided to support the fault-tolerance requirements of applications. The first two modify the local definition of the participant set. Specifically, `mask_out(participant)` removes a participant from its working definition of P, while `mask_in(participant)` return a participant back into the local definition of P. Once a given participant has masked out some other participant p, Psync ignores (discards) all messages m_p received from p unless it has in its holding queue a message m_q from some participant $q \neq p$, such that m_q is in the context of m_p. This is necessary so that messages will eventually stablize relative to the currently running set of participants. Note that both operations "mask" the participant set; they do not permanently delete existing participants or add new participants. Coordinating the execution of `mask_in` and `mask_out` operations by different processors is the function of the higher level membership protocol.

The last operation is used by a participant to initiate recovery following a failure. The form of this operation is as follows:

```
conv = restart(cid, pid, part_set, leaf_set)
```

Analogous to `active_open`, `restart` returns a handle for the conversation. The first argument is the system-wide unique identifier (*cid*) for the conversation, the second argument identifies the invoking participant, the third argument identifies the conversation's participant set, and the fourth argument gives the conversation-wide unique identifiers (*mids*) for the set of messages that are to form the leaves of the participant's view of the context graph upon recovery. Specifying the view is important because it defines the point at which the process starts receiving new messages. The `restart` operation is issued by a recovering participant in lieu of the standard operations for opening a new conversation.

The values used as arguments to `restart` are typically included in a checkpoint on

stable storage so that they will be available following a failure. Psync provides operations that allow the application to retrieve the values into local variables. The participant set is retrieved by the `participants` operation described in Section 3.2.1. The *cid* and *mids* are retrieved using the following two operations

```
cid = get_cid(conv)
mid_set = get_mids(node_set, conv)
```

respectively. The `node_set` given as an argument to `get_mids` is the collection of nodes for which identifiers are desired; *i.e.*, the set of messages the process wants to form the leaves of its view upon recovery.

The `restart` operation serves two purposes: to inform other participants that the invoking participant has restarted, and to initiate reconstruction of the local image of the context graph. Psync accomplishes this by sending a special restart message to all processors on which a participant resides. When a restart message is received at a processor, the local instance of Psync performs two actions. First, it notifies the local participant of the restart event; this is implemented as an out-of-band control message that is delivered to the local participant. As outlined above, this notification usually results in the initiation of the membership protocol.

Second, the local instance of Psync transmits the messages that make up the leaves of its context graph image to the participant that sent the restart message; these messages are sent as standard messages. As these messages are received at the restarting processor, the local instance of Psync reconstructs the lost context graph image according to the standard lost message protocol described in Section 3.2.2.1. That is, upon receipt of the first retransmitted messages m, Psync transmits a retransmission request to the sender of m requesting the contents of the graph from the root to the node representing m. Should that request fail, the request is broadcast to all participants. Portions of the graph that are not in the context of m (*e.g.*, siblings of m) are retrieved as required to fill in missing context of other messages as additional messages arrive from other processors. Note that this procedure recovers the processor's image of the context graph. Once the image has been recovered, the local participant's view is trivially reestablished according as specified

by the set of *mids* given as an argument to `restart`.

It is possible, given additional failures, that the entire graph will not be retrieved even when the request is broadcast. Define the *failure period* of a participant to be the time period beginning at the time of the failure and ending at the point when the participant's state and view have been reconstructed. If the failure period of $n-1$ other participants overlap with the failure period of a recovering participant p, it can be guaranteed only that the portion of the graph from the root to the lowest n-stable messages will be available upon recovery.[1] To see this, consider such an n-stable message m_s. Since m_s is in the context of messages sent by $n-1$ participants in addition to the participant that sent m_s, at least n context graph images will contain all messages from the root to m_s. Given that only $n-1$ participants have overlapping failure periods, one of the images containing that portion of the graph is assured to be available. It is worth emphasizing that the above is a worst-case scenario; it is possible that messages below m_s in the context graph will be retrieved, depending on exactly which participants fail when.

As described so far, the recovering processor depends entirely on the retransmission of messages from other processors to reconstruct its image. In fact, each processor is able to reduce its dependency on the other processors by saving a copy of the messages in its image to non-volatile storage. Thus, a restarting processor first directly recovers a portion of its image from non-volatile storage, and then "falls back" on the above procedure to recover the rest of the image. This scheme is described more completely in Chapter 5.

[1] This discussion assumes a one-to-one correspondence between images and participants.

CHAPTER 4

Ordering Protocols

The ordering protocols available in Consul allow messages from different processes in a group to be ordered in some consistent way at all receiving processes. The particular ordering used depends on the specific requirements of the application, but generally speaking, there are four possibilities: no order, partial order, semantic dependent order, and total order. Orderings at the beginning in the list are preferable since they provide faster message delivery and allow more concurrency, but they may not be strong enough to ensure the consistency of the replicas. Thus, the goal is to select the weakest ordering that still maintains the correctness of the application. To elaborate on this point, we now describe each ordering in more detail.

No ordering is the simplest kind of possible message ordering. In this case, messages may be delivered in any order at different processes and, in particular, in different orders at different processes. No extra processing is required to provide such an order and a message may be delivered to the process as soon as the message is received from the underlying communication network.

Extra processing is required to provide the remaining three kinds of ordering. Partial order is the delivery of messages following potential causality. Thus, two messages that are causally related are always delivered in the same order at every process and in an order consistent with their causal ordering. However, messages that are not related causally (*i.e.*, at the same logical time) may be delivered in different order at different process. The partial ordering is provided directly by Psync. Thus, in this case, no additional order protocol is required in the communication substrate.

Semantic dependent ordering provides a message delivery order that exploits the semantics of the operations contained in the messages. This ordering is obviously specific to the application, but one can identify classes of semantic dependent orderings that are

useful for multiple applications. In Consul, we provide a semantic dependent ordering that is based on the execution commutativity of operations. This particular ordering is one focus of this dissertation, so further discussion is deferred until Section 4.1.

Finally, the most restrictive ordering is total ordering. Here, every message is delivered in the same order at every process. In the communication substrate, total ordering is implemented by a protocol called Total based on the partial ordering provided by Psync. The algorithm for this protocol is given in [Pete89]. Basically, the total ordering is achieved by each process doing the same topological sort on the context graph. The topological sort is incremental in the sense that each process waits for a portion of its view to stabilize before allowing the sort to proceed. This is done to ensure that no future messages sent to the conversation will invalidate the total ordering. Specifically, the topological sort moves through the view in *waves*, where a wave is a maximal set of messages sent at the same logical time, *i.e.*, such that the context relation does not hold between any message pair. As soon as a wave is known to be *complete*—*i.e.*, the participant is certain that no future messages will arrive that belong to the wave—the messages in the wave are ordered according to some deterministic sorting algorithm and passed on to the application. A sufficient condition for the wave to be complete is that some message in the wave be stable; this guarantees that all future messages must follow it in the context graph and hence, that all possible members of the wave are contained in the participant's view.

4.1 Semantic Dependent Order

Although the ordering of messages guaranteed by the total order protocol provides a foundation for synchronizing distributed computations, there are certain cases in which the total ordering is not necessary. For example, the semantics of the application may allow the partial order for some operations, while requiring a total order for other operations. A semantic dependent ordering exploits the semantics of the operations to provide an ordering that is as flexible as possible, while maintaining the correctness of the application. In the communication substrate, we have developed a protocol that exploits the commutativity of operations used in certain applications. An operation is defined to be *commutative* if the execution of two or more consecutive instances of that operation, potentially with

different arguments, in any order leaves the same result; for example, increment, decrement, read are typically commutative. An operation that is not commutative is called *noncommutative.*

To make the discussion of our ordering protocol more concrete, we focus our attention on an application consisting of a replicated directory object. The directory object maintains a collection of name/value bindings and supports the following set of operations:

`list()`: return all the bindings in the directory.

`lookup(name)`: return the value with the given name.

`insert(name,value)`: insert a new name/value binding into the directory.

`delete(name)`: remove the named binding from the directory.

`update(name,value)`: update the named binding to have the given value.

Users, themselves distributed throughout the network, access the directory object by invoking one of these operations on the local copy. The local copy disseminates the operations issued by the user to other replicas. We assume that each copy of the directory object is maintained on stable storage that is unaffected by failures. We also assume that operations that change the state of the object are implemented *atomically*; in other words, execution of the operation is guaranteed not to leave the entry being modified in an intermediate state despite failures. Techniques for implementing atomic operations can be found elsewhere [Lamp81, Reed83]. Finally, we assume that the operations are *idempotent*; that is, inserting a binding that already exists, deleting a binding that does not exist, and updating a binding with its current value are all valid operations that result in the state of the directory being unchanged.

4.1.1 Ordering the Operations

The first problem in implementing a replicated object is to enforce an ordering on the operations applied to each copy of the object that is consistent with the ordering that would result if the operations were applied to a centralized copy of the object. It is clear that applying these operations in a total order solves this problem.

Notice, however, that such a solution is overly restrictive in that a pair of operations

invoked at the same time, *i.e.*, two messages (operations) sent by different clients, are coerced into a total order without regard for how the operations might or might not interfere with each other. For example, suppose that client programs running on two different processors invoke commutative operations at the same time. Requiring these operations to be executed in the same order at all processors restricts concurrency if the second operation (message) is received at a given processor before the first operation; the processor cannot execute the second operation while it is waiting for the first. To accommodate this possibility, some algorithms that use a total order take an optimistic approach: they execute the second operation on the chance that when the first one arrives it will not interfere with the second, and they rollback—undo the effects of the second operation—should it turn out that the first operation needed to be executed before the second operation.

The semantic dependent ordering presents an alternative solution to this problem of ordering operations. The solution takes advantage of both the partial ordering of messages preserved by Psync and the semantics of the operations, *i.e.*, whether or not they are commutative. In other words, instead of enforcing a total order on the operations, violating this order when it seems appropriate, and using rollback to recover when the algorithm guesses wrong, this approach starts out with a weaker partial ordering and uses the knowledge about the commutativity of operations to "break ties". For simplicity, we first describe this algorithm in the absence of processor failures; the next chapter deals with managing failures for applications such as these.

4.1.2 Overview

In the case of a replicated directory, notice that operations `list` and `lookup` are commutative. Thus, executing a collection of `list` and `lookup` operations in any order leaves the object in the same state and returns the same set of results. Operations `insert`, `delete`, and `update` are not commutative, however, because applying them in different orders may leave the object in a different state.

In addition to this static relationship among the operations, there is also a dynamic relationship among the operations based on when they were invoked. Suppose Figure

4.1 represents the partial order of invocations of a sequence of operations as given by a context graph, where the subscripts are used to distinguish between different invocations of the same operation. Here ls, l, i, d, and u stand for `list`, `lookup`, `insert`, `delete`, and `update` respectively. In this scenario, operations ls_1, l_1, l_2, d_1, and i_2 were invoked at the same logical time.

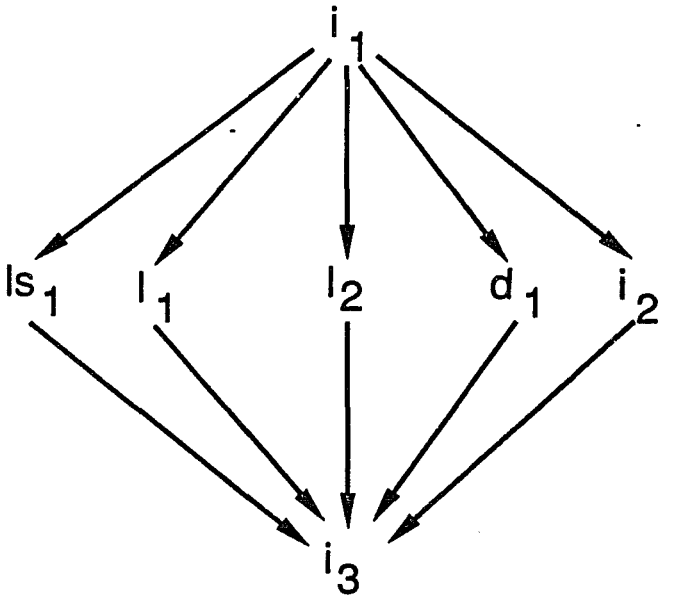

Figure 4.1: Context Graph Representing Operations

Our approach is to first order the operations based on the partial ordering—*e.g.*, i_1 is executed before l_1 because it was invoked first—and then to take advantage of the commutativity of the operations to enhance concurrency. For example, because operations `list` and `lookup` are commutative and because ls_1, l_1, and l_2 were invoked at the same time, they can be executed in any order, and in fact, in a different order by each processor. Furthermore, because `delete` and `insert` are not commutative, d_1 and i_2 must be executed in the same order by each processor and they must be totally ordered with respect to the group of commutative `list` and `lookup` operations. In other words, we assign a *precedence* to the operations and then use this precedence to break ties between operations

that were invoked at the same logical time. For example, if `list` and `lookup` are at the same precedence level, and both are preferred to `delete`, which is in turn preferred to `insert`, then the set of operations ls_1, l_1, l_2, d_1, i_2 can be executed in any of the following orders:

ls_1, l_1, l_2, d_1, i_2
ls_1, l_2, l_1, d_1, i_2
l_1, ls_1, l_2, d_1, i_2
l_1, l_2, ls_1, d_1, i_2
l_2, l_1, ls_1, d_1, i_2
l_2, ls_1, l_1, d_1, i_2

In contrast, a solution based on a total order would have limited each process that implements the directory to just one total ordering; *i.e.*, one of these six or some other that has d_1 and/or i_2 earlier in the ordering.

To understand how the algorithm works, initially assume that none of the operations are commutative. In this case, the operations will have to be sorted into the same total order at each processor. This can be done by the protocol described earler in this chapter.

Now consider the case where some of the operations are commutative. Intuitively, two or more commutative operations can be executed in any order—and in particular, in a different order by different processes—as long as there are no noncommutative operations "between" them. Formally, define an *op-group*, denoted O, to be a set of operations (nodes) in the context graph such that:

1. O contains all the noncommutative operations in a wave, or
2. O is a maximal set of commutative operations, such that every operation in this set has the same set of op-group predecessors, where an op-group α is a predecessor of an operation e if $e \notin \alpha$, and some operation in α precedes e.

We refer to the first type of op-group as a *noncommutative op-group* and the second as a *commutative op-group.* As an example, for the set of operations (nodes) in Figure 4.2, the op-groups are $\{i_1\}$, $\{i_2\}$, $\{u_1\}$, $\{d_1\}$, and $\{ls_1, ls_2, ls_3, l_1, l_2, l_3, l_4\}$.

Now, observe that for some op-group α that contains noncommutative operations and some other op-group β that contains commutative operations, one of the following two cases must be true: either the noncommutative operations in α precede zero or more of

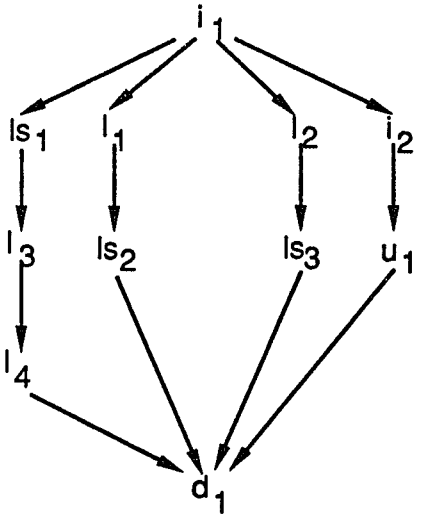

Figure 4.2: Example Partial Ordering of Operation Invocations

the operations in op-group β and are at the same logical time as the remaining operations in β, or the noncommutative operations in α are at the same logical time as one or more of the operations in β and are after the remaining operations in β. Thus, to maintain the directory object in a consistent state, the operations are processed as follows:

- All the op-groups are processed in the same total order at every replica,
- the operations in a commutative op-group are executed in an arbitrary order, and
- the operations in a noncommutative op-group are executed in the same total order.

For example, if we let α denote the op-group $\{ls_1, ls_2, ls_3, l_1, l_2, l_3, l_4\}$ in Figure 4.2, then the processors might sort the op-groups into the following total order: $i_1, \alpha, i_2, u_1, d_1$. The key idea is that while the processors must generate the same total order of op-groups, they may invoke the operations within each commutative op-group in an arbitrary order.

