CONFIGURABLE FAULT-TOLERANT DISTRIBUTED SERVICES

(Ph.D. Dissertation)

Matti Aarno Hiltunen

TR96-12

July 11, 1996

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

This research was supported by NFS grant CCR-9003161, ONR grants N00014-91-J-1015, N00014-94-1-0015, and N00014-96-0207, and grants from Suomen Kulttuurirahasto and Suomen Akatemia.

CONFIGURABLE FAULT-TOLERANT DISTRIBUTED SERVICES

by

Matti Aarno Hiltunen

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 6

STATEMENT BY AUTHOR

SIGNED: ______________________________

ACKNOWLEDGMENTS

I wish to express my sincere thanks to my advisor Rick Schlichting. He was the ultimate reason for my decision to study at the University of Arizona, and I do not think I could have made a better choice. He has always been there to guide me on my quest to become a researcher, provide inspiration, ideas, and constructive criticism, as well as to attempt to teach me how to write succinctly. I thank Greg Andrews for introducing me to SR, which had a profound impact in this research, and both Greg and Larry Peterson for serving on my committee. I thank my minor advisors Sid Yakowitz and Duane Dietrich for teaching me the probabilistic nature of fault tolerance and reliability.

Others have contributed to this research. I thank Laura Sabel and Keith Marzullo for their comments and Flaviu Cristian for discussions that improved the work on membership services. I thank Jack Goldberg for providing the inspiration and encouragement for the work on adaptive systems. Finally, I thank Nina Bhatti, Wanda Chiu, Dorgival Guedes, Zhanliang Chen, Xiaonan Han, and Ilwoo Chang for collaboration on the event-driven approach. In particular, Nina has implemented the *x*-kernel prototype, and Wanda and Xiaonan have implemented RPC micro-protocols using this prototype.

I want to thank people whose encouragement were essential in getting me to start my doctorate studies. I thank Martti Tienari for rescuing me from the real world by hiring me as a research assistant to work on my Master's thesis. Under his supervision, I rekindled my interest in being a researcher. I thank Matt Mutka for advising me during me initial steps studying fault tolerance and for providing encouragement and invaluable information at the time I applied to graduate schools. I thank others who provided advice, in particular, Henri Tirri, Shmuel Zaks, the Finland-USA Foundation, and the Institute of International Education. Finally, I thank Matti Penttinen for taking care of all my affairs in Finland.

Many people have made my stay in Tucson delightful. I thank the department's systems administrators and office staff for keeping things running smoothly. In particular, I thank Wendy Swartz for guiding me through all the paperwork. I thank my present and past fellow graduate students Nina Bhatti, Wanda Chiu, Patrick Homer, Mudita Jain, Shivakant Mishra, Prasad, and Vic Thomas for making my time here more pleasant. I thank Charles Cantrell, Jim Snyder, Richard Steede, Mike Teta, and my other friends in Tucson for their help and camaraderie over the years, and my "Tucson mom" Thomasina Teta for many delicious Italian meals.

I thank my parents, sisters, and relatives for their love and support over the years, and especially, my mom who always encouraged me to go as far in my studies as I'm capable.

Finally, I want to thank the National Science Foundation (grant CCR-9003161), the Office of Naval Research (grants N00014-91-J-1015, N00014-94-1-0015, and N00014-96-0207), Suomen Kulttuurirahasto, Suomen Akatemia, and USA-Scandinavian Foundation for funding my research and studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Fault tolerance—that is, the ability of a system to continue providing its specified service despite failures—is becoming more important as computers are increasingly used in application areas such as process control, air-traffic control, and banking. Distributed systems, consisting of computers connected by a network, are an important platform for many fault-tolerant systems. Unfortunately, it is difficult to construct fault-tolerant distributed software, so communication services such as multicast, RPC, membership, and transactions have been proposed as simplifying abstractions. However, although numerous versions of these services have been defined, no single implementation provides a perfect match for all applications and all execution environments.

This dissertation presents an approach to constructing highly configurable fault-tolerant services. A new model is proposed where a service is composed out of micro-protocol objects, each of which implements an individual semantic property of the overall service. This makes it easy to construct different customized versions of a service with properties tailored to the specifics of an application. The model allows micro-protocols to cooperate using user-definable events and shared variables, making the model more flexible than existing approaches. Three prototype implementations of the model are also described.

In addition, a new approach is introduced for specifying abstract properties of services using temporal logic over message ordering graphs, which are abstract representations of collections of messages on each site. Furthermore, the problem of which combinations of properties or corresponding micro-protocols are feasible is addressed by defining relations that identify those combinations that result in a functioning service. Dependency and configuration graphs are presented as tools for constructing operational configurations.

This new approach is used to develop configurable membership and group RPC services. Furthermore, the system diagnosis problem is contrasted with membership, and new membership and system diagnosis algorithms are derived based on the observations. Finally, the dissertation presents an application of the event-driven model to adaptive systems that dynamically change their behavior as a result of changes in the execution environment or user requirements.

CHAPTER 1
INTRODUCTION

Computers and computer networks are playing an increasingly important role in society. For example, embedded processors control electronic household appliances, personal computers are used for work and entertainment, and supercomputers are used for weather forecasting. Computers are also increasingly connected by computer networks; these range from local-area networks connecting personal computers within a company to wide-area networks, such as the Internet, that connect large parts of the world. Such networks are the foundation for *distributed systems*, which are collections of computers that cooperate to provide a service. The worldwide telephone system is an example of such a system. Distributed systems are also used in process control applications to control manufacturing plants, power stations, and airplanes. In these cases, each physical component in the application is controlled by one computer, and the collection of computers is connected by a network to support cooperative control over the whole system.

Using computer systems to control airplanes and power stations requires high levels of *dependability*, which means that reliance can justifiably be placed on the service it delivers [Lap92]. One important aspect of dependability is *fault tolerance*, which means the ability of the system to continue providing its specified service despite component failures. The relative importance of fault tolerance varies among application domains, largely because the consequences of failures can range from simple inconvenience to significant financial hardship or potential loss of life. Moreover, fault tolerance is important in distributed systems. For one thing, because of the large number of computers in some of these systems, the probability of at least one failing becomes large; therefore, fault-tolerance techniques are often required to ensure that these failures do not stop the entire system. On the other hand, the autonomy of computers in such systems means that failures are often independent; hence, a fault-tolerant service can be constructed on a distributed system by replicating the service on more than one computer so that it remains available as long as at least one machine is functioning.

The goal of this research is to simplify the difficult task of writing fault-tolerant distributed applications by providing powerful underlying communication services. A number of these services, such as *atomic multicast* [CASD85], have been proposed and implemented. Although such services are useful, each provides a fixed set of guarantees even though different applications often have different requirements. A way to solve this mismatch is *customization*, that is, the construction of specialized versions of a service that match the requirements of each application. Customization is not feasible unless the construction of specialized versions is easy and the resulting service efficient. Our solution to easy customization is *configurability*, that is, the construction of the customized version from prefabricated modules, each of which implements a portion of the service guarantees. A *configurable service*, then, is a service that can be customized in this manner. In this dissertation, we present a new approach to constructing configurable fault-tolerant distributed services in which fine-grained modules called *micro-protocols* can be configured together to give different variants of the desired service.

1.1 Fault Tolerance and Distributed Systems

Fault, failure, and error are the fundamental concepts of fault tolerance [Lap92]:

- A *failure* occurs when the delivered service of a system or a component deviates from its specification.
- An *error* is the part of the system state that is liable to lead to a failure.
- A *fault* is the hypothesized cause of an error.

An error is thus the manifestation of a fault in the system, while a failure is the effect of an error on the service. Therefore, a *fault-tolerant system* is one that continues to provide service without failures in spite of faults. A fault here can be, for example, a bug in the software, a hardware problem, a user error, or a power failure. Note that the failure of one component can be the cause—that is, the fault—that results in the failure of another component. For example, a failure in the electrical supply from the power company can be the fault that causes a computer to fail.

Fault tolerance is based on *redundancy*, which is the use of extra resources to detect, correct, or mask effects of faults. *Error detecting codes* are an example of redundancy where additional bits of information are added to data, such as memory, disks, or messages transmitted over a network, to detect errors [BM86, Toh86]. Redundancy can be divided into *time* and *space redundancy*. Time redundancy is based on using extra execution time, whereas space redundancy is based on using extra physical resources, such as extra memory, processors, disks, or communication links.

Fault-tolerance techniques implemented in software are often divided into *fault-tolerant software* and *software fault tolerance* based on which faults the techniques attempt to tolerate. Fault-tolerant software is designed to tolerate failures in the underlying layers, such as the underlying hardware platform or lower level operating system and network services. In contrast, software fault tolerance is designed to tolerate software bugs in the implementation of a service or application itself. The emphasis in this dissertation is on fault-tolerant software. Note, however, that these techniques are not disjoint. For example, software fault tolerance can sometimes tolerate failures in underlying layers, while fault-tolerant software can sometimes tolerate effects of software bugs. Both techniques apply both time and space redundancy to achieve their goals.

1.1.1 Measures of Dependability

Important measures of system dependability are *availability* and *reliability*, which are defined as follows [Kec91]:

- *Availability*, in particular *instantaneous availability*, is the probability that a system will be available for use at any random time t after the start of operation.
- *Reliability* is the probability that parts, components, products, or systems will perform their designed-for functions without failure in specified environments for desired periods at a given confidence level.

For example, for phone service availability is the probability that a phone call can be connected at any given time, and reliability the probability that the connection can be maintained until the caller terminates the call. For an airplane, reliability is the probability that a flight reaches its destination without an accident, while for a nuclear power station, reliability is the probability that the station provides power at a given level for a specified number of years without releasing more than a specified level of radioactivity. The reliability of a system is the product of the reliabilities of all the elements in the system. This includes not only the computer systems, but also the people working with the system and other mechanical components. This dissertation naturally concentrates on the computer aspect of the problem.

Providing 100% reliability and availability would be ideal for all systems and services, but unfortunately this goal is unattainable because it is impossible to tolerate all possible failures. Furthermore, increasing the reliability and availability of a system typically increases the cost of the system. Therefore, goals are usually set for reliability and availability that depend on the purpose of a system. For example, for the design of the AAS (Advanced Automation System) air-traffic control system, the availability goal of critical components was established at no more than 3 seconds of down-time a year [CDD90]. For telephone switching systems, the availability goal is 99.4%, which translates to 3 minutes of system down-time a year, and the reliability goal is to sustain at least 99.9875% of all established calls, which translates to inadvertently disconnecting no more than 1.25 calls out of 10,000 [Kec91]. In the manufacturing industry, information about the reliability of a product is used, for example, to determine the optimal guarantee periods. Finally, reliability and availability are even beginning to be used to advertise consumer goods and services. For example, long distance phone companies now advertise how quickly 1-800 service can be restored after link failures.

1.1.2 Failure Models

In a distributed computing system, it is impossible to tolerate all failures, because given N machines, there is always a nonzero probability that all will fail, either independently or due to a common cause. Therefore, the goal of fault tolerance is to improve the reliability and availability of a system to a specified level by tolerating a specified number of selected types of failures. *Failure models* have been developed to describe abstractly the effects of failures. The use of such a model simplifies the programmer's task by allowing the program to be designed to cope with this abstract model rather than trying to cope with the different individual causes of failures. Note, however, that the failure model used while writing a program is just an assumption about how components can fail. A part of the system reliability evaluation, then, is to determine if the failure model covers a large enough percentage of the expected failures for the system to satisfy its reliability goals.

A hierarchy of failure models has been developed for use in different application areas. The broadest failure model is the *Byzantine* or *arbitrary* failure model where components fail in arbitrary way [LSM82]. This model accommodates all possible causes of failures, including malicious failures where a machine actively tries to interfere with the progress of a computation. Naturally, the algorithms based on this failure model are extremely complicated and expensive to execute. The *Byzantine with authentication* failure model allows the same type of failure, but with the added assumption that a method is available to identify reliably the sender of any message. This assumption substantially simplifies the problem. The *incorrect computation* failure model is one where a component, given a correct input, produces an incorrect output [LMJ91]. The *timing* or *performance* failure model assumes a component will respond with the correct value, but not necessarily within a given time specification [CASD85]. The *omission* failure model assumes a component may never respond to an input [CASD85]. The *crash* or *fail-silent* failure model assumes that the only way a component can fail is by ceasing to operate without making any incorrect state transitions [PSB$^+$88]. Finally, the *fail-stop* failure model adds to the crash model the assumption that the component fails in a way that is detectable by other components [SS83]. A more thorough classification of failure models and their relations can be found in [Pow92]; numerous other classifications based on factors such as duration and cause have also been proposed [Lap92]. In general, the more inclusive the failure model, the higher the probability that it covers all failures that are encountered, but at a cost of increased processing time and communication.

1.1.3 Distributed Systems

As already noted, distributed systems and fault tolerance are inherently linked. For one thing, the potentially large number of machines in a distributed system makes the probability of at least one component failing large, which could lead to unavailability of the service. Consider a system of n machines where the failure probability for each machine during a system execution of T time units is p. Assuming independence of failures, the probability that none of the machines fail in time T is $(1-p)^n$. For example, if $n = 70$ and $p = 0.01$, the probability that the system can operate without a single failure for time T is only 0.49, i.e., on the average more than half of the executions of time T will fail. Although the reliability of modern computers is high, a distributed or parallel system may consist of hundreds or thousands of machines, making the probability of at least one failure large enough that the reliability is not satisfactory for many applications.

On the other hand, distributed systems possess inherent potential for fault tolerance, because the failure of some number of machines need not lead to unavailability of the service. Although the probability of at least one machine failing can be high, the probability of *all* the machines failing can be remarkably small. In particular, the probability that all components fail is p^n, which even in the case of moderately reliable components and moderate number of computers, can be extremely small. For example, given p = 0.1 and n = 4, the probability of all components failing is only 0.0001. This makes distributed architectures attractive for building applications that are required to provide highly reliable and available services. A distributed architecture by itself is, of course, only a starting point; fault-tolerance techniques need to be applied so that the application can continue operating despite the failures. An example of the use of distributed architecture for fault tolerance is the previously mentioned AAS air-traffic control system [CDD90]; the high availability goals are to be met by replicating critical applications on 3 or 4 computers structured as a distributed system.

1.2 Building Fault-Tolerant Distributed Software

Although fault tolerance is important, constructing distributed fault-tolerant software is hard. For one thing, programmers have to deal with the typical complications of concurrency, such as coordinating the use of shared data and other shared resources. Furthermore, the execution environment can be dynamic, often requiring the software to adapt its behavior; for example, response times or throughput of the underlying network may fluctuate due to congestion or changes in the load. Finally, various types of faults ranging from benign ones, such as a message getting lost in the network, to more severe ones, such as computer crashes, may occur. Since the faults may occur at arbitrary times during the execution of the software, designing the software so that it behaves correctly at all times is a significant challenge.

Different approaches have been proposed to simplify the development of this type of software. This section describes two of the most significant: use of a layered software structure and use of simplifying service abstractions. It also discusses the wide range of execution properties guaranteed by such service abstractions and the implications of this variety on applications.

1.2.1 Layers of Abstraction

The construction of complicated computing systems is often simplified by describing or implementing them as a hierarchy of separate layers. This approach has been used frequently in operating systems, where the layers range from hardware to the user interface. For example, the THE operating system was designed with six layers: hardware, CPU scheduling, memory management, operator console device driver, buffering for I/O, and user programs [Dij68]. This approach also has a traditional role in communication systems. For example, the seven layer ISO OSI model consists of physical, data link, network, transport, session, presentation, and application layers [DZ83].

In the layered approach, each layer provides a *service* to the layer above it using one or more *protocols* to implement the service. A service is defined by the set of operations that the layer supports, while a protocol defines one implementation of the service. For example, in the case of communication protocols, the protocol defines the rules governing the format and meaning of the messages that are exchanged by peer layers residing on different machines to implement the service [Tan88]. Naturally, a service may be implemented using a number of different protocols as long as each can guarantee the specified service. The term service is also often used to denote the software that provides the particular service. For example, by "membership service", we also mean the software that provides this service.

Although the service provided by a layer is defined in terms of concrete operations, each layer can also be seen as providing an abstraction of the system on which the higher layers are built. For example, the memory management layer in an operating system often provides the abstraction of virtual memory, which allows higher layers to assume the existence of a homogeneous segment of memory of arbitrary size. Similarly, in networking, the network layer of the OSI model provides the abstraction of a communication link between sender and receiver that in actuality may not be connected by one physical link. Thus, a system can be viewed as consisting of layers of abstractions.

1.2.2 Service Abstractions

A number of useful services and abstractions for distributed computing have evolved over the years. One of the most widely used is the TCP/IP protocol [Jac88], which provides the abstraction of a reliable pipe between two computers where bytes are delivered in the same order as they were sent. To implement reliable pipes, TCP/IP has to deal with messages that are corrupted, lost, or reordered by the underlying network, as well as issues such as congestion and resource control related to multiplexing the physical network. The abstraction of reliable communication hides events that designers would otherwise have to deal with, thereby simplifying development.

Although reliable communication is essential for building fault-tolerant distributed applications, more powerful services, often called *middleware* [Ber96], can further simplify the task. For example, *an atomic ordered multicast* or *broadcast* service with atomicity and message ordering guarantees makes it easy to send messages to a collection of processes [CM84, CZ85, CASD85, VM90, BSS91]. Similarly, a *membership* service provides consistent information about which computers are functioning and which have failed at any given time in a distributed system, thereby simplifying the problems associated with failures [Cri91, Bir85a, EL90, KGR91, MPS92]. A *time* service provides consistent information about time in a distributed system, either in the form of logical or virtual time that can be used to reason about the relative order of events [Lam78, Sch82, Mat89] or real time from synchronized clocks [KO87, WL88, RSB90, VR92]. Other important service abstractions are *atomic actions*, a collection of operations whose execution is indivisible despite concurrency and failures [Lam81, Lis85, SDP89]; and *stable storage*, storage whose contents is guaranteed to survive failures [Lam81, BY87].

Paradigms for structuring fault-tolerant software further simplify developing applications. The *replicated state machine approach* is a paradigm for building fault-tolerant services using replication [Sch90]. A service is constructed using a collection of identical deterministic state machines, with client requests being sent to all replicas for execution using atomic ordered multicast. This approach is an example of *active replication*, where every replica executes the same operations. In the *primary/backup paradigm*, only one of the replicas actively executes client requests [AD76, BJRA85, BMST92], with the state of the other backup replicas being updated periodically. This approach is an example of *passive replication*. In the *object/action paradigm*, the system is constructed of passive objects that export actions, i.e., operations, that modify the state of objects [Gra86]. Applications of this approach to reliable computing are discussed in [Whe89]. In all these paradigms, communication-oriented services such as multicast and membership are key components of the supporting infrastructure.

1.2.3 Properties of Services

Existing services can be characterized by the execution guarantees, or *properties*, they provide to their users. Properties can be defined in terms of constraints on the service, that is, if a service satisfies the constraints of property p, then we say that it has property p.

Properties can be illustrated using the atomic ordered multicast service in [CASD85] as an example. This service has the following properties:

- *Termination*: Every message broadcast by a correct sender is delivered to all correct receivers after some known time interval.
- *Atomicity*: Every message whose broadcast is initiated by a sender is either delivered to all correct receivers or to none of them.
- *Total order*: All messages delivered from all senders are delivered in the same order at all correct receiving nodes.

Consider a distributed application that builds on this service. The application consists of software components, here called *nodes*, executing on different computers in a distributed system. Let one node transmit a message to the others using an atomic multicast service. Due to the properties of the service, a node receiving this message has significant information about the distributed state of the application, independent of the message contents. For one, because of the atomicity property, it knows that all other correct nodes have already received or will receive the same message. Thus, if this message causes a state change in the node, it knows that this change will occur at all other nodes as well. Furthermore, because of the termination property, it knows that this change will occur within a known time bound. Finally, if two or more nodes send a multicast at approximately the same time, these messages will be delivered in the same order on all nodes. Therefore, the corresponding state changes occur in the same order.

Other multicast or broadcast services provide different sets of properties. The V-system [CZ85] provides only unreliable multicast. Consul [MPS93a, MPS93b], Isis [BSS91], and Transis [ADKM92b, DM96] provide multicast with the atomicity property and various ordering properties, but no timeliness properties. The Mars system [KO87, KDK$^+$89] provides atomicity, ordering, and timeliness properties. The existing multicast implementations cover only a small subset of all possible combinations of multicast properties, however. We can easily identify half a dozen different ordering properties and a few different variants of reliability, atomicity, and timeliness properties, which could be combined in dozens of different ways.

The properties defined for transactions are another example of abstract properties of a service. A transaction is a collection of operations that is executed as a unit despite concurrency and potential failures during the execution. Typically, transactions have the following four, so-called ACID, properties [HR83, Bla91]:

- *Atomicity* or *All-or-nothing*: Either the transaction completes or it has no effect, despite failures of some of the components involved in the transaction.
- *Consistency*: A transaction takes the database from one consistent state to another.
- *Isolation*: The intermediate states of the data manipulated by a transaction are not visible outside the transaction.
- *Durability* or *Permanence*: The effect of a transaction that has completed will not be undone by failure.

In essence, the ACID properties guarantee that each transaction is executed on what appears to be a dedicated system, even though in reality a number of other transactions may be executing concurrently. The ACID properties are a good example of abstract properties expressed in terms of their effect on the application, instead of how they are implemented.

In contrast with multicast, the ACID properties are relatively standard for transactions in most database systems. Naturally, different systems use different protocols to realize the properties, mostly for performance reasons. However, when database systems are used in certain specific application areas, such as CAD, engineering, and artificial intelligence, some of the ACID properties have been found to be too restrictive [BBG$^+$88]. Similarly, subsets of the ACID properties have been found to be useful for transactions in an operating system context [SMK$^+$94]. Extensible database systems, where the properties or their implementations can be adjusted to specific application requirements and execution environments, have been developed to address some of these needs [SR86, SCF$^+$86, CD87, BBG$^+$88].

1.2.4 Meeting Application Requirements

Different fault-tolerant distributed applications have different requirements for the underlying services, so they should be able to select which properties are enforced. Unfortunately, a given service implementation typically provides only a fixed set of guarantees, which may or may not be appropriate for the application. For example, consider building a distributed application using atomic ordered multicast. First, if the order in which messages are received is commutative—that is, processing messages in any order leads to the same result—the ordering property is not required. Second, if the application has no timeliness requirements, the time bound guarantee is not necessary. Finally, if the application can tolerate missing a message occasionally, as is often the case with video and sound, the atomicity and termination properties are not required either. Similar observations can be made with regard to transactions, as already noted above.

Building a service that incorporates all possible properties, and therefore satisfies all applications, is not viable for two reasons. The first is the execution cost associated with each property. Implementing a property typically requires some combination of processor time, extra messages, and synchronization time, thereby slowing down the progress of the application. This cost can be large, even orders of magnitude. As an example, consider a token-based total-ordered multicast in a distributed system consisting of N computers connected by an Ethernet. Provided that the network is not congested, an unordered multicast takes $O(1)$ time, whereas an ordered multicast may take up to $O(N)$ time since the sending site has to wait for the token. As a result, if a distributed application that does not require the ordering property has to use this service, its response time can be severely affected.

The second reason is the tradeoff between properties. Sometimes, particularly in the presence of failures, there are properties such that it is only possible to guarantee any one them but not all at the same time. For example, with transactions operating on distributed replicated data in a partitioned network, it is impossible to guarantee simultaneously the progress of transactions on all sites and consistency of the data. Furthermore, sometimes guaranteeing one property has a negative influence on another. For example, the implementation of a timeliness property typically requires preparing for the worst case and therefore often introduces extra waiting, affecting properties such as response time. Both these cases again illustrate the fundamental difficulties in attempting to enforce all properties with a single all-encompassing service.

Without a universal service, the user is forced to try and choose the best combination of properties from among existing implementations. Unfortunately, this is not a viable option either. First, because services are implemented on a unique combination of hardware, operating systems, and communication software, porting them to the platform where the application will be executing can be very difficult. Second, even if porting is possible, a service built for one particular environment may have bad performance when used in another. For example, an atomic multicast protocol may be fine for a local-area network where the failure rate is low and transmission latency short, but unsuitable in a wide-area setting where the failure rate and latency are much higher. Finally, if the failure model assumed for an implementation turns out to be unsatisfactory for the application's real execution environment, it is often virtually impossible to change without a complete rewrite.

1.2.5 Customization and Configurability

A way to solve the mismatch between application requirements and service guarantees is customization using configurable or extensible services. A *configurable service* is one that allows different versions of the service to be easily constructed to match application requirements. The configuration process may take the form of constructing an instance of a service from a set of chosen modules, or it may be based on simply setting compilation or runtime flags that cause the service to choose a certain operating mode. An *extensible service* is one where the functionality of the service can be augmented with either preprogrammed or user-provided modules. The terms configurable and extensible can in most cases be used interchangeably, because configurable services are typically extensible and extensible services are by definition configurable. The customization of a service can occur at compile or link time or dynamically during execution. In the latter case, we call it an *adaptive service*.

Extensibility has been explored in the database community since the early 1980s, and has recently started to gain interest in the operating systems and networking communities. In databases, well-known examples include Genesis [BBG+88] and Raid [BFHR90]. In operating systems, examples of configurable or extensible systems are Synthesis [PMI88], SPIN [BCE+94], Scout [MMO+94a], and V++ [CD94]. Also, in file systems, configurability has been used to enable new application types, such as databases and multimedia, to make use of file systems efficiently for their storage needs [HP94, KN93, Maf94]. In networking, the *x*-kernel [HP91] and Adaptive [SBS93] are examples of systems that support construction of customized communication protocols out of modules. All these systems are described further in chapter 2.

Although numerous projects have investigated issues related to communication services for fault-tolerant distributed computing, only a few have explored modularity and configurability. The most notable of these are the Consul and Horus systems. Consul is a collection of protocols developed for implementing fault-tolerant distributed programs based on the state-machine approach [MPS93a, MPS93b]. It provides support for ordered multicast, membership, and recovery, where the different services are implemented as modules using the *x*-kernel. Although configurable, the different choices available for the application builder are limited to different message-ordering properties. The Horus system provides similar services for distributed applications but with a higher degree of configurability than Consul [RBG+95, RB95, RBM96]. Both systems are oriented around hierarchical composition of code modules.

1.3 A New Model for Configurable Fault-Tolerant Services

This dissertation presents a new approach to constructing configurable fault-tolerant services. In this approach, a service is constructed of fine-grained modules called *micro-protocols* that implement the abstract properties of the service. Micro-protocols are executed using an *event-driven execution model*, which provides a sophisticated and flexible means of interaction between micro-protocols. In this model, each micro-protocol consists of a set of event handlers, which are bound to events such as arrival of a certain type of message from the network. Then, when a given event occurs, all handlers bound to that event are executed. In addition to providing predefined events that are raised by a runtime system, our approach provides user-definable events that are raised by micro-protocols, thereby providing for interaction between modules. Micro-protocols can also interact through shared data structures.

Given customized services built using micro-protocols, a system is constructed by combining these services hierarchically. This results in a two-level view of system construction, where services are constructed from micro-protocols and systems are constructed from services. For example, in a prototype implementation based on the *x*-kernel, micro-protocols are used to construct services, which are then combined with normal *x*-kernel protocol modules using the facilities of the *x*-kernel for hierarchical composition.

The combination of fine-grained modules and flexible interaction mechanisms make it possible to build configurable services that it would be difficult to build using traditional hierarchical models such as those used in Consul or Horus. Moreover, the model does not restrict how events and shared data are used and in particular, does not enforce a standard fixed interface between modules. As a result, the model supports various types of relationships between modules, ranging from hierarchical relationships to peer relationships to any combinations of these. This enables the logical relationship between modules to be preserved in the implementation, which facilitates the design and implementation of fault-tolerant services.

To design a configurable service using our approach, a necessary first step is to identify the individual abstract properties of the service. Unfortunately, forming a consistent set of abstract properties for a given service is a nontrivial task despite the large number of existing implementations. One problem is that the properties are often described in terms of a particular implementation strategy and may be difficult to translate into more abstract properties. Another is that different researchers use different methods and terminology for describing properties. As a part of our work, we have identified and defined sets of properties for group remote procedure call (RPC) and membership services.

Although the main emphasis of our work is on matching service guarantees to application requirements, configurability has two other applications that are addressed in this dissertation. First, it can be used to improve service performance by choosing the implementation of a property that best fits the conditions in the current execution environment, including issues such as network failure rates and communication latency. Second, configurability can be used to match the reliability guarantees of a service to the reliability requirements of an application by adjusting the failure model of a service. Configurability provides an elegant solution for both these problems.

The major contributions of this dissertation are:

- A two-level approach to system construction, where a system is constructed from services and each service is constructed from a collection of modules corresponding to abstract properties.
- A new and more flexible model for constructing highly configurable fault-tolerant distributed services and prototype implementations of this model.
- The design and prototype implementations of membership and group RPC services that offer high levels of configurability.
- Identifying, defining, and illustrating the abstract properties of fault-tolerant distributed services, such as membership and group RPC, and identifying of the relations between these properties.

Other contributions include a general model for adaptive systems and a corresponding implementation based on the event-driven approach. This work on adaptive systems provides an example of using configurability to improve the performance and reliability of a service by adjusting the implementation to the characteristics of the environment. We have also identified the relationship between the membership and system diagnosis problems, and utilized this relationship to develop new algorithms for both.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes related work in the areas of fault tolerance and configurable systems. It provides information about approaches that have been used to construct configurable services and serves as a foundation for comparing our approach to others.

Chapter 3 presents our approach to constructing configurable services. We first identify and define the properties of a service and then present a configurable implementation using the event-driven execution model. We also examine issues that affect which combinations of properties and micro-protocols result in operational services. Finally, we briefly describe three prototype implementations of the model: one using the SR programming language [AO93, AOC$^+$88], a second using the *x*-kernel, and a third using C++.

Chapter 4 identifies and formally specifies the abstract properties of membership services, as well as identifies and proves the relations between these properties that dictate which combinations of the properties are feasible in any implementation of a membership service. A number of well-known membership services are then characterized based on the set of abstract properties.

Chapter 5 describes a configurable implementation of membership. We outline the implementation strategy, the events and shared data structures, and the design of the micro-protocols. The set of 25 micro-protocols presented allows over 1000 different membership services to be configured. We also discuss the prototype implementation of the service.

Chapter 6 introduces the system diagnosis problem, a problem closely related to membership but traditionally treated separately. We contrast these two and conclude that they are closely related with the main differences being in the failure model assumed. Based on these observations, we derive new membership algorithms by changing the failure model of typical distributed system diagnosis algorithms.

Chapter 7 describes the application of our approach to another service, namely group RPC. We identify the abstract properties of this service and outline a highly configurable implementation.

Chapter 8 describes another application of the event-driven model: implementing adaptive systems that change their behavior during execution based on changes in the execution environment. We introduce a general model for adaptive systems, apply the model to a number of examples, and then outline how the event-driven execution model can be used as an implementation tool.

Chapter 9 summarizes the dissertation and offers future research directions.

CHAPTER 2
RELATED WORK

A number of other projects have investigated topics related to our work, including aspects of fault tolerance and configurability. First, we review distributed fault-tolerant systems that provide platforms or tools for building highly dependable applications. We introduce systems ranging from non-configurable, or *static* for short, to configurable. The prior work on fault-tolerant systems has served as the inspiration and the starting point for our work, while the work on configurable systems provides a measuring stick for success. Second, we describe work on configurability in other areas, such as networking, operating systems, file systems, and database systems. These systems provide more background on approaches that have been applied to configurability elsewhere. Finally, we summarize work in this area by characterizing the different approaches to configurability based on the structuring techniques used.

2.1 Fault-Tolerant Systems

A number of systems have been developed that facilitate the construction of fault-tolerant applications by providing useful services such as reliable multicast, membership, or transactions. Here, we examine the basic outline of some of these systems, without addressing in detail any of the particular services provided. Since the emphasis is on configurable systems, only two static systems are described: Isis [Bir85a, BJ87, BC91, BSS91, BR94] and Mars [KM85, KDK+89, KG94]. The configurable systems presented illustrate different approaches taken towards customization in fault-tolerant computing. A large number of other projects are not addressed in detail here, including ADS [IM84], Amoeba [RST89, KT91], AMp and xAMp [VRB89, RV91, RV92], Argus [Lis85, Lis88], Avalon [DHW88], Chorus [BFG+85], Clouds [DLA88, DLAR91], Delta-4 [PSB+88, BHV+90, Pow91], Saturne [DFF+90], Totem [AMMS+93, AMMS+95, MMSA+96], Transis and Lansis [ADKM92b, ADKM92a], and the V kernel [CZ85].

2.1.1 Static Systems

2.1.1.1 Isis

The Isis system is a toolkit developed to support construction of distributed applications, including those that have fault-tolerance requirements [Bir85a, BJ87, BC91, BSS91, BR94]. The toolkit is structured around the concepts of a *process group*, a group of processes cooperating to implement a service, and *virtual synchrony*, an abstraction that enables the writing of applications as if the execution of the system was synchronous. Essentially, virtual synchrony guarantees that events, such as message arrivals or membership changes, are received in a consistent order by all group members.

Isis provides a set of services that simplify the construction of distributed applications. The multicast services in Isis range from unreliable to totally ordered reliable multicast. The membership service of Isis provides information about the current membership of the group, including a ranking of the processes in the group based on how long they have been group members. This ranking can be used, for example, to choose a leader to execute certain tasks for the group. Processes are allowed to join and leave groups at will, and a process can create new processes, for example, to take the place of a failed process. When a new process joins a group, the current state of an existing group member is typically transferred to the joining process. To do this, the programmer provides routines for packing the state into messages and subsequently unpacking it at the joining process. Isis guarantees that the state transfer is atomic with respect to application messages sent concurrently with the transfer.

Process groups in Isis can be used in many ways. First, groups can be used to increase the fault tolerance of an application. This can be done either by structuring the group as an *actively replicated process group*, where every process executes and responds to every request sent to the group, or by structuring it as a *passively replicated process group*, where only a *primary* process responds to every request. Second, the group can be used to improve the performance of a computation by dividing the work associated with a request among the group members. Finally, various combinations of these approaches are possible. For example, each process can maintain the replicated state of the server group (fault tolerance), but when a computationally intensive request is received, the task can be divided among the group members (performance) as long as state changes are distributed to all members after the computation is finished. The services provided by Isis are not configurable, but the system includes a collection of multicast services with different ordering and reliability semantics.

2.1.1.2 Mars

Mars is a system for building fault-tolerant distributed real-time applications [KM85, KDK$^+$89, KG94]. Mars is targeted for hard real-time applications, where missing a deadline can be catastrophic. Therefore, Mars places extremely strict requirements on the timeliness of communication and task execution, as well as the overall reliability of the system. Mars provides various services such as synchronized clocks, atomic multicast, and membership services. An application design system has also been developed based on Mars. This system provides tools for designing applications, and for doing performance, dependability, and timing analysis.

Mars is built using a physical ring architecture in which the communication medium is accessed using a time division multiplexing access (TDMA) strategy based on synchronized local clocks. That is, access to the physical medium is divided into dedicated time slots that are allocated *a priori* to sites in a round-robin fashion. A *TDMA cycle* is defined as *N* consecutive slots, where *N* is the number of sites in the system.

Mars uses a wide array of techniques to increase the fault tolerance of applications to desired levels. Communication failures are masked by transmitting each message k times, either in parallel over k redundant rings or sequentially over a single ring. The exact value of k depends on the assumptions about the underlying network and the reliability requirements of the application. Computation failures such as crashes are tolerated by executing each task on redundant computers and/or executing each task more than once on a single machine [Kea90]. Configurability in Mars is limited to adjusting the degree of communication and computation redundancy to reach the dependability goals.

2.1.2 Configurable Systems

2.1.2.1 Consul

Consul is a collection of communication services developed for implementing fault-tolerant distributed programs based on the state machine approach [MPS93a, MPS93b]. To support the replicated processing implied by the state machine approach, Consul contains protocol objects (i.e., software modules) that allow user operations to be multicast to a set of replica processes reliably and in some consistent order, to reach agreement on failures and recoveries, and to facilitate replica recovery.

Figure 2.1 illustrates the specific protocols in the system and how they are configured using the *x*-kernel (see section 2.2.1 for more on the *x*-kernel). Psync [PBS89] realizes the functionality of a partially (or causally) ordered reliable multicast, while Order transforms that into either a total or semantic-dependent order. A membership service is implemented by the combination of Failure Detection, which monitors the message flow and triggers an alarm if it suspects that another machine has failed, and Membership, which implements an agreement algorithm to decide if a failure has indeed occurred. Recovery implements replica recovery using a combination of checkpoint and message replay.

Part of the motivation for building Consul was to determine whether it was possible to build a fault-tolerant system in a modular manner using traditional network protocol composition techniques. As can be seen from the figure, the system is indeed constructed from well-defined modules, each of which implements a specific function. Consul is also configurable, although the choice is limited to different message ordering properties. In particular, causal order is provided by Psync, while total and semantics-based orders are provided by additional protocols.

The Consul project was successful in demonstrating the feasibility of modular construction of fault-tolerant services and provided configurability features beyond most similar systems at that time. Another important contribution, however, was in identifying the limitations of the *x*-kernel for construction of communication services for fault-tolerant systems [MPS93b]. First, the *x*-kernel defines a simple interface between protocols that includes only operations for opening and closing connections, and for sending and receiving

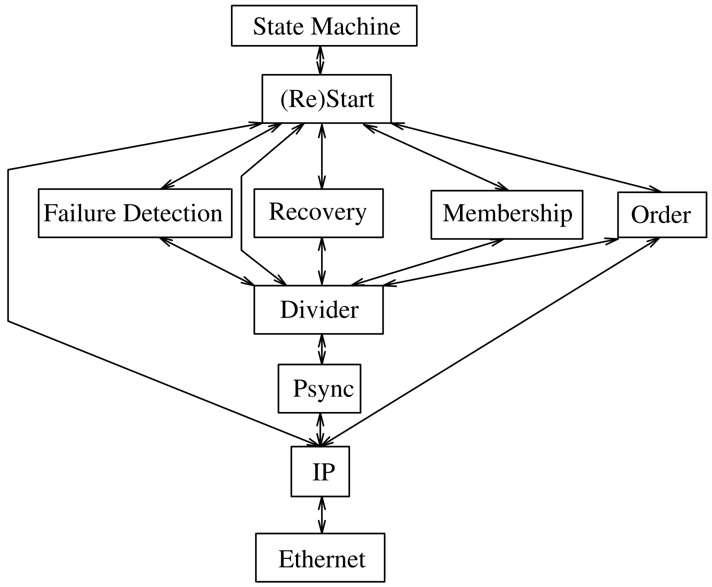


Figure 2.1: Consul Protocol Graph

messages; additional operations, if needed, must be encapsulated as *control operations*. Such an interface is reasonable for traditional network protocols, which have limited interactions with one another, but overly restrictive for protocols for fault tolerance, which interact much more closely. As a result, control operations are overloaded in Consul, making the system correspondingly more complicated and difficult to understand.

Second, the *x*-kernel is designed primarily to support hierarchical composition of protocols, where each protocol only interacts with protocols that are immediately above it or immediately below it in the protocol graph. In Consul, however, several protocols at the same logical level of the system must cooperate to implement services. For example, Membership, Order, and Failure Detection all cooperate, but without being related hierarchically. To find a way around this limitation in the *x*-kernel model, Consul has two additional protocols, (Re)Start and Divider, whose only function is to coordinate such interactions. Psync also serves a function in this regard by reflecting messages originating within protocols and destined for the network up to the other protocols. For example, the Failure Detection protocol notifies Membership about suspected membership change by multicasting a special message using Psync to all sites where it will be received by, among others, the Membership protocol objects. Similar problems were encountered in a project attempting a modular implementation of the xAMP atomic broadcast protocol suite in the *x*-kernel [RV92, Fon94].

2.1.2.2 Horus

Horus [RHB95, RBG^{+}95, RB95, RBM96] is a successor to the Isis system [Bir85a]. It is generally targeted for the same type of applications as Isis, but adds extensibility and configurability. The composition model in Horus is *linear*, that is, a system is constructed as a stack of protocols.