Without loss of generality, assume there are only two operations: `lookup` and `update`. The ordering algorithm can then be informally stated as follows:

- If a received `lookup` operation is not in the context of an unexecuted `update` operation, then it is executed immediately; otherwise its execution is delayed.

- An unexecuted `lookup` operation is executed as soon as all the unexecuted `update` operations that precede it in the context graph have been executed.
- A received `update` operation is never immediately executed. Define the *continuation property* as follows:

  Continuation Property: There is at least one unexecuted operation from every participant in the participant's view.

  Operation `update` is executed when the continuation property is satisfied and the wave that contains this `update` operation is the first unexecuted complete wave.

Note that once the continuation property has been satisfied, all future `lookup` and `update` operations will depend on the unexecuted `update` operations in the complete wave. The `update` operations in the complete wave are sorted using the same sort algorithm, *e.g.*, based on the sender's id, and then executed in that order. Thus, the execution of the `update` operations is performed in waves as these waves become complete and the continuation property is satisfied.

4.1.3 Algorithm

We now give an algorithm that realizes a semantic dependent ordering of operations for a replicated directory object. Again, for simplicity, we express the algorithm in terms of two operations: `lookup` and `update`. The algorithm uses following data structures:

`leaf_set` : Set of leaves in the participant's view.

`u_set` : Set of unexecuted `update` nodes in the participant's view.

`l_set` : Set of unexecuted `lookup` nodes in the participant's view.

`last_msg` : Array of booleans; `last_msg[i]` = true if participant i has an unexecuted operation in the participant's view.

`current_wave` : Set of nodes in the first unexecuted wave in the participant's view.

`last_wave` : Set of nodes in the last wave executed in the participant's view.

We begin by describing the auxiliary procedures upon which the main algorithm depends.

First, procedure `satisfy_continuation()` determines if the continuation property has been satisfied for a given set of unexecuted `lookup` and `update` operations. The procedure

determines if this property is satisfied by checking if every participant appears in last_msg. It returns true if every participant is in last_msg.

```
bool satisfy_continuation(last_msg)
{
   for (i = 0; i < N; i++)
      if (last_msg[i] == False)
         return (FALSE);
   return (TRUE);
}
```

Second, procedure l_executable() determines if a given node that represents a lookup operation is executable. Note that a given operation (node) cannot be executed if any of its direct predecessors have not been executed, which is easily computed by seeing if any of its predecessors are in l_set or u_set.

```
bool l_executable(n, l_set, u_set)
{
   ns = prev(n);
   for each node ∈ ns {
      if optype(node) == ''lookup'' && node ∈ l_set
         return(FALSE);
      if optype(node) == ''update'' && node ∈ u_set
         return (FALSE);
   }
   return(TRUE);
}
```

Third, procedure complete_wave() reports whether or not the current wave is complete. As described in the previous section, it decides that the wave is complete if at least one node in the wave is stable.

Fourth, procedure update_wave() determines if a given node belongs in the current wave. If it does, the node is added to wave.

We are now ready to describe the two main procedures that decide what lookup and update operations can be executed. Suppose the continuation property is satisfied. If this

```
bool complete_wave(wave)
{
   for each node ∈ wave
      if stable(node)
         return(TRUE);
   return(FALSE);
}


update_wave(wave, node)
{
   for each n ∈ wave
      if precedes(node, n) return(FALSE);
   wave = wave ⋃ node;
   return(TRUE);
}
```

---

is the case, then all update operations in current_wave can be deterministically sorted into some order and executed. Consider the context graph in Figure 4.3. Assume there are four participants and the superscripts in the messages indicate the participant that sent the message. Further assume that all the operations in the earlier part of the context graph have been executed. Here, l_1^2 in wave 1, and l_2^4 have already been executed. In Figure 4.3 (a), the continuation property is not satisfied because there are no unexecuted operations from participant 4 (note that the operation l_3^2 in wave 2 cannot be executed because it follows operation u_1^1 in wave 1). As a result, the update operations in wave 1 have not been executed. As shown in (b), an arrival of an update operation (u_5^4) and a lookup operation (l_4^2) make it possible to execute u_1^1 followed by u_2^3. Now we notice that operation l_3^2 in wave 2 can also be executed. Also, after the execution of l_3^2 in wave 2, the continuation property is satisfied, and as a consequence, u_3^1 and u_4^3 in wave 2 can be executed.

In general, execution of an op-group of update operation triggers a sequence of executions of the lookup operations that followed the update operations. Procedures do_update() and do_lookup() implement this algorithm; routines perform_lookup and perform_update actually apply the operations and are left unspecified.

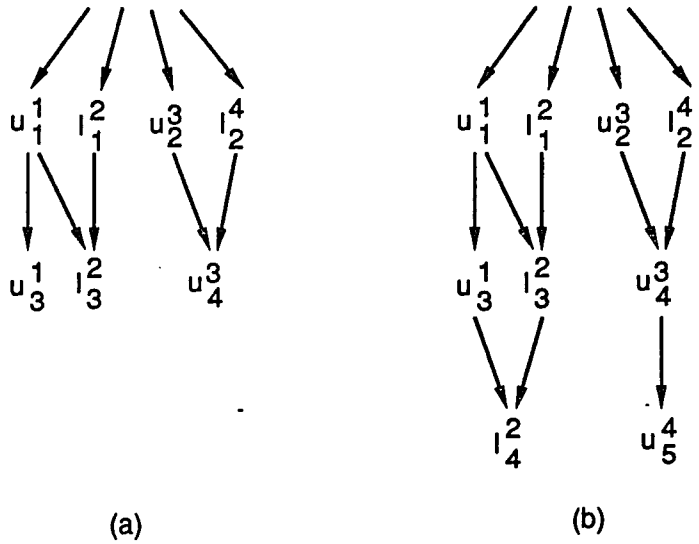

Figure 4.3: Execution of update operations

Finally, we are ready to describe the main program that implements the replicated directory object. This is realized by the process `dir_manager`. Client programs send requests to invoke operations on the directory to this process; the manager receives these requests using the operation `rcv_request`. A `wait_input()` operation is used to allow the manager process to block waiting for input from multiple sources, *i.e.*, from Psync below and from clients above.

The process first calls an `initialize` routine. This routine begins with a startup phase during which the conversation is created and messages are exchanged among the replicated processes. This routine also initializes `last_wave` and `last_msg`. After initialization, the process enters an infinite loop where it first blocks on `wait_input` for a request from a client program or a message from the conversation. When `wait_input` unblocks which indicates an input from some source, the process first processes all the messages received from the conversation and then checks to see if there are any client requests. The process immediately forwards any requests received from a client to the conversation. Finally, if the process has not sent any client requests in this iteration and if it has received an `update` operation message from the conversation, it sends an `ACK` message to the conversation.

```
do_update(node,leaf_set,l_set,u_set,last_wave,current_wave)
{
   while (satisfy_continuation(leaf_set)) {
      sort(current_wave);
      for each n ∈ current_wave
         if (optype(n) == "update") {
            perform_update(n);
            u_set = u_set - n;
            adjust_lastmsg(last_msg, node);
         }
      last_wave = current_wave;
      current_wave = ∅;
      for each n ∈ last_wave
         current_wave = current_wave ⋃ next(n);
      do_lookup(l_set, u_set);
   }
}
```

When the process receives a message from the conversation, it first updates the `current_wave` and then checks the type of the message. If the received message represents a `lookup` operation and if it is immediately executable—*i.e.*, procedure `l_executable` returns true—the process performs the `lookup` operation on its local copy of the object. For simplicity, we assume `perform_lookup` also sends the result back to the client process. If the received `lookup` operation is not executable or if the received operation is an `update` operation, then the participant calls procedure `do_update` to execute all the possible unexecuted `update` operations, followed by any subsequently enabled `lookup` operations.

4.1.4 Correctness Arguments

The algorithm essentially alternates execution between a sequence of commutative operations and a sorted wave of noncommutative operations. The sequence of commutative operations that are executed consecutively form a commutative op-group as defined in Section 4.1.2. Similarly, the sorted wave of noncommutative operations forms a noncommutative op-group. It is clear that an op-group so formed either contains all the commutative operations or all noncommutative operations. To prove the correctness of

```
do_lookup(l_set, u_set)
{
   do {
      possible = FALSE;
      for each node ∈ l_set
         if l_executable(node) {
            perform_lookup(node);
            l_set = l_set - node;
            adjust_lastmsg(last_msg, node)
            possible = TRUE;
         }
   } while(possible);
}
```

---

the algorithm, then it is sufficient to show the following:

1. The same op-groups are formed at every process,
2. All operations in an op-group are executed consecutively, and the execution of all the operations in a noncommutative op-group follows the same total order at every process, and
3. The order of op-group execution at every process is the same and this order preserves causality.

We formally prove each of these.

Theorem 1 *The same op-groups are formed at every process in the semantic dependent ordering protocol.*

Proof: We first observe that all the operations in an op-group are either commutative or noncommutative. First, assume that all the operations are noncommutative. Further assume that e_1 and e_2 are any two noncommutative operations. The op-group containing noncommutative operations is formed when a sequence of noncommutative operations is executed in the do_update procedure. This procedure puts all the noncommutative operations found in a single wave into one op-group. Since every process sees the same

```
dir_manager()
{
   u_set = l_set = leaf_set = current_wave = ∅;
   conv, last_wave, last_msg = initialize();
   while (TRUE) {
      snd_something = rcv_something = FALSE;
      wait_input();
      while (outstanding(conv)) {
         node, msg = receive(conv);
         update_wave(current_wave, node);
         switch (optype(node)) {
            case "lookup" :
               if l_executable(node, l_set, u_set) {
                  perform_lookup(node);
                  leaf_set = leaves(conv);
               }
               else {
                  l_set = l_set ⋃ node;
                  leaf_set = leaves(conv);
                  adjust_lastmsg(last_msg, node);
                  do_update(node, leaf_set, l_set, u_set,
                     last_wave, current_wave);
               }
               break;
            case "update" :
               rcv_something = TRUE;
               u_set = u_set ⋃ node;
               leaf_set = leaves(conv);
               adjust_lastmsg(last_msg, node);
               do_update(node, leaf_set, l_set, u_set,
                  last_wave, current_wave);
               break;
         }
      }
      while (!empty(request_queue)) {
         msg = rcv_request();
         send (msg, conv);
         snd_something = TRUE;
      }
      if (!snd_something && rcv_something)
         send(ACK, conv);
   }
}
```

context graph, e_1 and e_2 are either in the same wave in every process' view or they are in different waves in every process' view. Thus, either e_1 and e_2 belong to the same op-group at every process or they belong to different op-group at every participant.

Now, assume e_1 and e_2 are commutative operations. Further, assume the contrary, *i.e.*, that at process P_1, e_1 and e_2 are in the same op-group, while at process P_2, e_1 and e_2 are in different op-groups. Also, without loss of generality assume that e_1 is executed before e_2 at P_2 and let $u_1, u_2, ..., u_k$ be the noncommutative operations that are executed between the execution of e_1 and e_2. Since, e_1 is executed before any of these noncommutative operations, e_1 either precedes or is at the same logical time as each of these noncommutative operations. Assume that e_2 is sent by process j. Noncommutative operations are executed when the continuation property is satisfied. In particular, in order to execute $u_1, ..., u_k$, there must be an unexecuted operation from participant j. This operation is either one of the noncommutative operations or e_2 itself. In both the cases, e_2 must follow at least one of the operations (say u_j) in $u_1, ..., u_k$. Since this relation holds at every process, e_2 must be executed after the execution of u_j at every process including P_1 and e_1 must be executed before u_j at every process including P_1. Thus e_1 and e_2 cannot be in the same op-group at P_1. □

Theorem 2 *All the operations in an op-group are executed consecutively, and the execution of all the operations in a noncommutative op-group follows the same total order at every process.*

Proof: This proof follows from the way op-groups are constructed. Since an op-group is constructed by the order in which operations are executed, all the operations in an op-group are executed consecutively. Furthermore, procedure `do_update` uses the same deterministic sort algorithm on the op-group containing noncommutative operations to determine the order in which to execute these operations. As a result, the execution of operations in an op-group containing noncommutative operations follows the same total order at every process. □

Theorem 3 *The order of op-group execution at every process is the same and this order preserves causality.*

Proof: A commutative operation is executed when procedure `l_executable` returns true. Procedure `l_executable` returns true when all the predecessors of this operations have been executed. Thus, a commutative operation is executed only when all its predecessors have been executed. This implies that the order of the execution of commutative operations follows causality. A noncommutative operation is executed when it is in the current wave and the continuation property is satisfied. The current wave consists of the earliest wave containing an unexecuted operation. Thus, when noncommutative operations are executed, all their predecessors have also been executed. Thus the order of the execution of noncommutative operations also preserves causality.

Since the execution of all the operations preserve causality and the op-groups form a total order, the order of execution of the op-groups is same at every participant and that order preserves causality. □

4.1.5 Generalizing the Algorithm

In order to generalize the algorithm stated above to more than two classes of operations, assume a system of replicated objects where there are n different operations that can be applied on these objects by the client programs. Assume further that these operations can be subdivided into k disjoint sets $S_1, S_2, ..., S_k$ where any two operations within a set are commutative and any two operations from different sets are not commutative. Also, let different invocations of an operation in sets $S_1, S_2, ..., S_j$ $(j \leq k)$ be commutative. For such a system, define the continuation properties $\mathbf{C}_2$, $\mathbf{C}_3$, ..., $\mathbf{C}_k$ as follows:

> **Property $\mathbf{C}_i$:** There is an unexecuted operation from every participant in the sets $S_i, S_{i+1}, ..., S_k$.

We observe that operations in set S_1 can immediately be executed if they are not in context of some unexecuted operations from sets $S_2, S_3, ..., S_k$, like the `lookup` operation in the previous section. In order to execute operations from set S_2, property $\mathbf{C}_2$ must be satisfied and the wave containing the S_2 operation must be complete. In other words, in order to execute operations in set S_2, a total ordering is required. Since a total ordering is required to execute operations in set S_2, we can execute all the operations from sets $S_2, S_3, ..., S_k$ in the `current_wave` as soon as property $\mathbf{C}_2$ is satisfied and the `current_wave`

is complete. Thus, in case where the objects have operations that can be subdivided into more than two sets of commutative operations, the partial ordering is used to execute the operations in the first set and a total ordering is used to execute operations not in first set. The net result is additional concurrency for the first set S_1, but not for the rest.

4.1.6 Limitations

The algorithm presented so far ensures that the multiple copies of the directory are kept consistent under the assumption that no processors fail. Such failures may, in fact, block this algorithm indefinitely. Specifically, since a processor does not send any messages after it fails, the continuation property cannot be satisfied. As a result, no noncommutative operations and no commutative operations that follow these noncommutative operations can be executed in the presence of processor failures.

This algorithm can be modified to ensure consistency even in the face of such failures. Specifically, there are two requirements to make this algorithm fault tolerant: first, all the processes detect the failure of a process and remove it from the group of cooperating processes; and second, the recovering process is incorporated in the group and its state is restored. These two problems are discussed in detail in Chapter 5.

Another limitation of this algorithm is that it does not avoid starvation. In particular, a participant that continuously sends commutative operations without receiving prevents noncommutative operations submitted by other participants from being executed. This occurs because these noncommutative operations are not received and hence not acknowledged by the participant. Fortunately, this situation is easily avoided by restricting the number of sends that a client can do without performing a receive.

4.2 Related Work

Our approach is similar in many respects to approaches taken elsewhere. These approaches may be classified into two categories. The first category includes those protocols where the semantics of the operations are not exploited and a total order is imposed to implement replicated objects or a related constructs. Examples of this approach include [Alsb76, Birm85b, Birm85a, Birr82, Giff79, Herl86, Lamp86, Oki88b, Oki88a]

In the second category, the semantics of the application have been exploited to solve the ordering problem. In [Dani83], semantic information has been used to implement a replicated directory, while in [Davc85], the authors use semantic information to implement replicated files. Our approach differs from these two in that we maximize the concurrency by dividing different operations into op-groups, and our approach generalizes easily beyond files and directories. In [Herl87], semantic information is used to efficiently implement multiversion timestamping protocol for atomic transactions. While this work is similar to ours, the two approaches differ in two aspects. First, we efficiently implement operations on an object instead of atomic transactions. Second, we deal with objects replicated over multiple sites. That is, our emphasis is on increasing concurrency of independent operations over multiple sites rather than increasing concurrency among transactions on a single site.

Finally, we compare our work with [Ladi90]. Here, lazy replication has been proposed as a way to preserve consistency by exploiting the semantics of the service's operations to relax the constraints on ordering. Three kinds of operations are supported: operations for which the clients define the required order dynamically during the execution, operations for which the service defines the order, and operations that must be globally ordered with respect to both client-ordered and service-ordered operations. A problem with this approach is that it is best suited for client-defined ordering, even though many applications involve a collection of different kinds of operations, some requiring total ordering and some requiring partial ordering with respect to one another. Our approach performs much better in situations such as these when a mixture of these different operations needs to be applied to an object. The approach proposed in [Ladi90] must resort to a total ordering in such situations.

CHAPTER 5

Failure Handling Protocols

Support for agreement and order in a distributed system is complicated by the presence of failures. In Consul, two additional services, the membership service and the recovery service, are provided to support replica coordination in this situation. In this chapter, we discuss the details of both these services.

As defined in Chapter 2, the membership service maintains a consistent system-wide view of which processes are functioning at any given point in time. Changes in membership occurs when processes voluntarily leave or join a group, or when processes fail or recover. In the substrate, the membership service is implemented by two protocols: a *failure detection protocol* and a *membership protocol.* Both of these protocols depend on the partial order of messages provided by Psync. Our membership protocol is an instance of what is sometimes called a *monitor protocol*, *i.e.*, it is used by the system itself to maintain a consistent view of which processes are functioning rather than generating a membership change notice that is passed on to the application. The protocol, however, can easily be extended to provide such a function.

The recovery service deals with restoring the state of the failed process in such a way that the system state remains consistent. In the communication substrate, the recovery service is implemented primarily by the recovery protocol, the membership protocol, and the stable storage protocol. Other protocols in the substrate also have the ability to recover from failures, although this is oriented primarily towards restoring their own functionality rather than that of the application.

5.1 Membership Service

The membership service is initiated when the failure or recovery of a process is suspected. In the substrate, the task of detecting these events is assigned to a protocol called the

failure detection protocol. Specifically, this protocol monitors the messages exchanged in the system and on suspecting a change of state of a process, initiates the membership protocol by submitting a distinguished message to the conversation. In Section 2.3.3, we described different ways in which this might be done. In our scheme, a failure is typically suspected when no message has been received from a process in some interval of time, while recovery is based on the asynchronous notification generated when the recovering process executes the Psync `restart` primitive. Note, however, that the failure detection protocol is independent of the membership protocol and may employ any strategy to detect process status changes without affecting the membership protocol.

The membership protocol itself is based on the partial order provided by Psync. As a result, it requires less synchronization overhead and performs especially well in the presence of multiple failures. In particular, the membership protocol sits on top of Psync and coordinates the way in which processes modify their local participant list (also called the *membership list*) using the Psync primitives `maskin` and `maskout`. In the following discussion, we refer to the membership list as *ML*.

5.1.1 Correctness Criteria

Psync maintains the context graph and the membership list. In the presence of failures and recovery, an application may take an inconsistent action because of the changes being made in the membership list and context graph. To ensure the correctness of the application implemented over Psync in the presence of failures, the membership protocol must guarantee the following two properties: all functioning processes receive the same set of messages in partial order, and the graph queries by the application are consistent even in the presence of failures. We call these two properties *external consistency* and *internal consistency*, respectively. These are defined more precisely as follows.

External Consistency: The conversation graph is the same at all the processes. This has two aspects. First, all functioning processes reach the same decision about a failed (or suspected failed) process. Second, every functioning process starts accepting messages from a recovering process at the same logical time.

Internal Consistency: Decisions made by the application based on the process' view of the context graph are correct. This has two aspects. First, stability and completeness decisions are made correctly; *i.e.*, a message is considered stable only if it is followed by a message from all other functioning processes, and a wave is considered complete only when it has all of its messages. Second, processes receive all messages in the conversation.

We prove the correctness of the membership protocol by demonstrating that it guarantees both internal and external consistency.

5.1.2 Single Failures

Consider the case where at most one process fails at a time. Assume that ML initially contains n processes. The membership protocol is based on the effect that the failure has on the context graph. In particular, since a process obviously sends no messages once it has failed, it can be guaranteed there is no message from the failed process at the same logical time as the membership protocol's initiation message sent by the detection protocol. If, on the other hand, there *is* a message from the suspect process at the same logical time as the initiation message, then it can be viewed as evidence that the process has in fact not failed. In this case, it is likely that the original suspicion of process failure was caused by the process or network being "slow" rather than an actual failure. The membership protocol uses this heuristic to establish the failure of a process.

The goal of the protocol is to establish an agreement among the $n-1$ alive processes about the failure or recovery of the n^{th} process. As outlined above, the basic strategy is to agree on the failure of the process if and only if none of the $n-1$ processes have received a message from the suspect process at the same logical time as the protocol initiation message. In case of recovery, the process is incorporated in the membership list once all the remaining $n-1$ processes have acknowledged its recovery.

The actual details of the protocol are illustrated in Figure 5.1. Upon suspecting the failure of a process p, the detection protocol submits a $\langle p$ `is down`$\rangle$ message to the conversation. On receiving the message $\langle p$ `is down` $\rangle$, a process sends $\langle$`Ack,` p `is down`$\rangle$ if there is no message from p at the same logical time as this message. Otherwise, a

message ⟨Nack, p is down⟩ is sent. p is subsequently considered to have failed if the ⟨p is down⟩ message is later followed in the context graph by an ⟨Ack, p is down⟩ message from every other process.

Message	Actions of the Membership Protocol
⟨p is down⟩	If a message from p at the same logical time as ⟨p is down⟩ has been received, then send ⟨Nack, p is down⟩; otherwise send ⟨Ack, p is down⟩ and stop accepting messages from p.
⟨Nack, p is down⟩	Start accepting messages from p and terminate protocol.
⟨Ack, p is down⟩	If message ⟨p is down⟩ is stable, then remove p and terminate protocol.
⟨p is up⟩	Send ⟨Ack, p is up⟩
⟨Ack, p is up⟩	If ⟨p is up⟩ is stable, then add p to the membership list and terminate protocol.

Figure 5.1: Membership Protocol Assuming Single Failure

Internal and external consistency are easily demonstrated for this algorithm. Every process stops accepting messages from the failed process at the same logical time in the conversation—on receipt of the ⟨p is down⟩ message. Similarly, a process is incorporated at the same logical time—the wave containing the ⟨p is up⟩ message—at all the processes. As a result, every process starts accepting messages from the recovered process at the same time—as soon as it is incorporated. Since Psync guarantees the delivery of messages, every process receives the same set of messages, and as result, every process reaches the same conclusion about the failure of a process. The failed process is removed when every process has sent an ⟨Ack, p is down⟩ message. Since a process stops accepting messages from p before sending the ⟨Ack, p is down⟩ message, all messages from the failed process have been received at the time of its removal. Thus, a process receives all the messages in the conversation. Stability decisions are correct because the failed process is removed at the same time and a recovering process is incorporated at the same time at all the processes. This means that every process determines the stability of a message with respect to the

same set of processes. Thus, both internal and external consistency are satisfied.

Finally, notice that the events associated with the failure of a process, *i.e.*, the halt in accepting messages and its removal from the membership list, are only partially ordered with respect to other messages in the system. Compared with other protocols in which these events are totally ordered with respect to other messages, this approach enhances the concurrency and efficiency of the application.

5.1.3 Multiple Failures

We now extend the membership protocol to handle concurrent failures and recoveries. In the presence of such concurrent events, the protocol becomes much more complex. Perhaps the predominant reason for this is the inherent lack of knowledge about the set of processes that participate in the membership agreement process itself. That is, processes may fail or recover at any time and, in particular, they may fail or recover while the membership protocol is in progress. Another source of complexity stems from the requirement that a consistent order of removal or incorporation of processes in the membership list be maintained. This order must be the same at all the processes to ensure correctness of the application. However, it is not at all clear what this order should be, nor what the correct interpretation of "the same" is.

We first address this latter question by deriving an order in which these list modification events must be performed. We show that the semantics of *remove*—removing a failed process from the membership list ML—and *join*—incorporating a recovering process in the membership list ML—put a restriction on the order in which the modifications of the membership list take place. We then describe the actual membership protocol.

5.1.3.1 Ordering List Modification Events

Suppose that two processes p and q fail at approximately the same time. If the last message sent by p is at the same logical time as the last message sent by q (that is, neither is in the context of the other in the context graph), then p and q can obviously not participate in each other's failure agreement protocol. Since establishing agreement about the failure of a process requires concurrence of all functioning processes, agreement for processes that

fail in this way must be done simultaneously. On the other hand, if the last message sent by q is in the context of the last message sent by p, then q may contribute messages to the agreement about p having failed, *i.e.*, q may participate in the failure agreement of p.

Now, expand this scenario to include a third failing process r. Suppose that the last message sent by r is at the same logical time as the last message sent by q, but follows the last message sent by p. By the argument made above, this implies that the failure agreement of q and r must also be done simultaneously, leading to the conclusion that all three processes must be treated as a group. In general, then, the failure agreement of a set of processes must be done simultaneously whenever the last message sent by any process in the set is at the same logical time as the last message sent by at least one other process in the set.

We formalize this notion by defining a *simultaneous failure group* (*sf-group*) S as follows:

> S is an equivalence class of failed processes under the relation $\leftrightarrow^*$, where $p \leftrightarrow q$ if the last message sent by p is at the same logical time as the last message sent by q and $\leftrightarrow^*$ is the reflexive transitive closure of $\leftrightarrow$.

From the point of view of the membership protocol at a given process, all processes in an sf-group are treated as a unit: one failure agreement algorithm is used for the entire group and they are eventually removed from the membership list simultaneously. Thus, as the execution of a process proceeds, there are a series of sf-groups totally ordered with respect to one another. Specifically, an sf-group S_2 is said to *follow* another sf-group S_1 if the last message sent by any process in S_2 follows the last message sent by all processes in S_1. In this situation, the failure agreement for S_2 is typically performed after the failure agreement for S_1.

Notice, however, that it is also correct to perform the agreement for S_1 and S_2 simultaneously as if they were a single sf-group. This type of merging may be necessary in certain situations, such as if one or more processes in S_2 fail before they can participate in the failure agreement associated with S_1, or if an alive process receives the protocol initiation message for a process in S_2 before receiving all messages associated with the protocol for S_1. Interestingly, since messages are received in partial order, this latter situation can

result in different sf-groups being formed at different functioning processes. As discussed more fully in Section 5.1.3.3, our protocol exploits this fact to allow some processes to remove processes from the membership list earlier than others, while still preserving the semantic correctness of the application.

The other type of membership list modification is the addition of recovering processes. In this case, it is sufficient to add a process to the membership list at every process sometime before the recovered process sends its first message. Once this incorporation is complete, the process participates normally in system activity, including execution of future membership protocol agreement algorithms. Since the set of alive processes must be the same at all the processes while executing, for example, a failure agreement algorithm, the recovering process must be incorporated at the same logical time with respect to all other membership protocol events at all processes.

In summary, the order in which membership list modification events are handled is as follows:

1. All processes in the same sf-group are removed simultaneously. The order of removal of processes in different sf-groups follows the relative order of the sf-groups.

2. A recovering process is incorporated into the membership list at the same logical time at all the processes.

5.1.3.2 Protocol Preliminaries

For simplicity, we first define the terms and data structures used by the protocol. We say a message m_2 *immediately follows* m_1 if there is a direct edge from m_1 to m_2 in the context graph. We say m_2 *follows* m_1 if there is a path from m_1 to m_2 in the context graph. A process p is *suspected down* if it is in the membership list and a $\langle p$ `is down`$\rangle$ message has been received. Similarly, a process p is *suspected up* if it is not in the membership list and $\langle p$ `is up`$\rangle$ has been received.

The membership protocol maintains two lists: `SuspectDownList` and `SuspectUpList`. `SuspectDownList` contains the list of $\langle p$ `is down`$\rangle$ messages that have been received and `SuspectUpList` contains the list of $\langle p$ `is up`$\rangle$ messages that have been received. As

described below, messages are removed from these two lists once the process can reach a conclusion about the status of each p. The protocol also maintains an integer variable count that contains the total number of messages in `SuspectUpList` plus the number of unstable messages in `SuspectDownList`. Initially the value of count is zero and the two lists are empty.

We define the following logical times related to process failures and recoveries, where by logical time we are referring to a wave in the context graph.

Suspected Down Time (SDT): The SDT of a failed process p is the logical time containing the $\langle p$ `is down`$\rangle$ message.

Actual Down Time (ADT): The ADT of a failed process p is the earliest logical time such that there are no messages from p at or after ADT.

Realized Down Time (RDT): The RDT of a failed process p is the logical time when p is masked out of the membership list.

Suspected Up Time (SUT): The SUT of a process p is the logical time containing the $\langle p$ `is up`$\rangle$ message.

Realized Up Time (RUT) The RUT of a process ML is the logical time when p is masked back into the membership list.

Furthermore, define a *membership check state* as a state where `SuspectDownList` or `SuspectUpList` is non-empty. In a similar manner, let a *membership check period* be the time interval over which the system is in a membership check state. A membership check period always starts at the SDT or SUT of some process and ends when the two suspect lists are empty. In other words, a membership check period starts when count becomes non-zero and continues until count becomes zero. The end of a membership check period is always signified by the RDT or RUT of some process.

In the membership protocol, a message is considered *membership-stable* if it is followed by messages from all the processes that are not in `SuspectDownList`. Note that this definition of stability is applicable only in the membership protocol; all other protocols,

including those that run concurrently with the membership protocol, use the standard definition of stability given in Chapter 3.

5.1.3.3 Membership Protocol

The main idea of the membership protocol is to establish agreement among all the functioning processes about the membership list at the end of the corresponding membership check period. Thus, if there are n processes of which k are suspected to have failed, agreement on the failure of the k processes is reached if none of the other $n-k$ processes have a message from any of the suspected processes at the same logical time as the first protocol initiation message sent by the detection protocol. To maintain external consistency, the membership protocol also synchronizes the RUT of rejoining processes and the SDT of failed processes among all the functioning processes.

Informally, the membership protocol may be described as follows. Upon receiving a ⟨p `is down`⟩ message, the message is added to `SuspectDownList`. Upon receiving a ⟨p `is up`⟩ message, the message is added to `SuspectUpList`. The message associated with a suspected down process is subsequently removed from `SuspectDownList` if there is any process that has evidence to contradict the hypothesis that it has failed; that is, if a ⟨`Nack,` p `is down`⟩ message is received immediately following the ⟨p `is down`⟩ message. A suspected up process is removed from `SuspectUpList` and added to the membership list as soon as the appropriate ⟨p `is up`⟩ message becomes membership-stable. The membership check period ends when all messages in `SuspectDownList` become membership-stable and `SuspectUpList` becomes empty. At this point, all of the suspect down processes are removed from the membership list, and `SuspectDownList` is reinitialized to empty.

Figure 5.2 illustrates one possible scenario, in which processes p and r are checked for a possible failure and process q rejoins the membership list. The membership check period starts with the arrival of the ⟨p `is down`⟩ message. At the end of this period—that is, the RDT of p—process p is removed from the membership list. However, r remains since ⟨r `is down`⟩ is immediately followed by ⟨`Nack,` r `is down`⟩.

Figure 5.3 gives a more formal description of the protocol based on actions taken by a given process upon receipt of each type of message. The phrase "count is adjusted"

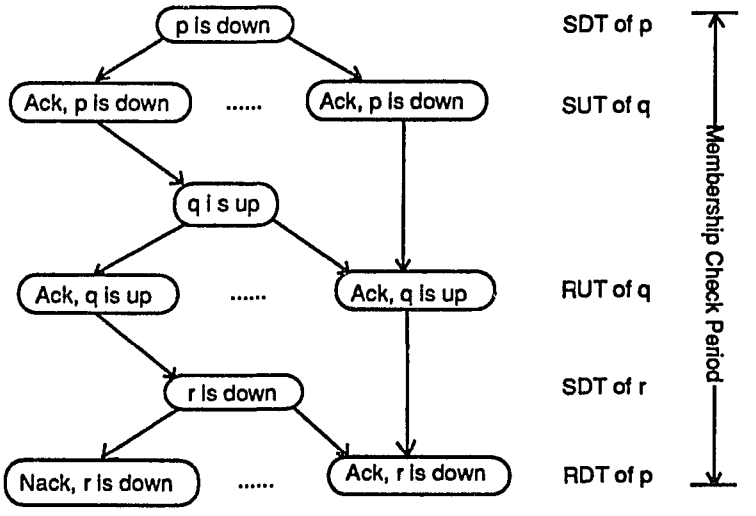

Figure 5.2: View Representing a Membership Check Period

means that the variable `count` is assigned the number of messages in `SuspectUpList` plus the number of membership-unstable messages in `SuspectDownList`. In addition, if `count` goes to zero as the result of processing a message, then `SuspectDownList` is reinitialized to empty and the corresponding processes are removed from the membership list.