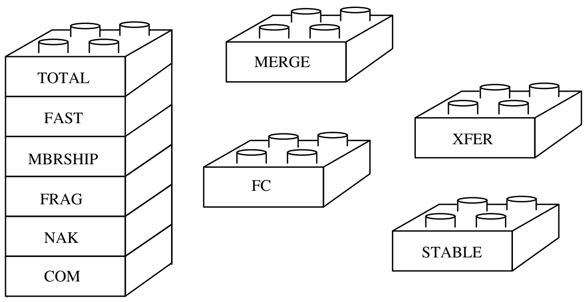


Figure 2.2: Horus Protocol Stack

Figure 2.2 gives an example of a Horus protocol stack. Conceptually, Horus protocols can be stacked at runtime like Lego^{tm} blocks. In this figure, the functions of the protocols are the following. COM provides unreliable communication over a low-level network of choice, such as ATM or Ethernet. NAK provides FIFO ordering using sequence numbers, FRAG provides fragmentation and reassembly of large messages, and MBRSHIP handles membership changes in a manner that guarantees virtual synchrony. FAST optimizes the transportation of messages from the application to the network by allowing these messages to bypass some of the underlying layers, both on the way from the application to the network and from the network to the application. This facility is used to improve the performance of communication during normal operation when no membership changes are occurring. TOTAL guarantees total order for messages sent within a group. STABLE provides information about message stability, where a message is *stable* if it has been received by every group member. FC implements flow control. MERGE locates group members by periodically sending a broadcast message in each partition. XFER implements state transfer for a process joining a group and the state merge that occurs when two partitions of the same group are joined. A large number of other protocols exist, including ones that implement clock synchronization, remote procedure call, message logging, and encryption.

The composition model is strongly motivated by that of *x*-kernel [RB95], with the issue of the limited interface between protocols being addressed by expanding the standard interface to 16 downcalls and 14 upcalls. Examples of downcalls are *cast* for multicasting a message, *send* for sending a message to a subset of group members, *view* for installing a new group view, *merge* for merging two views, and *join* for requesting addition to a specified group. Examples of upcalls are CAST for receiving a multicast message, SEND for receiving a message sent to a subset of members, LEAVE for leaving a group, VIEW for installing a new view, and JOIN_REQUEST for requesting to join a group. Note that, although all these calls are not of interest to all protocols, every protocol must support them since the operations are part of the standard interface. Also, while the standard interface syntactically allows protocols to be stacked in any order, most protocols require certain semantics from protocols below, imposing a partial order on the possible configurations [RHB95].

2.1.2.3 ANSA

ANSA (Advanced Networked Systems Architecture) is a software architecture for building distributed systems [Her89, Tea91, Her94]. ANSA provides a general computational model that defines how objects in a distributed system are specified and how objects interact. It also defines a set of services, such as communication, transaction, and security services, that an implementation of the ANSA model has to provide to the application designers. The ANSA model simplifies the construction of fault-tolerant applications by providing replicated servers and transactional techniques. ANSA supports customization of services by defining a collection of *transparency services* that hide different aspects of the distributed execution environment from the application, thereby making it easier to write applications. The transparency services defined by ANSA are the following:

- *Access transparency* hides the difference between accessing a local server and a remote server.
- *Location transparency* hides the physical location of clients and servers, enabling their physical location to change between invocations.
- *Migration transparency* hides the effect of servers moving between machines while interacting with clients.
- *Concurrency transparency* hides the effects of concurrent calls from several clients being processed at a server at the same time.
- *Failure transparency* hides the effects of partially completed interactions that fail, i.e., provides failure atomicity; builds on mechanisms that guarantee all-or-nothing semantics for interactions and replication.
- *Replication transparency* hides the difference between replicated and nonreplicated clients and servers.

The transparency services in ANSA are not standalone building blocks such as the modules in some of the configurable services discussed so far. However, a given transparency can be added to an application level service by replacing the original service by a new service that includes the chosen transparency [Tea91]. Some mechanisms provided by ANSA, such as replication and transactions, make it easier to construct services enhanced with the chosen transparencies, but in general, the transparencies are not provided as configurable modules that could be combined with any arbitrary service. An approach resembling the ANSA approach is described in [Bec94]. Here, a separate software layer completely hides the fault-tolerance aspects from applications.

2.1.2.4 Adaptive Parallel Real-time System

Configurability and adaptability are explored in the context of reliable parallel and distributed real-time systems in [BS91, SBB87]. The goal is to build systems that change their structure, both offline and during operation, to maintain good performance in response to such events as failures and changes in request latencies and utilization. A prototype implementation of the system has been constructed that places special emphasis of parallel execution of the underlying communication protocols [LAKS93]. In this model, a protocol consists of a set of *protocol objects*, each of which performs an isolated protocol processing task. Objects communicate with each other by asynchronous invocations implemented as messages delivered via shared memory mailboxes. Protocol objects cooperate in the processing of protocol packets. Due to parallelism, several operations may be in progress at the same time. For instance, an outgoing user packet may be in the process of being encrypted by the encryption object, while the sequence numbers for the packet are being computed by the reliability object. Any object can communicate with any other object, so the model does not restrict the relationship of the objects in any sense. In the following, we refer to this model as the *object/message model*.

Several extensions to the prototype have been proposed. In particular, to accommodate adaptability, the connections between objects may be changed at runtime. For example, a new object, say C, may be placed between two existing communicating objects, say A and B. After this addition, messages sent by A will go to C instead of B. Also, to improve efficiency, a synchronous form of object invocation is proposed, where the caller thread actually executes the called method of another object, instead of using asynchronous invocations that are implemented using mailboxes. However, the system still has certain limitations. First, because the objects do not share state, some operations, such as clearing all information concerning a particular message, becomes complicated. Furthermore, for the same reason, if one object holds information useful for some other object, the only way to acquire this information is by means of message passing, which is slower than if data was shared. Second, if the number of processors is smaller than the number of simultaneously active objects, the cost of context changes reduces system performance. Finally, although the communication between objects is reconfigurable, only existing communication can be redirected and it can only be directed to one object unless extra “multicast” type objects are added.

2.1.2.5 Group Communication Framework

A modular framework for group communication systems is proposed in [Gol92]. The framework has four fixed components: *application*, *message delivery*, *message ordering*, and *group membership*. Each component may have different implementations. For example, different ordering components can implement different variations, such as unordered, FIFO, causal, and total order. In this framework, the four components communicate though three predefined shared data structures: a *message log*, a *timestamp summary*, and a *group view*. The message delivery component implements a multicast communication service that exchanges messages with other group members. It writes incoming messages to the message log and maintains summary information of messages sent and received (timestamp summary) that can be used by the message ordering component. The group membership component maintains the set of group members in the group view data structure. When membership changes, this component communicates with the membership components of other group members. The message ordering component ensures that messages are presented to the application according to some ordering. This component also processes outgoing messages so that the matching components of other members have enough information to order messages properly. The framework components and their interactions are illustrated in Figure 2.3.

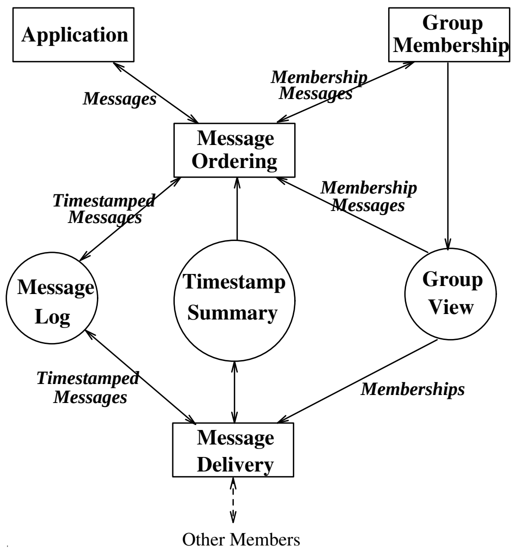


Figure 2.3: Group Communication Framework

Two applications of the framework are also presented in [Gol92]: a bibliographic database (Refdbms) and a distributed host reliability monitor (Tattler). These applications have different requirements for reliability and ordering of communication. All elements of the general framework are not explicitly present in all implementations. For example, in Tattler, no explicit message log is implemented because the message delivery and ordering components can work directly from the database that maintains the host reliability information. Therefore, it appears that the group communication framework is more of a conceptual framework than an actual implementation framework.

2.1.2.6 Arjuna

The Arjuna system provides tools for constructing reliable distributed object-oriented applications [SDP91]. Arjuna is based on the object/action paradigm [Gra86], where applications are structured as atomic actions operating on persistent objects, i.e., objects that survive site crashes. Arjuna is implemented using C++, and uses the inheritance mechanism provided by the language extensively. The distribution and replication of objects is hidden from applications by having a stub generator generate communication code. This code performs operations on remote objects through a remote procedure call (RPC) protocol. The protocol employs multicast, thereby allowing the invocation of operations on replicated remote objects. The layer that provides multicast also implements multicast groups, including operations that permit processes to join and leave groups.

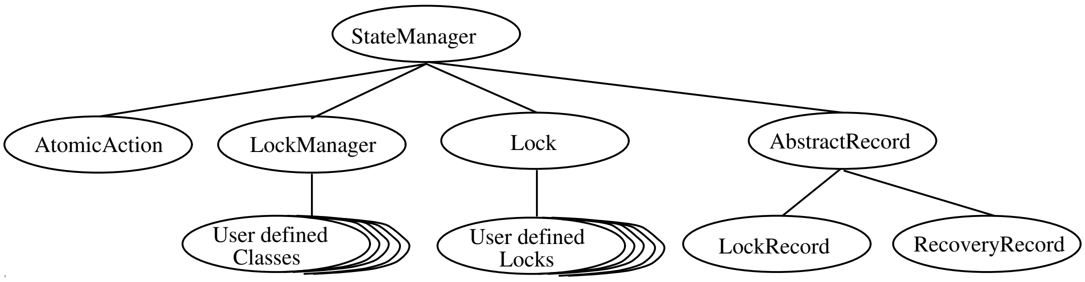


Figure 2.4: Arjuna Class Hierarchy

Mechanisms needed for constructing reliable distributed applications are presented to users of Arjuna as objects. A predefined class hierarchy, shown in Figure 2.4, specifies system services such as atomic actions and locks, which can be manipulated using operations like any other object [Whe89]. New classes can be defined as derived classes of existing ones, thereby inheriting the properties of the parent class. Arjuna provides a service, called the ObjectStore, for storing objects persistently. The StateManager class provides the interface to the ObjectStore, which means that objects from all classes derived from StateManager can be stored in the ObjectStore. If a new derived class of StateManager is created, the only additions required to make the class persistent or recoverable are to implement operations for saving and restoring the object state. Concurrency control for a new class can be provided simply by deriving the new class from the LockManager class. Operations on a class can be made atomic by using the operations of the AtomicAction class.

Arjuna is configurable in the sense that it provides a set of services out of which different application classes can choose the required ones. Furthermore, customized versions of services, for example locks, can be defined as derived classes of the existing service classes.

2.2 Other Configurable Systems and Services

In this section, we examine configurability in areas other than fault-tolerant distributed computing, including networking, operating systems, file systems, and database systems. Naturally, configurability has been addressed in other areas as well, but these are most closely related to our own work.

2.2.1 Networking

Extensibility and configurability have a traditional role in the computer networking field. Communication software has typically been viewed as consisting of logical layers, each of which builds on the layers below and adds functionality or properties to the service. An example of this view is the ISO OSI model [DZ83], which, although just a specification, is a good example of modular design of such services. In this model, lower levels are typically implemented in hardware, while higher levels are implemented in one or more software modules. The main emphasis of the OSI model is interoperability and configurability is mostly limited to choosing some execution parameters. Other projects in the communication field have taken more interest in configurability in addition to modularity.

The *x*-kernel is a system for constructing networking subsystems [HP91, OP92]. As discussed in the context of Consul (section 2.1.2.1), the *x*-kernel supports hierarchical composition of communication protocols, such as the standard Internet protocols IP, TCP, and UDP, where each protocol is implemented as an independent module. A communication service is constructed at compile time based on a specification in the form of a directed acyclic graph called a *protocol graph*. The nodes of the graph correspond to protocols and the edges represent communication paths between protocols. That is, if a protocol A sends a message to its peers using protocol B, then there is an edge from A to B. All *x*-kernel protocols implement a standard *x*-kernel uniform protocol interface (UPI). The UPI means that any combination of protocols is possible syntactically, but semantic requirements of the protocols restrict the configurability considerably. The *x*-kernel makes modular implementations efficient through the use of techniques such as having one thread carry a message through the protocol stack to avoid context switching and optimized data structures for message headers to avoid copying. The *x*-kernel work has demonstrated that modular implementation can be efficient. For example, user-to-user communication latency in the *x*-kernel has been shown to be much less, often less than half, than that of Unix on identical hardware [HP91].

The Adaptive system [SBS93] proposes a configurable and adaptive framework for building communication protocols for applications like multi-media that have special requirements for quality of service. The model used by Adaptive is based on dividing the communication service into functions, with each function being implemented by a chosen protocol. A logical communication connection between application level entities, a *session*, is divided into four functions: *connection handling*, *remote context management*, *error protection*, and *transmission control* as illustrated in Figure 2.5. Each of these functions has a slot in the communication service that can be filled with a number of different protocols for each logical session. For example, transmission control can be implemented using a sliding window, stop and wait, or rate-based control protocol. A function slot can also be filled with so-called *composite* components, which are a structure for binding more than one modules together to execute the function in question. In the figure, the *error protection* function is implemented with a composite component that has slots for protocols for *error detection*, *error recovery*, and *error reporting*. Configuration is based on automatic selection of library modules that satisfy user requirements—expressed either at compile time or during execution—and the status of the underlying network. The work has also been extended to support construction of configurable network daemons [SS94].

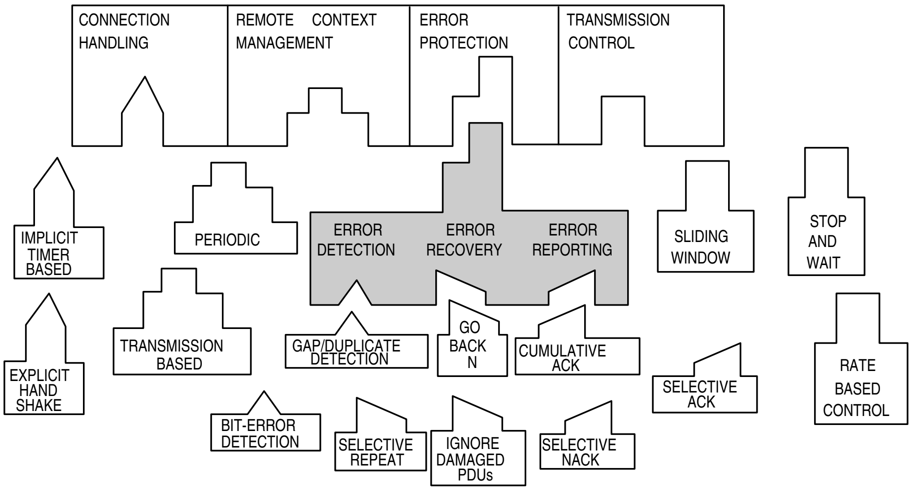


Figure 2.5: Session Configuration in Adaptive

2.2.2 Operating Systems

Several research projects in the area of operating systems are based on the premise that traditional operating system structuring limits the performance, flexibility, and functionality of applications. For example, [CFL94] demonstrates that application-level control of file cashing reduces application running time by 45%. Similarly, application-specific virtual memory policies increase application performance [HC92, KLVA93], while exception handling is an order of magnitude faster if the signal handling is deferred to applications [TL94]. Therefore, configurability and extensibility of both the abstractions provided by the operating systems and their implementations have recently been the targets of active research efforts.

The Synthesis system [PMI88] was one of the first projects to explore configurability and extensibility in the context of operating systems. Perhaps the key contribution of Synthesis is its use of optimization techniques from the area of dynamic code generation in its kernel design. Such techniques can produce efficient executable code since they can take advantage of the extra information in the runtime execution context. For example, frequently executed kernel calls can be regenerated at runtime using compiler code optimization ideas such as constant folding and macro expansion. Synthesis, therefore, is an example of adjusting or configuring the implementation to improve performance without modifications to the high-level operating system abstractions provided for the applications.

Recently, the work on configurable operating systems has concentrated on adjusting the operating system abstractions to fit the specific needs of application [CL95]. In the following, we take a closer look at a number of these projects. Numerous others, such as Choices [CJK$^+$87, CIM92], the Kernel Tool Kit (KTK) [GMSS94, MS96], Kea [VH96], and Apertos [Yok92, Yok93, TYT92, IY94], are not addressed here.

The SPIN operating system is based on an extensible microkernel [BCE$^+$94, BSS$^+$95]. The microkernel exports interfaces that offer applications fine-grained control over a few fundamental system abstractions, such as threads and virtual address spaces. SPIN is extensible in that application programs can install code sequences called *spindles* that execute in the kernel in response to hardware and software events, such as exceptions, interrupts, and context switches. Thus, the microkernel only provides the mechanisms for managing the system resources, not the policies. Spindles define the application-specific management policies, and also enable each application to define the precise interface and implementation for kernel services they require. Installing spindles at the kernel level allows for flexible and rapid response to system software and hardware events. Flexibility is achieved in large part because spindles have direct access to kernel data structures, a feature that is made safe by automatically verifying spindles before installation. The verification is based on the spindle programming language being type-safe and object-oriented. Thus, traditional type checking can ensure that spindles only invoke legal operations. Additionally, kernel operations made available to spindles can be guarded with a predicate expression that must be true for access to be legal. Rapid response time is achieved because spindles are part of the kernel and therefore do not require switching between kernel and user modes.

The Exokernel operating system architecture takes the approach of moving the physical resource management to the application level, thereby making it easy for each application to modify the resource management to best satisfy its requirements [EKO94a, EKO94b, EKO95]. The design of Exokernel is based on the argument that abstraction overhead is the root of inefficiency in most modern operating systems. Therefore, its goal is to eliminate all abstractions from an operating system and allow applications to craft their own. A minimal operating system kernel, called an *exokernel*, securely multiplexes available hardware resources. In particular, it provides operations only for *secure binding* of resources to the application, *visible resource revocation*, and an *abort protocol* that can be used by the exokernel to break secure bindings of uncooperative applications by force. The traditional operating system services are provided in an application-level library.

The V++ operating system is based on the abstraction of a cache [CD94]. Unlike conventional operating systems, where a cache is used to store memory data, the abstraction is extended in V++ to store operating system objects, such as threads and address spaces. The Cache Kernel is the kernel of the V++ operating system, and acts as the cache that stores these objects. For example, a thread is made available for execution by loading it, along with the associated address space objects, into the Cache Kernel, which will execute it. Unlike conventional operating systems, the Cache Kernel does not fully implement all the functionality associated with address spaces and threads. User-level *application kernels* provide the management functions required for a complete implementation, including the loading and writeback of these objects to and from the Cache Kernel. Therefore, the application kernels implement application-specific versions of typical operating system services, such as scheduling, exception handling, and memory management. Scheduling can be implemented by choosing which threads to load and unload from the Cache Kernel, and by setting the execution priorities of threads. A page fault, an example of exception handling, is handled by the application kernel associated with the faulting thread loading a new page mapping descriptor into the Cache Kernel as a part of a cached address space object. The application kernels also provide backing store for the object when it is unloaded. As a result, an application kernel has total freedom in selecting memory management policies, such as which pages are written to disk and which are kept in the application kernel memory. A number of standard application kernels are provided for different types of applications, but a customized kernel can be constructed if desired.

Scout, a communication oriented operating system, is based on the concept of a *path*, which is the extension of a network connection into the host operating system [MMO+94a, MMO+94b, MP96]. Scout makes the path its primary abstraction, with resource allocation, scheduling, and fault isolation done on a per-path basis. Thus, an application using Scout can associate with a given path all the resources—CPU, memory, bus, and cache—necessary to provide the same quality of service as provided by the network connection to which it is attached. Although the primary goal of Scout is good performance, configurability is also an important aspect, for two reasons. First, Scout must be able to support various forms of communication devices, such as network cameras, portable devices, and multicomputers. Second, Scout must be able to support different communication requirements, such as varying degrees of reliability, security, mobility, and real time. Scout is the successor of the *x*-kernel and, as such, preserves its hierarchical composition model.

2.2.3 File Systems

Traditional file systems have been found limiting in many application areas, a problem that configurability or extensibility could greatly alleviate. For example, the poor performance of traditional file systems for database applications has forced many database systems to bypass the file system and implement their own storage system directly on the physical devices [Sto81, Mos86]. Traditional file systems are also less than ideal for multimedia applications. In particular, multimedia requires high I/O throughput rates and quality of service guarantees such as constant minimum data rates not typically provided by file systems [AOG92]. Typical file systems also do not support easy addition of new services, such as compression or encryption [HP94]. Many of these problems have been addressed in research projects that study configurable or extensible file systems.

The stackable file system described in [HP94, HP95] allows the system to be augmented with new properties by adding layers to an existing file system. Examples of new layers might be encryption, compression, selective file replication, extended directory services, remote access, undo, undelete, or better support for transactions. Although the approach is called stackable, often the layers are organized in a hierarchical, but not necessarily linear, manner. The two major goals of the project are ease of configurability and extensibility. Configurability is easy if the interface between all layers is standard, meaning that there are no syntactic restrictions to how layers are configured. However, to support extensibility, a layer must have the ability to define new operations. For example, the layer implementing the undo operation must be able to make this operation available to the application and to have it pass unchanged through all other layers. The approach chosen is to have an extensible standard interface, where a layer typically just passes through operations that it does not support.

The extensible file system described in [KN93] addresses the same issues but takes a slightly different approach. This file system is designed for the Spring operating system, which is a distributed, multi-threaded operating system built using objects and interface inheritance. The file system is an object in Spring and its interface specifies the operations that the file system supports. The Spring file system stacking architecture enables new file systems to be added that extend the functionality of and build on existing file system implementations. This is achieved by adding new layers to the system, where each layer inherits the file system interface, and therefore presents the standard file system functions to the user. The implementation of each layer is written using the underlying file system. Unlike the stackable file system, all layers have an identical interface, and therefore, it is not possible to add new file operations.

The configurable mixed-media file system described in [Maf94] is another approach to configurability in file systems. This system is configurable in two different aspects. First, features such as the file replacement and space-allocation policies can be changed. Second, a variety of storage organization forms—for example, replicated storage, storage hierarchies, and striped storage—can be configured using a command language. The attribute mixed-media describes the file system's ability to integrate different media types, such as RAM, hard disks, and CD ROMs, into a virtual storage. This file system is not based on augmenting existing file systems with additional layers, but on defining storage devices using an object-oriented class hierarchy. The base class *Storage* implements the common core functionality of a storage device, so that any device-specific file system can be implemented as a derived class of Storage. Thus, implementing a new file system only requires defining three device-dependent methods: one to compact a storage region, and two to read and write data. Furthermore, the default policies provided by the Storage class, such as the replacement policy for determining when a unit of data will be moved to a secondary, slower, storage device, can be replaced by writing a new policy within the derived class.

2.2.4 Database Systems

Many important database applications, such as statistical databases, CAD and engineering databases, textual databases, and databases for artificial intelligence, are not well served by traditional database technology [BBG+88]. Although specialized databases have been developed for these different application areas, exploiting configurability is an appealing alternative. Here, we outline two different configurable or extensible database systems. Numerous others, including Postgres [SR86], Starburst [SCF+86], Exodus [CD87], Gral [Gut89], Ream [KNKH89], and P2 [TB95], are not addressed here.

The RAID system [BFHR90] has been used to study adaptive concurrency control. RAID is configurable in the sense that six components—the user interface, action driver, access manager, atomicity controller, concurrency controller, and replication controller—provide a choice of algorithms for implementing their functionality. For example, the concurrency control component implements timestamp ordering, two-phase locking, generic timestamp ordering, generic locking, and a generic optimistic algorithm as options for concurrency control.

Genesis [BBG+88] supports fast construction of complex and customized database systems from prefabricated components in libraries. New components can be added to the libraries, making the system extensible. A system is constructed as a hierarchical composition of components in predefined *realms* [BO92]. These realms range from those that define access methods and physical record allocation to those that define data models and their data access languages and query processing components. Realms are typed, with the type system restricting how components can be combined, as follows. Notation "t:T" means that component t belongs to realm T and "x[y: T1, z: T2]:T" means that component x (of realm T) takes two parameters, y of realm T1 and z of realm T2. This notation can also be seen as denoting a hierarchical composition, where component x is built using components y and z. Note that if a component has a parameter of the same realm as the component itself, any number of these components can be combined in any order. For example, x[y:T]:T and a[b:T]:T can be combined as x[a] or a[x[a]]. A component in this hierarchy can be replaced by any component of the same realm.

2.3 Characterizing the Approaches to Configurability

In the configurable systems described above, we can identify two major approaches to how a system is divided into configurable modules and how the modules are combined:

- *Hierarchical:* A system is constructed as a stack or a directed graph of protocols or other modules.

- *Function-based:* A fixed number of system functions are identified and different variations of these are implemented as configurable modules; a system is constructed by choosing a module for each function.

In the hierarchical approach, different variants of a service are constructed by stacking chosen protocols in a given way. The constraints that restrict how different protocols can be combined are based primarily on what a protocol assumes about the underlying protocols. Often, new protocols may be inserted between two protocols in the stack without any change in the existing protocols, or the order of two protocols may even be exchanged. Examples of this approach are the *x*-kernel, Consul, Horus, Genesis, and the stackable file systems.

In the function-based approach, a configurable service consists conceptually of a service backplane with openings, or slots, for the system functions, and a set of modules for each function. Construction of a service instance is strictly constrained by the backplane: each module can typically only be used for one slot and all the slots in the backplane must typically be filled. The interactions between functions, including who interacts with whom and the type of the interaction, are typically hardwired in the backplane. Examples of this approach are Adaptive, RAID, the group communication framework [Gol92], and most of the configurable operating systems. A good example of this approach is the concurrency control function in RAID, which can be implemented using different predefined or user-implemented modules.

The object/message model [LAKS93] and class-hierarchy based models like Arjuna and the configurable mixed-media file system [Maf94] can be viewed as somewhat generalized versions of the function-based approach. In the object/message model, the operation of the system is divided into functions, each of which is implemented by an object that interacts with other objects using messages. Unlike in the basic function-based approach, it is possible to install new objects between two communicating objects and redirect communication between objects. In class-hierarchy based models, different parts of the system operation are encapsulated in object classes, with variants being implemented as derived classes. A system or service is then constructed by choosing the right variants of the required objects. If the existing set of classes is not satisfactory, new classes can be implemented as derived classes of existing ones.

Another classification of configurable systems can be based on the type of interface between modules, that is, whether modules are required to support the same *standard interface* or whether each module type has an *individual interface*. If all the modules export identical standard interfaces, it is possible to combine syntactically any set of modules. This approach is taken in Consul, Horus, and the *x*-kernel. Alternatively, each module can export an individual interface that reflects the semantics of the module. An example of this approach is Genesis, where the interface of modules in each realm is identical but may differ between realms. Other examples are the function-based systems, since each module typically has a function-specific interface and can therefore only be used in one slot. A few systems, such as the stackable file system [HP94], support an extensible standard interface, which would fall somewhere between the two general approaches.

2.4 Conclusions

The different configurable services discussed in this chapter illustrate the different approaches taken to configurability. Most of these approaches can be characterized as being hierarchical or function-based. Both of these general approaches have their limitations. Hierarchical composition restricts the interactions between modules, thereby limiting configurability and making it more difficult to design and implement highly configurable services. Furthermore, hierarchical composition tends to impose a performance penalty because a message must typically traverse each protocol, even if it does not process the message. In the function-based approach, it is often impossible to divide the operation into functions so that the functions correspond to individual properties, and it is often difficult to add new functions. Furthermore, a fine grained system division, which allows good configurability, may lead to many of these functions being filled with null modules that can cause overhead.

Although there has been a considerable amount of work on configurability in different areas of computing, the use of configurability in fault-tolerant distributed systems has been limited. The most notable exceptions are Consul and Horus, both of which have the general limitations of hierarchical models. This dissertation introduces a new approach to constructing configurable services that is particularly appropriate for distributed fault-tolerant services. This approach, which is based on a two-level system view combined with event-driven execution of fine-grained micro-protocols, is hierarchical only on the system level, where a system is constructed from services. On the service level, where a service is built from modules implementing abstract properties, the model is neither hierarchical nor function-based. In the following chapters, we describe our approach in detail and apply it to different distributed fault-tolerant services.

CHAPTER 3

CONSTRUCTING CONFIGURABLE DISTRIBUTED SERVICES

This chapter describes in detail our approach to constructing configurable distributed services. As outlined in chapter 1, the approach is based on a two-level view: a system is constructed of services, with each service being composed of modules that implement the abstract properties of the service. Since services can be combined using traditional hierarchical methods, this aspect is not addressed here. Instead, the emphasis is on constructing the individual services out of fine-grained modules using the event-driven execution model.

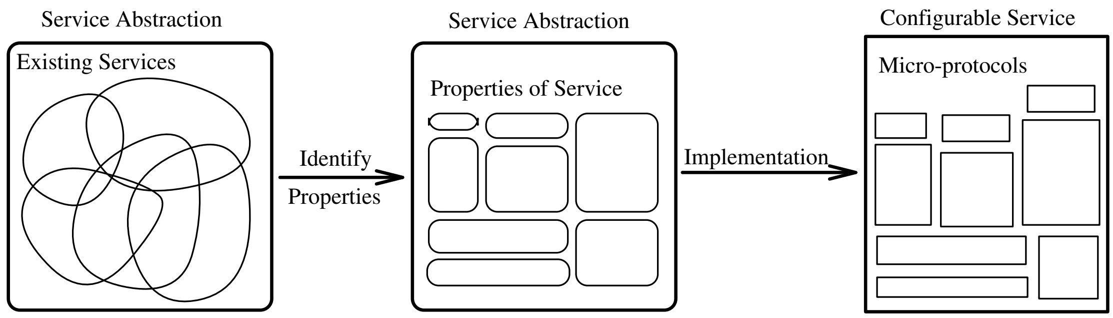


Figure 3.1: Construction of Configurable Services

Our approach starts with identifying the abstract properties or execution guarantees of a service and ends with a configurable implementation as illustrated in Figure 3.1. This chapter is structured around this process. First, we present an approach to specifying service properties based on message ordering graphs and identify relations between properties that dictate which combinations are feasible. Second, we describe the event-driven execution model, including the basic concepts of events and micro-protocols. Third, we outline the use of the event-driven model for constructing a configurable service, including the design steps involved in implementing properties as micro-protocols and the relations between micro-protocols that affect configurability. Fourth, to demonstrate the feasibility of the approach, we discuss the requirements for implementing the event-driven model and briefly describe three prototype implementations that have been used to experiment with different configurable services. Finally, we compare our approach to other approaches for constructing configurable fault-tolerant services and mention some related work on event-based systems.

3.1 Properties of Services

The first step in building a configurable service is to identify service properties that might be of interest to the users of the service. The goal is to divide the abstract service—that is, the union of all variants of the service that have been defined—into abstract properties. Forming a consistent set of such properties from existing service implementations is a nontrivial task, for several reasons. One problem is that properties are often described in terms of a particular implementation strategy and may be difficult to translate into more abstract properties. In essence, an algorithm is often designed first, and then the exact properties of the algorithm are determined. Another problem is that numerous different methods and terminology are used for describing properties. In the following, we introduce an implementation-independent technique for defining properties of services. In chapter 4, properties of existing membership services that have been defined using various notations are uniformly specified using this technique.

3.1.1 Specifying Properties using Message Ordering Graphs

The following notation is used in the rest of this dissertation unless otherwise stated. Capital letters A, B, C, ... are used to denote sites. Small letters with a subscript, for example a_i, are used to indicate the i^{th} message sent by site A. $\mathcal{S}$ denotes the set of all sites and $\mathcal{M}$ the set of all messages.

In our model, an application (or a higher level service) interacts with the underlying services only through messages that are passed in either direction across the interface. This is illustrated in Figure 3.2, where the underlying service on each site is seen as a black box from the application's point of view, and the only interface to this black box are messages sent and received. As a result, the only way to define properties for the underlying services is in terms of what messages are delivered to the application, in which order they are delivered, when they are delivered, how the messages and their order relate to events such as failures and recoveries in the system, and how the set of messages, their order, and delivery time relate to one another on different sites. For example, a requirement that messages be ordered FIFO between two sites means that the messages sent by a site A are constrained to be delivered to the application at any other site B in the same order as they were sent. This property sets a constraint on the order in which messages are delivered to the application at the receiving site based on the order in which they were sent by the originating site. Similarly, a requirement that messages be delivered in total order at all sites in a group constrains the delivery order on A to be identical to the delivery order on any other site B. Finally, a requirement that message transmission time be bounded by some Δ constrains the delivery of a message sent at some global time t_1 to occur at some global time t_2 such that $t_2 - t_1 < \Delta$.

Let $\mathcal{P}$ be a set of properties implemented by a given service. As noted above, $\mathcal{P}$ can be stated in terms of constraints that must be true for the sequence of messages delivered to the application at the various sites involved in the computation. These properties can be

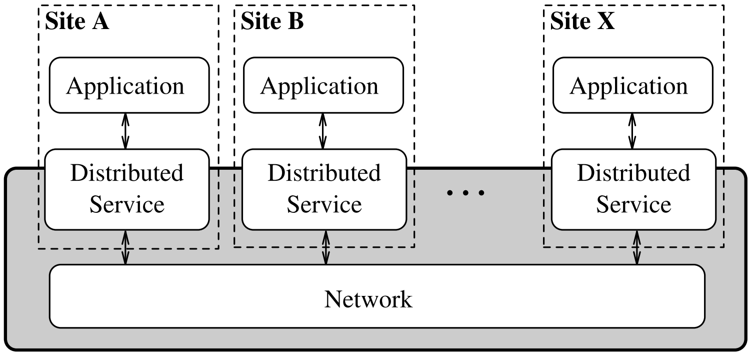


Figure 3.2: System Model

conveniently defined and illustrated using temporal logic formulas over *message ordering graphs*. Informally, the message ordering graph for a site A at time t is an abstract representation of all messages that have been received at A by time t and the ordering constraints between these messages. In essence, the graph represents all possible orders in which messages can be delivered to the application on A and still satisfy $\mathcal{P}$.

Formally, a message ordering graph is a directed acyclic graph, $G = (N, E)$, where the set of nodes, N, is a set of messages and the set of edges, $E = \{(m_i, n_j) | m_i, n_j \in N\}$, is the set of ordering constraints between messages. If O is an ordering graph, $N(O)$ denotes the set of nodes and $E(O)$ denotes the set of edges. If $(m_i, n_j) \in E$, then m_i is called an *immediate predecessor* of n_j, pred(n_j) for short, and n_j is called an *immediate successor* of m_i, succ(m_i) for short. If there is a directed path of edges connecting m_i and n_j, denoted by $m_i \to n_j$, we say m_i is a *predecessor* of n_j and n_j is a *successor* of m_i, PRED(n_j) and SUCC(m_i) for short, respectively. The meaning of the edges is that in order for $\mathcal{P}$ to be satisfied, any message m_i can only be delivered to the application after all its predecessors have been delivered.

O_A denotes the ordering graph at site A, while the set of ordering graphs from all sites at any given point of an execution is denoted by Θ. Essentially, Θ is a forest of the ordering graphs of the individual sites. The state of O_A at real time t is denoted by $O_A(t)$. Similarly, Θ at time t is denoted by $\Theta(t)$.

The basic ordering graph only reflects the set of messages and their possible delivery orders. In some cases, the actual order in which messages are delivered to the application or the actual time when the delivery occurs is of interest. For this purpose, define event $del_A(m)$ to denote the event of message m being delivered at site A. Symmetrically, define $send_A(a_i)$, or $send(a_i)$ for short, to be the event of the application on A sending message a_i. Furthermore, define $time(event_i)$ to be a function that returns the time t at which $event_i$ occurs. Note that t is not a timestamp generated by the system—i.e., the result of reading some real time clock—but rather a time as seen by an external observer that is used only for specification purposes. If an event has not occurred, $time$ is undefined.

Given these definitions, properties can be defined in terms of how they affect ordering graphs. Formally, a *property* is defined by a set of constraints on nodes and edges in a collection of ordering graphs, and how those nodes and edges relate to other system events. For example, a FIFO ordering property for a reliable multicast where every site is assumed to receive every message can be expressed as:

$$\forall\ A, B:\ i \in [1, \infty[\ : (b_i, b_{i+1} \in N(O_A)) \Rightarrow ((b_i, b_{i+1}) \in E(O_A))$$

This property specifies an ordering constraint for the graphs in Θ that must hold at all times across all executions. While the FIFO property can be stated in terms of a single ordering graph, most other properties relate ordering graphs from multiple sites. Let $a_i \rightarrow b_j \in O_A$ indicate that there is a path of length ≥ 1 from message a_i to message b_j in ordering graph O_A. Then, for example, a consistent total order in a reliable multicast system can be stated as:

$$\forall\ A, m_i, n_j : (m_i, n_j \in N(O_A)) \Rightarrow (((m_i \rightarrow n_j) \in O_A)\ \vee\ ((n_j \rightarrow m_i) \in O_A))$$

and

$$\forall\ A, B, m_i, n_j : ((m_i \rightarrow n_j) \in O_A) \Rightarrow \Box((n_j \rightarrow m_i) \notin O_B)$$

where $\Box$ is the temporal operator denoting "henceforth". Again, these formula must hold for Θ across all executions.

Given the formal definitions, we can reason about the properties. For example, it may be possible to show for some properties p_i, p_j, and p_k that $p_i = p_j \wedge p_k$ or that $p_i \Rightarrow p_j$. These propositions can sometimes be proven based on rules of temporal logic, but in general we use reasoning over executions. Let s be a system, e_s be an execution of s, and *SYS* be the set of all systems. Let $sat(e, p_i)$ be true if property p_i is satisfied for execution e and false otherwise. Then, in term of executions, we can state $p_i \Rightarrow p_j$ as

$$\forall\ s \in \textit{SYS}\ \ \forall\ e_s : sat(e_s, p_i) \Rightarrow sat(e_s, p_j).$$

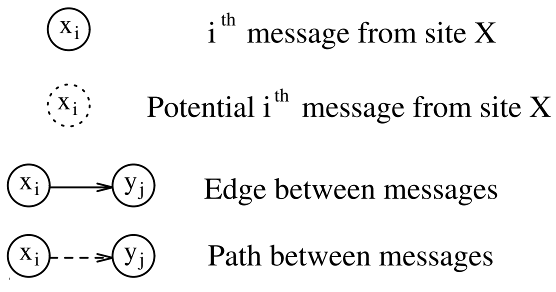


Figure 3.3: Ordering Graph Notation

Message ordering graphs are also used to illustrate graphically the various properties based on the notation outlined in Figure 3.3. Figure 3.4 illustrates FIFO and total order properties using this notation. In the figure, we assume an underlying reliable multicast mechanism, so the ordering graphs are identical at each site. Note, however, that although the ordering graphs are identical for FIFO, the order in which messages are actually delivered to the application at sites A and B may differ while still satisfying the ordering constraints. For example, A may deliver the messages in order a_1, a_2, a_3, b_1, b_2, b_3, while B delivers them in order a_1, b_1, a_2, b_2, a_3, b_3. The only requirement is that a message be delivered after its predecessor(s), so messages between which there is no ordering constraint can be delivered in any order.