The key observation about the algorithm is that different processes can form different sf-groups, or equivalently, different processes can have different membership check periods. The reason is that the membership protocol includes a process p for consideration as soon as it receives ⟨p `is up`⟩ or ⟨p `is down`⟩ protocol initiation message. To see the effect of this, consider the scenario outlined in Figure 5.4; here the numbers above the nodes represent the order in which messages are received. The process on the left receives ⟨q `is down`⟩ before the final ⟨`Ack,` p `is down`⟩ message.[1] This is possible because there is no path from the final ⟨`Ack,` p `is down`⟩ to the ⟨q `is down`⟩ message. Thus, both p and q are considered in one membership check period. In contrast, the process on the right receives all ⟨`Ack,` p `is down`⟩ messages before receiving ⟨q `is down`⟩. As a result, the right-most process uses two membership check periods. The implication of this situation

[1]Keep in mind that any pair of messages in the context graph for which there is no path leading from one to the other could have been received by the process in either order, and in fact, in different orders at different processes.

Message	Actions of the Membership Protocol
⟨*p* is down⟩	If a message from *p* at the same logical time as ⟨*p* is down⟩ has been received, then send ⟨Nack, *p* is down⟩; otherwise send ⟨Ack, *p* is down⟩, stop accepting messages from *p*, insert ⟨*p* is down⟩ in SuspectDownList, and adjust count.
⟨Nack, *p* is down⟩	Remove ⟨*p* is down⟩ message from SuspectDownList and start accepting messages from *p*.
⟨Ack, *p* is down⟩	If message ⟨*p* is down⟩ is membership-stable then decrement count.
⟨*p* is up⟩	Send ⟨Ack, *p* is up⟩, insert ⟨*p* is up⟩ in the SuspectUpList, and increment count.
⟨Ack, *p* is up⟩	If ⟨*p* is up⟩ is membership-stable, then incorporate *p* into the membership list, remove ⟨*p* is up⟩ from SuspectUpList, and adjust count.

Figure 5.3: Membership Protocol

is that p is removed from the membership list earlier at the right process than in at the left process, thus allowing its execution to proceed without unnecessary delay.

5.1.4 Correctness Arguments

We now argue that internal consistency and external consistency are maintained by the membership protocol. We first prove seven auxiliary theorems.

Theorem 1 *All messages from process p have been received by a process q before q decides the failure of p.*

Proof: The set of messages from process p is the union of all messages from p received at all alive processes. At the RDT of p, the message ⟨*p* is down⟩ is immediately followed by message ⟨Ack, *p* is down⟩ from every alive process. Thus, all messages sent by p that have been received by any alive process prior to sending ⟨Ack, *p* is down⟩ have been received by q. Since a process ignores any further incoming messages from p as soon as it sends ⟨Ack, *p* is down⟩ q receives no other messages from p. Thus, q has received all

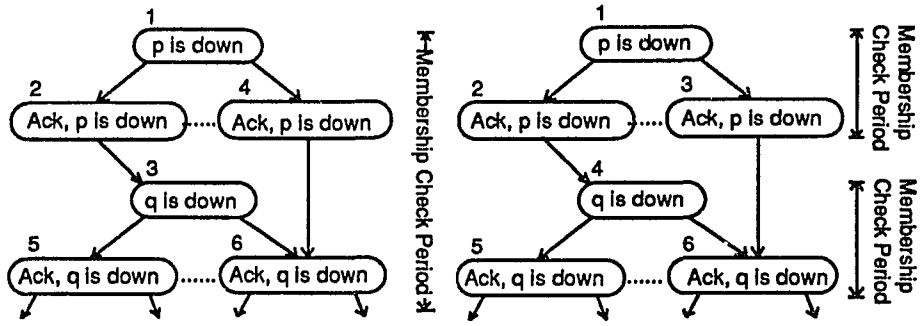

Figure 5.4: Two Different Processes' View

messages from p at its RDT; *i.e.*, when it decides that p is down. □

Theorem 2 *An alive process receives all the messages in the conversation.*

Proof: The membership check period ends at the RDT or RUT of any process From Theorem 1, we know that p has received all messages from failed process. Since the membership check period terminates at a wave that contains messages from all alive processes (`ACK` or `NACK`), all messages from all alive processes have also been received. Thus, a processes receives all messages in the conversation in the membership check period. Since Psync guarantees the delivery of all the messages in the normal state (*i.e.*, when a process is not in a membership check state), an alive process receives all the messages in the conversation. □

Theorem 3 *A recovering process is incorporated in the conversation at the same logical time at all alive processes.*

Proof: A recovering process p is incorporated into the membership list when the ⟨p `is up`⟩ message is immediately followed by a ⟨`Ack,` p `is up`⟩ message from every alive process, *i.e.*, when the wave containing the message ⟨p `is up`⟩ is complete. Since this wave is the same at all processes, p is added to the membership list at the same time by all processes. □

Theorem 4 *Stability and completeness decisions taken in the application during the execution of membership protocol are consistent.*

Proof: Stability decisions are correct if a message is decided stable when it is followed by messages from every alive process. The stability check of a message m can go wrong if some process that has a message m' following m is not considered for the stability check and m' has not been received. If a process p fails, then it is removed from the membership list at its RDT. Thus, the failed process is removed from consideration for stability checks only after its RDT. Since all the messages from a failed process have been received by its RDT (Theorem 1), all of its messages have been received before it is removed from consideration for stability checks. Thus, removal of a failed process does not introduce any inconsistency in the stability checks. When a process p recovers, it starts sending messages only after its RUT. Since the recovering process is incorporated into the membership list at its RUT, it is considered for stability checks for all the messages at or following RUT. Thus, stability checks remain consistent during the recovery of a process.

Completeness decisions are correct if a wave is decided to be complete only when it contains all of its messages. As noted in Section 4, this is typically done by testing the stability of a message in the wave. In the presence of failures, however, a failed process is not considered to determine stability, and a wave may contain messages from processes that later fail. This does not cause a problem because a failed process is not considered when determining the completeness of a wave only after it has been removed from the membership list at its RDT. But by Theorem 1, all messages from failed processes have been received by its RDT. Therefore, a wave that is determined to be complete has all messages from failed processes and all the alive processes (since all the alive processes are considered to determine stability), and as a consequence, removing a failed process does not cause any inconsistency in wave completeness checks. A recovering process p starts sending messages after its RUT, so no waves between its RDT and RUT inclusive contain a message from p. Since p is added to the membership list at its RUT, it is considered for completeness checks for all succeeding waves. Thus, completeness decisions are consistent during the process recovery. □

Theorem 5 *Alive processes have received all stable messages.*

Proof: A message m is declared stable when there is a message from all processes in ML following m in the context graph. Thus, all the processes that were alive at the time m was declared stable have received it. However, a process that was down at that time may not have received it. All the messages that are in the context graph of an alive process and that are missing in the context graph of the recovering process are retransmitted. Since message m is stable, it is retransmitted if it is not in the context graph of the recovering process. A recovering process starts participating in the conversation only after the retransmission of all the missing messages is complete, in particular, after the receipt of message m. Thus, a failed process that rejoins the membership list has received all the messages that are declared stable. □

Before considering the next theorem, we observe that we can always consider the logical time at which a given membership check period started as the same in all alive processes. This follows from the property that consecutive membership check periods (or equivalently, sf-groups) can be combined into a single membership check period without changing the semantics. Thus, given a specific check period, we can successively merge earlier check periods as required in each process to obtain a single membership check period that starts at the same logical time everywhere. This iteration is guaranteed to stop since the initial membership check period must start at the same time in every process. Note that this merging is done only for the purposes of the proof, and is not actually reflected in the protocol itself.

In the following theorem we prove that if one process decides that a process q has failed, every alive process eventually decides the same.

Theorem 6 *Let L_p and L_r be the set of alive processes at the end of membership check period containing $\langle q$ is down$\rangle$ in views of processes p and r respectively. If $r \in L_p$ and $p \in L_r$ then $q \in L_p \Leftrightarrow q \in L_r$.*

Proof: Using the above observation, we logically combine sufficient previous membership check periods so that the periods in the two processes start at the same time.

Let $Q = L_r - L_p = \{q_1, ..., q_k\}$. That is, Q is the set of processes that are in membership list of process r but not p at the end of the membership check period.

Assume that $q \in Q$. Since q is in the membership list of r, the message $\langle q$ `is down`$\rangle$ is immediately followed by $\langle$`Nack,` q `is down`$\rangle$ from some process (say s) in the view of process r. Since q is not in the membership list of p, message $\langle q$ `is down`$\rangle$ is not followed by any $\langle$`Nack,` q `is down`$\rangle$ message in p's view. Since p receives every message from all processes in L_p, and has not received the $\langle$`Nack,` q `is down`$\rangle$ message from s, then s must belong to Q. Thus, we have the following:

> [A] : For every $q \in Q$, the $\langle q$ `is down`$\rangle$ message is followed by a $\langle$`Nack,` q `is down`$\rangle$ message from some process $q_i \in Q$ in the view of r but not in the view of p.

Next, since $r \in L_p$, p has received every message from r. Since p has not received any of the NACKs, r must have sent $\langle$`Ack,` q `is down`$\rangle$ for every $\langle q$ `is down`$\rangle$ message where $q \in Q$. Thus, we have the following:

> [B] : r has sent $\langle$`Ack,` q `is down`$\rangle$ for every $\langle q$ `is down`$\rangle$ message for all $q \in Q$.

Now, since p has not received any of these NACKs and has received all of the $\langle q$ `is down`$\rangle$ messages, none of these NACKs are followed by a $\langle q$ `is down`$\rangle$ message. So, we conclude the following:

> [C] : For any $q \in Q$, the message $\langle$`Nack,` q `is down`$\rangle$ is not followed by any $\langle p$ `is down`$\rangle$ message where $p \in Q$.

In the following discussion, let s be some arbitrary process in ML. We say that "q is NACKed by s" if process r has received the message $\langle$`Nack,` q `is down`$\rangle$ from s, and this is the first NACK received immediately following the $\langle q$ `is down`$\rangle$ message. Also, we say "q is freed before s" if q is NACKed before s; *i.e.*, the first NACK message immediately following $\langle q$ `is down`$\rangle$ is received before the first NACK message immediately following $\langle s$ `is down`$\rangle$.

Let q be NACKed by s, where $s, q \in Q$. From [C], we see that $\langle$`Nack,` q `is down`$\rangle$ is not followed by $\langle s$ `is down`$\rangle$ message. Since $\langle$`Nack,` q `is down`$\rangle$ is sent by s and [B] implies that r sent $\langle$`Ack,` s `is down`$\rangle$ in response to $\langle s$ `is down`$\rangle$, r must have received

the ⟨s is down⟩ message before receiving ⟨Nack, q is down⟩. Since r received the ⟨s is down⟩ message and later received ⟨Nack, q is down⟩ from s, s must have been freed before r received ⟨Nack, q is down⟩. This implies that s has been freed before q. So, we have the following:

> [D] : If a process q_i is NACKed by another process q_j, then q_j is freed before q_i for all $q_i, q_j \in Q$.

From [A], every process in Q must be NACKed by some other process in Q, and from [D] we see that every process in Q has a process in Q that was freed before it. Since Q is a finite set and "freed before" is a total relation, this is impossible for any finite non-empty Q. Thus, the set Q must be empty.

Therefore, $Q = L_r - L_p = \phi$. Thus $L_r = L_p$. So, $q \in L_r \Leftrightarrow q \in L_p$. □

Theorem 7 *The removal of failed processes from the membership list follows the total order of sf-groups and all processes in one sf-group are removed simultaneously.*

Proof: The removal of all suspected failed processes takes place when the messages in the `SuspectDownList` are membership-stable. This condition exactly marks the completion of an sf-group. Thus, all the failed processes in an sf-group are removed simultaneously as soon as the sf-group is complete. Since an sf-group is complete before the sf-groups that follow it, the removal of failed processes from the membership list follows the total order of sf-groups. □

In summary, we have shown the following. First, since a process starts accepting messages from a recovering process as soon as the recovering process is added to the membership list, by Theorem 3 we conclude that every functioning process starts accepting messages from a rejoining process at the same time in the view. Theorem 6 confirms that the decision reached by the membership protocol about a failed (or suspected failed) process is consistent at all functioning processes. Thus, external consistency is ensured by Theorems 3 and 6. Similarly, the internal consistency is ensured by Theorems 2, 4 and 5. Finally, Theorem 7 guarantees that the order of the removal of failed processes is correct.

5.1.5 Related Work

Membership protocols have been proposed for both synchronous and asynchronous environments. Membership protocols in synchronous systems include [Cris88, Ezhi90, Kope89b, Lemo90, Walt82]. In [Kope89b], a mechanism based on synchronized clocks suitable for real-time systems has been proposed. This protocol is restricted to uses in systems in which the access to the communication channel is controlled by a synchronous time division multiplex strategy (TDMA) where a fixed slot of time interval is reserved for access to the channel for every processor. Included in each broadcast message is all the messages received by the sender in the previous TDMA cycle. This information is used to compute the set of functioning processes. The protocol in [Walt82] is synchronous in rounds, bounds the message transmission time, requires stable storage, and stops the system when reconfiguration takes place. The algorithm in [Cris88] requires synchronized clocks and a bound on the message transmission time. Periodically or on demand, each processor reaffirms its existence, and an atomic broadcast protocol is used to ensure that a message is received by all working processors.

Because the concepts of asynchrony and membership are incompatible, the membership problem is more difficult in asynchronous systems than in synchronous systems. The protocols proposed in [Birm87, Brus85, Chan84, Mish91, Ricc91] and the one proposed in this chapter assume an asynchronous environment. The protocol proposed in [Brus85] communicates failure information by diffusion. This algorithm, however, does not attempt to maintain a consensus view of the configuration. The protocols proposed in [Birm87, Chan84, Ricc91] do maintain a consistent view. However, in all of these approaches, the complete protocol has to be restarted when a process fails while the protocol is in progress. On the other hand, our protocol manages such failures differently—failures or recoveries detected while the protocol is in progress are taken in account incrementally by updating `SuspectUpList` or `SuspectDownList` appropriately. Moreover, the protocol proposed in [Ricc91] only establishes a consistent time when a failed process is to be removed. In particular, it assumes that detecting and establishing the failure of a process is implemented elsewhere. Since in asynchronous systems it is impossible to distinguish

with certainty between a failed processor and one that is merely slow, the best that can be done is reaching a tentative conclusion about a process that is suspected to have failed. Such a conclusion is reached by using some heuristics that typically involve communication among all the processes, for example, by using ack and nack messages, as we do above. Such an attempt is absent in [Ricc91].

Another advantage of our protocol relative to other approaches is that it relaxes the requirement that removal of a failed process from the membership list be totally ordered with respect to all other events. In particular, a process waits to update its membership list only until it has determined the last message sent by the failed process; it need not wait for other processes to update their membership lists. In contrast, other protocols force a process to wait until all alive processes have confirmed the failure.

A final advantage is that removal of failed processes from the membership list need not be done at the same time at all the processes. This results from the fact that sf-groups are created dynamically at each process, and these groups need not be the same at all processes. Thus, a process that does not have to merge two sf-groups will be able to remove the members of the first group before another process that does have to perform this merging operation. In contrast, other protocols wait until all processes have formed their sf-groups before removing the failed processes.

5.2 Recovery Service

When a processor recovers, three things must be done before it can resume normal processing. First, an appropriate state of the communication substrate must be reconstructed, including the states of all the protocols, the local image of the Psync context graph, and the process' view of the conversation. Second, processes at other sites must be informed so that the membership protocol described above can be initiated. Third, the actual state of the application must be brought up-to-date. We describe how these tasks are accomplished by outlining the sequence of events at a process that fails and subsequently recovers.