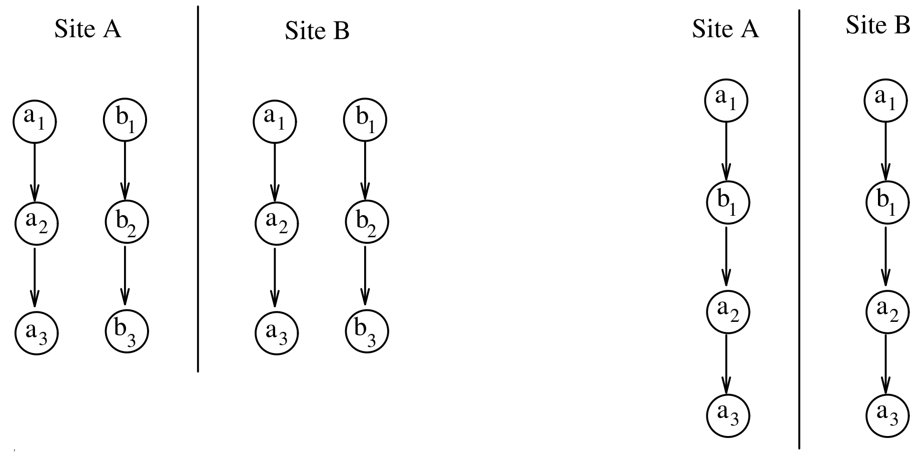


Figure 3.4: FIFO and Consistent Total Order Multicasts

Note that ordering graphs are only one method for defining, describing, or illustrating properties of distributed systems. Another approach, used for example in [RB91], is based on describing system behavior by *process histories*, where the history for process p is a sequence of events including send events, receive events, and internal events. A *system run* is a set of process histories, one for each process in the set of processes. Properties are defined in terms of temporal logic formulas over these histories. It would be possible to define the properties of services in this manner but we choose to use ordering graphs due to their more illustrative nature. Also, in contrast to process histories, ordering graphs model more closely the execution of the system and allow the expression of all legal orderings of message receptions in one graph instead of stating them as properties of linear histories. Numerous other methods are also possible. For example, in [RFJ93] membership properties are stated in terms of *membership runs*, which are defined as sequences of global membership states consisting of each site's view of the global membership. Transitions from one global membership state to the next occur whenever a site changes its view of the global membership. This approach is less comprehensive than ordering graphs, since it does not address the ordering of membership changes with respect to sending and receiving of application messages.

Finally, note that the concept of a graph of messages is very appealing as an implementation tool as well. For example, this technique is closely related to the *causality graphs* used in Psync [PBS89] and Transis [ADKM92b], which capture the causal ordering relation between messages.

3.1.2 Relations between Properties

Examining existing implementations of a service results in a set of properties for each implementation. That is, the properties of the implementations $I_1, \ldots, I_n$ can be stated as conjuncts of properties, as follows:

$$I_1 = p_1^1 \wedge p_1^2 \wedge p_1^3 \wedge \ldots$$

$$I_2 = p_2^1 \wedge p_2^2 \wedge p_2^3 \wedge \ldots$$

$$\ldots$$

$$I_n = p_n^1 \wedge p_n^2 \wedge p_n^3 \wedge \ldots$$

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ be the set of all the different properties guaranteed by I_1, I_2, …, and I_n. This set of properties can be normalized by eliminating redundant properties p_i such that $p_i = p_j \wedge p_k$. The set can also be expanded if desired by introducing new properties relative to those in existing implementations. For example, it is often possible to find some new property p' such that for $p_i,\ p_j \in \mathcal{P} : p_i \Rightarrow p' \Rightarrow p_j$. Of course, the inclusion of a new property should be determined by its practical value for realistic applications.

Relations between properties dictate which combinations of the properties in $\mathcal{P}$ are possible. We identify three basic relations:

- *Conflict:* $con(p_i, p_j) \Leftrightarrow \forall\ s \in \mathit{SYS}\ \exists\ e_s : \neg sat(e_s, p_i) \vee \neg sat(e_s, p_j)$

  That is, two properties conflict when no system can guarantee both p_i and p_j for all executions.

- *Dependency:* $dep(p_i, p_j) \Leftrightarrow \forall\ s \in \mathit{SYS}\ \forall\ e_s : sat(e_s, p_i) \Rightarrow sat(e_s, p_j)$

  That is, one property p_i depends on another p_j when satisfying p_i requires that p_j also be satisfied; in other words, there is no way to satisfy p_i without p_j.

- *Independence*: $ind(p_i, p_j) \Leftrightarrow \neg con(p_i, p_j) \wedge \neg dep(p_i, p_j) \wedge \neg dep(p_j, p_i)$

  That is, two properties are independent if they do not conflict and neither one depends on the other.

The dependency relation is transitive and asymmetric, whereas conflict and independence relations are symmetric but not transitive.

These relations determine, in essence, the set of feasible combinations, $\mathcal{C}$, of any two properties p_i and p_j:

- $ind(p_i, p_j)$: $\mathcal{C} = \{p_i, p_j, p_i \wedge p_j\}$.
- $dep(p_i, p_j)$: $\mathcal{C} = \{p_i \wedge p_j, p_j\}$.
- $con(p_i, p_j)$: $\mathcal{C} = \{p_i, p_j\}$.

Naturally these rules generalize to any number of properties.

Independence is the key to maximizing configurability of the resulting service. Often, two properties are naturally independent or they can be redefined so that they are independent. For example, in an atomic broadcast service, timeliness and the ordering properties are by nature independent. On the other hand, the ordering properties of atomic multicasts—for example, FIFO, causal, and total—are not naturally independent of atomicity, but can be made so by allowing gaps in the orders. Whether this redefinition has any practical justification depends, of course, on the projected applications of the service.

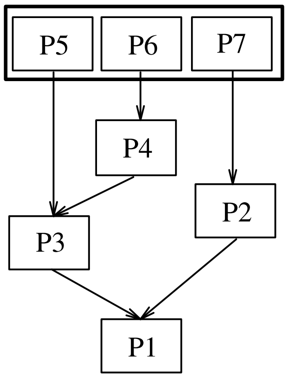
3.1.3 Dependency Graphs

Dependency graphs are a graphical method for recording and expressing the preceding relations between properties. A dependency graph is a directed, not necessarily acyclic, graph where each basic node represents a property and each edge a dependency. In addition, unlabeled *choice nodes* that encapsulate two or more nodes are provided to represent choice of properties. Specifically, the basic relations are graphically represented as follows:

- $dep(p_i, p_j)$: an edge from p_i to p_j.
- $con(p_i, p_j)$: p_i and p_j are included in a choice node.
- $ind(p_i, p_j)$: p_i and p_j are not included in the same choice node, and no path connects p_i and p_j.

Dependency graphs are simplified by omitting transitive dependencies. For example, if p_i depends on p_j and p_j depends on p_k, then the transitive edge from p_i to p_k is omitted.

A dependency graph represents relations between properties, and therefore all possible legitimate combinations of properties. Figure 3.5 shows a simple dependency graph and lists all possible combinations of the properties. In the figure, nodes P1, $\ldots$, P7 represent properties, with P5, P6, and P7 being in a choice node because they conflict. For a simple graph such as this one, it is straightforward to enumerate the possible combinations manually; for larger graphs, the process is easily automated. Here, there are 16 possible combinations even though the properties in the figure have a reasonable number of constraints. Obviously, for 7 different properties, the maximum number of different combinations would be $2^7 = 128$.



Legal combinations:

none	P1 P2 P3	P1 P2 P3 P5
P1	P1 P3 P5	P1 P2 P4 P5
P1 P2	P1 P4 P5	P1 P2 P4 P6
P1 P3	P1 P4 P6	P1 P2 P3 P7
P1 P4	P1 P2 P4	P1 P2 P4 P7
	P1 P2 P7	

Figure 3.5: Dependency Graph

3.2 Event-Driven Execution Model

Analyzing services gives property definitions and a dependency graph that summarizes the relations between properties. In this section, we introduce an event-driven execution model that can be used to implement these properties as configurable modules.

3.2.1 Overview

In our model, each service is implemented by a software module called a *composite protocol*, which is composed of fine-grained modules called *micro-protocols* that implement abstract properties of the service. The composite protocol provides an event mechanism and shared data structures that enable micro-protocols to co-operate. A micro-protocol consists of local data structures and a collection of *event handlers*, where an event handler is a procedure that implements an action upon the occurrence of an *event*. An event can be any status change in the system, such as the arrival of a message from the underlying network. Some events, such as message arrivals, are predefined meaning that every composite protocol supports these events. Others are defined by the designer to best fit the service in question and the interaction requirements of its micro-protocols. Figure 3.6 summarizes this two-level view of system composition.

The components of the model can be defined more precisely as follows. The syntactic elements of a micro-protocol are defined as a tuple

$$Mp = (Eh, MpInit, Ld, MpArgs),$$

where *Eh* is a set of event handlers, *MpInit* the initialization code that is executed when an instance of the micro-protocol is created, *Ld* the set the local variables accessible only from the event handlers and the initialization code of this micro-protocol, and *MpArgs* the arguments the micro-protocol is passed when created. Similarly, a composite protocol is defined as a tuple

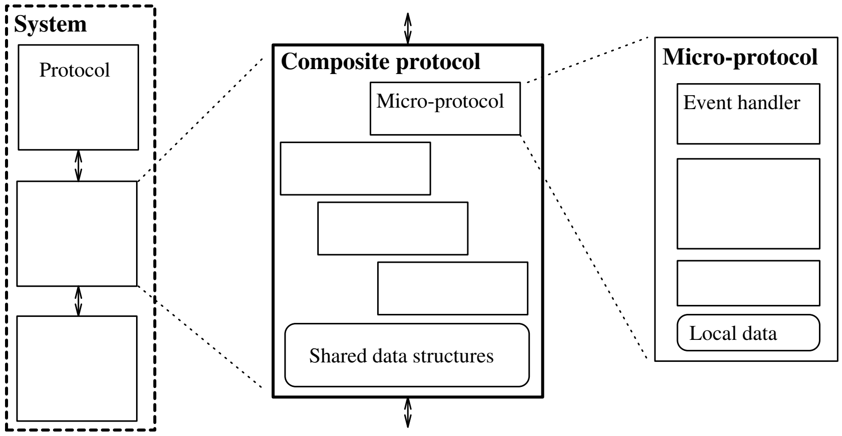


Figure 3.6: Two-Level View of System Composition

$$Cp = (Mp, Gd, Ev, Inter, CpInit, CpArgs),$$

where *Mp* is the set of micro-protocols, *Gd* the set of global data structures accessible from all micro-protocols, *Ev* the set of events defined for this composite protocol, *Inter* the service interface of the composite protocol, *CpInit* the initialization code of the composite protocol that, for example, creates chosen micro-protocols and initializes global data structures, and *CpArgs* the arguments that the composite protocol is passed when created. The interface *Inter* defines and implements the operations that the service exports. A customized composite protocol for a given service is created by defining the *Gd*, *Ev*, *CpInit*, *Inter*, and *CpArgs* elements.

Event handlers are bound to events by a registration operation that specifies that the handler is to be executed when the event occurs. The binding can also be deleted, in which case the handler is not invoked upon subsequent occurrences of the specified event. Furthermore, a handler can be bound to none, one, or several different events. Therefore, the state of a micro-protocol or a composite protocol at any given time t can be defined as a tuple that include the bindings at that time:

$$Mp(t) = (Eh, Ld, MpInit, MpArgs, MpBind(t)),$$

$$Cp(t) = (Mp, Gd, Ev, Inter, CpInit, CpArgs, CpBind(t)),$$

where *MpBind(t)* specifies the mapping from the event handlers of a micro-protocol to events at time t and *CpBind(t)* specifies the mapping from each event to the handlers registered for the event at time t.

This approach is depicted in Figure 3.7. It illustrates a composite protocol, which contains a shared data structure—in this case, a graph of messages. The boxes to the

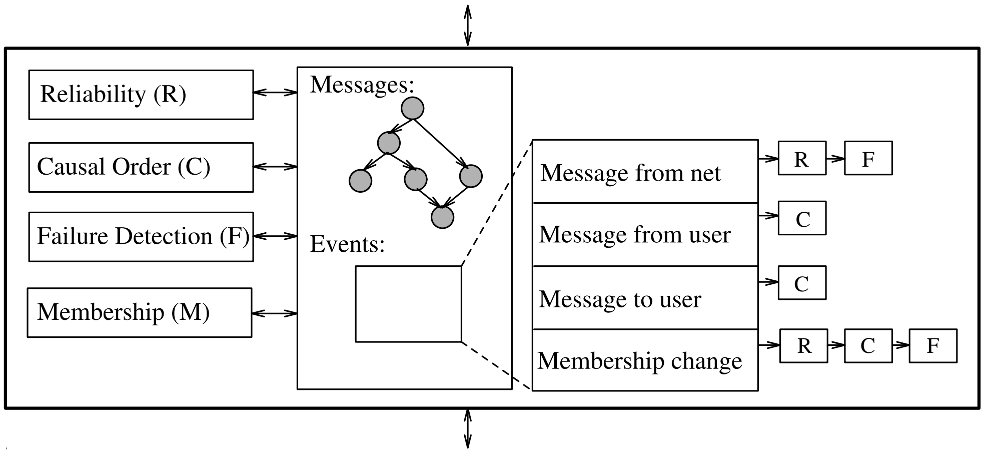


Figure 3.7: A Composite Protocol

left represent micro-protocols, while to the right are some common events with the list of micro-protocols that are to be invoked when the event occurs.

Given their importance in the model, we now describe events and event-related operations in more detail.

3.2.2 Events

An event is defined by a call to the runtime system specifying its name, event handler arguments, and optional event attributes. An event can be defined with two types of attributes. First, *execution attributes* are used to specify that the execution of the handlers associated with the event is to be *sequential* or *concurrent*. In the sequential case, handlers are executed one by one in specified order. In the concurrent case, handlers are executed logically in parallel so that each handler can make independent progress. Second, *invocation attributes* are used to specify that handler invocations associated with the event are to be *blocking* or *non-blocking*. In the blocking case, the invoker blocks until all handlers registered for the event have completed execution. In the non-blocking case, the invoker continues execution without waiting for execution of the handlers to complete. In particular, the invoker can continue execution even if a handler blocks, and the handlers can begin execution even if the invoker blocks.

Figure 3.8 illustrates the logical flow of control at the time an event is invoked using different combinations of event attributes. In the figure, the thicker line represents execution of the code that invokes the event, dots the invocation time, thinner lines the execution of event handlers bound to the event, and dotted lines execution being blocked.

Event attributes support different programming situations that arise when writing micro-protocols. For example, sequential execution allows an event handler to be written knowing that some other handler has already been executed. Concurrent execution is useful if a handler can block to ensure that other handlers are able to make progress.

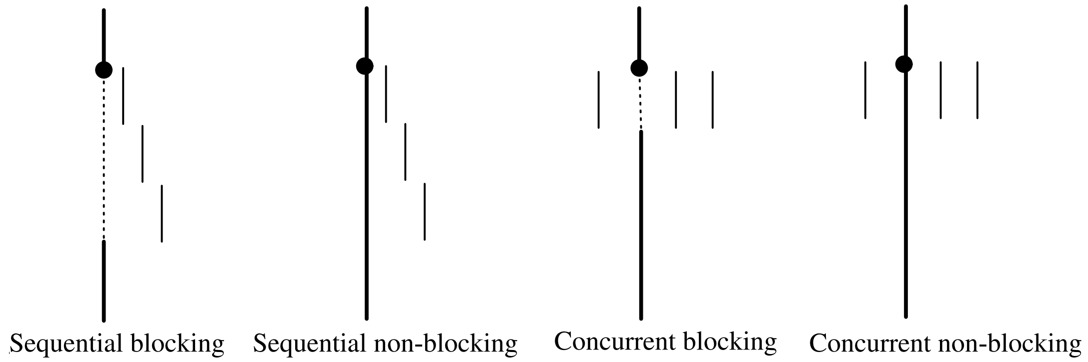


Figure 3.8: Effects of Event Attributes

Blocking invocation provides a convenient method for the invoker of an event to know when all the handlers registered for the event have been executed. Conversely, a non-blocking invocation ensures that the invoker is able to continue execution in spite of a handler blocking.

If an event attribute is left unspecified, the implementation of the event-driven model can choose the way invocations are handled and handlers executed. In particular, handlers may be executed sequentially in any order, logically in parallel by separate threads, or physically in parallel on a multiprocessor. In most systems, implementing events as blocking and sequential has the lowest overhead and therefore is the default.

Events are invoked either by the runtime system or by micro-protocols. The predefined events indicating, for example, the message arrival from a service above or below, are typically invoked by the runtime system. Other, user-defined events, are invoked by the micro-protocols.

The execution model is multi-threaded, i.e., several events may occur concurrently and several event handlers may be executing at the same time. If two or more events are invoked at approximately the same time, the relative execution order of handlers is unspecified. We do assume, however, that execution is fair in the sense that a handler that is eligible for execution is eventually executed. Note that execution of an event handler is not assumed to be atomic, that is, the scheduler may switch from one handler to another in the middle of execution.

3.2.3 Operations

We define five operations for handling events; these are used throughout the rest of the dissertation. An actual implementation of the event-driven model may define these operations slightly differently or only use a subset of the operations:

- **define**(*event_name*, *arg_types*, *ev_attr*). Defines an event *event_name* and the types of the arguments *arg_types* passed to the event handlers upon occurrence of the event. The optional event attributes *ev_attr* can be used to specify the event as concurrent or sequential, and blocking or non-blocking, using keywords CONC, SEQ, BLOCK, and NON-BLOCK, respectively.

- **trigger**(*event_name*,*arguments*). Notifies the runtime system that *event_name* has occurred. The runtime system then executes the appropriate handlers, passing *arguments* as invocation parameters.

- **register**(*event_name, event_handler_name, priority*). Notifies the runtime system that *event_handler_name* is to be executed when *event_name* is triggered. If the event is sequential, the handlers are executed in order according to the optional *priority* parameter.

- **cancel_event**(). Instructs the runtime system to cancel further event handler invocations associated with the same event occurrence that caused the operation to be invoked. This operation is useful mostly for sequential events.

- **deregister**(*event_name*,*event_handler_name*). Notifies the runtime system to remove the association between *event_name* and *event_handler_name*.

The last two operations, **cancel_event** and **deregister**, are useful for dynamically altering handler execution. For example, assume that one micro-protocol m_1 has a handler registered for the event corresponding to message arrival from the network, and that another micro-protocol m_2 filters out corrupted messages, messages send by unknown senders, or replicated messages. Now, if m_2 has an event handler registered for the same event but with a higher priority than m_1, m_2 can filter out chosen messages by executing the **cancel_event** operation. In this manner, different filtering micro-protocols can be configured into the composite protocol without the knowledge of m_1 or any changes to its code.

Similarly, the **deregister** operation is a very convenient and efficient way of handling state changes in a micro-protocol. For example, micro-protocols typically have different behaviors for different phases of system execution, such as startup, recovery from a failure, normal operation, and shutdown. During these different phases the micro-protocol might want to be notified of different events. This can be accomplished by registering the appropriate handlers at the beginning of a phase and then using **deregister** when these events are no longer of interest or when the handler must be changed. This potential to register and deregister handlers dynamically makes it possible for the minimal set to be registered at any given time and often allows code that tests whether this event is of interest in the current phase to be omitted. This feature is elaborated upon in chapter 8, where it is exploited to implement adaptive systems.

Finally, the model supports also a TIMEOUT event that is triggered by the passage of time. In this case, the *priority* parameter in the **register** operation is used to denote the time interval after which the specified handler is to be executed. Event handlers are usually persistent in the sense that they are invoked every time the specified event occurs until they are explicitly deregistered. The one exception is that handlers registered for TIMEOUT are executed only once and then implicitly deregistered.

Figure 3.9 gives a pseudo-code outline of a micro-protocol illustrating its main components.

```
define(EVENT_A,Type1,BLOCK);
define(EVENT_B,Type2,SEQ);
...

micro-protocol Protocol1(args: Type3) {
    var ... local data structures ...

    event handler HandleA(args: Type1) {
        ... code of event handler ...
    }
    event handler HandleB(args: Type2) {
        var par: Type1;
        ...
        trigger(EVENT_A,par);
        ...
    }
    initial {
        register(EVENT_A,HandleA)
        register(EVENT_B,HandleB,1)
        ... rest of initialization code ...
    }
}
```

Figure 3.9: Micro-Protocol Outline

3.3 Constructing a Configurable Service

The analysis of service properties and the resulting dependency graph give the starting point for the design of a configurable service, while the event-driven execution model provides the necessary tools. Actual design and implementation is still difficult, however, and certain design decisions can affect the configurability and performance of the resulting service. In this section, we review the design goals and describe steps in a design process that allow properties and their dependency graph to be transformed into events, shared data structures, and micro-protocols. In a manner analogous to properties, we define relations between micro-protocols that affect which combinations of micro-protocols result in a properly functioning service. A *configuration graph* is used to describe these relations, and as such, serves as a tool to facilitate the construction of properly working configurations.

3.3.1 Design Goals

The design of a configurable service has two major goals, to minimize execution overhead and to maximize the degree of configurability. The execution overhead is the performance difference between a monolithic implementation of a service and a configurable implementation that provides equivalent guarantees. The degree of configurability is the number of different functional configurations of a service given a set of micro-protocols. As already noted, the maximum degree of configurability is defined by all possible combinations of properties in the dependency graph.

The design of the micro-protocols comprising a service can impact the performance overhead of a service considerably. Although the goal is to implement each property as an individual micro-protocol, for efficiency reasons micro-protocols cannot be designed in isolation from one another. For example, if the implementation of property p_i requires a site to send a message to every other site and the implementation of property p_j has the same requirement, the implementations should only send one message, possibly with separate fields for p_i and p_j. Furthermore, redundant work in different micro-protocols should be avoided by having only one do the work, with the others utilizing those results. This can be accomplished, for example, by making the result available as global data and using events to notify other micro-protocols about its availability.

Note that, in principle, the maximum degree of configurability can be achieved by implementing each combination of properties as a separate monolithic program. Although this approach may result in more efficient implementations, the work associated with writing all these possibly hundreds of combinations is impractical. Our approach is to minimize the amount of code to be written by reusing modules in different configurations while aiming for a high degree of configurability.

3.3.2 Design Process

Typically, three different steps can be identified in the design of a configurable service: selection of a general implementation strategy, algorithm design, and micro-protocol design. These steps, outlined below, will be illustrated further in chapter 5 for membership and in chapter 7 for remote procedure calls. Note that sometimes services traditionally considered separately, such as reliable ordered multicast and membership services, may be so interrelated that it is more convenient to implement them together in one composite protocol. Typically, this is the case if some properties in each of the services have dependencies to properties in the other service.

Existing implementations of the service often provide good alternatives for the general implementation strategy. For example, many problems in distributed computing have coordinator-based and decentralized solutions. Issues to be considered in the selection include ease of implementation, efficiency in terms of number of messages or execution time, and applicability of the strategy to the underlying computing environment. For example, a broadcast based strategy may not be the best choice if the underlying network does not provide hardware broadcast facilities.

Given a general strategy, the second step provides the algorithms used to guarantee the various properties. At this point, algorithms can be designed in isolation, except that knowledge about relations between properties can be used to simplify the design. For example, if property p_1 depends on property p_2, the implementation of p_1 may assume that p_2 is guaranteed. Small details of the algorithms may be omitted here, since the important information are the general steps in the algorithm execution, the messages used, and required data structures.

Micro-protocol design translates the algorithms into co-operating micro-protocols that preserve configurability, while maximizing efficiency by reducing redundant work and messages. Often, some of the work required by a large number of micro-protocols can be encapsulated in so-called *base micro-protocols* that do not directly implement an abstract property, but rather make it easier to implement the property micro-protocols. Events and shared data structures for the composite protocol are also chosen here.

3.3.3 Configuring a Custom Service

Like their underlying properties, micro-protocols cannot be combined in arbitrary ways. Here, we identify relations between micro-protocols that affect configurability and introduce configuration graphs to describe these relations and aid in the construction of operational service configurations. Such graphs are similar to dependency graphs, but are based on physical software implementations rather than abstract properties.

Relations between Micro-Protocols

We can identify four relations between micro-protocols. Let $imp(m, p)$ denote that micro-protocol m implements property p, where p may be a combination of properties, e.g., $p = p_i \wedge p_j$. Similarly, let $m_1 + m_2$ denote a service configured from micro-protocols m_1 and m_2. As was the case with properties, conflict and independence are two of the relations between micro-protocols, with definitions as follows:

- Micro-protocols m_1 and m_2 *conflict* if they cannot be configured into the same system, which may be the result of the corresponding properties conflicting or design decisions made during the implementation.

- Micro-protocols m_1 and m_2 are *independent* if m_1 can be used without m_2, m_2 can be used without m_1, and m_1 and m_2 can be used together, where the combination guarantees both the properties implemented by m_1 and m_2.

The third relation between properties, dependency, is divided when considering micro-protocols into separate relations called *dependency* and *inclusion*. To motivate this, consider two properties p_1 and p_2 such that $dep(p_1, p_2)$. Based on the definition of dependency between properties, any system that guarantees p_1 must also guarantee p_2. While sufficient for abstract properties, in an implementation it is useful to identify two possible ways in which this can be achieved. First, we could implement m_2 such that it realizes p_2, $imp(m_2, p_2)$, and then implement p_1 as micro-protocol m_1 that builds on the guarantees made by m_2. This means that for m_1 to operate correctly, m_2 must also be present in the system, i.e., only the combination of micro-protocols m_1 and m_2 implements p_1, $imp(m_1 + m_2, p_1)$. This approach preserves the dependency that exists between properties as a dependency between micro-protocols. The second alternative is to implement m_1 such that it implements both properties directly, $imp(m_1, p_1 \wedge p_2)$, while m_2 is implemented to only realize p_2. This approach creates micro-protocols m_1 and m_2 where m_1 is strictly stronger than m_2. In this case, we say that m_2 *includes* m_1.

The choice of implementation approach and the resulting relation between the micro-protocols is based primarily on convenience. Often, the implementation of property p_1 cannot take advantage of property p_2 being satisfied, even though the properties have a logical dependency relationship. In this case, it is usually simpler to implement p_1 so that p_2 is satisfied directly, independent of the implementation of p_2. A good example is causal message ordering, a property that depends on the FIFO ordering property. In this case, the knowledge that messages are already FIFO ordered does not simplify the implementation of causal order, so it is easier to implement causal order independent of FIFO. As a result, the relation between the respective micro-protocols is an inclusion relation. In other situations, the implementation of a property can take advantage of the guarantees provided by some other micro-protocol. For example, the implementation of message ordering properties can exploit an atomicity micro-protocol that ensures that all sites will eventually receive every message. This leads to the dependency between properties being preserved as a dependency between micro-protocols.

The definitions and practical implications of dependency and inclusion relations for micro-protocols are as follows:

- Micro-protocol m_1 *depends on* micro-protocol m_2 if m_2 must be present in the configuration of a service and operate correctly in order for m_1 to provide its specified service. In practice, this means that if m_1 is to be configured into a service, m_2 must be configured in as well.

- Micro-protocol m_1 *includes* micro-protocol m_2 if m_1 implements a property strictly stronger than that implemented by m_2 without relying on m_2 being configured in the service. In practice, this means that m_1 and m_2 would be redundant if configured together into a service.

Configuration Graphs

Configuration graphs are a graphical method of representing configuration constraints caused by relations between micro-protocols in much the same way that dependency graphs represent relations between properties. In this graph, nodes represent micro-protocols, directed edges the dependency relation, node inclusion the inclusion relation, and unlabeled choice nodes the conflict relations. The configuration graph should also identify the minimal set of micro-protocols required to implement the service. This can be

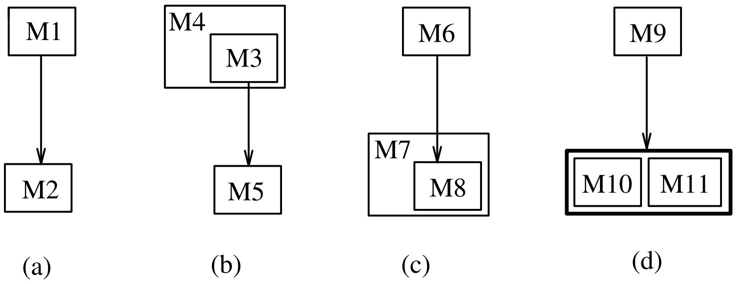


Figure 3.10: Configuration Graphs

done either by grouping together the minimal set of micro-protocols or by having a virtual *User* micro-protocol with dependency edges to those micro-protocols that are required to implement a minimal service.

Figure 3.10 illustrates these different concepts, where nodes labeled M1 to M11 represent micro-protocols. Figure 3.10(a) illustrates that micro-protocol M1 depends on M2. In 3.10(b), M3 is included in M4, with both depending on M5. In 3.10(c), M6 depends on M8, which is included in M7; since M7 includes M8, M8 can be replaced by M7 without affecting M6. Finally, in 3.10(d), M9 depends on a choice of two conflicting micro-protocols, M10 and M11. All these structures can be generalized to any number of micro-protocols.

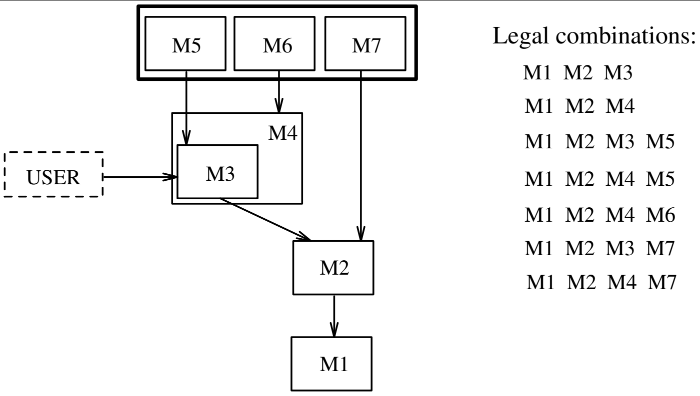


Figure 3.11: Example Configuration Graph

A configuration graph can be used as a tool for configuring customized services. The designer of a service first decides which properties are required and identifies the micro-protocols that implement those properties in the configuration graph. These micro-protocols are then included in the configuration, along with all micro-protocols on which the chosen ones depend. Any micro-protocol may be replaced by one that includes the original one. Only one micro-protocol from each choice node may be chosen and no micro-protocol m_1, such that micro-protocol m_2 is in the configuration and m_2 includes

m_1, may be chosen. The configuration graph can be used to generate and enumerate all possible different combinations of the micro-protocols.

Figure 3.11 illustrates an example configuration graph. In this graph, every configuration has to have micro-protocols M1 and M2, and M3 or M4. Other micro-protocols, M5, M6, and M7, are optional additions. In spite of the large number of conflicts and dependencies, the total number of operational configurations is 7.

3.4 Implementing the Event-Driven Model

3.4.1 Overview

The event-driven execution model can be implemented relatively easily as a collection of library routines in most multi-tasking systems using any programming language with function pointers and light-weight threads. For example, C or C++ on a Unix platform with a lightweight thread package is quite adequate. If the chosen programming language supports function pointers, event handlers can simply be implemented as functions and the event operations, such as register and deregister, reduce to keeping track of pointers to these functions. Light-weight processes, threads, or co-routines that share an address space make it easy to implement the full event semantics. For example, concurrent events can be implemented by creating a separate thread for each handler to be executed. Naturally, limited forms of the model can be implemented without concurrent threads. For example, sequential blocking events can be implemented using only ordinary procedure calls. Here, we provide an overview of three prototype systems that support this model.

3.4.2 SR Prototype

Initial experimentation was done using the SR concurrent programming language [AO93, AOC$^+$88] in the context of a reliable ordered group oriented multicast service [HS93]. In this prototype, each logical site hosting members of the multicast group is implemented as an SR *virtual machine*. A composite protocol is implemented as an SR *resource*, an object that contains local variables, procedures, and processes, and exports operations for use by other resources. The multicast composite protocol resource exports operations to the protocols above and below for transmitting messages up and down the protocol stack. In our experiments, each simulated site consists of three composite protocols: one that implements the multicast service, one that simulates the underlying unreliable network, and one that simulates the application. The latter two were degenerate composite protocols and did not implement full event handling.

An event handler in the prototype is simply a regular SR procedure. Association of an event handler with a named event is done by invoking a **register** operation with a *capability* (i.e., pointer) to the procedure as argument. The composite protocol resource maintains a table containing this event/handler association. Event handlers are triggered using the normal SR invocation mechanism; as a versatile concurrent programming language, SR supports operations for concurrent and sequential function calls, as well as blocking and nonblocking calls. In fact, these SR facilities served as inspiration for the different event attributes in the event-driven model.

The SR prototype does not provide explicit syntactic constructs for implementing micro-protocols. Rather, all micro-protocols, their event handlers, local data structures, and initialization sections, are simply placed together within the composite protocol resource. This means, of course, that all variables are shared and all names, including those of event handlers, are global. Problems such as name conflicts could be avoided by keeping the set of names used by different micro-protocols distinct. Different configurations of the service are created by copying the required micro-protocol sections into the composite protocol resource code.

Message transmission failures and machine failures are only simulated in the prototype. The network protocol simulates message transmission failures. Specifically, a multicast to N receivers is implemented by N point-to-point messages, so the program uses a probability distribution to decide if a particular point-to-point message will be transmitted or not. Simulating transmission failures turns out to be useful because with very large failure probabilities (0.2–0.5/transmission), short simulations are often sufficient to bring up interesting failure scenarios. Since SR does not allow virtual machines to be deleted, site failures have to be simulated in a more subtle way. In particular, such failures are simulated by having a shared variable, named *status*, indicate the status of the site as *operational* or *failed*. A failure is then simulated by the application protocol setting the variable to *failed*. When this occurs, the network and multicast protocols on that logical site cease forwarding messages, and are terminated along with the application protocol. To simulate recovery, the protocols are recreated following some specified time interval.

The SR prototype does not include all the features of the event-driven model as presented in this dissertation. In particular, the prototype does not support the **deregister** and **cancel_event** operations and does not allow the execution of event handlers to be ordered based on priorities. In particular, event handlers can only be ordered by changing the order in which the **register** operations occur. This proved tedious and provided the inspiration for including provisions to order the event handlers explicitly. Not having a language construct for micro-protocols is also a drawback and led to unintended name conflicts. Despite these shortcoming, the prototype was sufficient to suggest that this approach could be successfully used to build highly configurable services.

3.4.3 The *x*-kernel Prototype

The event-driven model has also been implemented as an extension of the *x*-kernel. This implementation augments the *x*-kernel's standard hierarchical object composition model with the ability to internally structure protocol objects. The result is a two-level model in which selected micro-protocols are first combined with a standard runtime system, or framework, to form a composite protocol. This composite protocol, whose external interface is indistinguishable from a standard *x*-kernel protocol, is then composed with other *x*-kernel protocols in the normal hierarchical way to realize the overall functionality required of the system. An initial prototype of the runtime framework has been completed, with a number of micro-protocol suites currently under development. Initial experiments with a group RPC micro-protocol suite show modest execution overhead [BS95]. The prototype executes on DecStation 5000/240s connected by a 10 Mbit Ethernet network running the Mach operating system. Details of this implementation can be found in [BS95, Bha96].

3.4.4 C++ Prototype

A prototype of the event-driven execution model has also been implemented using C++. The implementation of the model itself consists of approximately 1000 lines of code and uses the Sun Solaris operating system's thread package to implement event handling and other control aspects of the runtime system. Currently, the multiple sites of a distributed architecture are simulated within a single address space, although the code for all the micro-protocols and much of the runtime system would carry over unchanged to a true distributed implementation using C++. Using a simulated environment as an initial step has, of course, numerous advantages. For example, it facilitates rapid prototyping of micro-protocols since it is easier to execute test runs and collect results. It also makes it possible to control execution parameters to a degree not possible in a real system, including the number of sites, the message transmission times, and failure rates.

The prototyping environment is divided into three major portions that implement the application, network, and service layers, respectively. The application is simulated by class `User`, which generates application messages and receives messages from lower layers. In most of our experiments the `User` class is very simple—a few hundred lines of code—but could be of arbitrary size depending on the complexity of the application. One object of this class is created for each site in the simulated system.

The network is simulated by class `Network`, which implements the abstraction of an unreliable point-to-point and multicast communication medium. `Network` simulates the concurrency of a real distributed system by creating for each simulated site a separate thread that carries a copy of a message from the network to the service layer. Furthermore, `Network` implements a short transmission delay and generates communication failures by deciding for each message and destination whether or not to deliver the message based on random number generation. Network partitions are simulated by maintaining a table that specifies the partition for each site, so that a message from a given site is only delivered to a destination if it resides in the same partition. This table can be altered at runtime to simulate the creation and joining of partitions. The `Network` class is relatively simple, consisting of approximately 350 lines of code. A single `Network` object is created for each simulation.

The bulk of the prototype code is concentrated in the service layer. This layer includes the implementations of composite protocols and micro-protocols as C++ classes `CompositeProtocol` and `MicroProtocol`, respectively, and the service-specific micro-protocols and composite protocols. `CompositeProtocol` contains the runtime system of composite protocols, implementing the event-driven execution model and providing such operations as the registering and deregistering of events. It also implements the interactions with the layers above and below, such as providing operations for those layers to transfer messages to this layer. This class also contains the code that triggers predefined events. `MicroProtocol` is the base class from which the micro-protocols implementing the properties of the specified services are derived. This allows micro-protocols to be dealt with as uniform objects whenever possible. Appendix A gives a more detailed review of these base classes.

Each service such as RPC or membership is implemented as a derived class of `CompositeProtocol` by defining the service-specific events, shared data structures, and initialization. Typically, a service-specific composite protocol can be defined in a few hundred lines of code. The bulk of the code in each service is in the micro-protocols, each of which is typically from 50 to a few hundred lines of code. For example, in the membership service implementation described in chapter 5, the micro-protocols and their data structures make up about 7000 lines of code. In the experiments run so far, the service layer consists of one service-specific composite protocol class, an instance of which is created for each simulated site.

The system model provided by this prototyping environment is an asynchronous system where sites experience crash failures. The prototyping environment uses time provided by the system clock to model the progress of time, which is required, for example, for TIMEOUT events. This makes communication and computation in the system asynchronous since there is no guarantee with respect to the system clock time as to when a message will reach its destination or when an enabled event handler will be executed. The prototype could be made synchronous or even real-time by simulating the progress of time instead of using the system clock. Site crash failures are simulated by shutting down all micro-protocols in a controlled manner and zeroing out appropriate global data structures. Recoveries are simulated by recreating the micro-protocols objects. System startup is distinguished from recovery by the runtime triggering events STARTUP_EV and RECOVERY_EV, respectively, after the micro-protocol objects have been created.

The prototype has been used to implement the membership, system diagnosis, and group RPC services described in chapters 5, 6, and 7, respectively.

3.5 Conclusions

This chapter has outlined our approach to constructing configurable services and described the tools: the event-driven model, and message ordering, dependency and configuration graphs. We also discussed implementing the event-driven model and briefly described three prototype implementations. The rest of the dissertation is devoted to applications of this model and examples that further illustrate how the approach and the tools can be applied.

Our approach is different from the existing approaches to customization described in chapter 2, with the main differences being the flexibility of the event-driven model and the emphasis on customization based on abstract properties. Compared to hierarchical approaches such as Consul and Horus, our approach simplifies the design and implementation by preserving the natural, often non-hierarchical, interaction patterns between micro-protocols. Moreover, it distinguishes between services and micro-protocols, thereby allowing the service interfaces to be small and service-specific, while providing maximal configurability within a service. Hierarchical approaches can also suffer performance penalties caused by the large number of layers, leading into ad hoc solutions, such as the FAST protocol in Horus that by-passes portions of the protocol stack. Compared to function-based approaches such as Adaptive and Raid, our approach is more flexible and more extensible. In function-based approaches, dividing the operation of a service into specific functions makes it difficult to make configurable modules correspond to abstract properties and, in particular, the addition of new properties may require changing the division. Furthermore, a fine-grained, highly configurable, functional division may result in most configurations needing "null" modules that are only required to fill the function-slots but do not do any useful work. Finally, although a monolithic implementation can provide different variants of a service based on compile or runtime flags, a modular approach such as ours typically results in a more compact executable and an implementation that is easier to understand, to debug, and to extend with new properties.

Finally, it should be pointed out that event-based approaches have been used extensively to describe protocols and, to some extent, implement them. For example, an *x*-kernel protocol can be seen as an event-driven entity with handlers for events such as push, pop, open, and close. Recent work also includes event services [GJS92, SB95, MSS96], event-based programming [Ous96], and an event-based structuring approach where an application is constructed by customizing a standard framework with user-supplied handlers [Lew96]. Examples of this approach are Java AWT [Yu96], Taligent's CommonPoint [Tal96], and NeXT's OpenStep [Ne94].

CHAPTER 4

PROPERTIES OF MEMBERSHIP SERVICES

It is vital to maintain information about which computers are functioning and which have failed at any given time to build dependable distributed applications. This is often called the *membership problem*. A distributed service that maintains consistent information at all sites about the membership of a group of machines or, equivalently, processes is called a *membership service*, while the algorithm or implementation that realizes the service is called a *membership protocol*. Different variants of membership services are utilized for various purposes in computing, such as monitoring processors on a multiprocessor, sites in a token ring or bus, computers in a distributed computing system, processes in a distributed application level process group, or any other object or entity.

Membership services have proven to be fundamental for constructing systems and applications. The existing work can be classified based on the assumptions about the system model. In particular, some membership services assume a synchronous system where bounds are placed on network transmission time [Cri91, KGR91, KG94, EL90, LE90, SCA94], whereas other assume an asynchronous system where no such assumption is made [MSMA94, MAMSA94, DMS94, DMS95, EL95, AMMS$^+$93, MPS93a, RB91, SR93, ADKM92a, GT92, RFJ93, Bir85a, SM94, BDM95, CS95]. Distinctions can also be made based on the failure model. A majority of the work is based on the assumption that only crash failures will occur, while [Rei96] is based on the Byzantine failure model. Furthermore, *system diagnosis* [PMC67], which deals with the problem of detecting faulty processors, is closely related to membership. System diagnosis assumes a failure model where faulty processors can be detected by executing a test program on the processor [BMD93, BP90a, BGN90, BB91, BP90b, BB93, LYS93, Pel93, WHS95].

The different membership services provide a wide variety of different properties, ranging from ones that offer weak properties [RFJ93, GT92, Hil95] to others that guarantee strong properties [BSS91, AMMS$^+$95, DMS95]. The tradeoff is the strength of the guarantee versus the execution cost. For example, the property called *virtual synchrony* [BJ87] guarantees that messages reflecting membership change events are delivered to the application by the membership layer at every site at precisely the same point in the message stream. While making it easier to program many applications, virtual synchrony has an implementation cost in terms of extra messages and execution time, and also restricts the degree of concurrency between processes. As a result, other services have been implemented that offer properties that have smaller implementation cost and allow more concurrency, but at the expense of providing weaker guarantees for the application programmer. As might be expected, the semantics of the application has a strong influence on the type of membership protocol needed: some require strong properties, while others will execute correctly with something weaker.

Despite the above efforts, little has been done to examine the abstract properties important to membership independent of a given implementation. In this chapter, we attempt to rectify this shortcoming. We identify and specify these properties, and characterize relations between them using the message ordering graphs and dependency graphs that were introduced in chapter 3.

The advantages of developing an understanding of membership's constituent properties are numerous. For example, it helps clarify the structure and semantics of such services, which by their very nature are one of the most complicated, but also one of the most important, services in a distributed system. It also helps differentiate existing services, thereby assisting the distributed system developer in the choice of which is most appropriate for a given situation. Perhaps most importantly, it also facilitates the design of new services in which only those properties actually required by an application are included. Finally, it also forms the basis for constructing a configurable membership service, which will be outlined in chapter 5.

4.1 System Model and Notation

A membership service can be viewed as a protocol layer that generates *membership change messages* indicating changes in membership and forwards them to higher levels. Membership can be characterized in terms of any entity in a distributed system for which current status information is required, such as processes in a process group, processors, or larger entities such as entire computing systems. Here, *sites* are assumed to be the entity of interest, so that membership change messages refer to such events as site failures and recoveries within a specified group of interest. We use the term *group member* to refer to an unspecified site within this group.

An application can use the information in membership change messages in many ways. For example, it can be used to direct multicast messages to the current membership as seen by the application, to choose a leader of the group, or to make various decisions about the global state of the computation. Given this view, the properties of a membership service can be defined in terms of what membership change messages it generates and when they are delivered to the application with respect to other messages and real time [HS95].

Figure 4.1 illustrates the logical system structure. The *communication* and *membership* services add application and membership change messages, respectively, to the ordering graph. Although we separate them here logically, in practice the two components are often tightly interrelated, with dependencies between them. The communication component is responsible for realizing the required properties of the application level communication between group members. Some properties of membership services can be implemented without considering the form of the communication service, but most—especially properties involving ordering membership change messages with respect to

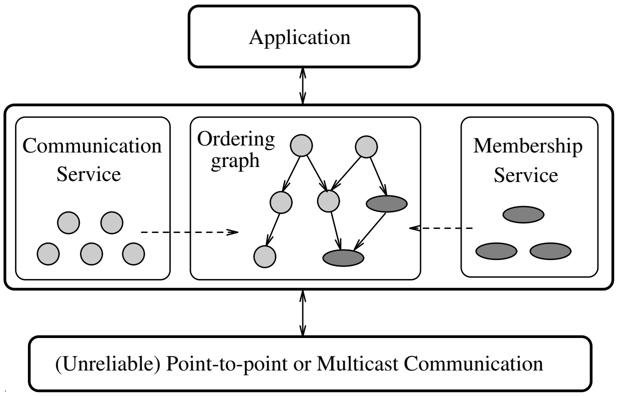


Figure 4.1: System Structure

application messages—require that application communication be based on reliable ordered multicasts that guarantee that every message is delivered by all functioning sites in some consistent order. The responsibility of the membership service is to guarantee that membership change messages appear in the ordering graph when and where they are supposed to according to the properties specified.

Given this structure, we specify a number of properties of membership services based on how these properties are reflected in message ordering graphs as defined in chapter 3. In our model, each site's lifetime consists of initial startup, followed by any number of alternating failures and recoveries. Each period during which a site is operational is identified by an incarnation number, which is assumed to be unique over the lifetime of a site. When required, the incarnation r of site A is denoted by A_r. Special membership change messages $F(A_i)$ and $R(A_j)$ are used to denote the failure (of incarnation i) and recovery (of incarnation j) of site A, respectively.

We augment the definitions of ordering graphs in chapter 3 by defining the concepts of view and membership. Define the *view* of the ordering graph at site A, $view_A$, to be the set of messages that have been delivered to the application on A. Thus, the relation between message delivery event $del_A(m)$, defined in chapter 3, and a view is:

$$\forall\ m \in \mathcal{M} : del_A(m) \Leftrightarrow m \in view_A$$

Note that in the above, $del_A(m)$ is used as a predicate. Such a predicate evaluates to false until the event occurs and to true afterwards. The view at some real time t, denoted $view_A(t)$, is defined to be the set of messages delivered on that site by time t.

We define the membership seen on site A, mem_A, to be a set of sites (including the incarnation numbers) that site A considers to be members in the group at the time. This set can be formally defined using the $del_A(m)$ event:

$$(del_A(R(B_i)) \wedge \neg del_A(F(B_i))) \Leftrightarrow B_i \in mem_A$$

Figure 4.2 illustrates the additional graphical notation used in this chapter.

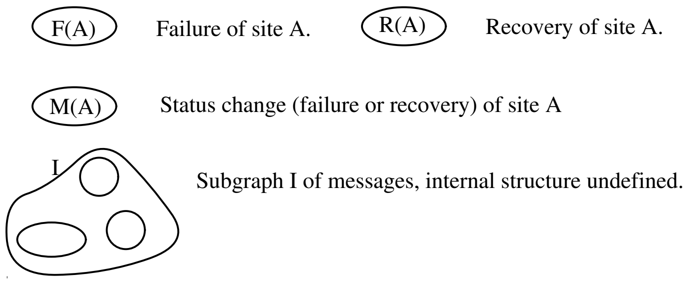


Figure 4.2: Ordering Graph Notation for Membership Service

4.2 Properties

In the following, we specify a number of properties of membership services based on how these properties are reflected in the ordering graphs on different sites. First, we address accuracy, liveness, and confidence properties, followed by agreement, ordering, startup and recovery, and partition handling properties.

4.2.1 Accuracy, Liveness, and Confidence Properties

Accuracy and liveness deal with reporting a change in status of a group member, either from functioning to non-functioning (*failure*), or from non-functioning to functioning (*recovery*). An *accurate* membership service is one that reports a change only if the change has indeed occurred (i.e., no false detections), while a *live* membership service is one that is guaranteed to report all changes eventually [BG93]. A special case of liveness is *bounded liveness*, where the failure or recovery is reported within a known bounded time.

Accuracy and liveness can be defined more formally in terms of ordering graphs. Let A and B be arbitrary sites and let $Failure(B_i)$ and $Recovery(B_i)$ signify events corresponding to incarnation i of site B failing or recovering, respectively. Then, a membership service is live if it guarantees that

$$Failure(B_i) \Rightarrow \Diamond(F(B_i) \in N(O_A)) \quad \wedge \quad Recovery(B_i) \Rightarrow \Diamond(R(B_i) \in N(O_A))$$

where $\Diamond$ is the temporal operator denoting "eventually". Note that in order for the definition to be satisfied, site A has to either not fail before receiving the membership change message or recover and upon recovery be notified of the change. The case of bounded liveness can be defined similarly, except that the membership change message for a change occurring at time t_1 must be delivered by some time t_2 such that $t_2 - t_1 < \Delta$, where Δ is a known constant. Likewise, a membership service is accurate if it guarantees that

$$Failure(B_i) \vee (F(B_i) \notin N(O_A)) \quad \wedge \quad Recovery(B_j) \vee (R(B_j) \notin N(O_A))$$

The liveness and/or accuracy of a membership service depends on the characteristics of the two steps involved in the process. That is, for a membership service to be live or accurate, it must first be able to detect a change in a live or accurate manner, and then be able to report the change on every site in a live or accurate manner. The change detection provides a local suspicion of a change, whereas the reporting of the change typically requires that some type of agreement is reached on the suspected change.

The change detection phase can be live, accurate, or both depending on the system model and the algorithm used. In asynchronous systems, it is impossible to have change detection that is both live and accurate [CT91, FLP85]. An example of an accurate detection that is not live is that of Mach, where the failed site notifies others about its own failure upon recovery [OIOP93]. Most membership services for asynchronous systems have chosen live but not accurate detection, for example, Isis [BSS91] and Consul [MPS93a]. The lack of accuracy in such systems comes from the use of timeouts to suspect the failure of a site, a technique that may trigger false suspicions. To deal with potentially inaccurate decisions, suspected sites that have in fact not failed are often isolated from the group and forced to fail and then recover before continuing execution. In synchronous systems, change detection based on timeouts is both accurate and live. Note, however, that a synchronous system is an abstraction that is maintained only as long as the bounded delivery time assumption is not violated. As a result, if this assumption is violated, a change detection algorithm that is intended to be live and accurate will lose its accuracy characteristics. This scenario is acknowledged and handled, for example, in the design of Mars [KGR91]. Although, in principle, it would be possible to have a change detection that is neither accurate nor live, in this chapter we only consider change detection that is at least live or accurate.

Typically, given an accurate detection, membership services do not generate spurious membership changes during the agreement phase, so accuracy is preserved. However, live detection is necessary but not sufficient to guarantee live service. For example, the membership protocol in Isis [RB91] that uses live detection based on timeouts has been shown not to be live in all situations [ACBMT95]. In synchronous systems, membership services are typically accurate and live [Cri91, KGR91].

In membership services in which change detection is inaccurate, the level of *confidence* indicates how certain it is that the suspected change has actually occurred. The typical way to increase confidence is to compare information from different sites before making a final decision. Thus, different levels of confidence can be defined by specifying how many sites must agree that a suspected change has occurred before a change indication is forwarded to the application. The possibilities range from a single site [RB91, RFJ93, SM94] to all functioning sites [MPS92]. In the following, *single site suspicion* is used to denote the former and *consensus* the latter. Any option between these two extremes is referred to as a *voted decision*. In general, the use of voted decisions has not been explored widely in the context of membership services. One exception is [Rei96], where voting is used to handle Byzantine failures.

Note that detecting failure can be dealt with separately from detecting recovery. A typical solution in asynchronous systems is to have failure detection be live but not accurate, with recovery detection being accurate but not live. This approach is natural since the most practical approach for detecting recovery is the receipt of a message sent by the recovered site.

4.2.2 Agreement Properties

The *agreement* property requires that any membership change message delivered to the application at one site eventually be delivered at all other sites. Figure 4.3 illustrates this concept. The property can be stated in terms of the ordering graph as follows: if a membership change message $M(C)$ appears in the ordering graph of one site, it will eventually appear in the ordering graph of all other sites. More formally, let A and B be arbitrary sites and $M(C)$ an arbitrary membership change message. For agreement to be satisfied, the following must hold for O_A and O_B:

$$M(C) \in N(O_A) \Rightarrow \Diamond(M(C) \in N(O_B))$$

Note that there are no ordering requirements between membership change messages, or between membership change messages and application messages.

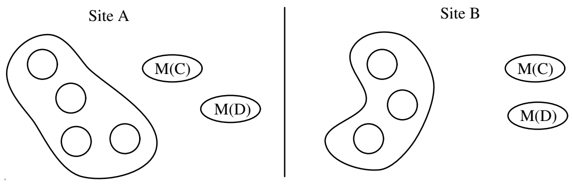


Figure 4.3: Agreement

If the underlying network can partition and the group memberships diverge in the different partitions, the above definition does not apply to sites in different partitions. In this case, the definition of agreement becomes the following:

$$\forall\ B \in mem_A : M(C) \in N(O_A) \Rightarrow \Diamond(M(C) \in N(O_B) \vee F(B) \in N(O_A))$$

This rule states that for two sites in the same partition, if one site observes a membership change, then any other site in partition will either observe the same membership change or be removed from the membership view.

Agreement specifies that the same membership change messages are delivered on all sites in the same partition. A weaker variant, *eventual agreement on views*, only ensures that all sites eventually reach the same membership set (or view), assuming no additional failures occur for a long enough period of time. Unlike regular agreement, with this property, the actual changes made by different sites may vary—for example, a site A may be considered to have failed and recovered on one site and not failed at all on a second—as long as the end result is the same. This property, although implemented by certain weak protocols such as [RFJ93], is insufficient for implementing message ordering properties. Hence, for the remaining properties, we assume that regular agreement is guaranteed. Furthermore, all the properties except the partition handling properties only apply to sites in the same partition.

4.2.3 Ordering Properties

Ordering properties specify constraints on the order in which membership change and application messages are inserted into the ordering graph and hence, the order in which they are delivered to the application.

4.2.3.1 FIFO Ordering of Membership Messages

The *FIFO ordering* property requires that membership change messages concerning any given single site be delivered to the application at every site in the same partition in the same order (Figure 4.4). In the ordering graph, this property requires that the membership changes of each individual site form a chain that is identical at every site. More formally, let A, B, and C be arbitrary sites in the same partition and $M(C_i)$ and $M(C_j)$ be arbitrary membership change messages indicating a status change of site C. Then, for FIFO order to be satisfied, the following two properties must hold:

$$M(C_i), M(C_j) \in N(O_A) \Rightarrow ((M(C_i) \rightarrow M(C_j)) \in O_A) \;\vee\; ((M(C_j) \rightarrow M(C_i)) \in O_A)$$

and

$$(M(C_i) \rightarrow M(C_j)) \in O_A \Rightarrow \Diamond((M(C_i) \rightarrow M(C_j)) \in O_B)$$

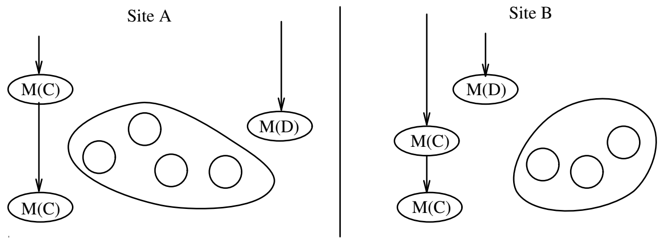


Figure 4.4: FIFO Order

FIFO order guarantees that when all membership change messages have been delivered, the membership at every site converges to the same view.

4.2.3.2 Total Ordering of Membership Messages

Total ordering requires that membership change messages be delivered to the application at every site in the same total order (Figure 4.5). In the ordering graph, total order requires that membership change messages form a single chain that is identical at all sites in the same partition. More formally, let A and B be arbitrary sites in the same partition, and $M(C)$ and $M(D)$ be arbitrary membership change messages. Then, for total order to be satisfied, the following two properties must hold:

$$M(C), M(D) \in N(O_A) \Rightarrow ((M(C) \rightarrow M(D)) \in O_A) \ \vee \ ((M(D) \rightarrow M(C)) \in O_A)$$

and

$$(M(C) \rightarrow M(D)) \in O_A \Rightarrow \Diamond((M(C) \rightarrow M(D)) \in O_B)$$

Note that total order here only applies to membership change messages and hence, makes no statement about the relative ordering of membership and applications messages at the different sites.

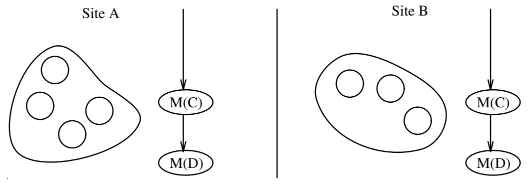


Figure 4.5: Total Order

Total order is a useful property for applications that rank processes or sites based on age, i.e., how long they have been members of the group, and then use that rank to reassign roles or tasks upon membership change. For example, an algorithm requiring a central coordinator could assign that role to the process with highest rank, with the second ranked taking over in case of failure. In order for the ranking seen on different sites to be identical, membership change messages must be processed in the same total order at each site.

The remaining ordering properties all order delivery of membership change messages with respect to application messages as well as other membership messages. As noted above, to establish such an ordering, application messages must be transmitted using an ordered reliable multicast service. Such a service guarantees that messages are delivered at all sites that remain functioning for the duration of the multicast and that are also within the same partition as the sending site. Sites that are in the process of joining the group may or may not receive the message. Furthermore, to simplify the presentation, we also assume that application messages are at least FIFO ordered.

4.2.3.3 Agreement on Last Message

The *agreement on last message* property requires that an agreed upon "final" message sent by a failed site be delivered to the application on each site prior to the membership change message announcing its failure (Figure 4.6). In the ordering graph, this property means that the membership change message indicating the failure of a site, say C, is a successor of the message c_i that all remaining sites agree is the last one to be delivered from the failed site. The notion that this is the last *agreed upon* message is important. There may, in fact, be a subsequent message, c_{i+1}, that was in transit when the agreement process was underway. Such a message will appear in the graph as a successor to the membership change message and be delivered to the application in the normal fashion. However, applications may choose to disregard this message as coming from a site that is no longer a valid member of the group.

More formally, for agreement on last message to be satisfied, there exists agreed last message c_i such that the following holds:

$$\exists\, c_i\ \forall\ A : (F(C) \in N(O_A)) \Rightarrow ((c_i \rightarrow F(C)) \in O_A\ \wedge\ \forall j > i : (F(C) \rightarrow c_j) \in O_A)$$

If C did not send any messages before it failed, the agreed last message will be the membership change message $R(C)$.

This property is useful in applications where a consistent distributed state needs to be maintained. In such situations, the final message may cause a state change, so agreement ensures that the change is applied either at all sites or at no site.

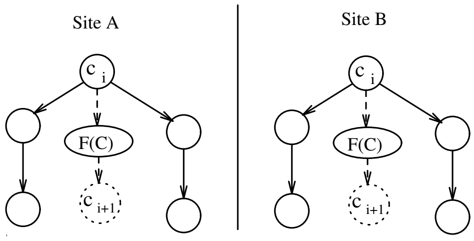


Figure 4.6: Agreement on Last Message

4.2.3.4 Agreement on First Message

The *agreement on first message* property requires that all sites start delivering messages from a new or recovering process to the application starting from the same first message (Figure 4.7). In the ordering graph, this property is represented by the appropriate membership change message being a predecessor of the agreed upon first message. More formally, for agreement on first message to be satisfied there exists agreed first message c_i such that the following holds:

$$\exists\, c_i\ \forall\ A \in \mathcal{S} : (c_i \in N(O_A)) \Rightarrow ((R(C) \rightarrow c_i) \in O_A\ \wedge\ \forall j < i : (R(C) \rightarrow c_j) \notin O_A)$$

If C does not send any messages before it fails, the agreed first message will be the membership change message $F(C)$.

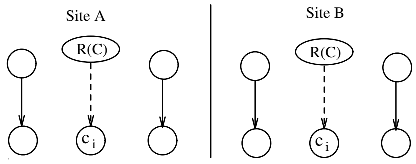


Figure 4.7: Agreement on First Message

Agreement on the first and last message is important if the communication subsystem is required to guarantee *validity*, defined as the property that only messages from group members are delivered to the application. If valid messages are defined to be those succeeding the membership change message indicating the recovery of a site and preceding the message indicating the failure of a site, the set of valid messages on different sites will be exactly the same. Validity is primarily a property of the multicast layer rather than membership, and so is not considered further in this chapter.

4.2.3.5 Agreement on Successors

The *agreement on successors* property requires that all sites deliver a membership change message before any message in an agreed upon successor set is delivered (Figure 4.8). For example, if site C is recovering, then the successor set might be the set of messages that will be received after the recovery message $R(C)$ on all sites. Unlike previous properties, agreement on successors requires ordering the membership change message with respect to application messages sent by all sites, not just the site that failed or recovered.

Formally, first define *cut*, $CUT(O_A)$, to be a set of nodes $S \in N(O_A)$ such that, for any node $m \notin S$, either $\exists s \in S : m \rightarrow s$ or $\exists s \in S : s \rightarrow m$, but not both. Let $IsCut(S, O_A)$ be a predicate that evaluates to true if the set of nodes S is a cut in ordering graph O_A, and false otherwise. Finally, let $succ_A(m)$ be the set of immediate successors of message m in the ordering graph of site A. Then, for agreement on successors to be satisfied, the following must hold for A and B in the same partition:

$$succ_A(M(C)) = succ_B(M(C))\ \wedge\ IsCut(succ_A(M(C)), O_A)\ \wedge\ IsCut(succ_B(M(C)), O_B)$$

Among other things, this property is useful for determining *message stability*, where a message is stable at the sending site once it has been acknowledged by every other operational site [PBS89]. If m is an agreed successor of $R(C)$, every site knows that m will have to be acknowledged by site C to be considered stable.

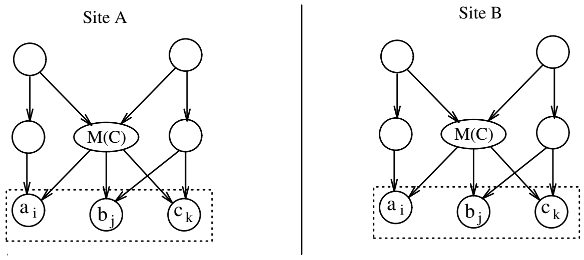


Figure 4.8: Agreement on Successors

4.2.3.6 Agreement on Predecessors

The *agreement on predecessors* property requires that a set of predecessor messages sent from every site be agreed upon and delivered to the application prior to the membership change message (Figure 4.9). Note that there may be some messages that are not ordered with respect to the membership change message.

Now, let A, B, C and $M(C)$ be defined as usual. For agreement on predecessors to be satisfied, the following must hold for A and B in the same partition:

$$pred_A(M(C)) = pred_B(M(C)) \wedge IsCut(pred_A(M(C)), O_A) \wedge IsCut(pred_B(M(C)), O_B)$$

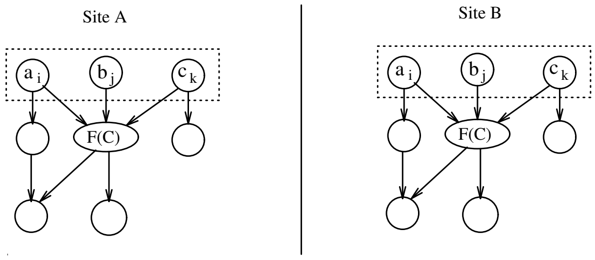


Figure 4.9: Agreement on Predecessors

This property is valuable because it can be used to represent the uncertainty that occurs when sites fail. Consider the case where a site C fails and the agreed-upon predecessor set consists of exactly those messages known to have been received by the communication layer at C prior to the failure. Messages not known to have been received at C—because no acknowledgment message has been received, for example—are not ordered with respect to the membership change message and so, are excluded from this set.

This information can be exploited to construct a variant of atomic multicast similar to those described in [KGR91, VM90] in which a message is only delivered to the application if it can be guaranteed to have been received at all sites to which it was addressed. Suppose that some site C in the destination set of multicast message m_i fails.

If m_i is in the predecessor set of the membership change message indicating the failure of C, then C received m_i and so m_i can be delivered at all sites. If, on the other hand, m_i is not ordered with respect to the membership message, then it is unknown whether C received m_i before failing or not. In this case, m_i is dropped at every site, thereby preserving the semantics of atomic multicast.

The specifications of agreement on successors and agreement on predecessors properties do not specify which messages constitute the agreed predecessor and successor sets. A family of these properties could be defined by specifying exactly which messages constitute these sets. For example, in the case of agreement on predecessors, the agreed predecessor set of a failure message could be defined as any of the following:

- Messages that were guaranteed to have been received by the failed site before it failed (for example, messages that were acknowledged by the failed site).
- Messages that were potentially received by the failed site before it failed (for example, messages that were sent before the failure was reported.)

An atomic multicast algorithm such as described above would, in most cases, prefer the first definition of predecessor set, whereas for some other applications the second definition may be more appropriate.

4.2.3.7 Virtual Synchrony

Virtual synchrony restricts the delivery order of application and membership change messages in such a way that it appears to the application as if events are occurring synchronously even though they are actually occurring on different sites at different times [BSS91]. Virtual synchrony is easy to explain in the ordering graph, as illustrated in Figure 4.10. Relative to membership, this property requires agreement among all operational sites on a division of the message stream such that each message is either in an agreed predecessor set to the membership change message or in an agreed successor set. In other words, virtual synchrony essentially creates an agreed cut in the message flow.

Let $\cap$ denote the intersection of two ordering graphs, i.e., the sets of vertices and edges that are common to both ordering graphs, and let $O_{A\cap B}$ denote an ordering graph that is the result of such an intersection. Then, for virtual synchrony to be satisfied, the following must hold for arbitrary A and B in the same partition:

$$\forall m_i : \Diamond((M(C) \rightarrow m_i) \in O_{A\cap B}) \vee ((m_i \rightarrow M(C)) \in O_{A\cap B})$$

As extensively discussed in the Isis literature, virtual synchrony makes it easy to write distributed applications. This is especially true for applications that can be viewed as replicated state machines that change their state when they receive application messages and membership changes [Sch90].

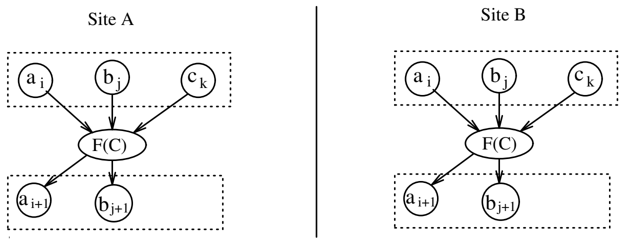


Figure 4.10: Virtual Synchrony

Note that, although virtual synchrony is closely related to agreement on successors and predecessors, it is not identical to combining these two properties. This follows because the combination does not require that every message be in one set or the other, whereas virtual synchrony does. Note also that, in contrast to the common definition of virtual synchrony, we choose here to define it without requiring that it also guarantee a total order of membership change messages.

4.2.3.8 Extended Virtual Synchrony

One drawback of virtual synchrony is that it does not necessarily relate the view of membership at the time a message is sent to the collection of sites that actually receive the message. For example, site A may multicast a message a_i when the membership is $\{A$, B, $C\}$, but before the message is received, another site D joins the group. Under virtual synchrony, the membership change message could be delivered to the application before a_i, thereby resulting in the delivery of the message to the application at D as well as A, B, and C. This is acceptable in cases where the actual membership of the destination group is not important, but stronger guarantees are useful in some cases. *Extended virtual synchrony* extends virtual synchrony by guaranteeing that all messages sent under the old membership are also delivered before the membership change message [AMMS+93].

Extended virtual synchrony can be defined more formally as follows. Let $ev_1 < ev_2$ denote event ev_1 happening before event ev_2 at the application on the same site. (Note that $<$ is a total ordering, assuming that the application is single threaded.) Then, for extended virtual synchrony to be satisfied, the following must hold in A and B in the same partition:

$$\forall a_i : (send(a_i) < del_A(M(C))) \Rightarrow \Diamond((a_i \to M(C)) \in O_B)$$

and

$$\forall a_i : (del_A(M(C)) < send(a_i)) \Rightarrow \Diamond((M(C) \to a_i) \in O_B)$$

Note that the second rule is implicitly implemented by the communication subsystem provided that the communication is at least causally ordered. Figure 4.11 illustrates this property; the shaded circles represent messages that were sent before the sender received the membership change message $M(C)$.

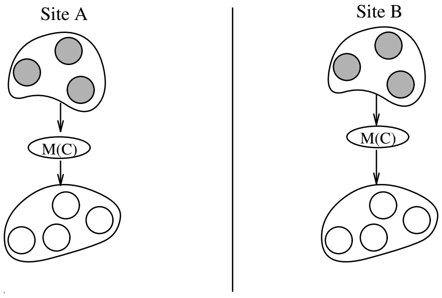


Figure 4.11: Extended Virtual Synchrony

Extended virtual synchrony has been explored in a number of papers, especially [AMMS+93] and [MAMSA94]. Our definition only addresses the ordering aspects of the property as they relate to membership change messages, and as such, does not include the full functionality of extended virtual synchrony as defined in those papers.

4.2.4 Bounded Change Properties

Ordering properties constrain the order in which state changes related to membership occur at different sites, but leave unspecified any notion of when a site makes a change relative to the other sites. *Bounded change properties* extends ordering by adding bounds on when such changes must be applied.

4.2.4.1 External Synchrony

External synchrony guarantees that if a site delivers a given membership change message, all other sites have either already delivered the message or are in a transition state in which delivery is imminent [RFJ93]. Having this property ensures that sites move into a new membership state with some degree of coordination, and that all sites have consistent membership information, modulo sites undergoing a transition. The idea is, in fact, related to the concept of barrier synchronization in parallel programs, in which execution at all sites must reach the barrier—i.e., move into the transition state—before any site can proceed—i.e., make the membership change. Following [RFJ93], we denote this transition state as state 0. Since underlying layers interact with the application only through messages, any protocol implementing external synchrony requires an extra message to generate a transition into state 0. We call this the *transition message* for membership change $M(C)$ and denote it as $Tr(M(C))$.

External synchrony can now be defined more formally for arbitrary sites A and B in the same partition as follows:

$$\forall\ M(C) : del_A(M(C)) \Rightarrow \Box(Tr(M(C)) \in view_B)$$

This definition specifies that once a membership change message $M(C)$ is delivered on an arbitrary site A, the corresponding transition message $Tr(M(C))$ has already been delivered on all sites, i.e., is in the view on all sites. Therefore, all sites have either delivered $M(C)$ or are in the transition state prior to delivering $M(C)$.

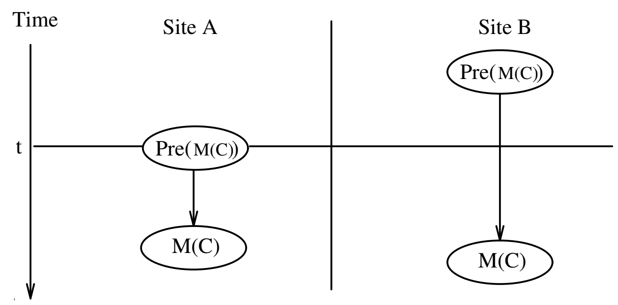


Figure 4.12: External Synchrony

Figure 4.12 illustrates this property. Based on the definition, there is some global time t such that all sites deliver the prepare message prior to t and the membership change message after t. Note, however, that although this definition references t, implementation of this property does not require access to a global time source. For example, the barrier could be implemented by having each site multicast a message to the group after receiving the transition message, and waiting until there is a message from all other sites before delivering the actual membership change message.

External synchrony is useful in a number of situations, especially in cases where an application must access an external device or send a message to a process outside the group. As an example, consider an application where a group leader is expected to take some action at a given time, where the action must be executed exactly once. Suppose further that a membership change that results in a leadership change happens to occur close to this time. Without external synchrony, there may be moments in real time when two different group members are designated as the leader on different sites, thereby potentially resulting in the external operation being executed more than once. External synchrony prevents this by guaranteeing that all sites share the same view of the membership or are knowingly in a transition state.

4.2.4.2 Timebound Synchrony

Timebound synchrony is a property of membership services in synchronous systems in which every site delivers a given membership change message within some known interval of real time [KGR91, Cri91]. The property has the same general applicability as external synchrony, but reduces the synchronization overhead by shrinking the window during which the membership is not identical on all sites. Also, it is important to have this property in most real-time systems, so that the system can respond in a predictable and timely manner to external events.

Timebound synchrony can be defined formally for arbitrary sites A and B in the same partition as follows:

$$\forall\ M(C) : (time(del_A(M(C))) = t_i\ \wedge\ time(del_B(M(C))) = t_j) \Rightarrow |t_i - t_j| < \Delta$$

where Δ is a known fixed constant. Figure 4.13 illustrates this concept. In the figure, dashed lines are used to represent the point in real time where the membership change message is delivered to the application.

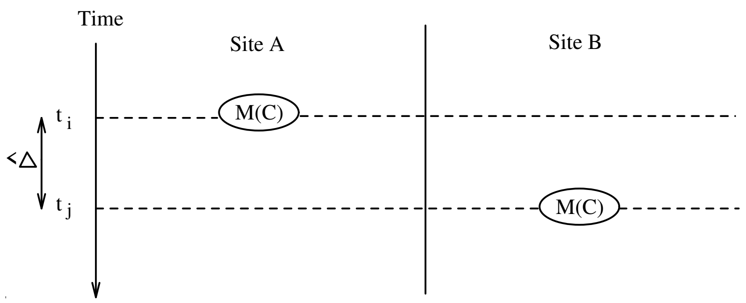


Figure 4.13: Timebound Synchrony

4.2.5 Startup and Recovery Properties

4.2.5.1 Startup

Two general approaches have be identified for coordinating the startup of a group: *collective startup* and *individual startup*. In the first case, the initial membership of the group is assumed to be known in advance, with all sites starting at approximately the same time. In the second case, each site starts with a membership consisting only of itself and sites merge membership views as they learn of one another. Collective startup is generally easier to handle since sites are known *a priori*, and can have implementation advantages if the initial membership is also assumed to be the maximum set of sites that might be group members [MPS93a]. Individual startup is more general, but also more complex. For example, this approach requires some known external mechanism for locating other sites, such as a shared name server.

To ensure that the root of the ordering graph on each site is well-defined, a startup message Su must be generated and delivered to the application before the application is allowed to send or receive other messages. This message carries with it the initial membership, which in the case of collective startup is a list of sites and in the case of individual startup is the identity of the site itself. More formally, if A is an arbitrary site, the following property is guaranteed for all messages in O_A:

$$\forall\, m \in N(O_A) : (Su \rightarrow m) \in O_A$$

Startup is closely related to the way in which network partitions are handled, so further details on execution options are deferred until section 4.2.6 below.

4.2.5.2 Recovery

Recovery involves restarting a site and reintegrating it back into the group. The problem includes recovering the application process, of course, but here we focus exclusively on aspects of recovery that impact the membership layer and the properties that it guarantees to the application.

The basic recovery requirement for membership is that the membership information of the recovering site C be re-initialized to a valid state, where the details depend on the properties being guaranteed. For example, if no agreement or ordering properties are guaranteed, the recovering site can use the last membership view it had before failing or a view consisting only of itself. On the other hand, if agreement or any ordering properties are being enforced, the membership information as well as the ordering graph at the recovering site has to be brought up to date.

We assume the recovering site will receive a special $Rec(C)$ message that includes all the necessary information to reestablish the state of the membership service. In particular, in addition to the information in the corresponding $R(C)$ message, $Rec(C)$ contains information about the current membership in the group. Other sites consider the recovering site C to be a group member after the $R(C)$ message has been delivered to the application. If the membership service guarantees some ordering with respect to application messages, any message that is agreed to be after $R(C)$ must be delivered at C after $Rec(C)$, while messages before $R(C)$ must not be delivered at C. Figure 4.14 illustrates these ordering graphs. Note that the shaded messages in the figure are not ordered with respect to $R(C)$ or $Rec(C)$, which means that they may or may not be delivered to the application on C.

More formally, let A and B be arbitrary sites in the same partition and m be an arbitrary (application or membership) message. For recovery, the following must be guaranteed in the ordering graph O_C for the membership state on site C to be consistent with the other sites in the group:

$$((m \rightarrow R(C)) \in O_A) \Rightarrow \Box(m \notin N(O_C))$$

and

$$(R(C) \rightarrow m) \in O_{A \cap B} \Rightarrow \Diamond((Rec(C) \rightarrow m) \in O_C)$$

Note, in particular, that there will be ordering guarantees at C only if similar ordering guarantees are being enforced on every other site as well.

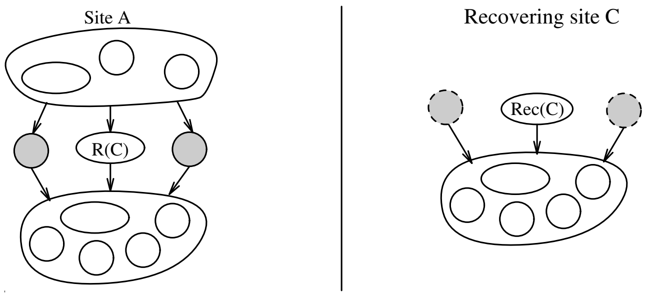


Figure 4.14: Ordering Guarantees at Recovery

4.2.6 Partition Handling Properties

4.2.6.1 Overview

Partition handling properties specify how the system behaves when a network partition occurs and when it is subsequently corrected. A network partition occurs when a subset of sites in a group is unable to communicate with the remainder of the sites. Partitions may be caused by disconnection of the underlying network or by problems such as network congestion or an overloaded gateway processor.

A number of different approaches are used in membership services to deal with partitions. One common approach is simply to assume they will not occur [Cri91, KGR91, MPS92]. This can be justified by increasing the connectivity of the network or by using other architectural assumptions. If this assumption is not valid, there are a number of ways to deal with multiple partitions [AMMS+95, DMS94, DMS95, MAMSA94, RFJ93, RB91, SM94]. We divide the policies for dealing with multiple partitions into three classes: the policy used at the time the partition occurs (*partition time*), how operation proceeds while the sites are partitioned (*partitioned operation*), and how sites in separate partitions are merged when communication is reestablished (*partition join*).

The operation of the membership layer is independent of issues regarding application state, so here we concentrate on describing guarantees or services that membership provides to the application. Specifically, the focus in this section is on describing properties that are enforced when the partition occurs and when separated sites are subsequently rejoined to reform the original group. The properties discussed in previous sections (e.g., agreement, message ordering) remain relevant even when partitions occur, although each is now enforced separately within each partition. We assume that membership operates continuously in all partitions, maintaining its own view of the membership and implementing the properties required by the application. It also forwards membership change messages to the application level on each site as usual, independent of policy choices made at that level.

4.2.6.2 Partition Time

A partition is impossible to distinguish from a site failure in a distributed system. Therefore, when a partition occurs, a basic membership protocol would forward a stream of membership change messages to the application, each indicating the failure of one of the sites in the other partition. We call this the *individual notification* property. *Collective notification* expands this notion by grouping together failure notifications for all the sites in a partition and forwarding them to the application in a single message. This message is ordered according to whatever criteria is being used to order individual membership change messages.

Collective notification is useful from the application's perspective, since it can be used to avoid a lengthy transition period in which the failures of multiple sites must be processed in succession. It is also straightforward to implement if the membership service already provides properties that require consensus of all sites, such as ordering with respect to application messages. In such cases, sites in the other partition will fail to participate in the required protocol and can therefore be collectively identified. This property allows agreement on the entire group of sites to be performed at once and forwarded to the application in a single message.

To realize the functionality of collective notification, we augment the failure notification message to indicate the failure of multiple sites A, B, ..., using the notation $F(A, B, \ldots)$. The analogous process occurs in both (or, in general, in all) partitions. Note that the group membership in the two partitions are non-overlapping after the failure notification messages are delivered.