5.2.1 Checkpointing and Message Logging

Every protocol in Consul uses stable storage to protect itself in the event of failure. As implemented in the substrate, the stable storage protocol provides the abstraction of stable variables together with five operations: `read`, `write`, `append`, `read_group`, and `write_group`. `Read_group` and `write_group` are used to non-atomically read and write multiple variables. These exist primarily to facilitate implementation of such things as logs.

The protocols in Consul periodically checkpoint their state onto stable storage. While most of the protocols in the substrate are stateless, an important part of the state of the substrate is the image of the Psync context graph and the view of the participant. To restore this, Psync checkpoints the participant set and the conversation identifier (`cid`), while the order protocol checkpoints the view of the local process, a value that consists of a set of message identifiers. This latter information is checkpointed by the order protocol only after all the operations in the view have been successfully applied; this guarantees that all preceeding operations in the context graph have been applied as well. Other parts of the checkpoint include various static parameters needed to restart each of the protocol and the system. The application is checkpointed separately in an operation that is coordinated with the checkpointing of the view. In general, the recovery service requires every protocol to write its own checkpoints during normal operation and to include enough information to allow recovery following a failure.

To write a checkpoint atomically, a standard two version technique is used [Lamp81]. In this technique, two copies of every checkpoint and a bit indicating the current copy is maintained on stable storage. To write a checkpoint, the information is first written onto the old copy and then the corresponding bit is switched to make this the current copy. Upon recovery, the current copy is retrieved.

Finally, Psync logs parts of the context graph periodically onto stable storage. This is done to minimize the message retransmission performed when the context graph image is rebuilt using the technique outlined in Chapter 3. The logging is performed asynchronously and independently of the checkpointing being done by other protocols. In particular, the

logged context graph does not need to include all messages up to checkpointed view of the context graph, since any missing messages are automatically retrieved using the retransmission technique.

5.2.2 Recovery Stages

Given this checkpointing and message logging strategy, the recovery service goes through three stages as shown in Figure 5.5. The first stage of this process restores the substrate to the checkpointed state, the second stage restores the substrate to the current state of the system, and the third stage initiates the membership protocol to incorporate the process into the membership list. Since the fact that the context graph encapsulates the entire communication history of the system makes it the key aspect of the recovery service, we outline these stages by focusing on the context graph.

Stage	Actions at the recovering site
Stage 1	• (Re)start protocol: Collect checkpoints from all the protocols. • (Re)start protocol: Restart each of the protocols with the required parameters. • Psync: Retrieve the logged messages from the stable store and reconstruct the context graph. • Psync: If the retrieved view is not included in the context graph, send retransmit requests for the missing messages.
Stage 2	• Recovery protocol: Set every protocol in passive mode. • Recovery protocol: Send `restart` message. • All the protocols function passively.
Stage 3	• Recovery protocol: If every alive process has acknowledged the ⟨p `is` up⟩ message, set protocols to active mode.

Figure 5.5: Stages of Recovery Service

As shown in Figure 5.6, the context graph can be divided into four regions. Define

wave(n) to be the wave containing node n. The first portion is from the root of the graph down to and including the nodes in *wave(n-view)*, where *n-view* is any node in the newly restored view. Any operations in this portion have already been applied. The second portion is from below the restored view down to and including the nodes in *wave(n-failed)*, where *n-failed* is the node corresponding to the ⟨p `is down`⟩ message generated by the process that detected the failure of p.[2] Any operations in this region may or may not have been applied. Similarly, the third portion is from below *wave(n-failed)* down to and including the nodes in *wave(n-restart)*, where *n-restart* is the node corresponding to the ⟨p `is up`⟩ message generated when p recovers. Operations in this area were missed due to the failure of p. The fourth and final portion of graph consists of those nodes below *wave(n-restart)*. Any operations here are new operations that p will apply once recovery is complete.

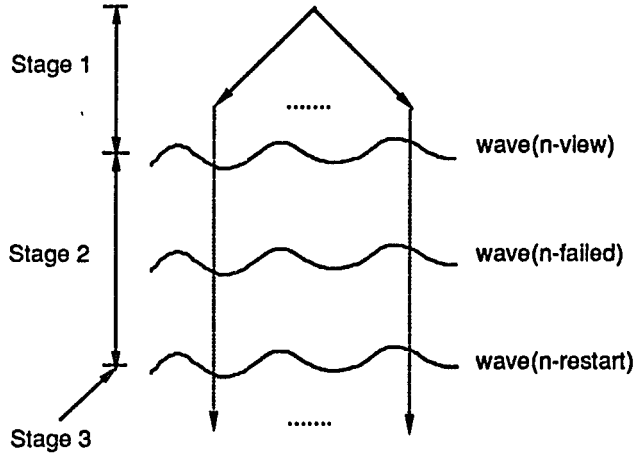

Figure 5.6: Context Graph at Recovery

It is clear from the above discussion that the operations that the recovering process needs to apply before resuming normal processing are those in the second and third portions. That is, it should apply those operations in the graph from below *wave(n-view)* up to and including *wave(n-restart)*. This is done in the various stages of recovery.

[2]There may be more than one such message if the failure was detected simultaneously by multiple processes. These messages will be siblings in the context graph, however, so the portion of the graph defined above is the same.

As mentioned above, the state of the substrate is restored to the checkpointed state in the first stage. This corresponds to restoring the context graph until *wave(n-view)* in Figure 5.6. Here, the (re)start protocol first collects the checkpoints from all the protocols and restarts each based on this information. The reconstruction of the context graph is done by Psync in two steps. Recall that Psync logs messages onto the stable store at regular intervals. In the first step, these messages are retrieved and placed into the new image of the graph. However, since the logging of messages and the checkpointing of the view by the order protocol are done independently, the logged messages may not be sufficient to reconstruct the complete context graph down to *wave(n-view)*. In such a case, a second step is invoked that retrieves the the missing messages from other functioning processes by using the Psync `retransmit` primitive. Since all the operations received during this stage have already been applied by the application, the other protocols in the communication substrate are quiescent during this stage.

The second and third stages of the recovery service are initiated simultaneously by the recovery protocol at the recovering site by transmission of the `restart` message. The second stage deals with updating the state of the process to the current state of the system. As explained in Section 3.2.2.4, on receiving the `restart` message, a functioning process retransmits all missing messages to the recovering process. The recovering process processes these messages as it would normally, but with two exceptions. First, all the protocols function in a passive mode, *i.e.*, they do not send any messages during this time. The intuition here is that, although the recovering site is replaying the past, it was not an active participant in the decisions and hence should not send messages. Second, the recovering site does not accept new operation requests from client programs. Such messages are associated with normal processing, and so are deferred until the recovery phase is complete.

The third stage deals with incorporating the process into the membership list and determining when the recovering process can start participating actively in the system. This is initiated by invoking the membership protocol at all functioning processes by an asynchronous notification on receipt of the `restart` message. Intutively, the recovering process starts participating actively in the system after it has been incorporated into the

membership list by every alive process, thus ensuring that it has received all messages prior to this time. Recall that every alive process sends an ⟨Ack, p is up⟩ message before incorporating the process in the membership list. Thus, the recovering process determines that it has been incorporated by all the alive processes when it has received ⟨Ack, p is up⟩ message from every such process. The recovery protocol checks for this condition and sets every protocol in active mode when it is satisfied. The completion of this stage completes the recovery service.

The remaining question is how the recovery protocol knows which processes are functioning so that it can determine when all ⟨Ack, p is up⟩ messages have been received. To determine this, the recovering site processes messages associated with the failure handling protocols during the passive phase as it would normally, *even when the process referred to in the message is itself.* In other words, the membership protocol masks *itself* out of the participant set when it receives the ⟨p is down⟩ message, and then back in when it receives the ⟨p is up⟩ message. This is necessary so that p will make correct stability decisions for those operations that occurred while it was down. This also implies that the mask on the participant set must be saved when a checkpoint is performed so that it is correctly restored upon recovery.

Finally, notice that by doing frequent checkpoints, the recovering site is able to minimize the size of the second portion of the context graph, thereby reducing the amount of duplicate work it does upon recovery. Because we assume idempotent operations, however, this checkpointing is actually unnecessary. That is, the site could *replay* the entire communication history in order to restore the state of the object, as is done in other systems [John87].

CHAPTER 6

Implementation and Performance

Consul has been fully implemented. The code consists of approximately 10,000 lines of C code, of which 3,500 is Psync. The implementation vehicle for the substrate is the *x*-Kernel, an operating system kernel designed explicitly for experimenting with communication protocols [Hutc91, Pete90]. In this chapter, we first give an operational overview of the *x*-Kernel and then describe the implementation of Consul itself. Finally, we give the results from some initial performance studies.

6.1 Overview of the *x*-Kernel

The *x*-Kernel is an operating system kernel explicitly designed to support the rapid implementation of efficient network protocols. It includes a uniform protocol interface and a support library to implement protocols. The support library reduces the level of effort required to implement protocols by providing efficient solutions to problems that nearly all protocols must address. This overview is borrowed largely from [Hutc91].

6.1.1 Communication Objects

The *x*-Kernel provides three primitive communication objects: *protocols*, *sessions*, and *messages*. Protocol objects are static and passive. Each protocol object corresponds to a conventional network protocol—*e.g.*, IP [Post81], UDP [Post80], TCP [USC81]—where the relationships between protocols are defined at the time a kernel is configured. Session objects are also passive, but they are dynamically created. Intuitively, a session object is an instance of a protocol object that contains a "protocol interpreter" and the data structures that represent the local state of some "network connection". Messages are active objects that move through the session and protocol objects in the kernel. The data contained in a message object correspond to one or more protocol headers and user data.

Figure 6.1(a) illustrates a suite of protocols that might be configured into a given instance of the x-Kernel. Figure 6.1(b) gives a schematic overview of the x-Kernel objects corresponding to the suite of protocols in (a); protocol objects are depicted as rectangles, the session objects associated with each protocol object are depicted as circles, and a message is depicted as a "thread" that visits a sequence of protocol and session objects as it moves through the kernel.

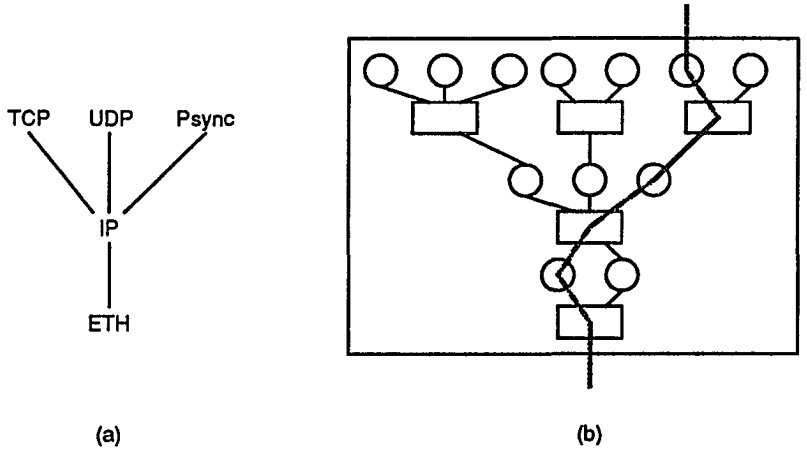

Figure 6.1: Example x-Kernel Configuration

6.1.2 Implementation techniques

Since the x-Kernel is explicitly designed to support efficient protocol implementations, all aspects of the x-Kernel—from process management, to buffer management, to the protection mechanism—are tuned for this task. Here, the important implementation techniques employed by the x-Kernel are identified.

First, the x-Kernel associates processes with messages instead of with protocols. A message is passed from one protocol to another by the former protocol invoking a procedure call on the latter protocol; no context switches are involved to pass a message between two protocols. Thus, when the message does not encounter contention for resources, it is

possible to send or receive a message with no context switches.

Second, processes are allowed to migrate across protection boundaries. When an incoming message arrives at the network/kernel boundary (*i.e.*, the network device interrupts), a kernel process is dispatched to *shepherd* it through the protocol graph; this process begins by invoking the lowest-level protocol. Should the message eventually reach the user/kernel boundary, the shepherd process does an upcall and continues executing as a user process. The kernel process is returned to a pool and made available for reuse whenever the initial protocol returns. In the case of outgoing messages, the user process does a system call and becomes a kernel process. This process then shepherds the message through the kernel.

6.1.3 Operations

A protocol object supports three operations for creating session objects:

```
session = open(protocol, invoking_protocol, participant_set)
open_enable(protocol, invoking_protocol, participant_set)
session = open_done(protocol, invoking_protocol, participant_set)
```

Intuitively, a high-level protocol invokes a low-level protocol's open operation to create a session; that session is said to be in the low-level protocol's class and created on behalf of the high-level protocol. Each protocol object is given a capability for the low-level protocols upon which it depends at configuration time. The capability for the invoking protocol passed to the open operation serves as the newly created session's handle on that protocol. In the case of open_enable, the high-level protocol passes a capability for itself to a low-level protocol. At some future time, the latter protocol invokes the former protocol's open_done operation to inform the high-level protocol that it has created a session on its behalf. Thus, the first operation supports session creation triggered by a user process (an *active* open), while the second and third operations, taken together, support session creation triggered by a message arriving from the network (a *passive* open).

In addition to creating sessions, each protocol also "switches" messages received from the network to one of its sessions with a

```
demux(protocol, message)
```

operation. demux takes a message as an argument, and either passes the message to one of its sessions, or creates a new session—using the open_done operation—and then passes the message to it.

A session object supports two primary operations:

```
push(session, message)
pop(session, message)
```

The first is invoked by a high-level session to pass a message down to some low-level session. The second is invoked by the demux operation of a protocol to pass a message up to one of its sessions. Figure 6.2 schematically depicts a session, denoted s_q^p, that is in protocol q's class and was created—either directly via open or indirectly via open_enable and open_done—by protocol p. Dotted edges mark the path a message travels from a user process down to a network device and solid edges mark the path a message travels from a network device up to a user process.

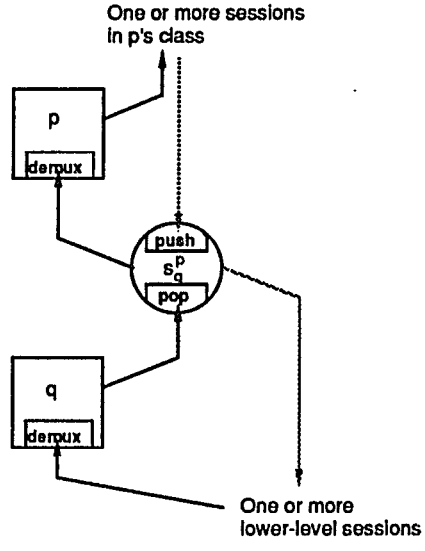

Figure 6.2: Relationship Between Protocols and Sessions

6.2 Substrate Implementation

In this implementation, there is a protocol object for each of the protocols in the substrate as shown in Figure 6.5—*viz.*, Psync, divider, membership, failure detection, recovery, order,[1] dispatch, and (re)start. Connections among the protocol objects are established by the (re)start protocol object. For every specific connection among the substrate protocols needed by the user, there is a separate protocol object implementing the corresponding (re)start protocol; the application picks up the appropriate (re)start protocol object that suits its needs.

There are three important aspects of the implementation: how the messages are structured, how the connections between various protocol objects are established initially, and how these connections are restored after a failure. To understand these aspects, we once again concentrate on the example of the replicated directory object described in Chapter 4.

6.2.1 Message Structure

The protocols in Consul share certain information about the structure of the message. The sharing of this information aids in communication between different protocols on the same processor, as well as between protocols on different processors. Specifically, there are two types of messages that are pushed onto the Psync protocol object in the communication substrate: **OT** (operation type) and **MT** (monitoring type). The OT message is used to invoke operations on the object, while the MT message is used to ensure the consistency of the communication substrate during failures and recoveries.