Collective notification can be defined more formally as follows. Without loss of generality, assume that the partition results in the sites being divided into two sets of sites P and Q whose intersection is empty. Let $A, B \in P$, $C \in Q$, and D be an arbitrary site in either partition. Furthermore, let F_P and F_Q be the respective collective notification messages delivered to the sites in P and Q; $S \in F_P$ is used to denote that site S is included in the failure notification message F_P, and similarly for $S \in F_Q$. Then, collective notification can be characterized as ensuring the following for the ordering graphs of A, B, and C:

$$F_P \in N(O_A) \Rightarrow \Diamond(F_P \in N(O_B))$$

and

$$(M(D) \rightarrow F_P) \in O_A \Rightarrow \Diamond((M(D) \rightarrow F_P) \in O_B)$$

and

$$(F_P \rightarrow M(D)) \in O_A \Rightarrow \Diamond((F_P \rightarrow M(D)) \in O_B)$$

and

$$\forall D \in \mathcal{S} : (D \in P \Rightarrow D \in F_Q) \;\wedge\; (D \in Q \Rightarrow D \in F_P)$$

The first rule states that agreement is reached within each partition on collective notification messages, while the second and third guarantee that any message ordering constraints that apply are enforced within each partition. Since $P \cap Q = \emptyset$, the fourth rule ensures that the membership views in P and Q are non-overlapping after the failure notifications have been delivered.

Figure 4.15 illustrates this property, where shaded nodes represent membership change messages reporting changes that happened prior to the partition, the dashed line represents the time when the partition occurred, and F_P and F_Q are as above.

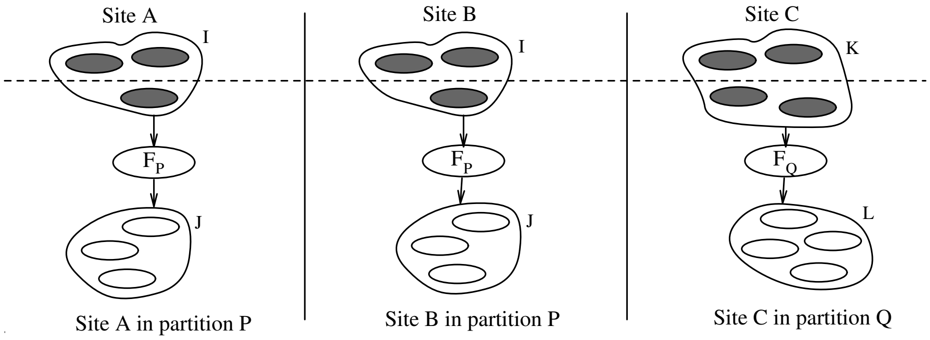


Figure 4.15: Collective Notification

4.2.6.3 Partitioned Operation

A number of different policies are available for the partitioned operation. If no special measures are taken after a network partition occurs, the computation will continue independently in each partition. We call this the *continued operation* policy. Sometimes this option is not appropriate because it will lead into inconsistency of the application state in the different partitions. The inconsistency can be avoided by requiring that computation continue only in one partition, such as the one with the majority of sites. We call this the *majority operation* policy. This approach has been chosen, for example, in [RB91, SM94]. The drawback of this approach, of course, is that it halts the application's execution in the rest of the system, thereby potentially affecting the progress or availability of the application. Alternatively, the computation may be allowed to continue in a limited form that does not lead into inconsistent state or only result in inconsistencies that are easy to resolve after the partitions are rejoined.

The specific policy for partitioned operation does not directly affect operation of the membership layer. However, membership can provide support for the application layer in certain cases. For example, a majority predicate of the type needed to realize majority operation is easy to implement in the membership layer, assuming that the service guarantees agreement and total order, and has knowledge about the maximum size of the group. To notify the application when a site is no longer a member of the majority group, a failure notification message $F(C)$ is augmented with an extra field indicating whether the (presumed) failure of C has caused the group size to shrink to the point that a majority can no longer be guaranteed. Note that such an indication only implies the *possibility* of a partition, not its actual existence; for example, so many sites may have actually failed that only a minority of the sites remain functioning. Similarly, a recovery message $R(C)$ is augmented to indicate whether the recovery of a site C has brought the group size back over the majority threshold. We call this the *consistent minority/majority status* property of membership services, or *majority status* for short.

4.2.6.4 Partition Join

When a partition is repaired, a basic membership algorithm would integrate each site from the other partition into its group membership individually. This will result in membership views being gradually merged. This type of partition join is only appropriate for a very limited set of applications. In particular, since during this gradual merging, the membership views on different sites will be inconsistent and overlapping, properties such as agreement or ordering are not well-defined while this merging is in progress. To be able to define properties such as agreement for joining partitions, the join operation must be atomic. In the following, we introduce two different policies for partition join that both allow properties such as agreement be defined. The first is a *collective join* policy, where partitions merge their memberships as one atomic membership change that is consistently ordered with respect to other membership change messages. The second is an *asymmetric join* policy, where partition join is reduced to sites in the minority partition simulating failure and joining the majority partition as individual recovering sites.

Collective Join

A special *merge* message is used to implement the collective join. This message includes a list of the members of the new group and is totally ordered with respect to other membership change messages. Figure 4.16 illustrates this property, where the shaded nodes represent membership change messages in the new membership after the merge and the larger circle in the middle is the merge message.

Collective join can be defined more formally as follows. As above, let P and Q be the set of sites in each partition, $A \in P$, $B \in Q$, and C be an arbitrary site in either partition. Furthermore, let $Me(P,Q)$ be the merge message indicating the merging of P and Q. Then, collective join can be characterized as ensuring the following for the ordering graphs of A and B:

$$\forall M(C) : (M(C) \rightarrow Me(P,Q)) \in O_A \Rightarrow M(C) \notin N(O_B)$$

and

$$\forall M(C) : (Me(P,Q) \rightarrow M(C)) \in O_A \Rightarrow \Diamond((Me(P,Q) \rightarrow M(C)) \in O_B)$$

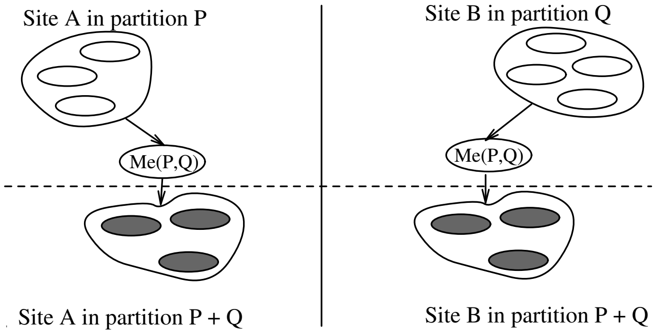


Figure 4.16: Collective Join

Asymmetric Join

The basic idea of asymmetrically joining multiple partitions is to reduce partition join to simple site recovery by forcing the sites in one of the partitions to fail and join the remaining partition as individual sites. This approach is adequate for some applications that cannot afford to enforce a single active partition. For this approach, we define a *domination* predicate that is evaluated by each site to decide if the site must fail and join the other partition. Domination can be defined in any number of ways, as long as the computation is deterministic and for any two partitions, one always dominates the other. Examples of possible predicates are "partition with more new updates since partitioning dominates", "larger partition dominates smaller", or a combination of the two. Figure 4.17 illustrates the membership change messages seen at the dominating and the dominated partition.

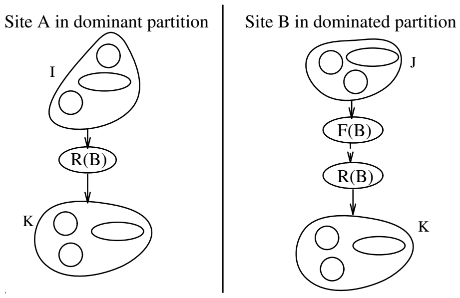


Figure 4.17: Asymmetric Join of Partitions

Asymmetric join can be defined more formally as follows. Let P be the set of sites in the dominated partition and A an arbitrary site in P. Then, asymmetric join can be characterized as ensuring the following for the ordering graph of A:

$$\forall A \in P : (F(A) \rightarrow Rec(A)) \in O_A$$

Agreement and total ordering of membership change messages are required for the domination predicate to work in the general case.

4.2.6.5 Ordering Partition Handling Messages

When a partition occurs, the ordering graphs of the sites in the two separated subgroups will generally evolve differently. Subsequent membership changes in one partition will result in a membership change message being issued in one subgroup but not the other, while application messages will also appear in the ordering graphs of only one set of sites. As a result, when collective notification is used, the relevant failure notification and join messages—the $F(...)$ and $Me(...)$ messages from above—delineate boundaries in the ordering graph at which the graphs at different sites diverge and then reconverge, respectively. In some sense, then, a failure notification message can be viewed as creating two independent streams of messages to be delivered, one in each partition, while a merge message can be viewed as merging the two streams back into one. This property distinguishes such *partition handling messages* from normal membership change messages, which are delivered as part of a single stream.

An implication of the differences between the two types of membership change messages is that the ordering properties discussed above in section 4.2.3 cannot be used directly to argue about ordering properties of partition handling messages. For example, although messages sent after receiving a merge message can easily be ordered after that message, it is less clear how to order messages that were sent by members of the separate partitions before receiving the merge message, or even to which sites they should be delivered. Perhaps the simplest solution is to deliver such messages only to sites that were in the partition in which they were sent. This strategy is, however, contrary to the semantics implemented by systems such as Psync [PBS89], which automatically propagates messages of this type to all sites for recovery purposes by virtue of its negative acknowledgment scheme for retransmitting lost messages.

Extended virtual synchrony is an example of a stronger property that can be augmented to include partition handling messages. This *extended virtual synchrony with partitions* property requires that messages sent before receiving the merge message be delivered before that message, and analogously, that all messages sent before receiving the failure notification message be delivered before that message. Note, of course, that this guarantee only applies within each partition. Figure 4.18 illustrates this property for partition join.

This property can be defined more formally as follows. As above, let P and Q be the set of sites in the partitions; F_P and F_Q be the failure notification messages delivered to sites in P and Q, respectively; and $Me(P,Q)$ be the merge message joining P and Q. Furthermore, assume A, B, and C are arbitrary sites such that $A, B \in P$ and $C \in Q$. Then, extended virtual synchrony with partitions can be characterized as ensuring the following for the ordering graphs of A and B at the time of partition:

$$\forall a_i : (send(a_i) < del_A(F_P)) \Rightarrow \Diamond((a_i \rightarrow F_P) \in O_B)$$

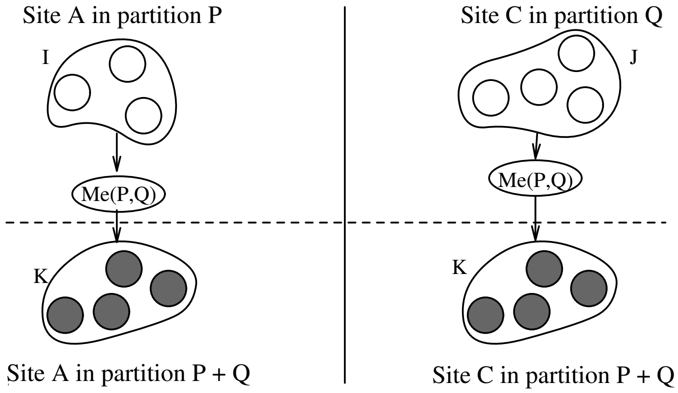


Figure 4.18: Extended Virtual Synchrony with Partitions

and

$$\forall a_i : (del_A(F_P) < send(a_i)) \Rightarrow \Diamond((F_P \rightarrow a_i) \in O_B)$$

An analogous property holds for sites in partition Q with respect to F_Q. Similarly, the following holds at the time of partition join:

$$\forall a_i : (send(a_i) < del_A(Me(P,Q))) \Rightarrow \Diamond((a_i \rightarrow Me(P,Q)) \in O_B) \ \wedge \ \Box(a_i \notin N(O_C))$$

and

$$\forall a_i : (del_A(Me(P,Q)) < send(a_i)) \Rightarrow \Diamond((Me(P,Q) \rightarrow a_i) \in O_{B \cap C})$$

Among other things, this property simplifies the problems associated with merging application states after a partition. In particular, since all application messages are either before or after the merge message, a consistent cut is created in the ordering graphs of all sites. To implement a merge of the application states, then, messages carrying the respective states can be sent from each site immediately after the merge message, with the assurance that they will be delivered after the merge and before any subsequent application messages. Of course, the difficult semantic problems associated with merging application states remain.

Our extended virtual synchrony with partitions property is derived from extended virtual synchrony as described in [MAMSA94]. The algorithm described in that paper provides the guarantees outlined above, plus additional ordering properties for messages that are sent around the time that the partition occurs.

4.3 Relations between Properties

Arbitrary combinations of the membership properties are not feasible because of relations between the properties that affect configurability as described in chapter 3. For example, totally ordering membership messages is impossible unless all membership messages are

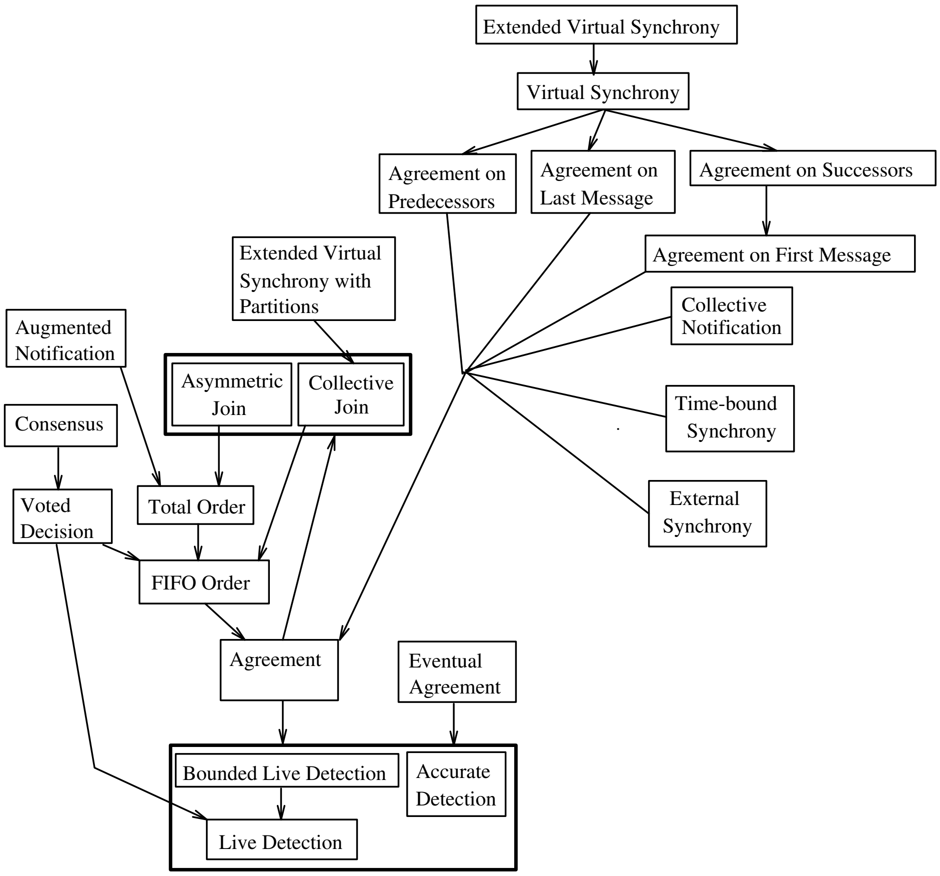


Figure 4.19: Membership Dependency Graph

in the ordering graphs of all sites, so total order depends on agreement. Also, some properties are weaker or stronger than others. For example, extended virtual synchrony is stronger than virtual synchrony in the sense that it satisfies virtual synchrony, but virtual synchrony in general does not necessarily satisfy extended virtual synchrony. Formally, relations such as these reduce to dependency, i.e, extended virtual synchrony depends on virtual synchrony.

Figure 4.19 gives the dependency graph containing the properties discussed in this chapter. To simplify the figure, dependencies between membership and multicast properties have been omitted. In particular, all ordering properties with respect to application messages depend on at least FIFO ordered reliable multicast. The graph is based on certain assumptions. First, we assume an asynchronous computing environment, making it impossible for change detection to be both accurate and live. Second, we assume that membership service has to be able to handle multiple partitions and, in particular, be able to join two partitions. If no partitions are expected or if the partitions are not required to join, the dependency from agreement to asymmetric or collective join is not required.

The dependency graph represents the relations between properties, and therefore all possible legitimate combinations of properties. The simplest possible membership services base the local view of the membership only on local live or accurate detection. More advanced services provide agreement augmented with various ordering and other properties.

Most of the relations in the dependency graph are relatively obvious. In the following, we provide proofs to some of the less obvious ones to illustrate the proof techniques. First consider the dependency relations, where dependency is formally defined as in section 3.1.2:

$$dep(p_i, p_j) : \forall\ s \in \textit{SYS}\ \ \forall\ e_s : sat(e_s, p_i) \Rightarrow sat(e_s, p_j).$$

Some dependencies are easy to prove directly based on this definition, that is, by showing that if property p_i is satisfied for any execution (of any system), then p_j is also satisfied for this execution. Theorem 1 below is an example of this type of proof. Alternatively, the definition can be expressed equivalently as:

$$dep(p_i, p_j) : \forall\ s \in \textit{SYS}\ \ \forall\ e_s : \neg sat(e_s, p_j) \Rightarrow \neg sat(e_s, p_i).$$

This form is convenient for proving certain dependencies by showing that for any execution, if p_j is not satisfied, then for this execution, p_i also cannot be satisfied. Theorem 2 below is an example of this type of proof.

Theorem: 1 *Total order depends on FIFO order.*

Proof: Assume an arbitrary execution of an arbitrary system, such that total order is satisfied for this execution. This means that any two membership change messages M(C) and M(D) are ordered the same on all sites. In particular, this is true also if C = D, that is, if the messages address the membership change of the same site. This means that FIFO order is satisfied. Therefore, total order depends on FIFO order. □

Similar arguments can be used to prove the dependencies between the properties that order membership change messages with respect to application messages.

Theorem: 2 *Total order depends on agreement.*

Proof: Assume an arbitrary execution of an arbitrary system, such that agreement is not true for this execution. This means that there is a membership change message M(C) that is received by site A, but not by site B. Assume also some M(D) that is received by both A and B. Now, based on the definition of total order, on site A messages M(C) and M(D) must be ordered, say M(C) $\rightarrow$ M(D). Now, however, this sequence of messages can never exist on B since B never receives M(C). Therefore, total order cannot be satisfied. Thus, total order depends on agreement. □

Note that, although there is no edge from total order to agreement in the dependency graph, total order depends on agreement because of the transitivity of the dependency relation. Similar arguments can be used to prove that any ordering property depends on agreement, that collective join depends on agreement, and that the ordering properties with respect to application messages depend on atomic multicast of application messages.

Theorem: 3 *Majority status depends on total order.*

Proof: Majority status provides indication about whether the partition in which the site resides is guaranteed to be a majority partition after each membership change. In section 4.2.6.3, majority status is described as being built using total order. Assume now that such is not the case, i.e., that total order is not guaranteed. To argue that majority status is impossible to implement without total order, we do a case analysis based on the two possible approaches to implementing the property: (1) based on local information, (2) based on global information.

In the first alternative, the majority status is calculated based on how many sites will be in the local application level membership view after the membership change message has been delivered to the application. Assume that two membership changes—reporting the failure of site C and the recovery of site D, respectively—occur approximately at the same time. Assume some site A delivers the messages in order F(C) $\rightarrow$ R(D), and some site B in order R(D) $\rightarrow$ F(C). Now, if delivering F(C) reduces the membership on A to a potential minority, an inconsistency may occur since B will not see the potential minority status. Obviously, this violates the majority status property.

In the second alternative, assume that global agreement on the minority/majority status of each membership change message is reached before it is delivered to the application. For example, in the above scenario, agreement may be on messages (F(C),minority) and (R(D),majority). Now, even though the information in the messages is consistent at all sites, if these messages are not delivered in total order—for instance, the orders in the above example—the membership on A has a majority status while on B the membership has a minority status. Obviously, this violates the majority status property. Thus, majority status depends on total order. □

Similar arguments can be used to prove that asymmetric join requires total order to make a consistent decision about which of the two joining partitions is dominant.

Theorem: 4 *Voted decision depends on FIFO order.*

Proof: Voted decision guarantees that a site failure is only reported to the application layer if a required fraction of the group members first suspects that the failure has occurred. To do this, each site maintains a list of sites that it suspects may have failed that is used in the voting process. Without loss of generality, consider the case where the required fraction is a majority of the group membership. Now, consider a system execution where FIFO order is not satisfied. This implies that the membership views seen by different sites may be different, even after all the agreed membership change messages have been delivered. Therefore, it is possible that more than half the sites consider some site A to have failed, i.e., not part of the group membership, while the remainder consider A to be operational and a member of the group. Assume now that one of the sites that considers A to be operational suspects the failure of A and starts the voting process. By definition, the sites that consider A be have failed cannot have A in their suspect lists and so vote "no" on the suspicion. Since the majority of sites consider A to have failed, the vote will fail to get the required number of "yes" votes. As a result, even if all the sites that have A in their membership suspect it to have failed, A will never be removed from the membership. Thus, for voted decision to work correctly, FIFO order must be satisfied. □

Theorem: 5 *Agreement depends on collective join (CJ) or asymmetric join (AJ).*

Proof: For agreement to be satisfied, every site in the same partition must deliver the same set of membership change messages. However, if partitions can overlap during the partition join step, agreement may be violated. To see this, consider a situation with two partitions, one with site $\{$A$\}$ and the other with sites $\{$B, C$\}$. Furthermore, assume that partition join is not atomic and that A and B add one another to their membership views first, giving membership views mem_A = $\{$A, B$\}$ and mem_B = $\{$A, B, C$\}$. Now, assume C fails and B detects that and delivers membership change message F(C). Now, since A $\in mem_B$, A must deliver the same membership change message based on the definition of agreement. However, the delivery of this message results in an attempt to remove C from mem_A even though C $\notin mem_A$. Therefore, agreement does not guarantee correct behavior if partitions are allowed to overlap during partition join.

The properties collective join and asymmetric join solve this problem by eliminating overlapping partitions during partition join. Collective join does so by making the partition join atomic, while asymmetric join transforms partition join into recovery of individual site. Of course, if we assume no partitions occur or partitions are never allowed to join, this problem does not exist. □

Finally, note that the dependency graph states that agreement depends on detection. This dependency exists to eliminate trivial solutions; agreement could exist without detection, but if no failures or recoveries are ever detected, agreement becomes unnecessary.

Next, property conflicts are proven. As mentioned in section 3.1.2, the formal definition of conflict is

$$con(p_i, p_j) : \forall\ s \in \mathit{SYS}\ \exists\ e_s : \neg sat(e_s, p_i) \vee \neg sat(e_s, p_j).$$

Unfortunately, this definition does not directly suggest a proof technique, since the definition requires proving for all systems that there is an execution that does not satisfy p_i and p_j. In some cases, like Theorem 6 below, it is possible to take advantage of a known impossibility result.

Theorem: 6 *Accurate detection and live detection conflict in asynchronous systems.*

Proof: Assume a failure detection algorithm that is both accurate and live in an asynchronous system. This means that it does not give false failure reports and it eventually reports every failure. Assume a set of sites wish to reach consensus. That is, each site has a binary value and wishes to reach an agreed value such that every participant decides on the same value and this value is the original value of at least one site. Assume furthermore that communication is asynchronous and unreliable, but that every message has a non-zero probability of reaching its destination.

Given these assumptions, we can construct a consensus algorithm that is guaranteed to terminate eventually despite the asynchrony of the system. The two steps are.

Step 1. Every site sends its binary value to every other site repeatedly until it receives an acknowledgment or the failure of the target site is detected. When a site detects a failure, it ignores all further messages from that site. Each site maintains a vector for values from other sites that is initialized with NULL values. When a site receives a value, it updates the vector. Step 1 is completed when every other site has either acknowledged the reception of the transmitted value or has been detected to have failed, and when a value has been received from every non-failed site.

Step 1 will terminate at each site since failure detection is live and repeated retransmission guarantees that every message is received and acknowledged, provided that the sender and the receiver remain operational. Note, however, that the value vectors may differ at this point, since some sites may have received the 0 or 1 value from a site before it failed, whereas others did not and still have the initial NULL value for the site. Step 2 is required to reach agreement on these messages.

Step 2. After Step 1 terminates, each site repeatedly sends its value vector to every other operational site until it receives an acknowledgment. When a site receives a value vector, it is combined with the existing value vector, as follows.

i) If the values are both 0 or 1, the result is 0 or 1, respectively.

ii) If one of the values is NULL, then the result is NULL.

Note that, if present, the binary values must agree since we assume that sites experience only crash failures and that the communication service does not corrupt messages in an undetectable manner.

If a site, say A, detects the failure of another site, say B, during this step, it stops accepting messages from B and sends out its updated value vector in a message indicating that the new vector was sent as a reaction to the failure of B. Then A waits until it receives both an acknowledgment to this message and a similarly updated vector from every other operational site. A site can terminate Step 2 when it has received the vector from every site, or it has detected the failure of a site and has received the subsequent updated vector from every other operational site.

Step 2 is guaranteed to terminate because of live failure detection and repeated message transmission. Since failure detection is also accurate, a site cannot be falsely considered failed by others, a scenario that would allow it to reach a different consensus value. As a result, Step 2 guarantees that all operational sites have the same values in their value vectors. Thus, if all sites use the same deterministic function to calculate the result, consensus will be reached. However, reaching consensus conflicts with the impossibility result presented in [FLP85], which implies that the original assumption about having failure detection that is both live and accurate must be false. □

Theorem: 7 *Collective join and asymmetric join conflict.*

Proof: In this case, the conflict is due to having mutually contradictory choices for dealing with a situation. In particular, consider a site A in the minority partition at the time two partitions are to merge. If the underlying membership service guarantees collective join, it must eventually generate a merge message and deliver it to the application on site A. However, if it guarantees asymmetric join, it must generate and deliver to the application on A message F(A) followed by Rec(A). Obviously, both of these event sequences cannot be true at the same time. □

Finally, we prove independence relations between properties. The formal definition of independence,

$$ind(p_i, p_j) : \neg con(p_i, p_j) \wedge \neg dep(p_i, p_j) \wedge \neg dep(p_j, p_i),$$

suggests a general proof strategy. In particular, independence can be proven by showing that neither of the properties depends on the other and that the properties do not conflict. The lack of conflict can be demonstrated by showing that there is an implementation that always satisfies both properties. In the following, we prove some of the more interesting cases from Figure 4.19.

Theorem: 8 *Agreement (AG) and eventual agreement (EA) are independent.*

Proof: AG and EA are independent because neither depends on the other, and they can be satisfied at the same time. First, recall that EA allows different sites to deliver different sets of membership changes. In particular, under EA, one site can deliver membership change messages F(C) and R(C), while another site does not. This means that AG is not satisfied. Therefore, EA does not depend on AG. Second, AG only requires that the same set of membership change messages is delivered on all sites, but does not restrict the order. In particular, A might deliver two messages in order F(C) $\rightarrow$ R(C), while B delivers them in order R(C) $\rightarrow$ F(C), resulting in different final membership views. Therefore, EA is not satisfied and AG does not depend on EA. Three, a system that implements AG and FIFO order guarantees that the membership views on different sites are identical after all membership change messages have been delivered, thus ensuring EA, which implies that AG and EA do not conflict. Combining these three facts means that agreement and eventual agreement are independent. □

Theorem: 9 *Agreement on last message (ALM) is independent from agreement on predecessors (AP).*

Proof: It is obvious that ALM cannot depend on AP since ALM only deals with messages from a site that is suspected to have failed, whereas AP deals with messages from all sites. We can also easily see that AP does not depend on ALM as follows. Let c_i and c_{i+1} be successive messages sent by a failed site C. Let c_i be in the agreed predecessor set of a membership change message F(C). In case of AP, the only restriction is that c_i must be delivered before F(C). However, since c_{i+1} is not in the agreed predecessor set, it is not ordered with respect to F(C) and may be delivered before F(C) on some sites and after F(C) on some other sites. Thus, there is no such agreed last message from C that is delivered on all sites before F(C). So, ALM is not satisfied and therefore AP does not depend on ALM. Finally, it is easy to construct a system that always satisfies AP and ALM. In particular, a system that guarantees virtual synchrony also always guarantees AP and ALM. Therefore, ALM and AP do not conflict, and the properties are independent. □

Theorem: 10 *Timebound synchrony (TBS) and external synchrony (ES) are independent.*

Proof: Obviously, neither TBS nor ES depend on the other because ES does not refer to global time bounds and TBS does not have the concept of a transition state used in ES before a new membership view is installed. Furthermore, in synchronous systems where TBS can be guaranteed, it is also possible to implement ES. Therefore, TBS and ES are independent. □

Similar arguments can be used to show that timebound synchrony and any of the other ordering properties are independent.

Theorem: 11 *Collective failure notification (CFN) and agreement on last message (ALM) are independent.*

Proof: CFN does not depend on ALM since it is not concerned with ordering of membership messages with respect to application messages. ALM does not depend on CFN since it is defined only for single membership change messages. Finally, CFN and ALM can easily be implemented together by simply having the agreed last messages of the sites in the collective failure notification message be the predecessors of the CFN message. □

Similar argument can be used to show that collective failure notification and any of the ordering properties are independent.

Theorem: 12 *Extended virtual synchrony (EVS) and extended virtual synchrony with partitions (EVSP) are independent.*

Proof: EVS and EVSP do not depend on one another since both can be satisfied separately from the other. First, EVSP can be satisfied without EVS being satisfied since EVSP is orthogonal to how application messages are ordered in the various partitions before the merge message. Second, EVS can be satisfied without EVSP being satisfied. Consider a membership change message R(A) and a merge message Me(P,Q). Now, assume an application message m_i is sent before the sender receives R(A) and therefore, is delivered on all sites before R(A), but before Me(P,Q) on one site and after Me(P,Q) on some other site. In this case, EVS is satisfied but not EVSP. Therefore, EVS does not depend on EVSP. Finally, it is easy to see from the definitions that EVS and EVSP do not conflict. In particular, EVSP only addresses ordering with respect to merge messages, while EVS addresses ordering with respect to membership change messages other than merge messages. Thus, the properties are independent. □

Note that, although these two properties are independent based on their formal definitions, in most practical situations EVS would be used with EVSP in conjunction with collective join. This is because the combination of these three properties ensures that a message is only delivered to those sites that were in the sender's membership view at the time the message was sent, independent of whether the membership change in question is the recovery of a single site or a partition join.

4.4 Characterizing Existing Services

4.4.1 Overview

The properties defined in previous sections can be used to characterize existing membership services. Doing so carries with it a number of caveats, however. For example, service properties defined in the literature are often properties of specific implementations that are difficult to relate to abstract properties as seen by the application. Furthermore, membership services are not uniform in how they interact with the application. In our approach, a service signals membership changes by forwarding membership change messages to the application in the regular message stream. We call services that follow this approach, including ISIS [BSS91], Consul [MPS92], and Transis [ADKM92a, ADKM92b], *delta-based services* since they deliver the changes ("deltas") to the application. Another approach is for the service to deliver the entire current membership set whenever a (possible) membership change occurs. We call services that follow this approach, including [Cri91, AMMS+93, RFJ93, SR93], *set-based services*. Although properties for the two types of services are typically stated differently, mappings between them can usually be constructed in a straightforward manner.

This section gives an overview of several existing membership services and characterizes their properties using the terminology defined in this chapter. These services are then summarized in tabular form in section 4.4.8.

4.4.2 Consul

The membership service of the Consul system [MPS92, MPS93a] assumes asynchronous communication and sites that experience crash failures. The service is built using Psync [PBS89, MPS89], a multicast service that preserves the causal ordering of messages using a *context graph* abstraction. Psync guarantees reliable multicast communication, so that context graphs on various sites are identical except for transmission delays.

Failures are detected at each site by monitoring the message stream from all other sites in the group. If no message is received from some site within a specified interval, the site is suspected to have failed. When a site suspects the failure of another site, say A, it initiates an agreement process by multicasting an "A is down" message. Upon receiving such a message, a site decides based on the state of its context graph whether or not it agrees with the suspicion. If it agrees, it multicasts "Ack, A is down"; otherwise, it multicasts "Nack, A is down". Since messages are multicast using Psync, each message will eventually be received by every other site, including those that may fail and later recover. If all responses are Acks, A is removed from the membership set at all sites.

Consul deals with simultaneous failures—i.e., the failure of one or more sites during the execution of the agreement protocol for an initial suspicion—by means of *simultaneous failure groups* (sf–groups). The sites in an sf–group are removed from the membership simultaneously. However, sf–groups may vary from site to site, so the ordering property guaranteed is weaker than the total order of membership changes property defined above. sf–groups are transparent to the application, and are used primarily as a means of optimizing the agreement process.

Consul's failure detection and reporting are live, with a level of confidence in failure detection that can be characterized as consensus. Recovery detection and reporting are accurate, based on receiving a message from a site considered failed. Although the system does not generate a singular membership change message in the sense used in this chapter, it can be shown that it guarantees agreement and FIFO ordering of membership change messages. The service also guarantees agreement on first and last messages, as well as agreement on successors.

4.4.3 Isis

The membership service of Isis described in [BJ87] consists of a distributed site view management component and an ordered multicast primitive (GBCAST) that is used to multicast and order membership change messages to ensure virtual synchrony. Site failures are detected by sending "Hello" messages between sites. If an Hello message from a site is not received within a specified period, it is assumed to have failed.

Each site maintains a *site view*, which is the set of sites it deems to be operational. The view management algorithm ensures that each operational site goes through the same sequence of site views. The sites in a view are ordered uniquely according to the view in which they first became operational, with ties broken by site identifier. The "oldest site" in this ordering is called the *view manager* and is responsible for initiating the view management protocol when it detects a site failure or recovery. If a site detects that all sites older than itself have failed, it takes over as the new view manager.

View changes are done using a two-phase commit protocol. First, the manager multicasts the proposed site view. If this view is new—i.e., a newer site view transmitted by some other manager has not been received—a site sends a positive acknowledgment. Otherwise, it replies with a negative acknowledgment and the more recent view. If all acknowledgments are positive, the manager multicasts a commit message. If a negative acknowledgment is received, or if new site failures or recoveries occur, the protocol is restarted. This protocol guarantees that the view managers on all operational sites process site views in the same order.

GBCAST is used to multicast membership change messages among the sites comprising the group. GBCAST guarantees total ordering with respect to all messages; that is, no message is delivered before a message sent using GBCAST on one site and after it on another site. Furthermore, membership change messages sent by GBCAST are delivered after every message from the failed site. Because of these properties, informing the application about a membership change is just a matter of multicasting the appropriate message using GBCAST.

The Isis membership service has live failure detection based on single site suspicion. Although the service was assumed to be live [RB91], it has later shown not to be live in all cases [ACBMT95]. Since GBCAST messages are totally ordered, the service realizes both total ordering of membership changes and virtual synchrony. Isis deals with partitions by allowing computation to continue in at most one partition, an approach supported by the majority status property described in section 4.2.6.3.

4.4.4 Cristian's Synchronous Membership Protocols

In [Cri91], Cristian presents three group membership protocols built on the assumption that the underlying system provides synchronous reliable atomic broadcast primitives. The protocols handle faulty sites leaving the membership and fault-free or repaired processors joining. All three protocols provide the same service abstraction, but make slightly different tradeoffs in the implementation. For example, in the *periodic group creation* protocol, each site multicasts "Present" messages at agreed times, which allows a consistent membership set to be calculated at all sites based on the assumption of synchronous reliable communication. Another protocol, the *attendance list* protocol, reduces message overhead in the absence of joins and failures by circulating an attendance list through all sites once per period instead of using Present messages.

The protocols proposed by Cristian guarantee a number of properties. In the terminology of [Cri91], these include the following:

- *Agreement on group membership*: Any two sites in the same group have identical membership views.

- *Reflexivity*: A site that has joined the group belongs to the membership (excludes the trivial solution of an empty membership list).
- *Bounded join delay*: The time for a site to join a group is bounded by a constant.
- *Bounded departure detection delay*: The time to detect the departure of a site (e.g., because of a failure) is bounded by a constant.
- *Bounded group formation delay*: The time between the point when the first and last sites join the group during initial group formation is bounded by a constant.
- *Bounded group change delay*: The time elapsed between a site leaving one group and joining a new group is bounded by a constant.

Mapped into our abstract properties, the bounded join and departure detection delay properties combined with the reliability and synchrony assumptions guarantee our liveness and accuracy properties. Agreement on group membership corresponds to our agreement property. Reflexivity is guaranteed by liveness. Bounded group formation delay is equivalent to timebound synchrony. Bounded group change delay only makes sense in the context of this particular algorithm, since in our framework, a membership view never expires. Although not explicitly listed as a property, these protocols also guarantee total order of membership views for sites within the same partition, which corresponds to our total order of membership messages property. Note that these protocols do not attempt to order membership changes with respect to application messages.

4.4.5 Mars

The membership service in the Mars system is another example of a synchronous protocol [KGR91]. Mars builds on a physical ring architecture in which the network is accessed using a time division multiple access (TDMA) strategy based on common global time, i.e., access to the physical medium is divided into dedicated time slots that are allocated *a priori* to sites in a round-robin fashion. A *TDMA cycle* is defined as *N* consecutive slots, where *N* is the number of sites in the system. Based on these assumptions, the Mars membership protocol handles processor crash failures and sites failing to send or receive, under the assumption that at most one failure occurs in each TDMA cycle. Failure detection exploits the TDMA communication strategy: since each site is expected to send data in each of its slots, the lack of transmission is interpreted as an indication of failure. To reduce network failures, every message is transmitted twice in the same time slot. Agreement on membership changes is reached by forwarding information in each message about all messages received in the previous cycle. This essentially propagates the list of sites the sending site knows to have been alive in the previous TDMA cycle.

Due to the way in which membership is integrated with the communication system, the service realizes ordering with respect to application messages. Total order of membership changes is trivially guaranteed given the assumption of no more than one failure per

TDMA cycle and since each processor observes the same membership changes in each cycle. Agreement on first and last messages is also guaranteed, since if one site receives a message, every non-failed site also receives the message, while if no site receives the message, the sending site is deemed to have failed. Although not guaranteed by the basic algorithm, agreement on successors and predecessors is easy to implement given the communication system and failure assumptions.

4.4.6 Totem

The Totem message ordering and membership protocol is described in [AMMS+93, AMMS+95]. The protocol is based on a logical token passing scheme, where the token is used for total ordering of messages, reliable message transmission, flow control, and membership. All messages in Totem are totally ordered reliable multicasts.

Membership in Totem is based on reforming the group whenever a group membership change is suspected, i.e., whenever a site failure or token loss is detected or a new message from a site outside the group is received. The site that initiates the membership change multicasts an "Attempt Join" message and then shifts to a "Gather" state while it waits for Attempt Join messages from other sites. After a specified time period has elapsed, it then multicasts a Join message that contains the identifiers of those sites from which it received messages, and shifts into a "Commit" state. In this state, sites reach agreement on the new membership, as follows. Each time a site receives a Join message containing sites of which it was previously unaware, it transmits a new Join message with the updated information. Once a site receives Join messages from all sites it included in its most recent transmission and the membership in all these messages agree, the protocol on that site terminates.

The Totem membership service has a live failure detection based on single site suspicion; the agreement algorithm is not live, however. It also guarantees FIFO ordering of membership messages, extended virtual synchrony, and extended virtual synchrony with partitions.

4.4.7 Weak, Strong, and Hybrid Membership Protocols

A family of three membership protocols labeled as *weak*, *strong*, and *hybrid* is described in [RFJ93]. All three deal only with ordering membership views and establishing agreement between views on different sites, with no attempt made to order membership changes with respect to application messages. The weak protocol simply guarantees that the views of all sites converge to a single consistent view if there are no failures for some period of time. More precisely, define $V_i \prec_s V_j$ if site S installs membership view V_i before installing view V_j. Let $\overset{*}{\prec}$ be the transitive closure of $\prec_s$, i.e.,

$$\forall i, j : V_i \overset{*}{\prec} V_j \Leftrightarrow \exists s : V_i \prec_s V_j$$

Then, the weak protocol ensures that $\overset{*}{\prec}$ is a partial ordering relation, i.e., it is irreflexive, transitive, and asymmetric. In particular, asymmetry guarantees that if one site installs V_i before V_j, then no other site will install them in the opposite order. We have not defined a property that corresponds to the partial ordering of membership views, but it would be easy to define such a property within our framework. The protocol also guarantees convergence: if site S has view V_i and site T has a different view V_j, then assuming no failures, there is eventually a state in which both sites have the same membership view V_k such that $V_i \overset{*}{\prec} V_k$ and $V_j \overset{*}{\prec} V_k$. Translated into our properties, the weak protocol is live, guarantees no agreement on membership changes but eventual agreement on membership views, and ensures none of our ordering properties.

Informally, the strong membership protocol ensures that membership changes are seen in the same order by all members. More precisely, it guarantees the properties of the weak protocol plus the following:

P_1: In any global state, if a site s has joined g locally, all other sites in g have either joined g locally or are in a transition state (i.e., members of no group).

P_2: In the presence of performance failures or network partitions, members of concurrently active groups are disjoint.

P_3: In the absence of performance failures or partitions, all active members go through the same sequence of groups (total order).

Based on the definitions in this chapter, the strong membership protocol guarantees agreement and total order of membership changes. P_2 also implies that collective failure notification and ordered collective join notification are satisfied when partitions occur. Also, as discussed in section 4.2.4.1, P_1 is the source of our external synchrony property.

The hybrid membership protocol provides guarantees that are intermediate between those of the weak and strong protocols. In particular, it ensures the properties of the weak protocol, with the additional guarantee that there is a single leader for each group. Thus, an algorithm similar to strong membership is executed when the leader of a group changes; otherwise, an algorithm similar to weak membership is executed. Essentially, the hybrid protocol guarantees that in any global state, if a site S decides locally that site T is the leader of the group g, then all other sites in g either consider T the leader or are in a transition state (i.e., have not decided a leader). This property would be equivalent to having all membership changes that affect the leader have our external synchrony property.

4.4.8 Summary

Table 4.1 summarizes the properties guaranteed by each service discussed in this section. The properties are abbreviated as follows:

	Lv	Ac	Co	Ag	Fo	To	Af	Al	As	Ap	Vs	Ev	Es	Ts	Np	Ms	Cn	Oj	Ep
Consul	x		All	x	x		x	x	x						x				
Isis			1	x	x	x	x	x	x	x	x					x			
Cristian	x	x	1	x	x	x								x	x				
Mars	x		2	x	x	x	x	x	x*	x*				x	x				
Totem			1	x	x	x*	x	x	x	x	x	x	x				x	x	x
Weak	x		1																
Strong	x		All	x	x	x							x				x	x	

Table 4.1: Properties Enforced by Existing Services

- *Accuracy and Liveness*: Liveness (Lv), accuracy (Ac), level of confidence (Co) ranging from one site suspicion (1) to consensus (ALL). Note that in asynchronous systems the liveness only applies to failure detection and reporting.
- *Agreement*: Agreement (Ag).
- *Ordering Properties*: FIFO ordering (Fo), total ordering (To), agreement on first (Af), agreement on last (Al), agreement on successors (As), agreement on predecessors (Ap), virtual synchrony (Vs), extended virtual synchrony (Ev).
- *Bounded Change Properties*: External synchrony (Es), timebound synchrony (Ts).
- *Partition Handling*: No partitions (Np), majority status (Ms), collective failure notification (Cn), ordered collective join (Oj), extended virtual synchrony with partitions (Ep).

In the table, “x*” denotes a property that is not guaranteed by the service, but could easily be added. Recall also that a system that provides a given property P also provides all properties that satisfy the inclusion relationship with P, as discussed in section 4.3.

4.5 Conclusions

In this chapter, we have specified abstract properties of membership services and described how they relate to one another. Ordering graphs were used to define the properties by illustrating their effect on the order in which messages are delivered to the application. For example, the agreement property guarantees that every membership change message is received at all sites, thereby providing the basis for consistent decision making. Ordering properties extend this notion to ensure consistent message ordering information as well, differing in the degree of consistent information provided. The tradeoff is that the stronger properties require more expensive algorithms and more synchronization between sites, which potentially results in less efficient execution. The way in which properties relate to one another was illustrated using dependency graphs.

As noted in section 4.4, existing membership services can be characterized in terms of these properties. With the exception of [SR93], however, membership papers concentrate on describing the properties of a particular algorithm or system, rather than providing a more global view. [SR93] gives a decomposition of membership services into three components: Failure Suspector, Multicast Component, and View Component. The Failure Suspector is responsible for detecting membership changes and propagating changes that it has detected to other Failure Suspectors. In our framework, this corresponds to change detection properties. The Multicast Component is responsible for implementing virtually synchronous communication. Compared to our approach where multicast is separate from membership, their Multicast Component combines these two functions. The View Component ensures that all sites have the same view of the membership. As such, it essentially implements our agreement property. However, our decomposition is much more extensive and identifies a large number of properties in contrast to just identifying more implementation oriented components of membership services.

CHAPTER 5

GROUP MEMBERSHIP SERVICE

Existing membership solutions typically provide only one fixed service, making it impossible to tailor the service to the specific needs of an application. In this chapter, we describe a configurable membership service that addresses this problem. This service is based on the abstract properties of membership described in chapter 4 and the event-driven execution model described in chapter 3. The specific family of modules introduced is based on a token-passing paradigm, and allows choices of whether the service is accurate or live, what kind of agreement is performed, how membership messages are ordered with respect to application messages, and how partitions are handled.

The design presented here is structured around the system model described in chapter 4. In particular, the key data structure is an implementation of the abstract ordering graph used there to specify properties. Every site maintains an ordering graph consisting of messages and their ordering constraints. Messages are added to the ordering graph at a site when they arrive from the network or are generated by the membership service. Each message is delivered to the application when all its predecessors in the graph have been delivered. Furthermore, we separate the service into the same two logical components used in chapter 4: a reliable communication component and a membership component. The communication component implements reliable causally ordered communication between application level processes and maintains the ordering graph. The membership component implements the various membership properties. In this design, only the membership component is configurable, so the remainder of this chapter concentrates exclusively on that portion of the service.

The implementation is based on the assumption that the underlying communication network is asynchronous with no *a priori* time bounds on message delivery. The failures considered are site crash and performance failures, as well as typical network failures such as lost messages and network partitions.

5.1 Design Overview

The basic components of a configurable membership service built using our model are events, messages, shared data structures, and micro-protocols. This section gives an overview of the first three; the micro-protocols implementing membership are described in the following section. First, however, we overview the general implementation strategy, which is based on token passing.

5.1.1 General Implementation Strategy

Perhaps the key requirement for implementing many of the properties of membership is some means of information collection and dissemination. The various approaches for accomplishing this can be classified into three major categories: (1) broadcast based (e.g., [MPS93a, ADKM92b]), (2) coordinator based (e.g., [RB91, RFJ93]), and (3) token based (e.g., [RM89]). In examining each approach in light of our requirements, we selected the third based on the resulting simplicity of the micro-protocols.

The basic idea behind this approach is to organize the group members into a (logical) ring and then have a token that circulates around the ring. In contrast with other multicast and membership protocols that use token passing, the token in our scheme is used only for membership; regular communication need not be restricted to the ring or based on token passing. The role of the token is to collect and distribute the information required to realize the various properties. Specifically, the token has one record with multiple fields for each membership change underway at any given time. We call such a record a *membership entry*. Various micro-protocols exploit the information in the token in various ways.

Different properties of membership impose different requirements on how the token is used. For example, properties that involve primarily information dissemination, such as agreement, require that the token be rotated only once around the ring, while properties that also involve information collection, such as virtual synchrony, require that the token be rotated twice. The number of rotations actually used in a given configuration is the maximum of the number required across all micro-protocols that are included. The reliability of token passing is increased by requiring that the receiver acknowledge receipt of the token. Among other things, this strategy enables some aspects of failure detection to be integrated into the token passing mechanism.

To realize ordering-related properties, we use an ordering graph as described above, i.e., membership change messages are inserted into a graph of messages that constrains the delivery order to the application. We assume that actual delivery of messages from the graph to the application is handled independently from the membership service by the reliable communication component of the system, which is configured within the same composite protocol. As already noted, we also assume that the underlying network provides asynchronous unreliable point-to-point message delivery, and that sites may suffer from crash or performance failures.

5.1.2 Events

As described in chapter 3, execution of code within micro-protocols is initiated when events occur. In general, events are used in this model for a variety of purposes, including to indicate a change of state in the composite protocol such as message arrival, to signal the opportunity to update shared variables or message fields, or as a procedure call to transfer control and data between two micro-protocols. Like signal variables in the monitor construct found in some concurrent programming languages, an event is a no-op if no handler is bound to that event. This feature de-couples micro-protocols to a certain degree and helps increase the configurability of the resulting system by removing the need to explicitly reference other micro-protocols.

In the membership service design, events can be classified into four categories:

- *Membership entry events*. For managing membership entries that circulate in the token.
- *Failure and recovery events*. For dealing with site failures, recoveries, and partitions.
- *Token handling events*. For dealing with token passing, regeneration, and merging after a partition.
- *Message handling events*. For managing application and membership change messages within the composite protocol.
- *Startup and restart events*. Two events, STARTUP_EV and RECOVERY_EV, that are generated at a site upon initial startup and recovery, respectively.

The first four are now described in more detail. For simplicity, all events are sequential and blocking.

Membership Entry Events. Events FIRST_ROUND and SECOND_ROUND signal that a membership entry has been seen at the site for the first or second time, respectively. ADD_ENTRY is generated once current entries have been processed to signal the opportunity to add another entry to the token prior to it being passed to the next site. Similarly, NEW_ENTRY is generated when a new entry is added to allow various micro-protocols the opportunity to initialize fields of interest.

Failure and Recovery Events. Event SUSPECT_NEXT_DOWN is generated when the conditions for failure suspicion are met for the next site in the ring. In the case of a live membership service, this occurs when a certain number of token retransmissions are attempted without success; with an accurate service, this occurs when a message with a new incarnation number is received from a recovering site, indicating the earlier failure of the old incarnation. SUSPECT_CHANGE is a more general event indicating the suspected failure or recovery of any site in the system. POTENTIAL_ENTRY is generated to allow an opportunity to increase confidence in a suspected change or deny it prior to reporting it to the application. Finally, PARTITION is generated when a partition has been confirmed; note that since a partition cannot be distinguished from individual site failures at partition time, this event occurs after communication has been reestablished.

Token Handling Events. Event TOKEN_RECEIVED is generated when the token arrives at the site. FORWARDING_FAILED indicates that a particular attempt to pass the token to the next site in the ring has failed; this is also sometimes used as part of the failure detection process, as mentioned above. MERGE_TOKENS is generated when two tokens are merged into a single token, such as when partitions are merged.

Message Handling Events. Events are also used to manage application and membership messages. MSG_FROM_NET is generated when a message arrives at the composite protocol from lower-level protocols, while MSG_FROM_USER is generated when upper-level protocols pass along a message to be delivered to application processes on other sites. Both are triggered automatically by the runtime system. The event MSG_RECEIVED is generated within the membership service after some initial processing has been performed on every application message that arrives from lower-level protocols.

Events are also used to implement interactions between the membership and communication components of the composite protocol, as illustrated in Figure 5.1. Membership

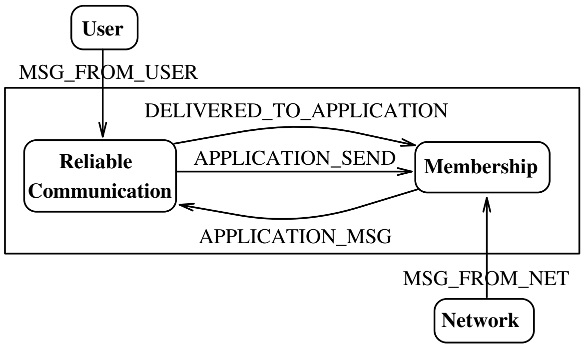


Figure 5.1: Interaction between Components

signals that a membership change message is ready to be added to the ordering graph by triggering APPLICATION_MSG. The communication service indicates that an application message is ready to be sent to lower-level protocols with event APPLICATION_SEND, while DELIVERED_TO_APPLICATION is generated when a message has been delivered upwards towards the application. These latter two are often used by membership micro-protocols involved with ordering messages.

Note that the semantics of events make their use for component interaction qualitatively different than function calls. In this case, for example, some reasonable system configurations include no micro-protocols that field the event APPLICATION_SEND. If function calls were used, the communication code would have to be changed to avoid making this call, while with events, no change is needed since it can safely be generated with no effect. More importantly, since more than one handler can be bound to a single event, using this mechanism also makes it simple for multiple independent micro-protocols to be notified when an event occurs. In this case, for example, it is quite likely that multiple micro-protocols will field the APPLICATION_MSG event.

Finally, the event MEMBER_MSG is generated when a membership change message has been created, thereby allowing micro-protocols an opportunity to set fields in the message.

5.1.3 Membership Change Messages

Messages transmitted from the composite protocol up to the application process on a given site are either application data messages from other sites or membership change messages. As noted in section 4.1, membership change messages are the mechanism by which the membership service informs the application of site failures and recoveries so that it can, for example, update a local membership list.

The membership change messages used in this design are the following:

- STARTUP indicates that the application can begin normal processing; includes a list of initial group members.
- SHUTDOWN indicates that the application should stop.
- FAILURE reports the failure of one site; if the site is this site, the application should stop.
- RECOVERY reports the recovery of one site; if the site is this site, the application can resume normal operation using the current membership information in the message.
- MERGE reports the merging of two partitions; the message contains the identities of the new group members.
- C_FAILURE reports the collective failure of more than one site, possibly due to a partition.
- PRE_MERGE indicates that the merging of two partitions is in progress; used to implement certain message ordering guarantees that require a site to stop sending messages until the merge is complete.
- PRE_CHANGE indicates that a change in membership is about to occur so that the application can alter behavior if necessary; for example, with some message ordering guarantees, the application must stop sending messages or change its membership into a transition state.

Note that the specific messages that an application may receive depends on the particular micro-protocols configured into the composite protocol. For example, a MERGE message will only be received if the micro-protocol that implements partition merging is included. Also, note that these are just the membership messages that are delivered to the application; other messages, such as those that implement token passing, are used by the membership service itself to communicate with peers on other sites.

5.1.4 Shared Data Structures

One of the most important benefits of using composite protocols is that it allows micro-protocols to share data as noted in chapter 3. services. In the configurable membership service, the shared data structures are the following:

- `MsgGraph`. The ordering graph of messages.
- `Token`. The contents of the most recent token received at this site.
- `Membership`. An image of the application's view of the current membership.
- `ParList`. The membership service's current view of the group membership, i.e., the list of participants in the membership protocol; membership changes take effect earlier in `ParList` than `Membership`.
- `Delivered`. A vector with the identifier of the most recent message delivered from each site, which is used to determine when a message is eligible for delivery because all of its predecessors have been delivered; shared since other micro-protocols such as those concerned with recovery must be able to update it.
- `SuspectList`. All suspected membership changes that have not yet been reflected in `ParList`.

5.2 Micro-Protocols

This section describes the various micro-protocols that comprise the membership service, presenting the code for several. The goal is not to be exhaustive, but rather to give an overall view of how the service operates and some examples of the algorithmic and programming style used to write composite protocols. For expository purposes, we divide the micro-protocols into five categories:

- *Base micro-protocols*. Provide the base functionality needed by other micro-protocols, including message and token handling, and recovery.
- *Accuracy, liveness, and confidence micro-protocols*. Deal with detecting site failures and recoveries.
- *Agreement micro-protocols*. Implement the agreement process required for most variants of membership.
- *Ordering and synchrony micro-protocols*. Implement different varieties of message ordering guarantees.
- *Partition handling micro-protocols*. Implement different partition handling policies.

5.2.1 Base Micro-Protocols

The base micro-protocols are **MessageDriver**, **TokenDriver**, **Recovery**, and **StartUp**. The subsequent paragraphs summarize their functionality.

MessageDriver. This micro-protocol coordinates the traversal of application messages from the network to the application. It triggers MSG_RECEIVED when an application message arrives from the network and APPLICATION_MSG when the message is ready to be forwarded. Application messages sometimes have to be temporarily retained in the membership layer, for example, to implement virtual synchrony. To do this, **MessageDriver** provides a mechanism for releasing a message when all micro-protocols have finished operating on it. Specifically, two arrays are used: a global `Hold` array, which specifies which micro-protocols must execute for each message, and a corresponding **hold** array associated with each message, which specifies which micro-protocols have already executed. Thus, when `hold` is equivalent to `Hold`, the message can be forwarded.

```
micro-protocol MessageDriver() {
    export procedure forward_up( var msg: ApplMessage, hold_index: int) {
        msg.hold[hold_index] = true;
        if for each i: Hold[i] = msg.hold[i] then trigger(APPLICATION_MSG,msg);
    }
    event handler msg_from_net(var msg:NetMessage) {
        if msg.type = DATA then { msg.amsg.hold[ ] = false;
            trigger(MSG_RECEIVED,msg.amsg); forward_up(msg.amsg,DEFAULT); }
    }
    event handler delivered_msg(var msg: ApplMessage) {
        if msg.type = FAILURE then Membership –= msg.changed;
        elseif msg.type = RECOVERY then Membership += msg.changed;
        . . . similar for other message types . . .
    }
    initial { Hold[DEFAULT] = true; register(MSG_FROM_NET,msg_from_net);
        register(DELIVERED_TO_APPLICATION,delivered_msg); }
}
```

Figure 5.2: **MessageDriver** Micro-Protocol

The pseudo-code for **MessageDriver** is shown in Figure 5.2. Its general form is similar to most micro-protocols: a few event handlers, initialization code, and possibly some local variables and functions. As can be seen from the pseudo-code, messages have a number of fields. These include its type (`type`), a unique identifier (`mid`), the sender (`sender`), the hold array mentioned above (`hold`), the message to be passed to the application (`amsg`), and, in the event of a membership change message, the identity of the failed or recovered site (`changed`). Not shown here, but used below, is an array of message identifiers (`pred`), which holds the predecessors of the message in the ordering graph.

TokenDriver. This micro-protocol's task is to implement for each partition the abstraction of an indestructible token that circulates among all functioning sites in the order dictated by the logical ring. In addition to the actual message passing involved in sending the token to the next site, much of the code in **TokenDriver** involves dealing with exceptional conditions such as lost token regeneration, site failures during regeneration, and the possibility of multiple tokens during the merging of partitions.

TokenDriver at a given site suspects that the token has been lost when it fails to receive it again a specified amount of time after passing it on. When this occurs, the micro-protocol first finds the most recent copy of the token that has been seen by any site in the ring. To facilitate this, **TokenDriver** at each site maintains a copy of the most recent token it has seen, together with its *version number* and *circulation count*; the former is incremented every time the token is regenerated, while the latter is incremented every time a site processes the token. The most recent copy of the token is discovered by circulating a *regeneration token*, which at any time holds the most recent copy seen so far. Once the regeneration token has circulated the entire ring once, it contains the most recent token, which is then used to create a new token. Multiple sites can—and often will—issue a regeneration token, but the algorithm ensures that only one site will have its regeneration token make the complete circuit.

Site failures may, of course, occur while the regeneration token is circulating. If this occurs, a membership entry will be added to the regeneration token and merged with the entries from the most recent copy of the lost token. New entries will not, however, be processed or any membership change messages forwarded to the application during the regeneration process

Although the regeneration protocol guarantees that each partition recreates at most one token, there are cases where a partition may temporarily have more than one. Consider three sites A, B, and C, such that each follows the previous in the logical ring. Suppose that failure detection is based on timeouts. Then, it is possible that A passes token to B, but fails to receive any acknowledgment, and therefore declares B faulty and passes the token to C. If B received the token and did not fail, both B and C have a token simultaneously. **TokenDriver** handles this situation by ensuring that only one of the tokens is considered valid within the current partition. In this particular situation we have two possible cases. First, if C receives a token from A before it receives one from B, it will note that B has failed and therefore will not accept the token from B. Second, if C receives a token from B before receiving one from A, it ignores the one from A based on the circulation count number.

The **TokenDriver** micro-protocol triggers the event TOKEN_RECEIVED when a normal (i.e., not regeneration) token is received, MERGE_TOKENS when two tokens are merged as a part of token regeneration, and FORWARDING_FAILED when the site to which a token is passed fails to acknowledge the reception within a specified time bound.

Recovery. This micro-protocol handles recovery of a site after failure. Its execution is initiated when the RECOVERY_EV event is triggered automatically by the runtime system upon recovery. At this point, **Recovery** begins sending JOIN messages to other sites, either using a previous membership list saved on stable storage or exploiting broadcast-based network hardware, if available. When another site receives this message, it triggers whichever recovery detection micro-protocol is configured into the composite protocol (see section 5.2.2), which begins the process of reintegrating the site back into the membership. The **MembershipDriver** micro-protocol (see section 5.2.3) manages the necessary agreement protocol, which involves adding a recovery entry to the token. While the token circulates, each site reads the recovery entry to obtain relevant information about the recovery, and updates it as needed with information needed to reinitialize the membership state of the recovering site. Based on what ordering properties are being enforced, the token will circulate either once or twice. Once a site has received the token the required number of times, it inserts a membership change message announcing the recovery into its ordering graph, with the specific location depending on the ordering properties.

The site that originated the recovery entry is also responsible for updating the membership state of the new member and inserting it into the logical ring. To do this, it sends a STATE message containing the relevant state of the membership service (e.g., the current membership list) and the token to the recovering site. Upon receiving this message, the **Recovery** micro-protocol updates its local data structures. It also uses information in the recovery entry of the token to initialize the state of other components of the composite protocol, such as the location in the ordering graph that the communication component should use as the starting point for forwarding messages up to the application. Once this has been done, the recovering site forwards the token to the next site and begins normal operation.

As is the case with token handling, the recovery and reintegration process is designed to tolerate site failures, additional recoveries, and similar exceptional conditions.

StartUp. This micro-protocol manages the startup process. When the STARTUP_EV event is generated by the runtime system, it sets the incarnation number, initializes the local copy of the token, and forwards the STARTUP message with the initial membership to the application.

5.2.2 Accuracy, Liveness, and Confidence Micro-Protocols

For detecting site failures, micro-protocols that implement both live and accurate algorithms are provided as alternatives. Live detection is based on lack of response from a site, i.e., timeouts. Accurate detection, on the other hand, cannot be based on communication since the network is assumed to be asynchronous. As a result, our implementation, like that described in [OIOP93] for Mach, detects a site failure only when the failed site recovers and reestablishes communication. Similarly, accurate recovery detection—the only kind possible in asynchronous systems—is implemented by the recovering site contacting other sites upon recovery as described in the previous section. The following micro-protocols implement these properties.

```
micro-protocol LiveFailureDetection(LIMIT:int, check_period: real) {
    var SilentList: set of int;    % list of sites not heard of lately

    event handler handle_failure(var site:int, attempts:int) {
        if attempts < LIMIT and (site,FAILURE) ∉ SuspectList then attempts++;
        else { SuspectList += (site,FAILURE); attempts = 1;
            trigger(SUSPECT_NEXT_DOWN,site); }
    }
    event handler handle_msg_from_net(var msg: NetMessage) {
        if msg.type != JOIN then {
            if (msg.sender,FAILURE) ∈ SuspectList then
                SuspectList –= (msg.sender,FAILURE);
            if msg.sender ∈ SilentList then SilentList –= msg.sender; }
        elseif msg.sender ∈ Membership and (msg.sender,FAILURE) ∉ SuspectList then {
            SuspectList += (msg.sender,FAILURE);
            trigger(SUSPECT_CHANGE,msg.sender,FAILURE); }
    }
    event handler handle_new_membership_msg(msg:ApplMessage) {
        if msg.type == FAILURE then SuspectList –= (msg.changed,FAILURE);
    }
    event handler monitor() {
        for each m:int ∈ Membership do {
            if m ∈ SilentList and (m,FAILURE) ∉ SuspectList then {
                SuspectList += (m,FAILURE);
                trigger(SUSPECT_CHANGE,m,FAILURE); }
            SilentList += m; }
        register(TIMEOUT,monitor,check_period);
    }
    initial { register(FORWARDING_FAILED,handle_failure);
        register(MSG_FROM_NET,handle_msg_from_net);
        register(MEMBER_MSG,handle_new_membership_msg);
        register(TIMEOUT,monitor,check_period); }
}
```

Figure 5.3: **LiveFailureDetection** Micro-Protocol

LiveFailureDetection. Triggers event SUSPECT_NEXT_DOWN signaling a suspected site failure if token retransmission fails a specified number of times. Triggers SUSPECT_CHANGE instead if a site that is expected to communicate for some other reason does not respond in a timely manner, or if a site that is already in the membership list attempts to join. The pseudo-code for **LiveFailureDetection** is shown in Figure 5.3.

AccurateRecoveryDetection. Triggers SUSPECT_CHANGE signaling a suspected site recovery upon receiving a message from a site not currently in the membership. Used in combination with **LiveFailureDetection**.

AccurateDetection. Implements accurate detection of both site failures and recoveries. Triggers SUSPECT_CHANGE signaling a suspected site failure and succeeding recovery when a message arrives with an incarnation number greater than the current incarnation number for that site. Also inserts the current incarnation number in outgoing messages.

Our design supports two versions of the confidence property. The first is single site suspicion, where no confirmation is needed from other sites. In this case, a suspected membership change can simply be entered into the token and circulated among all group members. The second is a voting-based process, which is implemented by the micro-protocol **VotedDecision**. When the event POTENTIAL_ENTRY is triggered, **VotedDecision** sends out a request for votes and sets a timer using the facilities for TIMEOUT events provided by the runtime system. When the timer expires, all votes that have been received are examined. If any site has responded "no", the result in negative, that is, the conclusion is that no membership change has occurred. Otherwise, the result is positive. Individual sites base their responses on whether or not the suspected site is in their `SuspectList`. Many other variants of voting-based policies are, of course, possible.

5.2.3 Agreement Micro-Protocols

Implementing agreement on site failures and recoveries is straightforward given the abstraction of an indestructible token. In particular, since the token is guaranteed to be received periodically by every operational site, all that is required is to enter the change in the token and circulate it. Sites then read the entry when the token arrives and deliver a membership change message to the application at the appropriate place in the message stream.

The micro-protocol **MembershipDriver** implements agreement and coordinates the overall execution of the membership protocol. Coordination is primarily based on events. Figure 5.4 illustrates the event interactions between **MembershipDriver** and other micro-protocols. In the figure, edges originating at **MembershipDriver** represent events triggered by this micro-protocol and edges pointing towards **MembershipDriver** represent events for which this micro-protocols has event handlers registered. **MembershipDriver** also maintains information about the number of rotations needed for each membership entry in the token. Special attention is paid to entries whose *reporter*—the site that originally added the entry to the token—fails during execution of the protocol. This situation is handled by having such entries be "adopted" by other sites, which then behave as the reporter for the remainder of the protocol.

A second membership driver micro-protocol called **SimpleMembershipDriver** is provided as an option for applications not requiring agreement. Rather than circulate information in the token, it simply translates local detection of failures and recoveries into membership change messages that are delivered to the application. It also implements a simple recovery facility for this type of application.

All remaining micro-protocols assume that **MembershipDriver** is configured into the system.

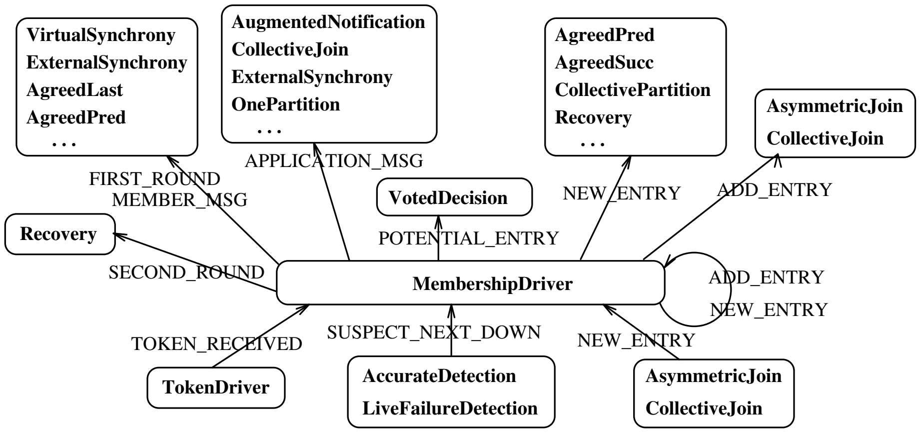


Figure 5.4: Event Interactions between **MembershipDriver** and Other Micro-Protocols.

5.2.4 Ordering and Synchrony Micro-Protocols

The **MembershipDriver** micro-protocol implements FIFO ordering of membership change messages as a free side-effect of the agreement process. Other orderings are implemented by separate micro-protocols, as follows.

Total order. Total ordering of membership change messages is implemented by simply forwarding membership change messages to the application in the order the changes are recorded in the token. Since every site sees the same token, every site delivers the messages in the same total order using only one round of token rotation.

The **TotalOrder** micro-protocol (Figure 5.5) implements this property by translating the ordering of entries in the token into a total order in the ordering graph using message predecessor fields. Note that the strong guarantees provided by **TokenDriver** and **MembershipDriver** greatly simplify this micro-protocol.

```
micro-protocol TotalOrder() {
    var previous_mid: int;    % id of the previously processed msg in total order

    event handler handle_mship_msg( var msg:ApplMessage,entry:EntryType) {
        msg.pred[ ] += previous_mid; previous_mid = msg.mid; }

    initial { previous_mid = 0; register(MEMBER_MSG,handle_mship_msg); }
}
```

Figure 5.5: **TotalOrder** Micro-Protocol

Agreement on Last Message. Properties such as agreement on last message that require ordering membership change messages with respect to application messages are somewhat more complex. The **AgreedLast** micro-protocol implements this property by collecting information in the token about the last message received from the failed site. This information, which is stored in the membership entry, is updated at a site if that site has received a message with a higher identifier than the one currently in the token. After one rotation, then, the token holds the identifier of the most recent message that any site has received from the failed site at the time it updated the entry. This message is taken to be the agreed-upon last message, and the token rotated a second time to disseminate the result. After receiving the token a second time, each site places the appropriate membership change message in the ordering graph immediately after the agreed-upon last message. Note that during this process, delivery of application messages from the suspected failed site must be stopped. Figure 5.6 presents the pseudo-code for **AgreedLast**.

Virtual Synchrony. The **VirtualSynchrony** micro-protocol implements this property by guaranteeing that the sets of application messages received before and after a membership change message are identical on all sites, respectively. This is accomplished by first collecting information on which messages have already been delivered and then constructing an agreed predecessor set consisting of those messages that have been delivered on at least one site. set. As was the case above, delivery of application messages is stopped during the execution of the agreement.

Agreement on Successors. This property states that every site has an agreed view of which messages are guaranteed to be delivered after the membership change message. Note that causal ordering of messages creates an agreed successor set consisting of those messages that are sent after the sender receives the membership change message. Different options are available for creating agreed successor sets that are larger, i.e., include earlier messages. The algorithm used for **AgreedSucc** is similar to **VirtualSynchrony**, where information about the most recent messages that have not been delivered is collected and used to form the agreed successor set. This approach requires stopping message delivery while agreement is in progress. Another micro-protocol, **LateAgreedSucc**, is based on an inexpensive solution where the successor set consists of those messages sent after the agreement has started. This requires only one token rotation.

Agreement on Predecessors. This property states that every site has an agreed view of which messages are guaranteed to have been delivered on all sites prior to a membership change message. Note that there may be messages that are not in this set but are delivered before the membership change message on some sites. This is not inconsistent with the definition since the agreed set only specifies those messages delivered at every site. There are several possible choices for what constitutes the agreed predecessor set. For example, it could be those messages that every site has received by the time the agreement protocol starts, those messages that at least one site has received by this time, or even all messages that were sent by this time. The first alternative is the least restrictive and allows the membership change message to be delivered as early as possible, so it is the alternative implemented by **AgreedPred**. The protocol simply collects information about delivered

```
micro-protocol AgreedLast() {
    var LastSeen[ ], LastAllowed[ ]: int; mutex: semaphore;

    event handler first_round(var entry:EntryType) {
        var s: int;
        if entry.type == FAILURE then { P(mutex); s = entry.changed;
            entry.pred[s] = max(LastSeen[s],entry.pred[s]);
            LastAllowed[s] = entry.pred[s]; V(mutex); }
        else if entry.type == C_FAILURE then { P(mutex);
            for each s ∈ entry.members do {
                entry.pred[s] = max(LastSeen[s],entry.pred[s]);
                LastAllowed[s] = entry.pred[s]; }
            V(mutex); }
    }
    event handler new_membership_msg( var entry:EntryType, msg: ApplMessage) {
        var s: int;
        if entry.type == FAILURE then { P(mutex); s = entry.changed;
            msg.pred[s] = max(msg.pred[s],entry.pred[s]);
            LastAllowed[s] = msg.pred[s]; V(mutex); }
        else if entry.type == C_FAILURE then { ... similar to above ... }
    }
    event handler handle_delivered_msg(var msg:ApplMessage) {
        if msg.type == FAILURE then { P(mutex);
            LastAllowed[msg.changed] = MAXINT; V(mutex); }
        else if msg.type == C_FAILURE then { ... similar to above ... }
    }
    event handler handle_msg(var msg:ApplMessage) { P(mutex);
        if LastAllowed[msg.sender] < msg.mid then { V(mutex); cancel_event(); }
        else {
            LastSeen[msg.sender] = max(LastSeen[msg.sender],msg.mid);
            V(mutex); }
    }
    initial { register(FIRST_ROUND,first_round);
        register(MSG_RECEIVED,handle_msg);
        register(DELIVERED_TO_APPLICATION,handle_delivered_msg);
        register(MEMBER_MSG,new_membership_msg);
        LastSeen[ ] = 0; LastAllowed[ ] = MAXINT; }
}
```

Figure 5.6: **AgreedLast** Micro-Protocol

messages during the first round of token passing and this set forms the predecessor set on the second.

External Synchrony. This property states that when a site updates its membership view, all other sites either already have this new view or are in a transition state. The **ExternalSynchrony** micro-protocol implements this property by using a a special membership change message PRE_CHANGE that is forwarded to the application on the first round. Upon the reception of this message, the application changes its membership state to

"Transition". Once this occurs, the token is forwarded to the next site. Thus, by the second rotation, each site is in a transition state and the normal membership change message can be delivered to the application.

Extended Virtual Synchrony. This property states that application messages sent before the sender receives a membership change message are delivered at their destinations before the membership change. The **ExtendedVirtualSynchrony** micro-protocol implements this property using a PRE_CHANGE message similar to **ExternalSynchrony**. In this case, however, upon receipt of this message, the application refrains from sending more messages. In addition to forwarding the PRE_CHANGE message on the first round, the micro-protocol collects information about the most recent messages sent by the application prior to receiving the PRE_CHANGE message. These messages are made the predecessors of the membership change message on the second round, thereby guaranteeing that they will be delivered to the application before the actual membership change message. Figure 5.7 presents the pseudo-code for **ExtendedVirtualSynchrony**. Note that special attention must be paid to sites that have failed; such sites cannot participate in the algorithm, so messages sent by these sites are not ordered with respect to the membership change message. This is handled by every site keeping track of not only the latest message it sent, but also the latest message from every site that it has received.

5.2.5 Partition Handling Micro-Protocols

As noted in chapter 4, the policies that dictate how a system operates in the presence of partitions can be divided into three phases: partition time, partitioned operation, and partition join. The micro-protocols relevant to each phase are described below.

Partition Time. By default, the membership service implements individual notification, where the membership changes associated with a partition are treated as individual site failures. The alternative collective notification policy is provided by the **CollectiveNotification** micro-protocol, which reports the failure of all sites in other partitions in a single membership change message.[1] It does this by waiting for the NEW_ENTRY or MERGE_TOKENS events, and then when they occur, combining all failure entries in the token into a single entry. The entries are combined so that the ordering properties guaranteed for the combined entry are inherited from the first entry in the token. Once the entries are combined, the token is circulated again to ensure that every site sees the combined entry. To guarantee that no site generates a membership change message for an entry before **CollectiveNotification** has a chance to combine entries, each entry is circulated at least once around the ring before a membership change message is delivered to the application. This also guarantees that all sites in other partitions are included in the collective entry.

Partitioned Operation. The policy for what level of service a system should provide during partitions is inherently an application decision, so the membership service generally

[1]Note that simultaneous true site failures will also be reported collectively since such situations are impossible to distinguish from partitions in asynchronous systems.

```
micro-protocol ExtendedVirtualSynchrony() {
    var wait: semaphore; % wait for delivery of PRE_CHANGE message to application
        view[ ]: int; % latest message seen from every site;

    event handler record_view(var msg: ap_msg_type) {
        if msg.mid > view[msg.sender] then view[msg.sender] = msg.mid;
    }
    event handler first_round(var entry: entry_type) {
        var msg = new(ap_msg_type);
        if entry.change != MERGE then {
            msg.mid = e.entry_id; msg.type = PRE_CHANGE; msg.pred[ ] = 0;
            trigger(APPLICATION_MSG,msg); P(wait);
            entry.view = max(entry.view,view); }
    }
    event handler handle_delivered_msg( var msg: ap_msg_type) {
        if msg.type = PRE_CHANGE then V(wait);
    }
    event handler new_membership_msg( var msg:ap_msg_type,entry:entry_type) {
        if msg.type ∈ {FAILURE,RECOVERY,C_FAILURE} then
            if not(msg.type = RECOVERY and msg.changed = MyId) then
                msg.pred += entry.view;
            else msg.pred[ ] = 0;
    }
    initial { view[ ] = 0; wait = 0; register(MSG_RECEIVED,record_view);
        register(APPLICATION_SEND,record_view);
        register(FIRST_ROUND,first_round);
        register(DELIVERED_TO_APPLICATION,handle_delivered_msg);
        register(MEMBER_MSG,new_membership_msg); }
}
```

Figure 5.7: **ExtendedVirtualSynchrony** Micro-Protocol

only provides supporting information. In our design, this is done by the **Augmented-Notification** micro-protocol, which augments each membership change message with majority/minority status information depending on whether the site is in the majority partition or not. Note that the appropriate value is simple to calculate, assuming that the maximum membership of the group is known and that membership change messages are delivered in total order. The application can use this information, for example, to halt processing or reduce the level of service in minority partitions.

Partition Join. The merging of partitions is initiated by the **PartitionDetection** micro-protocol, which attempts to detect the existence of other partitions by periodically sending "I am alive" messages containing the current membership to all sites that are currently considered failed. Upon receipt of such a message at some site *A*, the event PARTITION is triggered if the membership list in the message has no sites in common with the membership list on *A*. If the lists do overlap—which might occur, for example, if the partition was of such short duration that the process of removing sites from the list was

```
micro-protocol AsymmetricJoin() {
    var OtherPartition: set of int;    % sites in the other partition

    event handler handle_partition(Members: set of int) {
        if dominate(Members,ParList) then {
            OtherPartition = Members; register(ADD_ENTRY,enter_shutdown); }
    }
    event handler enter_shutdown(var token: TokenType) {
        var entry: EntryType;
        if SHUTDOWN ∉ token.entries then {
            entry.type = SHUTDOWN; entry.members = OtherPartition;
            token.entries += entry; trigger(NEW_ENTRY,entry); }
        deregister(ADD_ENTRY,enter_shutdown);
        register(DELIVERED_TO_APPLICATION,handle_delivered_msg);
    }
    event handler handle_delivered_msg(msg: ApplMessage) {
        if msg.type == SHUTDOWN then {
            deregister(DELIVERED_TO_APPLICATION,handle_delivered_msg);
            status = DOWN; shutdown_and_restart(msg.members); }
    }
    initial { register(PARTITION,handle_partition); }
}
```

Figure 5.8: **AsymmetricJoin** Micro-Protocol

incomplete in one or both partitions—then the sites in the intersection are removed from the membership of *A*'s partition prior to beginning the merge process.