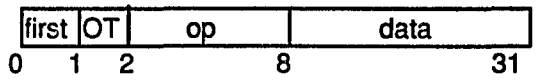

Figure 6.3: Operation Type Message

Figure 6.3 schematically depicts the **OT** message. As shown here, it is four bytes long

[1]There is a separate protocol object for each type of ordering provided in the substrate since as mentioned in Chapter 3, there is a protocol for each ordering. The one to be used is selected at the configuration time.

and the numbers in the figure indicate the starting bit of each of the fields. Here, *first* indicates whether this is the first message of the system, *op* is the operation to be invoked on the object and the *data* includes the arguments of the operation. The MT message is schematically depicted in Figure 6.4. It is 12 bytes long. Here, *mode* indicates the type of the membership message and *p_addr* indicates the address of the process. As described in Chapter 5, there are five types of membership messages: `<P is down>`, `<P is up>`, `<Ack, P is down>`, `<Nack, P is down>` and `<Ack, P is up>`.

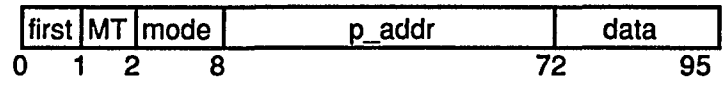

Figure 6.4: Monitoring Type Message

In Consul, a protocol object receives one or more types of messages. For example, the order protocol object receives both the OT and the MT messages, while the membership protocol object receives only the MT messages. The protocol objects specify which messages they expect to receive to the divider protocol object at the substrate initialization time and the divider protocol object, in turn, delivers the appropriate messages as they are received.

6.2.2 Establishing Connections

The system of replicated directory objects consists of a well-defined set of processes—one for each replica—that explicitly open connections among themselves in order to exchange messages. To establish these connections, one replica does an active open, while the remaining replicas do passive opens. This process of starting Consul is similar to that used for Psync, and in fact, an active open in Consul results in an active open of the Psync protocol object, while a passive open results is a passive open of Psync. Figure 6.5 shows the sessions that are created at one site in Consul; again the protocol objects are depicted as rectangles and the corresponding session objects are shown as circles. In the following discussion, we describe how these sessions are created and how the connections are established among the protocol objects; in doing so, we consider both active and passive

opens. For convenience we use the terms protocol and protocol object synonymously in the following.

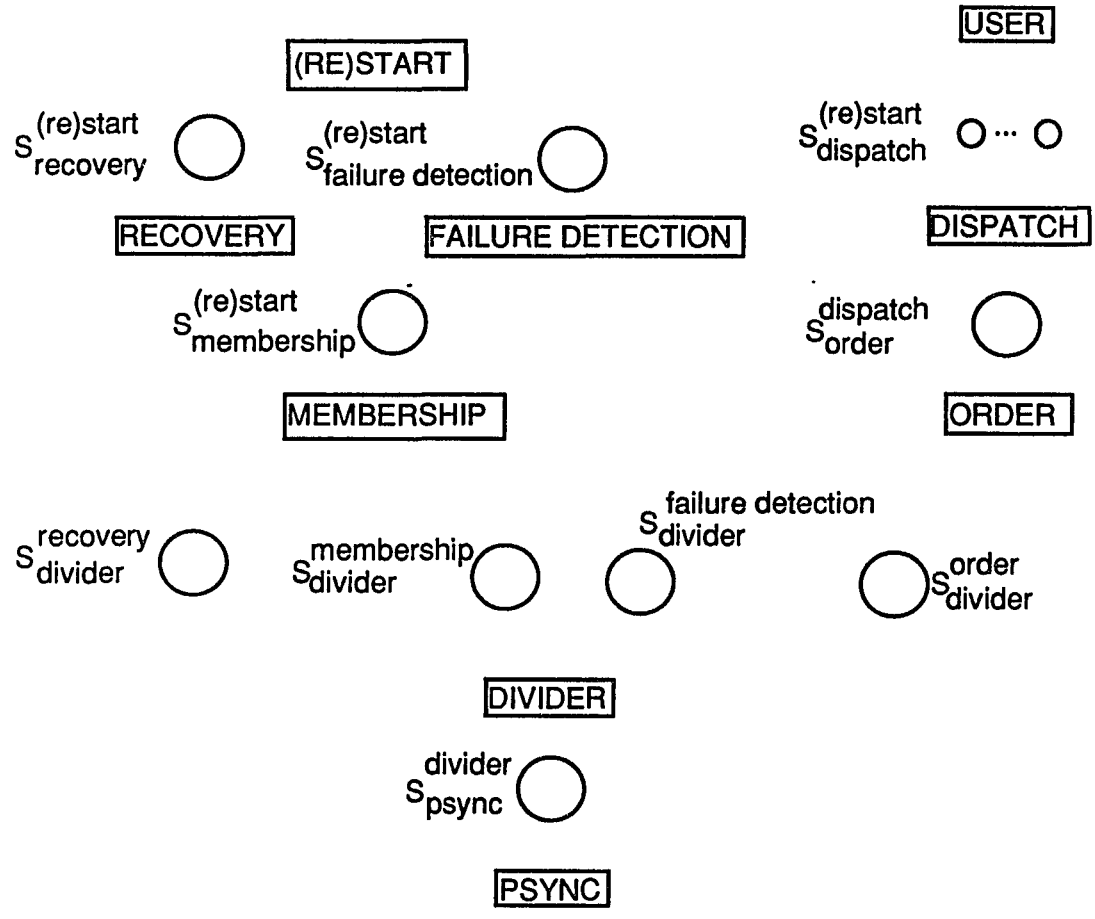

Figure 6.5: Protocol and Session Objects in the Communication Substrate

6.2.2.1 Active Open

Each user on a processor is identified by a *port_id*. A user of the replicated object opens the (re)start protocol once for every operation the object exports. This is done using the open primitive provided by the *x*-Kernel. The parameters of this call include the *operation_id*, *port_id*, and participant set. The session returned from this call is then used by the user to invoke the corresponding operation by doing a push onto this session.

The (re)start protocol is responsible for opening every protocol needed by the user. The open procedure of the (re)start protocol object opens the divider protocol when it is invoked for the first time for a given *port_id*. The arguments to this call include the *composer_id*, *port_id*, and participant set. The *session_id* of the session returned from this call acts as the unique system-wide identifier of the system and is referred to as the *system_id*. The (re)start protocol maintains a mapping of *port_id* to *system_id* for future reference. On receiving this *system_id*, the (re)start protocol opens the failure detection, membership, recovery, and dispatch protocol objects. The failure detection, membership, and recovery protocol objects are opened exactly once for a given *port_id*, while the dispatch protocol object is opened every time the open procedure of the (re)start is invoked. The arguments for all these invocations include the *system_id* and the *participant_id* of the process invoking the call. The arguments for opening the dispatch protocol also include the corresponding *operation_id*. The session returned by the dispatch protocol is returned to the user.

The open procedure of the dispatch protocol object opens the appropriate order protocol object once for a given *system_id*. It creates a session and returns it to the invoking protocol object. The pair $< S_{order}^{dispatch}, operation\_id >$ is used by the dispatch protocol to demultiplex incoming messages to the appropriate sessions above.

When the open procedure of the failure detection, membership, recovery, and order protocol objects is invoked, these protocol objects in turn open the divider protocol. The arguments to these calls include the *system_id* and the types of messages that these protocol objects expect expect to receive. These protocol objects also create a session on this invocation. The session returned by the divider protocol is used by these protocol objects to demultiplex the incoming messages to the appropriate sessions above.

The divider protocol is used to demultiplex the incoming messages to one or more protocols above. When it is opened by the (re)start protocol, it opens the Psync protocol object and returns the Psync session, returned from Psync open procedure, to the (re)start protocol. When opened by some other protocol object, it creates a session and returns it to the invoking protocol. It also maintains a map from $\langle system\_id, message\_type \rangle$ to a set of sessions, which is used to demultiplex the incoming messages to the appropriate

protocols and is updated every time a protocol other than (re)start opens the divider.

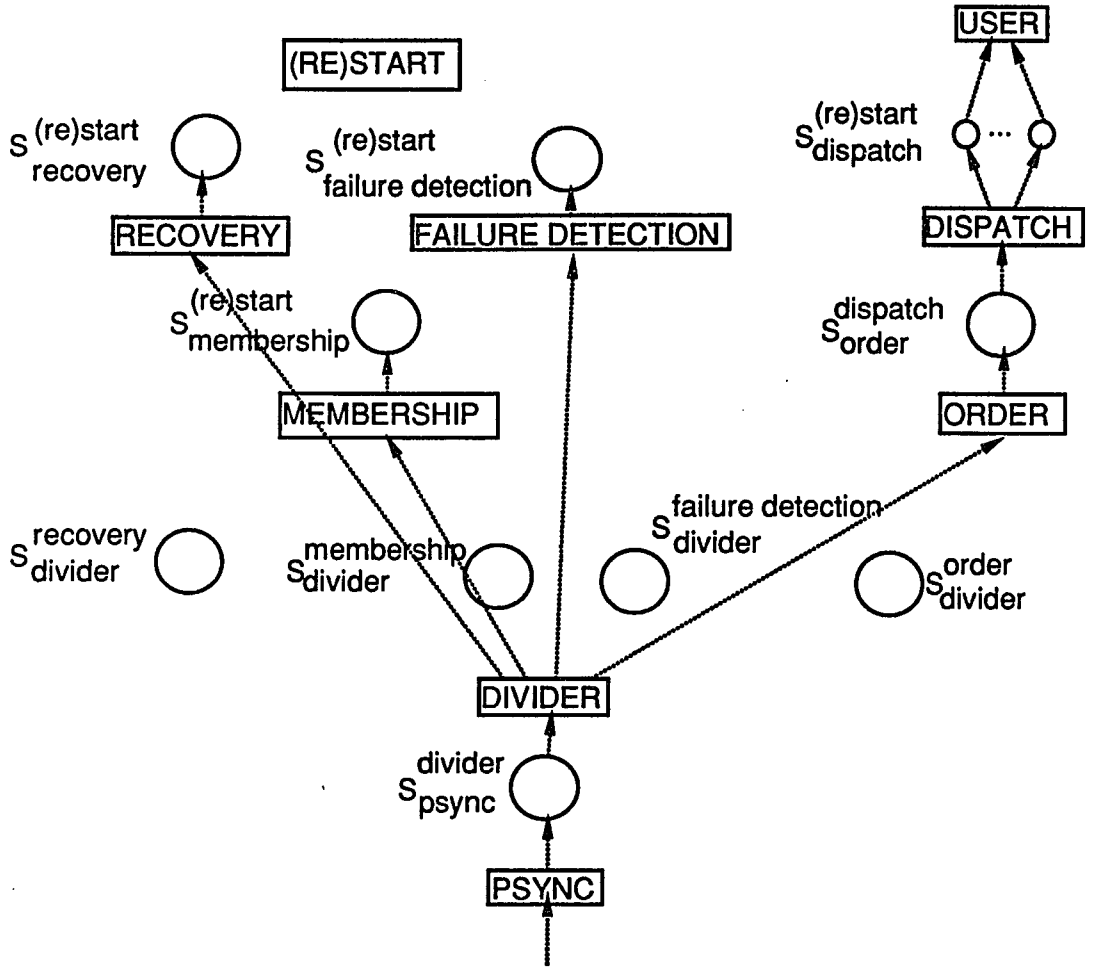

Figure 6.6: Message Flow Upwards in the Communication Substrate

6.2.2.2 Passive Open

The users on the sites containing passive replicas do `openenable` on the (re)start protocol, once for every operation the object exports. The corresponding sessions are returned by the (re)start protocol by invoking the user's `opendone` procedure.

The `openenable` procedure of the (re)start protocol invokes `openenable` of the divider

protocol exactly once for a given *port_id*. This procedure also maintains a map from *port_id* to a list of < *operation_id*, *participant_id* > to be used when the corresponding opendone procedure is invoked. The opendone procedure of the (re)start protocol is invoked by the divider protocol when the first message is received with the *port_id* as argument. In this procedure, (re)start opens the failure detection, membership, recovery, and dispatch protocols. The opendone procedure then invokes the opendone procedure of the user with the appropriate dispatch session as argument.

When the openenable procedure of the divider protocol is invoked, it performs an openenable on the Psync protocol. When the divider's opendone procedure is invoked, it invokes the opendone procedure of the (re)start protocol with the Psync session as one of the arguments. The failure detection, membership, recovery, dispatch and order protocols do not have openenable procedures since these protocols are always opened actively.

6.2.2.3 Optimizations

In the implementation, several optimizations have been made to improve system performance. Specifically message flow is optimized to direct messages only to those protocol and session objects that actually process it. Figure 6.6 shows the path a message takes as it moves upwards from the Psync protocol through the substrate. There are two optimizations done as the message flows in this direction. First, since the (re)start protocol is needed only for establishing connections, it does not need to see messages. Accordingly the (re)start protocol object and the (re)start session object are bypassed and the message moves directly from the dispatch sessions to the user protocol object. The second optimization is bypassing the divider sessions. The divider protocol needs to see the incoming message to demultiplex it to the appropriate protocols above, but there is no function to be performed in its sessions as the message flows up. Thus, these sessions are bypassed and the message moves directly from the divider protocol to the failure detection, membership, recovery or order protocols.

Figure 6.7 shows the path a message takes as it moves down through the communication substrate. Again, there are two optimizations performed as the message flows in this direction. The first optimization involves bypassing the (re)start sessions. Since the

(re)start sessions do not process an outgoing message, the message is directly pushed from the user to the appropriate dispatch sessions. In the second optimization, the divider sessions are bypassed. Since divider is a headerless protocol and is needed only for demultiplexing the message as it flows upwards, the corresponding sessions do not need to see the message as it moves in the other direction. Accordingly, a message is pushed directly from the failure detection, membership, recovery, or order sessions to the Psync session.

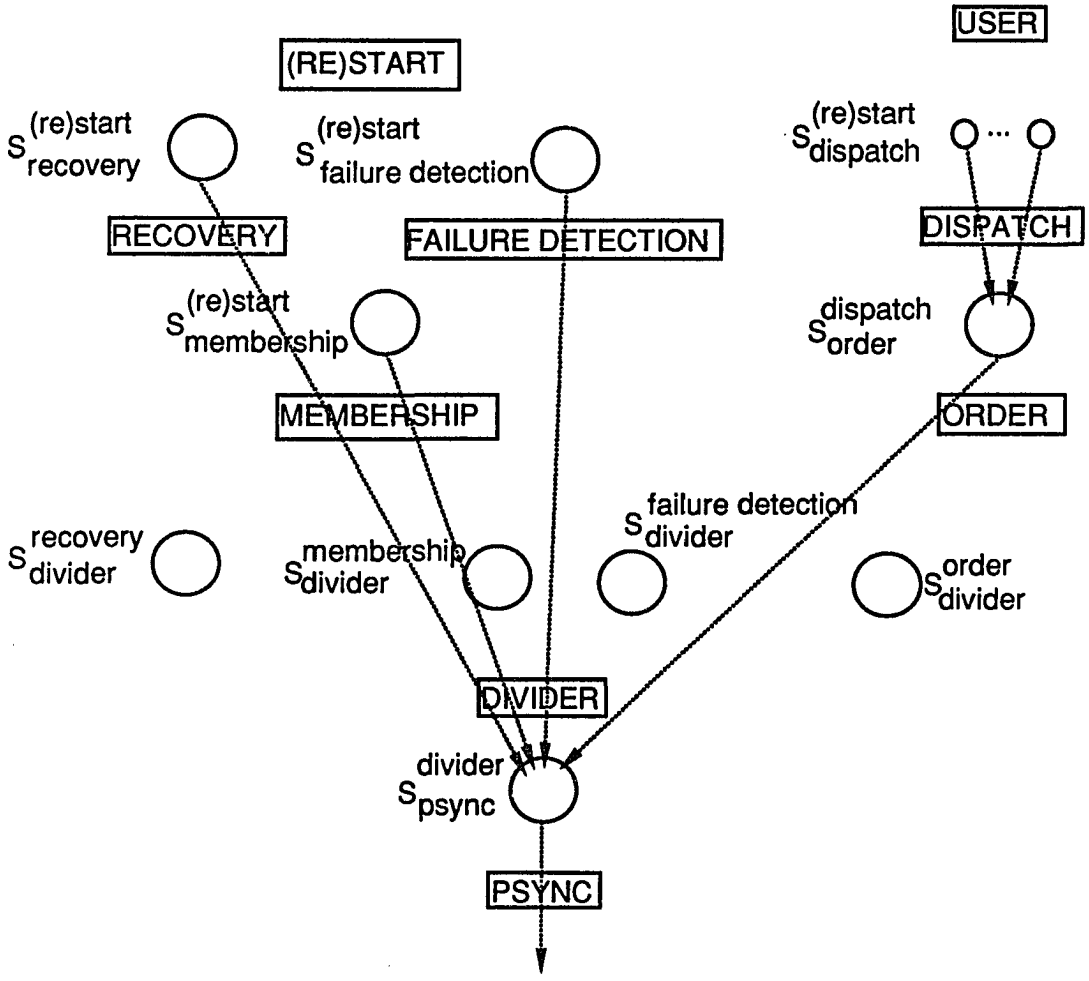

Figure 6.7: Message Flow Downwards in the Communication Substrate

The net result of the optimizations is that the (re)start sessions are not created since

they end up performing no function in the substrate. On the other hand, the divider sessions, though not used in the message flow, are created, since the *session_id* of these sessions are used by the failure detection, membership, recovery, and order protocols to demultiplex incoming messages to the sessions above.