Two alternative micro-protocols are provided for implementing the partition merge when PARTITION is triggered, **CollectiveJoin** and **AsymmetricJoin**. The first combines two partitions into a single one, including merging the two logical rings used for communication. This is accomplished by one partition giving up its token to a site in the other partition, which then combines the tokens into a single token as described in section 5.2.1. A special membership change entry of type MERGE is then inserted into the token and circulated to inform all sites in the combined membership of the new membership information. **AsymmetricJoin** handles partition join by forcing sites in the minority partition to fail prior to being allowed to rejoin the majority partition as individual recovering sites. The shutdown is coordinated by adding a SHUTDOWN entry to the token in the minority partition.

The pseudo-code for **AsymmetricJoin** is shown in Figure 5.8. Functions *dominate* and *shutdown_and_restart* are defined elsewhere. The former defines the dominance of two partitions based on group size and site identifiers, as discussed above. The latter, which is executed only by sites in the non-dominant partition, takes the assumed membership of the dominant partition as argument and simulates the failure and restart of the composite protocol. It also triggers the event RECOVERY_EV.

Finally, the **ExtendedWithPartition** micro-protocol implements extended virtual synchrony between application messages and membership change messages reporting partition merges, similar to that defined in [MAMSA94]. This micro-protocol is distinct from **ExtendedVirtualSynchrony** since the predecessor sets of such messages are different in the sites in the two merging partitions.

Other partition handling micro-protocols. In numerous membership services [Cri91, HS95, KT91, KGR91, MSMA94, MPS93a, RB91], it is simply assumed that partitions will not occur, or that only one partition will continue to operate. The **OnePartition** micro-protocol implements a simple strategy that approximates this behavior. In particular, when any message other than JOIN is received from a site outside the current membership, **OnePartition** sends a STOP message that forces the offending site to halt. A simple dominance relationship based on group size and maximum site identifier is used to ensure that only one partition remains active.

5.3 Configuring a Custom Membership Service

The collection of micro-protocols outlined above provides the basis for building a membership service with properties customized to the needs of a given application. In principle, those micro-protocols that provide the desired properties are combined at system configuration time to construct an instance of the service. However, as discussed in chapter 3, relations between micro-protocols restrict which combinations are operational. Here, we discuss the relations between membership micro-protocols and present the membership configuration graph that summarizes the operational combinations.

5.3.1 Relations between Membership Micro-Protocols

Most relations between micro-protocols are inherited from the relations between the corresponding properties described in chapter 4. For example, totally ordering membership change messages is impossible unless such messages are present on all sites, so the property of total ordering depends on agreement. Thus, in our design, this means that **TotalOrder** depends on **MembershipDriver** that implements the agreement property.

Sometimes a dependency relation between two properties changes naturally into an inclusion relation between the corresponding micro-protocols, as outlined in chapter 3. Examples of this are the ordering properties with respect to application messages. For example, extended virtual synchrony depends on virtual synchrony, but it is easier to implement the **ExtendedVirtualSynchrony** micro-protocol using **MembershipDriver** instead of attempting to build it using **VirtualSynchrony**. Essentially, the knowledge that virtual synchrony is guaranteed does not make the implementation of extended virtual synchrony any easier in our implementation framework. On the contrary, from the point of view of **ExtendedVirtualSynchrony**, **VirtualSynchrony** collects irrelevant information and adds irrelevant edges to the ordering graph that **ExtendedVirtualSynchrony** would have to change to guarantee the corresponding property.

Other relations are results of design decisions, however. For example, in our design **ExternalSynchrony** is included in **ExtendedVirtualSynchrony** although the corresponding properties are formally independent. This is because the behavior of **ExternalSynchrony** is a subset of the behavior of **ExtendedVirtualSynchrony**. In particular, **ExternalSynchrony** is unnecessary when **ExtendedVirtualSynchrony** is used, since the application can change into transition state when it receives the PRE_CHANGE message created by **ExtendedVirtualSynchrony**.

Finally, all the real-time properties discussed in chapter 4 are naturally omitted since this design and implementation are based on an asynchronous environment.

5.3.2 Membership Configuration Graph

Figure 5.9 presents the membership configuration graph for the micro-protocols presented in this chapter. Using the graph we can, for example, configure a simple membership service consisting of the micro-protocols **MessageDriver**, **SimpleMembershipDriver**, **AccurateDetection**, and **StartUp**. This service provides accurate but uncoordinated membership change indications to the higher level protocols. An example of a more complicated membership service would be one that provides virtual synchrony, and handles network partitions by allowing computation to continue in each partition and then combines partitions by forcing sites in one to fail and rejoin as individual members. Such a service consists of the micro-protocols **AsymmetricJoin**, **PartitionDetection**, **VirtualSynchrony**, **TotalOrder**, **MembershipDriver**, **TokenDriver**, **MessageDriver**, **Recovery**, **StartUp**, **AccurateRecoveryDetection**, and **LiveFailureDetection**. In this manner, 1136 semantically distinct membership services can be configured from the micro-protocols in this collection.

Although the number of possible combinations may appear excessive for any practical purposes, one has to realize that it is a result of a small number of mostly independent aspects of membership services. In particular, there are the two types of failure detection, three different approaches for dealing with partitions, a number of different ordering properties, and a couple of extra features such as **CollectivePartition** and **AugmentedNotification** that can be added to most combinations. The need for the different failure detection methods can be justified by different systems taking the different approaches. Similarly, different applications have different needs for dealing with partitions. For example, the consistency and availability goals of the system dictates the behavior required in the event of partitions. Finally, although one could always choose the strongest ordering property, **ExtendedVirtualSynchrony**, the cost of the different ordering properties in terms of the number of messages required and the delay imposed on message delivery makes it appealing to choose the property providing the minimum required ordering guarantee.

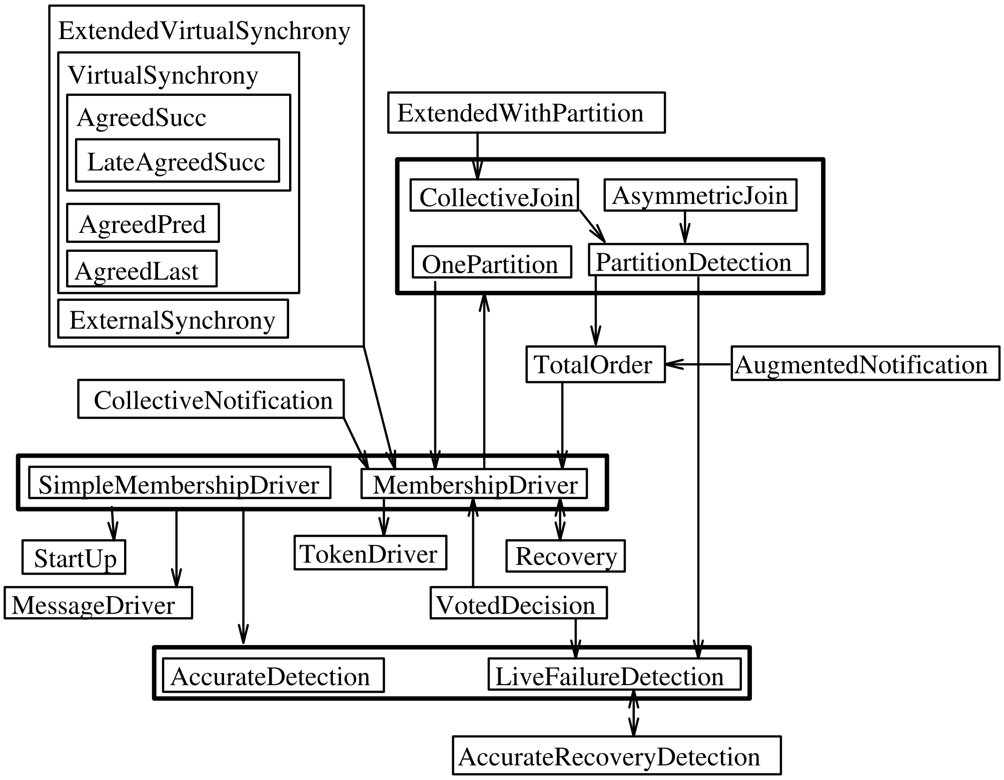


Figure 5.9: Membership Configuration Graph

5.4 Prototype Implementation

The C++ prototype implementation of the event-driven execution model described in chapter 3 was used to implement the configurable membership service. In the following, we elaborate on how the C++ prototype was used and on our initial experience with the implementation.

The membership prototype uses the `User` and `Network` object classes as described in chapter 3, with the exception that the `User` class logs application and membership change messages received for debugging purposes. The actual membership service and required communication services are implemented as derived classes of base classes `CompositeProtocol` and `MicroProtocol`. Class `MembershipService`, which is derived from `CompositeProtocol`, is the membership composite protocol. This class defines the events, shared data structures, and initialization for the service. All membership micro-protocols described in this chapter are implemented as object classes derived from the `MicroProtocol` class. Furthermore, an additional micro-protocol `CommunicationService` implements the functionality of a reliable communication service not addressed by the membership micro-protocol. In particular, it maintains the ordering graph and controls delivery of messages to `User` so that all ordering constraints are satisfied. Furthermore, it implements reliable causally ordered multicast communication between the `User` objects.

The prototype implementation of the membership service has been used to test a variety of different membership services. Given the large number of micro-protocol combinations possible based on the configuration graph in figure 5.9, the services tested were representative rather than exhaustive. First, all possible combinations that did not involve any of the micro-protocols that implement ordering properties with respect to application messages were tested. Then, to test these micro-protocols—**AgreedLast**, **AgreedSucc**, **LateAgreedSucc**, **AgreedPred**, **VirtualSynchrony**, **External Synchrony**, and **ExtendedVirtual Synchrony**—each was combined with five representative configurations of the remaining micro-protocols, one with **AccurateDetection** and four with **LiveFailureDetection** plus different ways of dealing with partitions. The test suite included scenarios involving multiple failures and recoveries, network partitions, and token loss, as well as normal processing.

Testing was performed as a black-box process in which various output results were monitored for a given set of inputs. For a membership service, the primary determinants of correctness are the messages received by the application level on each site and their ordering, so message logs maintained in `User` objects were the main source of information. Token passing was also traced to validate **TokenDriver**, arguably the most complicated micro-protocol. Execution of other individual micro-protocols could also be traced by setting the appropriate bit in a tracing mask; this causes event handling code in the micro-protocols to generate trace information every time one of its event handlers is invoked.

Building a version of the configurable membership service in this simulated environment has demonstrated several things, in our view. One is the overall viability of our modularization approach, where properties are mapped directly to fine-grained software modules to enhance configurability and customization. Another is the value of event-driven execution for decoupling modules and thereby minimizing the software changes needed to support configurability. The property-based modularity and configurability of the service also turned out to be an asset during the testing process itself. For example, during debugging, it was often easy to identify the property that was not being properly enforced, which automatically identified the offending micro-protocol. The ability to include or exclude micro-protocols easily also helped narrow the collection of modules that had to be examined when other types of problems occurred.

5.5 Discussion

5.5.1 Micro-Protocol Execution Costs

This choice of which micro-protocols to include when building a customized membership service is based primarily on the functional guarantees required by the application. However, another consideration is the incremental execution cost associated with guaranteeing the associated property, such as the number of extra messages required and the additional delay that it imposes on the delivery of messages to the application. To examine these costs relative to the micro-protocols discussed above, we first divide the functionality implemented by the membership service into two phases:

1. *Detection*: Initial detection of suspected site failures and recoveries.
2. *Coordination*: Subsequent processing required for sites to coordinate decision and deliver membership change message to the application.

Each phase incurs separate execution overhead based on the specific properties being enforced.

For the detection phase, the metric of interest is *detection delay*, that is, the time it takes from when a change occurs until the initial suspicion is signaled at some site. For recovery detection, both **AccurateRecoveryDetection** and **AccurateDetection** are based on receiving a JOIN message from the recovering site, so the detection delay depends on how quickly this message is sent and received after the site restarts. For failure detection, the delay depends on whether **AccurateDetection** or **LiveFailureDetection** is used. **AccurateDetection** only detects a failure once the failed site recovers, so the detection delay may be arbitrarily long. On the other hand, as with most live failure detection schemes, the delay associated with **LiveFailureDetection** depends on the frequency of message exchange and the timeout interval used before a suspicion is signaled. In our particular design, detection delay can be reduced either by circulating the token faster, which increases the network load, or by altering the token passing protocol to reduce the maximum number of retransmissions or shrink the timeout interval, both of which increase the probability of false failure suspicions. **VotedDecision** reduces the probability of false detections at the cost of increased detection delay.

For the coordination phase, the metrics of interest are *agreement cost* and *delivery delay*. Agreement cost is the number of messages it takes to collect and distribute information about a membership change to all sites so that the selected properties are guaranteed. Given our token-based protocols, the agreement cost can most easily be analyzed in terms of how many token rotations a specific property requires. For the micro-protocols in our service, these costs are:

- *One Rotation*: **TotalOrder**, **LateAgreedSucc**, **AugmentedNotification**, **Recovery**, and **AsymmetricJoin**.
- *Two Rotations*: All other ordering micro-protocols and **CollectiveJoin**.
- *Between Two and Three Rotations*. **CollectiveNotification**.

Delivery delay is the extra delay imposed by the membership service on the time it takes a message to be delivered to the application. Although it is difficult to calculate such delays in absolute terms, examining the relative delays between micro-protocols can be instructive. The delays for the relevant micro-protocols are as follows:

- **AgreedPred**. None, since algorithm will construct an agreed predecessor set consisting of messages already delivered on every site.
- **AgreedLast**. Halts delivery of messages from suspected site during agreement, and requires delay of membership change message until agreed last message is delivered.
- **AgreedSucc**. Halts delivery of all messages during agreement, but does not delay membership change message after agreement has been reached.
- **VirtualSynchrony**. Halts delivery of all messages during agreement, and requires delay of membership change message until agreed predecessor set is delivered.
- **ExternalSynchrony**. No extra ordering delay, but requires each site to deliver PRE_CHANGE message during first token rotation prior to forwarding token to next site.
- **ExtendedVirtualSynchrony**. Same as **ExternalSynchrony** for token passing and **VirtualSynchrony** for membership change messages, except that extra delay may be incurred since agreed predecessor set consists of all messages sent before the initiation of agreement.

As would be expected, the delays are roughly proportional to the strength of the guarantees provided.

5.5.2 Related Work

Membership services and protocols have been the subject of a large number of papers. Some of the work has been based on a synchronous system model, where bounds are placed on the network transmission time [Cri91, EL90, KGR91, LE90, SCA94]. Other work assumes an asynchronous model similar to that used in this chapter [ADKM92a, AMMS$^+$95, DMS94, EL95, GT92, MPS92, MPS93a, MAMSA94, RB91, SM94]. Unlike our configurable service, however, all these services guarantee only a single collection of properties, or at most, offer a small number of choices.

Schemes based on logical rings or token passing are used by many multicast, membership, and system diagnosis protocols. For example, token passing is used as a means of implementing reliable totally ordered multicast in the Reliable Broadcast Protocol [CM84], Token-Passing Multicast (TPM) protocol [RM89], Multicasting Transport Protocol (MTP) [AFM92], Totem [AMMS$^+$95], Pinwheel [CM95], and Reliable Multicast Protocol (RMP) [WMK95]. In these protocols, the site possessing the token is either the only site that is allowed to send a message or the site that assigns a global ordering to messages sent by all sites. Most of these protocols deal with site failures and recoveries, as well as the possibility of token loss. With the exception of TPM, however, all deal with membership changes using broadcast-based schemes that are independent of the token passing used during normal processing; in these cases, the token is recreated only after agreement has been reached on the new membership.

The algorithms used in TMP are perhaps closest to those used in our membership service, especially its use of the token to recreate group membership after a failure. When a site suspects that a token loss or site failure has occurred, it generates a *recovery token* and circulates it to collect the identities of operational sites. Multiple recovery tokens are eliminated by having each site only forward a token if it was created by a site with a larger identifier. After agreement on the new membership has been reached, a *clean-up token* is circulated to collect and disseminate information about the messages received on each site so that missing messages can be requested. This token also collects the maximum sequence number across all delivered messages as it circulates, which is then used to initialize the new token during recovery.[2]

Although the TMP protocol employs a token to collect and disseminate information in much the same way as we do, the underlying algorithms are quite different. For example, unlike TPM, in our approach, information about concurrent membership changes—including partitions—is collected and disseminated using a single token. This difference changes many of the details of token handling, such as the steps taken to regenerate the token when failures occur. Furthermore, our design emphasizes configurability and facilitates the construction of customized membership services, rather than implementing a single set of properties as does TMP.

A number of membership and system diagnosis protocols organize sites into a logical ring structure without using a token. For example, the protocols in [RFJ93] use a ring to detect membership changes by having each site monitor its neighbor. Once a failure or recovery is detected, however, a membership protocol that employs a coordinator process is employed rather than using the ring. Some system diagnosis protocols, such as the Adaptive DSD protocol [BB91], use a logical ring for failure monitoring and information propagation. A simple membership protocol derived from Adaptive DSD that also uses a ring organization but not a token is introduced in chapter 6.

5.6 Conclusions

The modular configurable membership service described in this chapter is an example of the use of our techniques for constructing configurable fault-tolerant services. The membership service facilitates the construction of a customized fault-tolerance support layer that can provide the specific execution guarantees needed by a given application. By supporting this type of customization and configuration, our approach reduces the size and complexity of the system, thereby increasing the likelihood that it will operate as intended. It also has the potential to improve application performance by giving the designer explicit control over the tradeoff between the strength of the guarantees provided and the performance; rather than having to accept guarantees stronger than needed and

[2] Totem uses a *commit token* for a similar purpose [AMMS^{+}95].

thereby incur extra execution costs, the designer can select—and pay for—only those guarantees that are truly required. The approach is based on mapping abstract properties to individual micro-protocols, which are then configured together with a standard runtime system to form a composite protocol. The mapping of abstract properties to software modules and the event-driven model supported by the runtime system both enhance the overall configurability of the resulting system.

CHAPTER 6

MEMBERSHIP AND SYSTEM DIAGNOSIS

System diagnosis is the problem of keeping track of which processing elements in a computing system are functioning correctly and which have failed [PMC67]. That paper stated that a system operating in a tightly or loosely coupled distributed environment must avoid giving tasks to or using results from faulty processing elements. Therefore, it is necessary for a central authority, or for every processing element, to be aware of the condition of all the active processing elements. This ability to agree on the state of the system allows the fault-free processors to make correct and consistent progress. A model was presented where each subunit is able to test other subunits. Each test involves the controlled application of stimuli and the observation of the corresponding response. On the basis of the responses, the outcome of the test is classified as "pass" or "fail". In either case, the testing unit evaluates the tested unit as either fault-free or faulty. Numerous papers on system diagnosis have followed [BMD93, BP90a, BGN90, BB91, BP90b, BB93, LYS93, Pel93, WHS95].

Despite the close resemblance of the system diagnosis problem and the membership problem discussed in chapters 4 and 5, little has been done to compare or contrast these two fields. An important exception is [BMD93], which reviewed the field of system diagnosis thoroughly and compared it to membership. Unfortunately the emphasis was heavily on system diagnosis and the comparison brief. A number of other papers, for example [EL95, KGR91], acknowledge the relationship between these problems but do not explore it any further. In this chapter, we introduce system diagnosis, contrast it with the membership problem, and show that they can be viewed as essentially the same problem with slightly different assumptions. Given this observation, we conclude that the choice of service to keep track of functional and faulty processes, processors, or computers—whether called membership or system diagnosis—should be based only on user requirements and assumptions made about the computing environment, especially the failure models and synchrony assumptions. This means that system diagnosis and membership can be viewed as customized instances of a general service to keep track of faulty and functioning elements. Finally, the observations are applied by transforming two typical distributed system diagnosis algorithms into new membership algorithms, and by transforming a family of membership algorithms into system diagnosis algorithms that provide a service stronger than any other of which we are aware. In the conclusions, we outline how these observations could be utilized to construct a general configurable change detection and reporting service that could be configured to provide service guarantees similar to traditional membership and system diagnosis services.

6.1 System Diagnosis

The so-called PMC model for system diagnosis was introduced in [PMC67]. The model uses a graph *G(V,E)* to model the system's testing convention. Subunits make up the set *V*, and directed edges in *E* represent one subunit applying a test to another subunit, i.e., the directed edge *(A,B)* denotes that *A* tests *B*. Each edge is labeled with either 0 or 1 if the corresponding test produces a passing (0) or failing (1) result. The set of results from a given test sequence is known as a *syndrome*.

In the PMC model, a centralized arbiter interprets the syndrome after completion of testing according to *G* and deems each subunit to be either faulty or fault free. Fairly strict assumptions are made about the behavior of faulty subunits: all faults are permanent, a fault-free subunit is always able to determine accurately the condition of a subunit it is testing, and no more than t subunits may be faulty. As noted in [BMD93], these assumptions are not necessarily valid or desired in a fault-tolerant distributed network, and later work has dealt with removing these restrictions. The first problem is supervisor-controlled diagnosis. The implication is that all test data must be gathered and analyzed by a centralized supervisor and the result distributed back to the system. This is costly in terms of time, messages, and system reliability, which is directly related to the reliability of this supervisor. The assumption about permanent faults disallows intermittent and transient faults. The assumption that tests are *complete* or have 100% fault coverage may not be realistic. And finally, setting an upper bound for faulty subunits may not always be practical.

Systems that allow unambiguous diagnosis in all cases under the assumptions of the PMC model are said to be *t-diagnosable*. Based on the model, it was shown that if as many as t members of the subunit population may be faulty, then it is necessary for the system to contain n members, $n \geq 2t + 1$, to be diagnosable in all cases. Moreover, each subunit must be tested by at least t distinct other subunits. In the special case where no two subunits test each other, these necessary conditions are also sufficient for t-diagnosability [HA74].

In addition to the general model, [PMC67] gives a convenient testing subnetwork formulation for t-diagnosable systems that is used widely in later papers. Given an enumeration of the units u_1, ..., u_n, a system S is said to belong to design $D_{\delta t}$ when a testing link from u_i to u_j exists if and only if $j - i = \delta m$ (modulo n) and m assumes values $1, 2, \ldots, t$. This formulation simply states that each unit u_i tests t other units, and that the other units are chosen so that they are a step of δ away from one another. For example, when δ = 1, unit u_i tests u_{i+1}, u_{i+2}, ..., u_{i+t}.

An interesting variant to the testing used by the PMC model is comparison testing [Mal80]. In comparison testing, tests are based on comparing computation results of productive tasks from different units in the system. In case the size of the result is too large, a function of the result such as checksum can be used instead. In this approach, a syndrome is created by units comparing results with those units that they are testing, and edges in the testing graph are labeled with zero or one based on if the units agree or disagree similar to the PMC model. The resulting syndrome can be analyzed as in PMC model.

More recently, work in the system diagnosis field has concentrated on distributed diagnosis and probabilistic diagnosis. The concept of t_p *self-diagnosable* systems was introduced in [HKR84] for distributed systems where there is no central coordinator that executes the diagnosis. A testing network is shown to be t_p self-diagnosable if for any set of faulty processors, V', with $|V'| \leq t_p$, there exists a directed path of fault-free nodes and links from each fault-free node to any other fault free node. It is shown that this property requires that any node be tested by at least $t_p + 1$ nodes. For example, D_{1t} is t_p self-diagnosable if $t = t_p + 1$. More recent work on distributed diagnosis can be found in [BGN90, BB91, BB93, SA89, WHS95].

The first attempt at probabilistic diagnosis is in [MH76], which proposes assigning a reliability to each subunit in the network. This reliability measure is simply the probability of a fault occurring in a given subunit. A probabilistically t-diagnosable (p-t-diagnosable) system is defined as one having, for every allowable syndrome, a unique, consistent fault set whose probability of occurrence is greater than p. Assigning a probability of correctness to each test rather than to the subunits themselves is proposed in [Blo77]. Procedures are given for determining the probability of correct diagnosis for a particular fault set, and for the entire system. The general problem is to diagnose a system that suffers from intermittent failures and with tests that have imperfect coverage. The problem using probabilistic diagnosis under the comparison approach is first studied in [DSK87]. This system model avoids many of the pitfalls of the PMC model, including the need for the complete tests, the permanent nature of faults, off-line testing, and an upper bound on the number of simultaneously faulty subunits. More recent work on probabilistic diagnosis can be found in [BP90a, BSM89, BP90b, LYS93, Pel93].

6.2 Comparison

We argue that the major differences between algorithms traditionally viewed as system diagnosis or membership are in the failure model and the strength of the properties provided by the service. Since membership algorithms typically are distributed by nature, the comparison here is concentrated on distributed system diagnosis.

The failure model assumed by membership algorithms is in most cases either fail-stop, crash, or performance failures, although recently some work has been done assuming Byzantine failures [Rei96]. System diagnosis, on the other hand, seems, in most cases, to deal with a somewhat obscure failure model, namely one that can be detected by whichever test is employed by the system diagnosis algorithm. This failure model is strictly smaller than Byzantine failures [LSM82], since faulty processors suffering from such a failure could fool any test. The best characterization of this failure model is *incorrect computation failure*, which occurs when a processor fails to produce the correct result in response to the correct inputs [LMJ91]. Note, however, that if we define the test result to be "fail" in case no response arrives or the response arrives too late, the failure model employed by most membership algorithms is a subset of that used in system diagnosis.

The assumed failure model affects the way testing is performed. In membership services, testing is sometimes explicit by means of sending test messages, either to indicate that the processor itself is functional or to probe the status of another processor, or implicit, where the failure of another processor is suspected based on lack of messages from that processor. In system diagnosis, the testing is always active and the test consists of the tested computer executing a self-test or some task assigned to it by the tester and returning the results to the tester who compares the result to an expected result. A good test probes as many parts of a computer as possible, including the processor, memory, I/O subsystem, etc. In the case of comparison testing, the two computers involved in the test exchange results or checksums of results and the test result is achieved by comparing the results.

Related to the failure model is also the question of how many failures can be tolerated or diagnosed. Traditional system diagnosis algorithms have fixed testing assignments and can therefore tolerate a fixed number of failures, whereas most membership algorithms are prepared to tolerate any number of failures. However, lately this distinction appears to be disappearing since some recent system diagnosis algorithms, for example [BB91], can tolerate any number of failures.

In contrast with most system diagnosis algorithms, membership algorithms acknowledge that the detection mechanism may give false detections—especially those designed for asynchronous systems where it is impossible to tell the difference between a failed processor and a slow communication link. Some membership algorithms include some type of distributed vote that compares local information from different group members about the status of a suspected processor before a failure is declared [HS94a, MPS92]. Even in this case, there is a possibility of false failure detection, so membership algorithms are prepared to deal with this eventuality by forcing a member suspected to have failed to fail before it is allowed to recover and rejoin the group.

One of the most important differences between membership and system diagnosis algorithms may be the degree of integration with the application computation. In most cases, system diagnosis algorithms are run independently from the application and the results are used solely for system management. In some cases, this system management may require stopping the application and restarting it from an uncorrupted state. In contrast, membership algorithms are in most cases very tightly integrated with the application, providing it with a service that enables the programs themselves to deal with failures in the distributed computing environment. Furthermore, to make it easier for the applications to deal with the environment, a membership service may provide agreement, ordering, and synchronization properties as discussed in chapter 4. Using the classification developed there, system diagnosis algorithms usually only provide change detection and agreement properties. Although often this difference is intentional, i.e., the required function of the system is different, it is worth considering whether system diagnosis algorithms could be augmented to provide some of the stronger properties found in membership services.

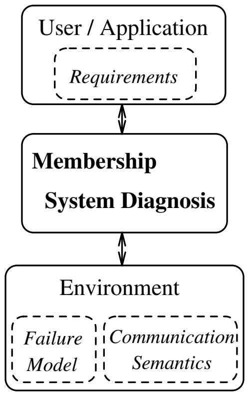


Figure 6.1: Role of Membership/System Diagnosis

Figure 6.1 illustrates the similarities and differences between membership and system diagnosis. To summarize, essentially they both serve the same purpose, i.e., to provide the user or the application with information about correct and faulty processors so that the requirements of the user/application can be met. These requirements could be agreement, ordering, or synchrony properties as discussed above, or they could be requirements on how many failures have to be tolerated, how quickly a failure has to be reported, etc. The membership or system diagnosis algorithm implements a service that satisfies these requirements, given what the environment makes available. We make assumptions about how the processors fail (failure model), if the communication is synchronous or asynchronous, and partially dictated by these facts, what kind of fault coverage the tests have and if false detections may occur.

In [BMD93] the problems of system diagnosis and membership are reviewed and the problems compared. The major difference noted in this paper is that membership algorithms mostly test for failures in the time domain whereas system diagnosis algorithms usually test for failures in the data domain. These two domains were considered orthogonal. A suggestion was made that techniques from membership algorithms could be used in system diagnosis algorithms to detect time domain failures and similarly techniques from system diagnosis algorithms could be used in membership algorithms to detect failures in the data domain. However, in practical system diagnosis algorithms, such as [BB91, BB93, WHS95], lack of response is considered a failure, i.e., time domain failures are essentially detected as well. The differences between the service guarantees or possible transformations between membership and system diagnosis algorithms were not addressed in [BMD93].

6.3 From System Diagnosis to Membership

In this section, we apply the observations above and show how typical distributed system diagnosis algorithms NEW_SELF [HKR84], EVENT_SELF [BGN90], and Adaptive DSD

[BB91] can be transformed into membership algorithms. The transformation is based on changing the failure model and thereby the testing method. The last example generates a new membership algorithm that appears to be competitive with published membership algorithms due to its simplicity and low message and execution overhead.

6.3.1 NEW_SELF System Diagnosis Algorithm

NEW_SELF [HKR84] assumes that each processor in the distributed system is capable of testing its neighbors. Correctly operating processors pass on results of these tests to other processors in the network. No assumptions are made about faulty processors. Diagnosis messages, which contain test results, flow between neighboring processors and reach non-neighboring processors through intermediate processors. Each processor determines diagnosis of the network with its given information independently.

Each node P_i tests a set of its neighboring nodes, denoted by TESTED_BY(P_i). After these tests, P_i receives additional diagnostic information from the fault-free members of TESTED_BY(P_i) and stores the information in an input buffer. P_i then re-tests all nodes in TESTED_BY(P_i). If node $P_j \in$ TESTED_BY(P_i) tests fault-free again, all previously received diagnostic information from P_j is validated and stored in an array structure. Once diagnostic information is validated at node P_i, the information is forwarded to all of the nodes that test P_i, denoted TESTERS_OF(P_i). Diagnostic information propagates through the network along testing paths in the reverse direction of the tests being performed. In summary, node P_i operates as follows:

1. P_i tests all $P_j \in$ TESTED_BY(P_i);
2. P_i receives diagnostic information from all $P_j \in$ TESTED_BY(P_i);
3. P_i re-tests all $P_j \in$ TESTED_BY(P_i);
4. P_i validates information from all fault-free $P_j \in$ TESTED_BY(P_i);
5. P_i forwards the validated information and the results of all its tests to TESTERS_OF(P_i).

A good testing subnetwork for this algorithm is D_{1t} [PMC67], since it minimizes the required tests and keeps the majority of the tests between local processors. As noted above, D_{1t} is t_p self-diagnosable when $t = t_p + 1$.

The theoretically optimal NEW_SELF algorithm has been found to be too expensive for practical systems, so a modified algorithm named EVENT_SELF was developed [BGN90]. The original algorithm requires a high number of diagnostic messages to be forwarded through the network. This message count is significant and can exceed network capacity for testing networks with even a small number of nodes. The algorithm also requires the testing time to be insignificant as compared to the testing period. The EVENT_SELF algorithm solves these problems by only propagating diagnostic information when it has changed and by not testing all nodes in the TESTED_BY set on every test cycle.

EVENT_SELF also adds automatic reconfiguration of the testing assignments upon failures or recoveries.

6.3.2 NEW_SELF Membership Algorithm

To construct the corresponding membership algorithm, we first change the failure model to one typical for membership algorithms, in this case crash failures. In order to test for these failures, a test consists of sending a *test* message to the tested processor, which should generate a *reply* message. If no *reply* message arrives within a specified time period, the test is assumed to indicate failure. Now, since we are dealing with crash failures, we do not need to validate the information from the processors in the TESTED_BY(P_i) set, so the re-testing step 3 can be omitted from the NEW_SELF algorithm. Furthermore, since no test result other than the message indicating the reception of the *test* message is returned, the *reply* message can be made to carry the diagnostic information. Finally, based on the assumption about the failure model, no processor ever forwards incorrect information, so the validation step 4 is not required. Thus, assuming that the communication network is synchronous and the processors only suffer from crash failures, the following variation of the NEW_SELF algorithm would guarantee that every failure is known by every correct processor in the system after the required number of iterations.

1. P_i sends *test* message to all $P_j \in$ TESTED_BY(P_i);
2. P_i receives *reply* message with diagnostic information from all $P_j \in$ TESTED_BY(P_i);
3. P_i combines the information and sends the results to TESTERS_OF(P_i).

However, a problem arises if the network is assumed to be asynchronous or the processors experience performance failures. This means that two different processors testing some processor P_j might get different results. A simple approach would be to consider a processor faulty if at least one non-faulty processor detects it to have failed. If this approach is taken, the algorithm can be simplified; in particular, the information passed around can be reduced to a simple boolean vector, *Members*$_j$, where *Members*$_j[i]$ indicates whether processor j considers processor i to be correct and functional. Assume now that initially these vectors are initialized to true and that a processor changes an entry to false when the corresponding processor fails a test. Now, the information that has to be passed around consists of these vectors, with information being combined by an elementwise "and" operation. If the suspicion of one processor is not considered to be enough, all *Members* vectors have to be propagated and a failure is only declared when all t (or a certain portion of the t testers) consider a processor to have failed.

This algorithm guarantees that every correct processor has the same view of the group membership after some required number of iterations. If those processors that are considered faulty are not explicitly excluded from the algorithm, i.e., they can continue testing the processors in their TESTED_BY set, they will eventually find out that the other processors have decided that they have failed, assuming no network partitions.

Similar changes can be made to EVENT_SELF to derive a new membership algorithm more efficient than the one derived from NEW_SELF.

6.3.3 Adaptive DSD System Diagnosis Algorithm

Adaptive DSD [BB91] is a variation of the NEW_SELF algorithm where testing assignments change adaptively during the execution of the system diagnosis. Adaptive DSD assumes a distributed network, in which nodes perform tests of other nodes and determine them to be faulty or fault-free. Test results conform to the PMC model. The Adaptive DSD algorithm is based on a similar test cycle to NEW_SELF and EVENT_SELF, with four phases: test, receive information, test again, accept information. The difference is that the testing set changes with time. In contrast with NEW_SELF, no restrictions are placed on the number of faulty nodes, and the algorithm can diagnose any fault situation with any number of faulty nodes. For correct diagnosis, each node must be tested by at least one fault-free node; in Adaptive DSD, each node is tested by exactly one fault-free node. Each node typically tests one other node, but can be required to test multiple nodes, of which one must be fault-free. Adaptive DSD is shown to be a considerable improvement over previous algorithms, including being optimal in terms of the total number of tests and messages required.

The Adaptive DSD algorithm operates at each node by first identifying another unique fault-free node and then updating local diagnostic information with information from that node. Functionally this is accomplished as follows. List the nodes in sequential order, as $(n_0, n_1, \ldots, n_{N-1})$. Each node, n_i identifies the next sequential fault-free node in the list. This is accomplished at n_i by sequentially testing consecutive nodes $n_{i+1}, n_{i+2}, \ldots$, until a fault-free node is found. Diagnostic information received from the fault-free node is assumed to be valid and is utilized to update local information. All addition is modulo N so that the last fault-free node in the list will identify the first fault-free node. As a result of this process the fault-free processors will form a directed cycle. After N testing rounds every fault-free processor knows about every other fault-free processor in the system. The detailed algorithm and correctness proofs can be found in [BB91].

As noted above, the Adaptive DSD algorithm is optimal in terms of the total number of tests required. The algorithm can be improved, however, with respect to total number of messages and diagnostic latency. In a manner similar to the EVENT_SELF algorithm, it is possible to reduce the number of diagnostic information messages by only transmitting when the information has actually changed. This can be accomplished by means of timestamping information and comparing timestamps of new diagnostic messages to currently stored information. Also, the diagnostic latency can be decreased by not waiting until the next testing period before changed information is forwarded. This can be accomplished by storing information about processors to whom the information has to be forwarded in FORWARD_TO tables. This table is essentially the reverse of the testing subnetwork: if a node n_i tests n_j, n_j includes n_i in its FORWARD_TO table. Now, when n_i gets new diagnostic information, it forwards the information to the nodes in the FORWARD_TO table instead of waiting until the next testing period. Upon receiving the information, these nodes test node n_i to validate the correctness of the information.

6.3.4 Adaptive DSD Membership Algorithm

Adaptive DSD is fairly close to membership algorithms in the sense that the testing arrangement is not fixed and that any number of failures can be tolerated. It could be used as the membership algorithm in a synchronous system where the lack of response can be reliably credited to the failure of a processor. However, to deal with more realistic asynchronous systems some modifications must be made. In the following, we present a complete membership algorithm designed for asynchronous distributed systems based on the Adaptive DSD algorithm. This algorithm deals with both failures and recoveries, and guarantees the agreement property defined in chapter 4.

Assume that processors in the group start with the same initial membership. Now, as in Adaptive DSD each processor finds a correct processor based on the identifiers of the processors. Testing is based on *test* and *reply* messages, as in the membership algorithm derived from the NEW_SELF algorithm. Similarly, the *reply* message carries the current membership view of the sending processor. For now, assume that the communication layer provides best effort message delivery, i.e., messages are retransmitted until there is reason to believe that the receiver has failed. We assume that each message includes at least the sender and an incarnation number that identifies how many failure/recovery cycles the processor has experienced. It is assumed that the incarnation number is stored in such a manner that it can be recovered after failure. Figure 6.2 outlines two procedures used by the membership algorithm. Procedure *reconfigure*, given the current view of the membership, calculates the new identities of *Tester*, the processor that tests this processor, and *Tested*, the processors that is tested by this processor. Procedure *combine*, given the current membership m_1 and the membership in a reply message m_2, decides which membership changes have occurred, notifies the application accordingly, and updates the local membership. Figure 6.3 outlines an event-driven pseudo-code for the membership algorithm.

The algorithm operates as follows. Initially, it creates a logical ring of correct processors in a manner similar to Adaptive DSD. If a processor fails, it will be detected by its tester and this information propagated to everybody using the *Tester* links. If a processor p_i recovers, it finds a correct processor p_j analogously to the initialization of the logical ring. p_j, in turn, finds out about the recovery of p_i when it receives a *test* message from p_i. p_j notifies its *Tester* about the recovery of p_i, with the information being propagated to every other correct processor. Every time the membership changes, each processor recalculates the identities of its *Tested* and *Tester* processors.

The algorithm can be verified by showing that if a correct processor p changes its local view of membership, this change will be reflected eventually on all the other correct

```
type MemType = array[1:N] of record {ok: bool; inc: int};

procedure reconfigure(mem:MemType;tested,tester,id:int) {
    var i: int;

    i = (id+1)mod N; while !mem[i].ok do i = (i + 1)mod N;
    tested = i; i = (id–1)mod N;
    while !mem[i].ok do i = (i – 1)mod N; tester = i;
}
procedure combine(mem,new_mem:MemType;id,inc,tester,tested:int) {
    var change: bool = false;

    if inc = new_mem[id].inc and !new_mem[id].ok then {
        notify application: failure of this processor;
        initiate recovery; return();
    }
    for i = 1 to N: i ≠ id do {
        if mem[i].ok and !new_mem[i].ok and mem[i].inc = new_mem[i].inc then {
            notify application: failure of i;
            mem[i].ok = false; change = true;
        } elseif !mem[i].ok and new_mem[i].ok and mem[i].inc < new_mem[i].inc then {
            notify application: recovery of i;
            mem[i] = new_mem[i]; change = true;
        } elseif mem[i].ok and new_mem[i].ok and mem[i].inc < new_mem[i].inc then {
            notify application: failure and recovery of i;
            mem[i] = new_mem[i]; change = true;
        }
    }
    if change then {
        send reply message with mem to tester; reconfigure(mem,tested,tester,id);
    }
}
```

Figure 6.2: Procedures for Adaptive DSD Membership Algorithm

processors in the same partition, provided that p does not fail. It is also easy to show that every failure and recovery will eventually be detected by every correct processor in the same partition. Furthermore, this algorithm guarantees that if a processor has been falsely detected as faulty, it will eventually receive a membership set indicating it is faulty, which causes it to initiate recovery. The details of the proof have been omitted for brevity.