6.2.3 Restoring Connections

The connections among various protocol and session objects, as well as their states, are lost when a failure occurs. As a result, when a process recovers, all of these objects and interconnections must be recreated. To restore these connections, every protocol and session object stores information in the stable store at a well known logical address. Typically, a protocol object stores the number of its associated session objects and for every session, the logical addresses in the stable store where the session state is checkpointed, while a session object stores its state. This is performed during the periodic checkpointing that every session performs while the system is in operation. After this is done, connections among protocol and session objects are restored by the (re)start protocol. However, there is an additional complexity: the session states cannot be fully restored given only the information stored by the corresponding protocol and session object, since these states also depend on the checkpoints taken by the other protocols. This problem is solved as follows. First, the (re)start protocol gathers the relevant checkpoints from all the protocols; these checkpoints include the *port_ids* that invoked the (re)start protocol, the corresponding *system_ids*, and all the operation identifiers for every *port_id*. The (re)start protocol then invokes the divider protocol with a control operation to restore the sessions corresponding to each *system_id*. The divider protocol, in turn, invokes the Psync protocol object to reconstruct the session state corresponding to the *session_id* retrieved from stable storage. The Psync protocol object creates a Psync session, reconstructs the context graph from the stable storage as described in Chapter 5 and returns the new *system_id* to the divider protocol, which returns it to the (re)start protocol. (re)start then invokes the failure detection, membership, dispatch, and recovery protocol objects to recover their appropriate session states, while the dispatch protocol in turn invokes the order protocol to recover the state of its session.

This completes restoration of the connections among various protocol and session objects of the communication substrate. The connection between the user protocol and the substrate is restored when the user invokes the (re)start protocol with the appropriate *port_id*.

6.3 Performance

We have built two different applications using Consul: the replicated directory object described in Chapter 4 and a distributed word game. In the distributed word game, various user processes at different processors share a list of words and a grid of letters in which these words are hidden either horizontally or vertically. The objective of the game is to locate each of the words in the grid. This is done by each process picking a word from the list and searching for it in the grid. As it is searching, the positions of the search are displayed graphically at every processor and, once found, the word is highlighted in the grid at every processor. This application requires a total ordering protocol to select a word from the list, a partial ordering protocol to display the search position of various processes, and a total ordering protocol to highlight the word once it is found.

Both of these applications have been tested under varying configurations for two, three and four replicas. These configurations differ in several ways. One is the type of ordering protocol used; some use semantic dependent ordering, while others use total ordering. Another is whether or not they contain various failure handling protocols. A third is whether checkpointing is performed and at what interval. Our experience has been that it is easy to move from one configuration to another without any modifications to the substrate.

This section reports on the performance of various protocols in Consul and the overheads they impose on the overall performance of the system. All of the numbers reported here have been taken from the replicated directory object application running on a collection of Sun 3/75 workstations connected by a lightly loaded 10Mbs Ethernet. Various experiments were designed to measure the performance of Psync and the semantic dependent ordering protocol, as well as the overhead of failure handling and checkpointing protocols.

Psync Timings

To measure the performance of Psync, one-byte messages were exchanged between a pair of user processes directly on top of Psync. In this test, the resulting average round trip delay was measured as 2.9 msec. This number is derived by exchanging messages for 10,000 trips (20,000 total messages) and reporting the elapsed time for every 1,000 round trips. Each of these measurements was then divided by 1,000 to produce the average.

Performance Using Semantic Dependent Ordering

To determine how well the semantic dependent ordering protocol performs, we compared the performance of the replicated directory object using the semantic dependent protocol with the same application using a total ordering protocol. In this experiment, we focused on measuring the average *response time* of the system, *i.e.*, the elapsed time between the time the operation is issued by the client and the time that operation is applied to the local copy of the directory. The time needed to actually perform the operation is not included.

For this experiment, the communication substrate was configured to include Psync, the divider protocol, and the appropriate order protocol. There was no logging or checkpointing done by any of these protocols. The system was configured to run on two processors. In the case of the semantic dependent ordering protocol, the average response time depends heavily on the overall mix of the commutative and the noncommutative operations, so the mix was varied across different runs. In each case, the response time is derived by having clients on each processor apply 10,000 operations (20,000 total operations), with a given percentage of commutative operations uniformly distributed, and reporting the elapsed time for every 1,000 operations (approximately 2,000 total operations). Each of these measurements was then divided by 1,000 to produce the average response time.

The results for the system configured for two replicas are shown in Table 6.1 and graphically in Figure 6.8. As expected, the semantic dependent ordering protocol improves the response time of the system as the percentage of commutative operations increase. The response time is 2.7 msec when all the operations applied are commutative, giving an improvement of about 25% over the use of a total ordering protocol. Another important

point to note is that the semantic dependent ordering protocol performs almost as good as the total ordering protocol when all the operations are noncommutative, *i.e.*, the overhead of the protocol is negligible leading to minimal effect on system performance. Similar improvement was observed for the 3-replica and 4-replica systems. These results are shown in Table 6.2.

% of comm. operations	Semantic Dep. Order	Total Order
0	3.7	3.6
50	3.55	3.6
75	3.2	3.6
90	2.9	3.6
99	2.7	3.6
100	2.7	3.6

Table 6.1: System Response Time (in msec) for a 2-replica system

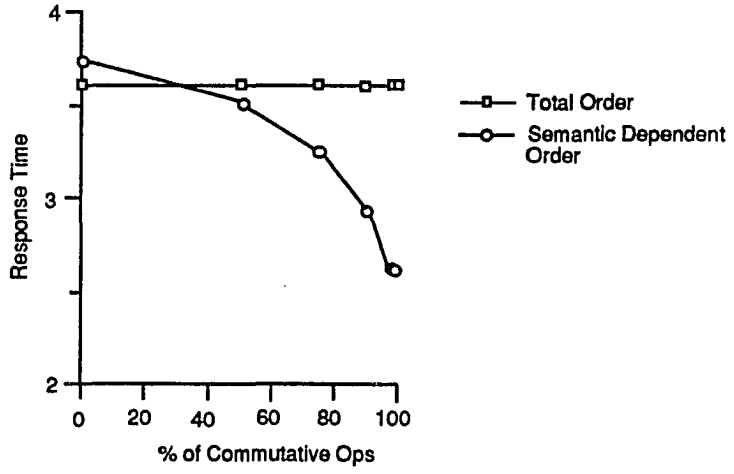

Figure 6.8: Response Time of the System

% of comm. operations	Semantic Dep. Order		Total Order	
	3-replica	4-replica	3-replica	4-replica
0	4.1	4.45	4.0	4.3
100	2.75	2.8	4.0	4.3

Table 6.2: System Response Time (in msec) for a 3-replica and 4-replica system

% of comm. operations	Response Time
0	4.2
50	4.1
75	3.8
90	3.6
99	3.3
100	3.2

Table 6.3: Response Time with Failure Handling Protocols (in msec)

Failure Handling Protocols

The overhead of the failure handling protocols in the absence of failures is measured by extending the semantic dependent ordering configuration to include the membership, failure detection, and recovery protocols. Once again, none of the protocols do any logging or checkpointing in this experiment. The response time was measured in the same way as described above for various mixes of commutative and noncommutative operations. The results are shown in Table 6.3.

The overhead imposed by the failure handling protocols is about 0.6 msec per operation. This overhead is due to two factors. First, since the communication substrate includes more protocols, the divider protocol has a larger set of protocols to which to demultiplex incoming messages. Second, the failure detection protocol needs to receive every message exchanged in the system. Thus, in addition to the order protocol, the divider protocol also demultiplexes every message to the failure detection protocol.

Checkpointing Overhead

The experiment to measure the checkpointing overhead was done as follows. Two clients at different processors issued operations at a fixed rate of 10 ops/sec and 100 ops/sec and the elapsed time for every 100 operations is measured. In the experiment, 5,000 operations were issued by each client. The elapsed time for 100 operations is measured under different checkpointing intervals. The results are shown in Table 6.4. All the operations issued were commutative operations and the semantic dependent ordering protocol is used throughout along with failure handling protocols.

Two observations can be made from these measurements. First, the checkpointing

Ops/sec	Checkpoint interval (in sec)	Time for 100 Ops (in sec)
10	No Checkpointing	10.0
10	5.0	10.0
10	2.0	10.0
10	1.0	10.1
100	No Checkpointing	2.0
100	5.0	2.1
100	2.0	2.2
100	1.0	2.4

Table 6.4: Measure of Checkpointing Overheads

overhead increases with the increase in the rate at which clients issue operations. Thus, the overhead is 0.4 sec per 100 operations when the clients issue operations at 100 ops/sec and the checkpointing interval is 1 sec, while the overhead under the same conditions is 0.1 sec per 100 operations when operations are issued at 10 ops/sec. The reason for this is that the system performs fewer operations per unit time when the rate of issue of operations is lower, leading to more idle time to do the checkpointing. As a result, its effect on the time to process an operation is less.

The second observation is that the overhead of checkpointing increases as the checkpoint interval is reduced. Thus, this overhead is 0.1 sec per 100 operations when operations are issued at 100 ops/sec and the checkpointing is done every 5 sec, while this overhead increases to 0.4 sec per 100 operations when the checkpointing is done every 1 sec. This is expected, as the time spent to do the checkpointing per unit time increases as the checkpoint interval is reduced. The effect of checkpointing is almost negligible for checkpoint intervals of 5.0 sec or higher for the observed operation rates.

CHAPTER 7

Conclusion

7.1 Summary

In this dissertation, we have presented the design and implementation of Consul, a communication substrate for fault-tolerant, distributed programs. The overall objective of the system is to provide fault-tolerant services for enhancing computing system dependability based on the state machine approach. Specifically, the system is designed to realize three important objectives. The first is to provide support for interprocess communication and for different kinds of consistent orderings among the messages exchanged in the system. The second is to provide support for recovering from failures and for continued processing in the presence of failures. The third is to have an architecture flexible enough to satisfy the diverse requirements of many different applications.

Consul consists of a suite of fault-tolerant communication protocols that together provide various fault-tolerant services such as broadcast, membership, and recovery. These protocols together form a substrate that can be used to build fault-tolerant applications. Various protocols included in the substrate are Psync, order, membership, failure detection, recovery, stable storage, divider and (re)start.

Psync is the main communication mechanism in Consul. It provides a multicast facility that also maintains a consistent partial order among the messages exchanged in the system. Other protocols in Consul make use of this partial ordering to implement their respective services. An overview of Psync has been given in Chapter 3.

Chapter 4 described various broadcast services provided in Consul. These broadcast services differ from one another in the type of order they provide. The partial ordering is available directly from Psync. In addition, protocols providing total ordering and semantic dependent ordering are also included. The semantic dependent ordering is based on operation commutativity, which allows concurrent execution of commutative operations

in certain situations. Specifically, this ordering is achieved by (a) dividing the operations into commutative and noncommutative op-groups, (b) executing these groups in the same total order at every process, (c) executing operations within a commutative op-group in any order, and (d) executing operations within a noncommutative op-group in the same total order.

The membership service is implemented by the combination of failure detection and membership protocols. The failure detection protocol monitors messages traffic and submits a notification message to the conversation when lack of activity indicates a possible failure. The membership protocol then confirms this failure in a way that maintains a consistent system-wide view of which processes are functioning at any given point in time. The heuristic used to establish the failure is based on every alive process (other than the suspected process) concurring on this suspicion. This protocol forms sf-groups—set of processes that fail simultaneously—and removes all the processes in an sf-group simultaneously. These removals may be done at different times in different processes.

The recovery of a process' state after a failure has been described in Chapter 5. The recovery service consists of a combination of the membership, recovery, and (re)start protocols. This service combines checkpointing with message replay to avoid rollbacks at functioning processes and minimize recomputation at the recovering process.

Consul provides a flexible architecture in which an application can pick the required protocols and build a system using these protocols. The (re)start and divider protocols aid in building such a system. The (re)start protocol establishes a connection among the protocols selected by the application at the beginning of the execution and following a crash. The divider protocol demultiplexes an incoming message to multiple protocols based on their specific requirements.

Consul has been completely implemented using the *x*-Kernel and runs standalone on a network of Sun 3/75 workstations. The system has been tested for using several different applications and the initial performance is encouraging. This implementation and its performance have been reported in Chapter 6.

7.2 Contributions and Limitations

This dissertation contributes to both the theory and practice of providing fault tolerance in distributed systems. In particular, there are two main contributions. The first is the design of new algorithms that make the system more dependable and efficient. In particular, novel algorithms have been presented for semantic dependent ordering, membership, and recovery. The semantic dependent ordering exploits the execution commutativity of operations to provide more concurrency, thereby improving system performance in some cases by 30% over a similar system that uses total ordering. Moreover, the overhead is negligible and the algorithm performs as well as a total ordering algorithm when every operation issued in the system is noncommutative. The membership protocol manages concurrent failures and recoveries incrementally, and does not have to restart when failures or recoveries occur while the protocol is in progress. Processes that fail concurrently are removed from the membership list as a group, while still allowing variance at different processes to reduce synchronization. The recovery protocol combines message logging with message retransmission to recover the lost states of failed processes efficiently and without rolling back the states of functioning processes.

The second contribution of this dissertation is in the application of new system structuring techniques. These techniques make it easy to modify the system architecture or add new protocols to the substrate without affecting existing components. The configuration protocols in particular allow an application designer to build a system around a given collection of protocols with minimum effort. As a result, the system can satisfy the diverse needs of many different applications with little overhead and in a way that allows an application to pay only for the functionality that it needs. These techniques have been possible due to the use of x-Kernel as the implementation vehicle.

Although Consul has performed well for the tested applications, there are certain limitations that should be noted. One is that the system is not scalable to any large degree; this follows from the use of Psync with its explicit context graph, as well as the fully distributed nature of the other protocols. However, this is not a problem in practice since the number of replicas used in the state machine approach is typically small.

Another limitation of Consul is in the recovery service. Specifically, if the failed process remains down for a long time, the second stage of recovery will result in a large number of messages being retransmitted, possibly overloading the communication network. This problem is fundamental to recovery in any system, but can be alleviated to a degree by either checkpointing frequently or changing the recovery service to retrieve the current state of a functioning replica rather than relying on retransmissions.

7.3 Future Directions

There are a number of related areas that we plan to explore in the future. These include the redesign of Psync, development of appropriate tools, protocol decomposition, and application of the techniques developed in this work to other fault-tolerant programming paradigms. We discuss each of these in turn.

One of the future directions of this research is redesigning Psync to make it more efficient. For example, Psync provides multicast communication, but the group members must be specified completely at conversation initiation time. This is overly restrictive for some applications in which various processes join and leave the group as the computation progresses. Another aspect to be explored is smart garbage collection. As the computation progresses, the context graph increases in size, so a garbage collection capability is needed to trim the context graph by deleting nodes that are no longer needed by the application. Finally, we will investigate the possibility of decomposing Psync into smaller pieces. The functions provided by the protocol include a group abstraction, partial ordering, guaranteed message delivery, maintenance of the participants' views, and various failure handling primitives. Some of these functions are independent of one another, so it may be possible to construct them as independent modules to achieve a more efficient and understandable system.

In the general area of fault-tolerant protocols, our future research is motivated by two important observations that we have made in our current work. First, every fault-tolerant protocol needs certain functions such as stable storage, logging and checkpointing. In Consul, each protocol implements its own versions of these functions, but given their widespread use, a general purpose tool seems warranted. The second observation is that

every fault-tolerant protocol has a similar structure, *i.e.*, every protocol uses certain standard operations in a common way. Given this standard structure, it seems reasonable that the system should provide capabilities to implement such a structure efficiently. In both these cases, providing tools to implement more common functions and capabilities should result in a reduction in the complexity of fault-tolerant protocols.

We also plan to extend this research by looking at the issue of decomposing fault-tolerant protocols into smaller, more fundamental pieces. One of the reasons for the complexity of many such protocols is that they implement many functions, so by implementing each of these functions as a separate individual module, we hope to simplify the implementation. Moreover, this decomposition should also make it easier to identify the inherent dependencies among the various protocols. The challenge here is, first, determining exactly how to decompose each existing protocol, and then, how to use the resulting fundamental modules to provide the required functionality.

Finally, this research will be extended to explore the applicability of the techniques we have developed to other fault-tolerant programming paradigms. Consul is closely tied to the state machine approach, but we feel that the techniques developed may also be useful with the other paradigms mentioned in Chapter 1, such as the object-action model or the conversation model. Although these other paradigms have enough similarities to the state machine approach to make us confident that our approach can be extended, only further investigation can answer this question in any definitive way.

REFERENCES

[Alsb76] Alsberg, P. A. and Day, J. D. A principle for resilient sharing of distributed resources. In *Proceedings of 2nd International Conference on Software Engineering*, pages 627–644, Oct 1976.

[Ande76] Anderson, T. and Kerr, R. Recovery blocks in action: A system supporting high reliability. In *Proceedings of 2nd International Conference on Software Engineering*, pages 447–457, San Francisco, CA, Oct 1976.

[Ande83] Anderson, T. and Knight, J. C. A framework for software fault tolerance in real time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, 1983.

[Aviz85] Avizienis, A. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[Bari83] Barigazzi, G. and Strigini, L. Application-transparent setting of recovery points. In *Proceedings of the 13th Symposium on Fault Tolerant Computing*, Jun 1983.

[Bhar88] Bhargava, B. and Lian, S. Independent checkpointing and concurrent rollback for recovery in distributed systems — An optimistic approach. In *Seventh Symposium on Reliable Distributed Computing*, pages 3–12, Columbus, Ohio, Oct 1988.

[Birm85a] Birman, K. Replication and fault-tolerance in the ISIS system. In *Tenth ACM Symposium on Operating System Principles*, pages 79–86, Orcas Island, WA, Dec 1985.

[Birm85b] Birman, K., Joseph, T., Raeuchle, T., and Abbadi, A. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, Jun 1985.

[Birm87] Birman, K. and Joseph, T. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[Birm91a] Birman, K., Cooper, R., and Gleeson, B. Programming with process groups: Group and multicast semantics. Technical Report 91-1185, Department of Computer Science, Cornell University, Jan 1991.

[Birm91b] Birman, K., Schiper, A., and Stephenson, P. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[Birr82] Birrell, A., Levin, R., Needham, R., and Schroeder, M. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, Apr 1982.

[Boeh73] Boehm, B. W. Software and its impact: A quantitative assessment. *Datamation*, 19(5):48–59, May 1973.

[Brus85] Bruso, S. A. A failure detection and notification protocol for distributed computing systems. In *Proceedings of the IEEE 5th International Conference on Distributed Computing Systems*, pages 116–123, Denver, CO, May 1985.

[Chan84] Chang, J. and Maxemchuk, N. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[Chen78] Chen, L. and Avizienis, A. N-version programming: A fault-tolerance approach to reliability of software operation. In *Eighth Annual International Conference on Fault-Tolerant Computing*, pages 3–9, Toulouse, Jun 1978.

[Cris85] Cristian, F., Aghili, H., Strong, R., and Dolev, D. From simple message diffusion to byzantine agreement. In *Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[Cris88] Cristian, F. Agreeing on who is present and who is absent in a synchronous distributed system. In *Eighteenth International Conference on Fault-tolerant Computing*, pages 206–211, Tokyo, Jun 1988.

[Cris89] Cristian, F. Probabilistic clock synchronization. In *Ninth International Symposium on Distributed Computing Systems*, pages 288–296, Newport Beach, CA, Jun 1989.

[Cris90] Cristian, F., Dancey, B., and Dehn, J. Fault-tolerance in the advanced automation system. Technical Report Research Report RJ 7424, IBM Almaden Research Center, Apr 1990.

[Cris91] Cristian, F. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[Dahl72] Dahl, O.-J., Dijkstra, E., and Hoare, C. *Structured Programming.* Academic Press, London, 1972.

[Dani83] Daniels, D. and Spector, A. Z. An algorithm for replicated directories. In *Second Annual ACM Symposium on Principles of Distributed Computing*, pages 104–113, Montreal, Canada, Dec 1983.

[Davc85] Davcev, D. and Burkhard, W. A. Consistency and recovery control for replicated files. In *Tenth ACM Symposium on Operating System Principles*, pages 86–96, Orcas Island, WA, Dec 1985.

[Dijk68] Dijkstra, E. W. The structure of the the multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dijk76] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs (NJ), 1976.

[Dole83] Dolev, D., Dwork, C., and Stockmeyer, L. On the minimal synchronism needed for distributed consensus. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science*, Tucson, AZ, Nov 1983.

[Dole84] Dolev, D., Halpern, J. Y., and Strong, R. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of 16th Anual ACM STOC*, pages 504–511, Washington, D.C., Apr 1984.

[Ezhi90] Ezhilchelvan, P. D. and Lemos, R. A robust group membership algorithm for distributed real-time system. In *11th Real-Time Systems Symposium*, pages 173–179, Lake Buena Vista, Florida, Dec 1990.

[Fidg88] Fidge, C. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988.

[Fisc85] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[Garc82] Garcia-Molina, H. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):49–59, Jan 1982.

[Garc88] Garcia-Molina, H. and Kogan, B. An implementation of reliable broadcast using an unreliable broadcast facility. In *Seventh Symposium on Reliable Distributed System*, pages 101–111, Columbus, OH, Oct 1988.

[Garc91] Garcia-Molina, H. and Spauster, A. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, Aug 1991.

[Giff79] Gifford, D. K. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, Dec 1979.

[Gray87] Gray, J. Notes on database operating systems. In *Lecture Notes in Computer Science 60*, pages 393–481. Springer-Verlag, Berlin, 1987.

[Hadz82] Hadzilacos, V. An algorithm for minimizing rollback cost. In *First ACM Symposium on Principles of Distributed Computing*, pages 93–97, Ottawa, Canada, 1982.

[Halp84] Halpern, J. Y., Simons, B., Strong, R., and Dolev, D. Fault-tolerant clock synchronization. In *Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, Vancouver, Canada, Aug 1984.

[Herl86] Herlihy, M. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, Feb 1986.

[Herl87] Herlihy, M. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, C-36(4):443–448, Apr 1987.

[Horn74] Horning, J. J. A program structure for error detection and recovery. In Gelenbe, E. and Kaiser, C., editors, *Lecture Notes in Computer Science 16*, pages 171–187. Springer-Verlag, Berlin, 1974.

[Hutc89] Hutchinson, N. C., Peterson, L. L., O'Malley, S., and Abbott, M. RPC in the x-kernel: Evaluating new design techniques. In *The Twelfth ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, Dec 1989.

[Hutc91] Hutchinson, N. C. and Peterson, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[John87] Johnson, D. and Zwaenopoel, W. Sender based message logging. In *Seventeenth International Symposium on Fault-Tolerant Computing*, pages 14–19, Pittsburgh, PA, Jun 1987.

[John90] Johnson, D. and Zwaenepoel, W. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, pages 462–491, 1990.

[Kaas89] Kaashoek, M. F., Tanenbaum, A., Hummel, S. F., and Bal, H. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, Oct 1989.

[Kim78] Kim, K. H. An approach to program-transparent coordination of recovering parallel processes and its efficient implementation rules. In *Proceedings of 1978 International Conference on Parallel Processing*, Aug 1978.

[Kim86] Kim, K. H., You, J. h., and Abouelnaga, A. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Proceedings of 16th IEEE Symposium on Fault Tolerant Computing*, Jun 1986.

[Koo87] Koo, R. and Toueg, S. Checkpointing and rolback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan 1987.

[Kope85] Kopetz, H. and Merker, W. The architecture of MARS. In *15th Annual Symposium on Fault-Tolerant Computing*, pages 274–279, Ann Arbor, Mi, Jun 1985.

[Kope87] Kopetz, H. and Ochsenreiter, W. Clock synchronizatin in distributed, real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, Aug 1987.

[Kope89a] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, Feb 1989.

[Kope89b] Kopetz, H., Grunsteidl, G., and Reisinger, J. Fault-tolerant membership service in a synchronous distributed real-time system. In *International Working Conference on Dependable Computing for Critical Applications*, pages 167–174, Santa Barbara, California, Aug 1989.

[Ladi90] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Lazy replication: Exploiting the semantics of distributed services. Technical Report MIT/LCS/TR-484, MIT Laboratory for Computer Science, Cambridge, MA, Jul 1990. to appear in ACM Transactions on Computer Systems.

[Lala85] Lala, P. K. *Fault Tolerant and Fault Testable Hardware Design.* Prentice Hall International, London, 1985.

[Lamp78] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.

[Lamp81] Lampson, B. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[Lamp84] Lamport, L. and Melliar-Smith, P. M. Byzantine clock synchronization. In *Third ACM Symposium on Principles of Distributed Computing*, pages 68–74, Vancouver, Canada, Aug 1984.

[Lamp86] Lampson, B. W. Designing a global name service. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pages 1–10, Aug 1986.

[Lapr91] Laprie, J. C., editor. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, New York, 1991. to appear.

[Lee90] Lee, P. A. and Anderson, T. *Fault Tolerance : Principles and Practice*, pages 55–57. Springer-Verlag, Vienna, 1990.

[Lemo90] Lemos, R. and Ezhilchelvan, P. Agreement on the group membership in synchronous distributed systems. In *4th International Workshop on Distributed Algorithms*, Otranto, Italy, Sep 1990.

[Leu89] Leu, P. and Bhargava, B. A model for concurrent checkpointing and recovery using transactions. In *The Ninth International Conference on Distributed Computing Systems*, pages 423–430, Newport Beach, California, Jun 1989.

[Lipp72] Lippman, S. B. *C++ Primer.* Addison Wesley, Reading, MA, 1972.

[Lund84] Lundelius, J. and Lynch, N. A new fault-tolerant algorithm for clock synchronization. In *Third ACM Symposium on Principles of Distributed Computing*, pages 75–88, Vancouver, Canada, Aug 1984.

[Marz84] Marzullo, K. *Maintaining the Time in a Distributed System.* PhD thesis, Stanford University, Department of Electrical Engineering, Mar 1984.

[Matt89] Mattern, F. Time and global states in distributed system. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, North-Holland, 1989.

[Mell89] Melliar-Smith, P. M. and Moser, L. E. Fault-tolerant distributed systems based on broadcast communication. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 129–134, Newport Beach, CA, Jun 1989.

[Meye88] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mish91] Mishra, S., Peterson, L., and Schlichting, R. A membership protocol based on partial order. In *The Second IFIP Working Conference on Dependable Computing for Critical Applications*, pages 137–145, Tucson, AZ, Feb 1991.

[Myer76] Myers, G. J. *Software Reliability: Principles and Practices.* Wiley, New York, 1976.

[Nava88] Navaratnam, S., Chanson, S., and Neufeld, G. Reliable group communication in distributed systems. In *The Eighth International Conference on Distributed Computing Systems*, pages 439–446, San Jose, California, Jun 1988.

[Ng88] Ng, P. A commit protocol for checkpointing transactions. In *The Seventh Symposium on Reliable Distributed Computing*, pages 22–31, Columbus, Ohio, Oct 1988.

[Oki88a] Oki, B. M. Viewstamped replication for highly-available distributed systems. Technical Report TR MIT/LCS/TR-423, MIT Laboratory for Computer Science, Cambridge, MA, Cambridge, MA, Aug 1988.

[Oki88b] Oki, B. M. and Liskov, B. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, Toronto, Canada, Aug 1988.

[Panz88] Panzieri, F. and Shrivastava, S. K. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, SE-14(1):30–37, Jan 1988.

[Parn72] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec 1972.

[Pete89] Peterson, L. L., Buchholz, N. C., and Schlichting, R. D. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[Pete90] Peterson, L. L., Hutchinson, N. C., O'Malley, S. W., and Rao, H. C. The x-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[Post80] Postel, J. User datagram protocol. Request For Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug 1980.

[Post81] Postel, J. Internet protocol. Request For Comments 791, USC Information Sciences Institute, Marina del Ray, Calif., Sep 1981.

[Powe88] Powell, D., Seaton, D., Bonn, G., Verissimo, P., and Waeselynk, F. The Delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The 18th International Symposium on Fault-Tolerant Computing*, Tokyo, Jun 1988.

[Rama88] Ramanathan, P. and Shin, K. G. Checkpointing and rollback recovery in a distributed system using common time base. In *Seventh Symposium on Reliable Distributed Systems*, pages 13–21, Columbus, OH, Oct 1988.

[Rama90] Ramanathan, P., Shin, K. G., and Butler, R. W. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, pages 33–42, Oct 1990.

[Rand75] Randell, B. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, Jun 1975.

[Reed83] Reed, D. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, Feb 1983.

[Ricc91] Ricciardi, A. and Birman, K. Using process groups to implement failure detection in asynchronous environments. Technical Report TR 91-1188, Dept of Computer Science, Cornell University, Feb 1991.

[Russ80] Russell, D. L. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, Mar 1980.

[Schl83] Schlichting, R. and Schneider, F. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, Aug 1983.

[Schl90] Schlichting, R. D., Mishra, S., and Peterson, L. L. Fault-tolerance aspects of the Psync IPC mechanism. Technical Report TR 90-23, Dept of Computer Science, University of Arizona, 1990.

[Schn82] Schneider, F. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, Apr 1982.

[Schn87] Schneider, F. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Dept of Computer Science, Cornell University, Aug 1987.

[Schn90] Schneider, F. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[Shin84] Shin, K. G. and Lee, Y. H. Evaluation of recovery blocks used for checkpointing processes. *IEEE Transactions on Software Engineering*, SE-10(6):692–700, Nov 1984.

[Shri88] Shrivastava, S. K., Mancini, L. V., and Randell, B. On the duality of fault tolerant system structures. In Nehmer, J., editor, *Experiences with Distributed Systems*, volume 309. LNCS Springer-Verlag, 1988.

[Shri89] Shrivastava, S. K., Dixon, G. N., and Parrington, G. D. An overview of Arjuna: A programming system for reliable distributed computing. Technical Report 298, Computing Laboratory, University of Newcastle upon Tyne, Nov 1989.

[Siew82] Siewiorek, D. P. and Swarz, R. S. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford (MA), 1982.

[Sist89] Sistla, A. P. and Welch, J. L. Efficient distributed recovery using message logging. In *Proceedings of Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, Edmonton, Canada, Aug 1989.

[Somm89] Sommerville, I. *Software Engineering (Third Edition)*. Addison-Wesley, Wokingham, 1989.

[Srik87] Srikanth, T. K. and Toueg, S. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, Jul 1987.

[Stro85] Strom, R. and Yemini, S. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug 1985.

[Stro87] Strong, R., Skeen, D., Cristian, F., and Aghili, H. Handshake protocols. In *7th International Conference on Distributed Computing Systems*, pages 521–528, Berlin, Sep 1987.

[Tami84] Tamir, Y. and Sequin, C. H. Error recovery in multicomputers using global checkpoints. In *Proceedings of the 13th International Conference on Parallel Processing*, Aug 1984.

[USC81] USC, . Transmission control protocol. Request For Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sep 1981.

[Veri89] Verissimo, P., Rodrigues, L., and Baptista, M. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.

[Veri90] Verissimo, P. and Marques, J. Reliable broadcast for fault-tolerance on local computer networks. In *Ninth IEEE Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, AL, oct 1990.

[Verr90] Verrissimo, P. Real-time data management with clockless reliable broadcast protocols. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–24, Houston, TX, Nov 1990.

[Walt82] Walter, B. A robust and efficient protocol for checking the availability of remote sites. *Computer Networks*, 6(3):173–188, Jul 1982.