As noted above, the algorithm can handle the case where processors are falsely assumed faulty. The key here is that those processors falsely accused continue communication with other processors and so will eventually receive a membership view where they have been deleted. The situation will be totally different if a true network partition occurs. In this case, each partition will form a group of correct processors of its own, with no provisions for further communication with other groups. This can be fixed simply by having each processor occasionally test a random processor that it assumes is faulty.

For the code outline, we assume that communication is reliable. This assumption could easily be omitted by having the membership algorithm take care of retransmitting

```
protocol Membership(MyId:int) {
   var Mem: MemType;       % membership status of each site
       Tested: int;        % site tested by this site
       Tester: int;        % site that tests this site
       MyInc: stable int;  % current incarnation of this site

   when msg: message from network do {
      var sender: int = msg.sender;

      if msg.type = test then {
         if !Mem[sender].ok and msg.inc > Mem[sender].inc then {
            notify application: recovery of sender;
            Mem[sender].ok = true; Mem[sender].inc = msg.inc;
            send reply message with Mem to Tester;
            reconfigure(Mem,Tested,Tester,MyId);
         }
         send reply with Mem to sender;
      } elseif msg.type = reply then {
         cancel reply timer; cancel test cycle timer;
         combine(Mem,msg.Mem,MyId,MyInc,Tester,Tested);
         set test cycle timer;
      }
   }
   when reply timer timeout do {
      Mem[Tested].ok = false; notify application: failure of Tested;
      reconfigure(Mem,Tested,Tester,MyId);
   }
   when test cycle timer timeout do {
      cancel reply timer; send test message to Tested;
      set reply timer; set test cycle timer;
   }
   when recovery do {
      for i = 1 to N do { Mem[i].ok = true; Mem[i].inc = 1; }
      MyInc = MyInc + 1; Mem[MyId].inc = MyInc;
      notify application: recovery of each processor i;
      reconfigure(Mem,Tested,Tester,MyId); set test cycle timer;
   }
   initial { MyInc = 0; initiate recovery; }
}
```

Figure 6.3: Adaptive DSD Membership Algorithm

the *test* message a certain number of times before suspecting the failure of the receiver. The *reply* messages need not be retransmitted since if one is lost, the reception of a new *test* message will cause a new *reply* message to be generated. Other extensions to the algorithm are also possible. For example, the assumption about the known identities of all possible processors in the group could be omitted and the group could be built on the fly when processors find out about other processors. Furthermore, as will be discussed in the following sections, the algorithm could be augmented to provide ordering and other properties.

Although the algorithm uses similar ideas as some published membership algorithms—

for example, using a logical ring structure [AMMS$^+$93, HS94a, RFJ93]—the complete algorithm is different from any of which we are aware. In terms of message complexity it is very attractive. If the test messages and replies are excluded, the algorithm uses exactly $N-1$ messages to propagate information about a membership change to every processor in the group, which is the optimal for a system with no multicast communication. Also, in contrast to token or coordinator based membership algorithms, there are no complicated steps required to deal with coordinator failure or token loss. Finally, the extension required to deal with network partitions is remarkably simple.

A version of the algorithm that can tolerate communication failures and can recover from network partitions has been implemented using the C++/Solaris prototype of the event-driven execution model. In spite of the extensions, the algorithm is less than 300 lines of C++ code, and has behaved as expected under all test scenarios.

6.3.5 General Guidelines

It is premature to draw any general conclusions on how to transform an arbitrary system diagnosis algorithm into a membership algorithm mechanically. The above examples show that in the case of the family of system diagnosis algorithms derived from the NEW_SELF algorithm, the steps consist mostly of eliminating the validation step due to the difference in the failure model, and dealing with false failure detections.

In general, most non-distributed system diagnosis algorithms are not suitable for transformation into membership algorithms since membership algorithms typically do not admit centralized solutions. There may be some exceptions to this rule for parallel applications, where the user initiates the computation from a distinguished site that is assumed not to fail.

6.4 From Membership to System Diagnosis

In this section, the reverse transformation is illustrated. First, similar to above, a typical membership algorithm is transformed into a system diagnosis algorithm while preserving some of the properties of the membership algorithm. This makes the resulting algorithm superior in the strength of the properties it guarantees to the user or the application. Second, a typical system diagnosis algorithm is augmented with some of the properties a membership algorithm might provide. This makes it easier for an application that utilizes the information from the system diagnosis service to deal with failures in the environment.

6.4.1 Family of Membership Algorithms

As an example, we take the family of membership algorithms consisting of weak, strong, and hybrid described in [RFJ93] and briefly outlined in section 4.4.7. All the algorithms in this family assume an asynchronous communication network and processors that experience only crash failures. The algorithms are based on a logical ring where every processor monitors its immediate neighbors by means of *heartbeats*, i.e., periodic *hello* messages.

If a processor suspects a failure, it notifies a processor elected to be the leader, which then distributes the information to all the members in the group. The failure of the leader is dealt with by having an agreed processor be the *crown prince* that takes over. The exact properties guaranteed by the different variants were described in section 4.4.7.

6.4.2 System Diagnosis Algorithm

Assume first that we have a reliable way of testing a processor for a failure, in particular, that communication is synchronous. Now, it is very easy to transform the above membership algorithms into system diagnosis algorithms by simply changing the heartbeats into tests and validating all information sent by another processor along the lines of the NEW_SELF algorithm. The heartbeats can be transformed into tests in two different ways. If the test is a self-test, each processor can run the test program and send the results to its immediate neighbors using the heartbeat messages. If, on the other hand, the test is determined by the tester, each heartbeat message is simply replaced by a pair of messages: test and reply. If a test fails, the leader is notified of the failure of the tested processor. These steps are common for all the algorithms.

The validation step consists of testing the sender of a message before accepting the information as valid. If the test fails, the tester assumes the tested processor is faulty and does not accept any information from that processor. For the weak algorithm, the leader has to validate the correctness of the processor reporting the failure, while each member receiving the new membership from the leader must validate the correctness of the leader. The strong algorithm has a two-phase commit step for the transition to the new membership. In the first step, the leader informs all group members to prepare for the new membership and then waits for responses. In the second, it sends out a commit message, which causes recipients to install the new membership. All messages sent by the leader in the two-phase algorithm have to be validated. This guarantees the properties discussed in section 4.4.7.

Note that all the changes to the membership algorithm are very easy, practically mechanical, and that the resulting algorithm conforms to the style of traditional system diagnosis algorithm. However, the ordering guarantees are stronger than any system diagnosis algorithm of which we are aware. Moreover, the message complexity of the algorithm is comparable to the best system diagnosis algorithms. For example, the weak algorithm and Adaptive DSD both require N validated message exchanges to propagate information about a failure or a recovery. Finally, the detection delay is close to optimal since distributing the change information only involves one validated message exchange and one or two validated multicasts. The exact delay depends on the implementation of validated multicast.

Now, consider the case where communication is asynchronous. With this model, there is no way to distinguish between a failed processor and a slow communication network. However, since the original membership algorithms were designed for asynchronous systems, the derived system diagnosis algorithm will work as well given a few minor changes. In particular, if a processor fails to receive a response to a test it is applying to a neighbor within a specified time limit, a failure is suspected. Furthermore, if validation fails to complete within a specified time limit, the information that was being validated is simply considered to never have been received. All the properties specified for the membership algorithms are still guaranteed.

6.4.3 Augmented Adaptive DSD

Recall that the Adaptive DSD algorithm described above was characterized as being fairly close to membership algorithm in many respects. In this section, the Adaptive DSD algorithm is augmented to provide ordering properties so that the application layer on every site that executes the algorithm sees the failures and recoveries in a consistent total order. We also show how the algorithm can be augmented to deal with false failure detections.

One option for implementing a total order is to use a leader process like above. Here, however, we outline an implementation that relies on logical clock based timestamp ordering [Lam78]. Assume that the communication between processors is FIFO ordered and that each processor uses its local logical clock to timestamp the information it passes on about failures and recoveries that it has detected. Now, since the communication for the system diagnosis is confined to the logical ring, any processor that receives this information has its logical time greater than the timestamp of the event. To implement a total order based on these timestamps, each processor must delay until it is guaranteed that no correct processor can issue a timestamp smaller than the timestamp of the event being ordered. However, since the order of the processors in the logical ring is known to every processor, it is easy to calculate which processors have logical time greater than the event's timestamp and which therefore cannot submit an event with a smaller timestamp. Thus, every processor upon receiving an event indication updates its own view of the functioning processors and stores the event in a queue in the order based on the timestamp. Also, based on the processors that have already seen the event indication, each processor keeps track of the minimal timestamp each processor could still submit. If the event at the head of the queue has a timestamp smaller than this value, the event can be forwarded to the user or the application with the guarantee that the order on each of the correct processors is identical. This algorithm requires that each event indication be rotated at most twice around the logical ring.

The Adaptive DSD algorithm does not address the possibility that a site is ever erroneously assumed faulty. This may not be the case in a realistic system, and in particular, if the communication network is asynchronous, tests may falsely accuse a processor of failure. Section 6.3.4 presented a membership algorithm that deals with false detections by means of eventually notifying the falsely detected processors about their suspected failure so they can fail and recover and rejoin the group. A similar modification to the Adaptive DSD system diagnosis algorithm would give the same outcome.

6.4.4 General Guidelines

In general, when developing a new system diagnosis algorithm using a membership algorithm as a starting point, the changes that have to be made are modifying the test to one that conforms to traditional system diagnosis algorithms and validating all information exchanges by testing the sender before accepting the information. Many membership algorithms do not explicitly test other members for failures. If an algorithm like this is taken as a starting point, an explicit test must be added. However, if the membership algorithm is such that any member can suspect the failure of any other member, such as in Consul [MPS92], implementing corresponding semantics in the system diagnosis algorithm by means of explicit tests may be too expensive for a practical system. As was shown, the validation step can be fairly simple given leader-based information distribution. The same is true for information distribution based on a logical ring as demonstrated, for example, by the Adaptive DSD algorithm. However, numerous membership algorithms use multicast for distributing change information and for reaching agreement either on the accuracy of the information or the new membership after the change. Although multicast may be an efficient solution for membership algorithms provided that the underlying network provides physical broadcast (or a close approximation), in the case of system diagnosis, the validation of the information may be too expensive. Naturally, if testing is based on self-tests, optimizations such as having each processor execute the self-test and including the test result with the broadcast message may be possible. The validation step may not require any extra message exchange in this case. Finally, membership algorithms that assume synchronous communication are often more straightforward to transform into system diagnosis algorithms due to the similarity of assumptions.

6.5 Conclusions

This chapter contrasted system diagnosis and membership, two services that have similar goals but that have been traditionally treated separately. These problems can be characterized as, given the assumptions about the failure model and communication service, how to detect a failure of the given type, and how to distribute information about the failure to every processor in a manner that tolerates the given failure model and guarantees required service properties. Some of the assumptions made by membership and system diagnosis algorithms have traditionally been different, as have the service guarantees provided. However, the problems are closely related enough that by changing the testing method and the manner in which information is distributed, algorithms in one area can be transformed into the other. Based on these findings, we demonstrated how well-known system diagnosis algorithms can be transformed into membership algorithms and how a family of membership algorithms can be transformed into system diagnosis algorithms that guarantee stronger properties than other similar algorithms. Furthermore, it was demonstrated how a system diagnosis algorithm can be augmented to provide properties traditionally provided by membership algorithms.

While these observations potentially give rise to a large number of new algorithms for both membership and system diagnosis, the most important contribution is showing the relationship between these problems. Lately, the distinction in the literature between membership and distributed system diagnosis algorithms seems to be diminishing. In particular, some of the new system diagnosis algorithms are starting to exhibit characteristics typical to membership algorithms, such as dynamically assigning testing subnetwork and tolerating any number of failures. Ideally, the fields of system diagnosis and membership should be considered as one, with the choice of which algorithms to use being based solely on the requirements placed on the system and the properties of the underlying computing environment.

Much work in this area remains to be done, however. Here, we concentrated only on distributed system diagnosis based on the PMC model, which is only one approach. For example, we did not address probabilistic system diagnosis and how those strategies might be applied to membership. Finally, an obvious future goal is to use the micro-protocol approach described in this dissertation as a way to combine system diagnosis and membership. The result would be a general configurable change detection and reporting service, where configurability is used to match the failure model to the requirements of the application. In particular, the failure model chosen would dictate which failure detection module is used and which information exchange module is used for communication between sites to ensure that the expected failure model cannot corrupt the data that is exchanged.

CHAPTER 7
GROUP REMOTE PROCEDURE CALL

Remote Procedure Call (RPC) [BN84, Nel81] is a communication abstraction designed to simplify the writing of distributed programs. With RPC, a request for service from a *client* to a *server* process is structured to give synchronization semantics at the client similar to normal procedure call. Numerous examples of different RPC services and implementations exist, including Firefly RPC [SB90], Alphorn [AGH$^+$91], lightweight RPC [BALL90], Peregrine [JZ93], [RSV94], and SUPRA-RPC [Sto94]. Among the commercial RPC packages released have been Courier from Xerox [Xer81], Sun RPC [Sun88], Netwise RPC from Novell Netware, and NCA from Apollo [Apo89]. [TA90] gives a survey of work in this area.

On the surface, the semantics of RPC seem very simple, yet the reality is that there are subtleties and variations. For example, there are many ways to define how an RPC service deals with server and communication failures. The set of options grows even more when considering *group RPC*, a variant of RPC often used for fault tolerance where the request is sent to a group of servers rather than one. For example, there are numerous ways to define how requests are ordered at a server, and how the multiple replies to a given request are collated for return to the client. Making choices in each of these cases gives a different variant of RPC. This explains at least in part why so many RPC systems have been defined and implemented: since each system typically realizes one and only one set of semantics, a new system is built whenever different semantics are called for by the application requirements.

In this chapter, we apply our approach of constructing configurable fault-tolerant distributed services to group RPC. To simplify the presentation, we concentrate on RPC features related to distribution and fault tolerance. Other issues, although important, are not addressed here. These include stub generation and heterogeneity [Sun88, Gib87, HS87, Apo89, TB90, WSG91], binding [BN84, LT91, BALL90], performance or performance optimizations [PA88, RST89, SB90, BALL90], and security issues [Bir85b]. Our design assumes unreliable asynchronous communication and crash failure model.

7.1 Properties of RPC Services

The construction of any configurable service starts from identifying the abstract properties of the service, as was described in chapter 3 and illustrated for the membership service in chapter 4. Here, we start with the properties of *simple RPC*, by which we mean RPC calls to non-replicated servers. Then, these properties are augmented to accommodate group RPC.

7.1.1 Simple RPC

The properties of simple RPC can be classified into five categories: *failure semantics*, *call semantics*, *orphan handling semantics*, *communication semantics*, and *termination semantics* defined as follows.

Failure semantics specify what guarantees are given to the client about the execution of the server procedure, both when the call returns successfully and when the call returns unsuccessfully. The two properties are *unique execution*, which states that the server procedure is not executed more than once, and *atomic execution*, which states that the server procedure is either executed completely or not at all.

Call semantics specify the synchrony of the client call. A call is *synchronous* if the client thread is blocked until the call to the server is completed, while a call is *asynchronous* if the client thread returns immediately. In the latter case, the RPC system may include another system call that allows the thread to retrieve results later. Although synchronous is most commonly used, a number of systems provide an asynchronous option as well (e.g., [ATK91]).

Orphan handling semantics specify how *orphans*—that is, server computations associated with clients that have failed—are dealt with. Orphans not only waste computing resources, but may also interfere with new calls issued by a recovered client. Options for dealing with orphans include *interference avoidance*, where the orphans finish their computation before the recovered client is allowed to issue new requests, and *orphan termination*, where orphans are terminated upon detection [PS88, Shr83].

Communication semantics specify properties about the communication between the client and server. Here, we concentrate on *reliable communication*, which can be implemented by message acknowledgments and retransmissions. Of course, if the reliability guarantees provided by the underlying communication layer are strong enough, then the RPC layer may not need to implement reliability. Furthermore, an application builder might choose not to have this property for other reasons, such as, efficiency or cost.

Termination semantics specify the guarantees that are given about termination of a call. Due to communication and server failures, the client site may retry a call an arbitrary number of times. *Bounded termination* states that a call always terminates and the client thread returns within a bounded, specified time. If the server has not responded by the deadline, the call returns with an indication of failure.

Note that our classification of failure semantics subsumes more traditional distinctions, which can be summarized as follows [PS88]. *At least once* guarantees that if the invocation terminates normally, the remote procedure has been executed one or more times, and if it terminates abnormally, no conclusion is possible [Spe82]. *Exactly once* guarantees that if the invocation terminates normally, the remote procedure has been executed exactly one time, and if it terminates abnormally, no conclusion is possible other than that it has not been executed more than once. *At most once* is the same as exactly once if the invocation terminates normally, while if the invocation terminates abnormally, the execution of the remote procedure is guaranteed to be atomic, that is, either executed completely or not at all [LS83]. In our classification, each of these semantics can be realized as some combination of the unique and atomic execution properties, as illustrated in Table 7.1.

	Unique execution	Atomic procedure execution
At least once	NO	NO
Exactly once	YES	NO
At most once	YES	YES

Table 7.1: Decomposition of Traditional RPC Semantics

Many papers that discuss RPC do not even address the failure semantics. Exactly once appears to be the most popular in implemented systems. For example, it is the semantics chosen in Rajdoot [PS88], Courier RPC [Xer81], OSI RPC [LH94]. Some systems guarantee at least once, such as [Mar89]. At most once is rare because of the cost of implementation, although it is provided in both Atomic RPC [LG85] and Arjuna [SDP91].

7.1.2 Group RPC

Group RPC is any RPC service where the request is sent to more than one server—that is, a *server group*—using either multicast or point-to-point communication. Group RPC has numerous applications. For example, it can be used to implement replicated servers to increase availability of the service in the event of failures, to implement parallel computation, or to improve response time. Examples of group or multicast RPC include [CGR88, Coo85, WZZ93, YJT88].

Here, we consider only one-to-many group RPC, in which one client uses RPC to invoke a procedure implemented by a server group. Other variants not discussed here are many-to-one RPC, where a replicated client invokes an RPC on a non-replicated server, and many-to-many RPC, where a replicated client invokes a procedure implemented by a server group. The semantics of group RPC are identical to ordinary RPC when considering the call, orphan handling, communication, and termination semantics discussed above. However, group RPC also includes the *ordering semantics*, *collation semantics*, and *failure semantics for group RPC* defined as follows.

Ordering semantics specify the order in which concurrent calls are executed by different members of the server group. *FIFO order* guarantees that all calls issued by any one client are executed in the same order by all group members, while *total order* guarantees that all calls are executed in the same total order. Other variants such as *causal order* have also been defined.

Collation semantics specify how responses from the multiple members of the group are combined before being returned to the client. Different possibilities include return any reply, return all replies, or return the result of a function that maps all replies into one result (for example, average). Of course, any of these alternatives can be described as a function, so we take the general approach of having the user provide the desired collation function at initialization time.

Failure semantics for group RPC are defined as failure semantics for simple RPC but augmented for more than one server. In particular, different combinations of failed and successful executions must be considered. Specifically, we must now consider how many servers must succeed in order for the group RPC to be considered successful, a property we term *acceptance semantics*. Possibilities range from requiring successful execution at only one server to successful execution at all servers. Note that the concept of "all" is non-trivial. For example, if we assume the server group has a fixed membership and that all sites eventually recover, a response will be forthcoming from all servers if the client waits long enough. On the other hand, the client might not want to wait for recovery, but is willing to settle for the responses from all servers that are still functioning. Dealing with site failure and recovery in this way constitutes the *membership semantics* of the group RPC.

Note that conventional failure semantics can also be extended to group RPC. For example, *at least once on one* semantics in this case guarantees that if the invocation terminates normally, the remote procedure has been executed one or more times on at least one server site, and if it terminates abnormally, no conclusion is possible. This semantics is useful for getting one read-only response quickly and is used, for example, in the lookup RPC of GRPC [WZZ93]. *At least once on all* semantics is defined similarly, except that the invocation must have been executed on all server sites. Note that the number of sites where the call must succeed can be anything between "one" and "all". Similarly, we can define *exactly once on ...* and *at most once on ...* semantics. Note that all these semantics can be trivially mapped to the unique and atomic execution properties with appropriate acceptance policies.

Figure 7.1 summarizes the properties of group RPC and by implication, simple RPC. Note in the figure that *Collation*, *Acceptance*, and *Membership Semantics* are represented as choice nodes since they all represent sets of different collation, acceptance, and membership semantics properties. Furthermore, the *Basic RPC* node represents a simple RPC with no ordering, reliability, boundedness, atomicity, uniqueness, or orphan handling properties. Based on this dependency graph there are 192 possible functional combinations of the given properties.

7.2 Implementing a Configurable Group RPC Service

In this section, we outline a modular and configurable implementation of RPC using the event-driven model described in chapter 3. This section outlines the general implemen-

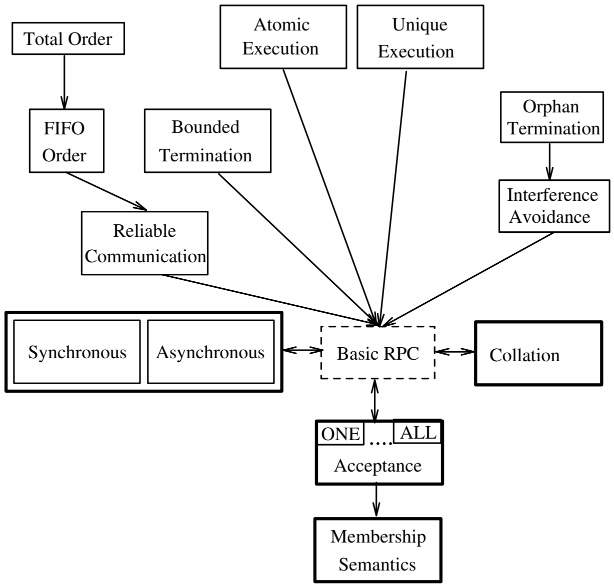


Figure 7.1: Group RPC Dependency Graph

tation strategy, events, and shared data structures, with the actual micro-protocols being presented in the next section. The focus here is on group RPC. Simple RPC can be seen as a special case in this implementation, although in practice it would likely be implemented separately to obtain a more compact and efficient protocol.

7.2.1 Outline

RPC services are typically structured along the lines described in Nelson's thesis [Nel81] consisting of the *user* (or client), *user-stub* (or client stub), the RPC communication package (*RPC runtime*), the *server-stub* and the *server*. The user (client) and server look like regular non-distributed programs, with the stubs masking the distribution and communication from the user and server. Stubs responsibilities include converting parameters between different representation formats and layouts, and transmitting converted parameters across networks. Stubs may also use encryption techniques to make communication more secure. Stubs are often generated automatically using a *stub generator*.

In large part our design reflects this general structure of RPC service implementations. In following the tenets of the event-driven execution model, the RPC service is implemented as a composite protocol **GroupRPC**. We assume that the system also includes the following composite or simple protocols: **UnreliableCommunication**, **User** (that is, the server and client code), and possibly **Membership**. The unreliable communication protocol provides the transport service needed to deliver messages between **GroupRPC** on the client and server sites. We also assume that the client above **GroupRPC** has a stub for each RPC call that marshals arguments and does binding. A similar stub on the server site unmarshals the data and invokes the actual procedure. From the perspective of **GroupRPC**, then, the arguments are treated as one continuous untyped field that is copied to and from messages.

7.2.2 GroupRPC Composite Protocol

The composite protocol contains a number of shared data structures. The first is `pRPC`, a table for storing pending remote procedure calls at the client. For each pending call, the table contains one record (type `ClientRecord`) with fields for unique call identifier (`id`), operation identifier (`op`), input and output parameters (`args`), identity of server group (`server`), semaphore for the pending client thread (`sem`), number of responses required for the call (`nres`), list of server processes from which a response to the call is waited (`pending`), and status of the call (`status`). The call identifier is carried along with the call to the server and its response so that calls and responses can be matched. The number of responses required field has a value of one for simple RPC and a value depending on the acceptance policy for group RPC. The `pending` list has, for each entry in the list, the process id of the server process (`p`) and a counter for the number of call attempts made (`attempts`). The `status` field has values OK for a normally terminated call, WAITING for a pending call, and TIMEOUT for an incorrectly terminated call. The `pRPC` table is indexed using the call identifier (for example, `pRPC(id)`), while other fields are accessed using record notation (for example, `pRPC(id).sem`). Access to the `pRPC` table is controlled using semaphore `pRPCmutex`.

A similar data structure, `sRPC`, is used at the server to store information about each pending client call. For each client call, `sRPC` has one `ServerRecord` that contains the fields `id`, `op`, `args`, and `server` that are identical to the fields in `pRPC`, and fields for the identity of the client (`client`) and a boolean vector for keeping track of which properties have been satisfied for the call (`hold`). Access to `sRPC` is controlled using semaphore `sRPCmutex`.

As described above, messages are exchanged between the user and **GroupRPC** (type `UserMsgType`), and between **GroupRPC** and the underlying communication service (type `NetMsgType`). The `NetMsgType` has fields for type (`type`), the sender (`sender`), and the incarnation number of the sender (`inc`), in addition to fields `id`, `op`, `args`, and `server` that are similar to above. Finally, if a message is an acknowledgment, it has a field for the call being acknowledged (`ackid`). The message type is either CALL or REPLY for the standard RPC messages, ORDER for messages carrying total ordering information, or ACK for acknowledgment messages. The `UserMsgType` has `type`, `id`, `op`, `args`, `server`, and `status` fields analogous to above.

Finally, a number of global variables are accessed by all micro-protocols. Variable `Net` is a pointer to the communication protocol and a point-to-point send or multicast operation that can be executed with operation `Net.push`. Variable `Server` is an analogous pointer to the user protocol and an operation `Server.pop` that is used to pass messages up the protocol stack. A call to this operation is blocking. Variable `inc_number` contains the number of the current incarnation. The boolean `HOLD` vector is used to indicate which properties must be satisfied before a call can be passed up to the server, or, in other words, which micro-protocols must process the message. An analogous vector is associated with each call indicating which properties have been satisfied, and when the two are equal, the call is given to the server. The `Members` data structure contains the identities of the sites considered functioning at the moment.

7.2.3 Events

The events used by **GroupRPC**'s micro-protocols are the following; for simplicity, we assume all events are blocking and sequential:

- CALL_FROM_USER(umsg:UserMsgType): Triggered at the client site when a new call from the user protocol arrives.
- NEW_RPC_CALL(id:int): Triggered at the client site when a call is ready to exit **GroupRPC** and be sent to the server site. This event is used primarily by micro-protocols to update data structures before the invocation is sent.
- CALL_TO_SERVER(id:int): Triggered at the server site before **GroupRPC** passes the call to the server.
- REPLY_FROM_SERVER(id:int): Triggered at the server site when the server passes a call response to **GroupRPC**.
- MSG_FROM_NET(msg: NetMsgType): Triggered when a message arrives from the network; used at both the client and server sites.
- RECOVERY(IncNumber:int): Triggered when a failed site is recovering; used at both client and server sites. The argument `IncNumber` is the sequence number of the current incarnation.
- MSHIP_CHANGE(who: pid, change: MemChange): Triggered by the membership service when a process fails or recovers. Most properties identified in section 7.1 do not require this information in their implementations, so the membership component of the system is omitted in these cases.

7.3 RPC Micro-Protocols

In this section, we outline the micro-protocols for the configurable RPC service. All major micro-protocols are described, with pseudo-code for several shorter micro-protocols given to illustrate the programming style. Detailed pseudo-code for all micro-protocols can be found in [HS94b]. The section starts with a base micro-protocol **RPCMain** followed by collection of micro-protocols for various aspects of remote procedure calls ranging from ones for user thread managements to dealing with orphans.

7.3.1 RPCMain

The **RPCMain** micro-protocol handles the main control flow of an RPC on both the client and server sides. Specifically, it stores the call request in `pRPC`, sends the request over the network, issues the call to the server, sends the response over the network, and stores the results in `pRPC`. The call is issued to the server only when all the specified properties (e.g., ordering properties) are satisfied. Checking is done by comparing the global `HOLD` array to the `hold` array of each individual call. This checking and eventual forwarding is done by calling the `forward_up` procedure, which is exported by **RPCMain**. This micro-protocol constitutes the basic, unchangeable, foundation for the configurable RPC service.

7.3.2 User Thread Management

As described in section 7.1, an RPC invocation can be either synchronous (blocking) or asynchronous (non-blocking). The **SynchronousCall** micro-protocol implements synchronous RPC semantics by blocking the caller thread and matching responses with pending threads. The code segment in Figure 7.2 illustrates this micro-protocol. Notice that the event handler registered for the event CALL_FROM_USER is executed after all the other micro-protocols have processed the call and sent it to the server.

```
micro-protocol SynchronousCall()

    event handler msg_from_user(umsg:UserMsgType) {
        if umsg.type = Call then {
            P(pRPC(umsg.id).sem);    % block at private sem
            umsg.args = pRPC(umsg.id).args;    % copy results
            umsg.status = pRPC(umsg.id).status;
            pRPC -= pRPC(umsg.id);    % cleanup
        }
    }
    initial { register(CALL_FROM_USER,msg_from_user,LAST); }
}
```

Figure 7.2: The **SynchronousCall** Micro-Protocol

AsynchronousCall implements a very simple asynchronous RPC where the caller thread is not blocked when the call is issued, but may later request the result using a REQUEST message. If the result is pending, the request message returns immediately; otherwise, the caller is blocked until the result arrives or the call is otherwise terminated.

7.3.3 Communication Aspects

The standard approach to making RPC reliable is to retransmit the call to the server site until the response or some other form of acknowledgment arrives. The **ReliableCommu-**

nication micro-protocol implements these retransmissions. The micro-protocol uses the `pRPC.pending` field to determine which sites have not yet responded.

RPCMain combined with **ReliableCommunication** provides for unbounded termination, that is, the **GroupRPC** protocol at the client side keeps trying until it gets a response. In order to guarantee bounded termination, either a limit of the amount of time that can pass or the number of retransmissions can be used. The implementation of **BoundedTermination** illustrated in Figure 7.3 uses a retransmission limit. The micro-protocol is based on periodically invoking the event handler `handle_timeout` that checks if the attempt limit has been exceeded for each pending call in `pRPC`. If it has, the call is released by signaling the private semaphore on which the call thread is waiting.

7.3.4 Response Handling

The **Collation** micro-protocol in Figure 7.4 implements collation semantics, taking the function used to combine the results and the initial value of the result from the user protocol as parameters. Notice how the return value is initialized when a new call is issued (event NEW_RPC_CALL) and the value is updated every time a response arrives from the network (event MSG_FROM_NET).

```
micro-protocol BoundedTermination(Limit:int;timeout:real) {

    event handler handle_timeout() {
        var expired: bool;

        for each id:int in pRPC do {
            expired = false;
            for each p:pid in pRPC(id).pending do
                if pRPC(id).pending(p).attempt > Limit then expired = true;
            if expired then { pRPC(id).status = TIMEOUT; V(pRPC(id).sem); }
        }
        register(TIMEOUT,handle_timeout,timeout);
    }
    initial { register(TIMEOUT,handle_timeout,timeout); }
}
```

Figure 7.3: The **BoundedTermination** Micro-Protocol

The **Acceptance** micro-protocol implements the corresponding property. For a call to be accepted, it must be executed successfully by at least `ALimit` (Acceptance Limit) members of the server group, where `ALimit` is specified as a parameter at initialization time. If the acceptance limit is greater than the number of group members, we chose to set the number of required responses to the size of the group. An alternative would be to abort the call. The micro-protocol keeps track of responses using an event-handler registered for the MSG_FROM_NET event, and when enough responses have been received, the calling thread's local semaphore is signaled. The micro-protocol also keeps track of membership changes.

```
micro-protocol Collation(cum_func:func,init:arg_type) {

    event handler msg_from_net(msg: NetMsgType) {
        var old_val: arg_type;

        if msg.type = Reply and exists pRPC(msg.id) then
            if exists pRPC(msg.id).pending(msg.sender) then {
                P(pRPC_mutex);
                old_val = pRPC(msg.id).args;
                pRPC(msg.id).args = cum_func(old_val,msg.args);
                V(pRPC_mutex);
            }
    }
    event handler handle_new_call(id:int) { pRPC(id).args = init; }

    initial {
        register(MSG_FROM_NET,msg_from_net,SECOND);
        register(NEW_RPC_CALL,handle_new_call); }
}
```

Figure 7.4: The **Collation** Micro-Protocol

7.3.5 Failure Semantics

RPCMain and **ReliableCommunication** combined with **SynchronousCall** or **AsynchronousCall** provide the equivalent of at least once semantics. To implement exactly once semantics, **GroupRPC** must guarantee that a call will not be executed more than once at each server, that is, the unique execution property from section 7.1. This is implemented by the **UniqueExecution** micro-protocol. The basic strategy is to keep track of requests that have already been executed. In our solution, the server stores its response to the original request until the client acknowledges the response. If a duplicate request is received after the acknowledgment has been received, the message is assumed to be old and simply discarded.

To provide at most once semantics, **GroupRPC** also has to guarantee that execution of the server procedure is atomic, that is, the atomicity property from section 7.1. In situations where the server has no *stable state*—that is, state that would persist across failures, such as values stored on disk—execution is automatically atomic. On the other hand, if the server does have stable state, transactional techniques must be used to guarantee atomicity. These techniques can either be implemented in the server itself, or, with some extra support, within the RPC layer. The tradeoff is efficiency versus transparency: implementing the atomicity within the server means that the technique used can be more application specific, while doing it within the RPC layer simplifies the task of programming the server at the cost of some execution overhead.

Here, we outline an **AtomicExecution** micro-protocol that follows the second approach of implementing atomicity in the RPC layer. To support this, the micro-protocol must have the ability to take an atomic checkpoint of the state of the server and write it to stable storage. Here we assume that the server (application layer) provides an operation `checkpoint(file)` that writes the state of the server to `file`, and an operation `load(file)` that restores the state of the server from `file`. In the following, simple variables that reside in non-volatile storage are labeled **stable**; assignment to these variables is assumed to be atomic.

```
micro-protocol AtomicExecution() {
    var old, new: stable ptr file;    % checkpoint file ptrs

    event handler handle_reply(id:int) { checkpoint(new); old = new;}

    event handler handle_recovery(inc:int) { load(old); }

    initial {
        register(REPLY_FROM_SERVER,handle_reply,FIRST);
        register(RECOVERY,handle_recovery,LAST); }
}
```

Figure 7.5: The **AtomicExecution** Micro-Protocol

Note that this micro-protocol only deals with restoring the state of the server following site recovery, not the state of the **GroupRPC** composite protocol itself. To accomplish this, appropriate checkpoints of composite protocol's state would have to be written at each site or provisions included for retrieving the current state from other group members.

Finally, for **AtomicExecution** to work correctly, calls must be processed one at a time by the server, so an additional micro-protocol, **SerialExecution**, is also needed. The micro-protocol operates by locking semaphore *serial* just before the user level gets the call (event CALL_TO_SERVER) and releasing the semaphore after the call is finished (event REPLY_FROM_SERVER). Note that the event handler for REPLY_FROM_SERVER is constrained to execute after the analogous handler in **AtomicExecution**, thereby guaranteeing that no other call can enter the server before checkpointing is done.

7.3.6 Ordering Calls

The default execution order of the client calls at the server group members is entirely arbitrary, even to the point where calls from the same client may be executed in a different order by different servers. Restricting the order is, however, straightforward by incorporating a suitable micro-protocol.

```
micro-protocol SerialExecution() {
    var serial: semaphore = 1;

    event handler handle_call(id:int) { P(serial); }

    event handler handle_reply(id:int) { V(serial); }

    initial {
        register(CALL_TO_SERVER,handle_call,SECOND);
        register(REPLY_FROM_SERVER,handle_reply,SECOND); }
}
```

Figure 7.6: The **SerialExecution** Micro-Protocol

Two micro-protocols for ordering have been defined: **FIFOOrder** and **TotalOrder**. **FIFOOrder** guarantees that the calls from each client will be served in a FIFO order at every server. The order is based on keeping track of the maximum call identifier received from each client and then executing each call only after the previous one from that client has been completed. For this algorithm to work correctly, the call identifiers must be sequential and each call must reach each server site, that is, micro-protocol **ReliableCommunication** is required. Furthermore, a micro-protocol like **BoundedTermination** that may terminate the retransmission before every server has received the call cannot be used with **FIFOOrder**.

TotalOrder, on the other hand, guarantees that calls from all clients are processed in a consistent order by all servers. The technique used to implement this ordering is to have one group member, the *leader*, assign the order in which calls are to be executed and disseminate it to the group. The leader at any point is defined to be the server with the largest unique identifier of all non-failed servers. Thus, for example, if the initial leader fails, the server with the second largest identifier takes over. Like **FIFOOrder**, it must be guaranteed that every call reaches every server in order to avoid deadlocks.

7.3.7 Dealing with Orphans

The basic set of micro-protocols presented so far ignores orphans in the sense that any responses generated by orphan computations are simply ignored. This approach may, however, cause problems. For example, a client may issue a request, fail, recover, and issue the request again while the previous request is still being processed by the server. As described in section 7.1, two ways of dealing with these problems are interference avoidance and orphan termination.

The micro-protocol **InterferenceAvoidance** implements the first option. The solution technique is based on using client incarnation numbers to partition calls into generations. In particular, if a call from the client arrives with a new incarnation number, execution of the requested procedure can only be initiated once execution of any pending calls

```
micro-protocol InterferenceAvoidance() {
    type client_info_table = table of {client: process_id, inc: int, count: int, next_inc: int}
                                    indexed by client;
    var Cinfo: client_info_table;

    event handler msg_from_net(msg: NetMsgType) {
        var client: process_id;

        if msg.type = Call then {
            client = msg.sender;
            if not exists Cinfo(client) then
                Cinfo += (client,msg.inc,0,msg.inc);
            if Cinfo(client).inc > msg.inc then {
                cancel_event(); return();
            } elsif Cinfo(client).inc < msg.inc then {
                Cinfo(client).inc = MAX_INT;
                Cinfo(client).next_inc = msg.inc;
                if Cinfo(client).count = 0 then
                    Cinfo(client).inc = msg.inc;
            }
            if Cinfo(client).inc = msg.inc then
                Cinfo(client).count++;
        }
    }
    event handler handle_reply(id:int) {
        var client: process_id;

        client = sRPC(id).client; Cinfo(client).count–;
        if Cinfo(client).count = 0 and Cinfo(client).inc = MAX_INT then
            Cinfo(client).inc = Cinfo(client).next_inc;
    }
    initial {
        register(MSG_FROM_NET,msg_from_net,THIRD);
        register(REPLY_FROM_SERVER,handle_reply); }
}
```

Figure 7.7: The **InterferenceAvoidance** Micro-Protocol

with old incarnation numbers have been completed. Rather than storing these calls with new numbers, we use the approach of simply dropping them until all current calls have been finished, relying on retransmission from the client to ensure they will eventually be executed. To avoid starvation, no more calls with the old incarnation number are started once the first one with a new number has been seen.

The micro-protocol **TerminateOrphan** implements the second option of immediately killing orphans as soon as they are detected. Detection can be based either on receiving a message from a newer incarnation of the client, indicating that the previous incarnation died, or by periodically probing the client. **TerminateOrphan** uses the first approach. The actual termination of the server thread executing the orphaned computation is done using thread management operations provided by the underlying operating system.

7.4 Configuring a Group RPC Service

A group RPC service is configured by choosing those micro-protocols implementing the desired properties and then combining with the **GroupRPC** composite protocol to form a customized group RPC service. Figure 7.8 shows the group RPC configuration graph.

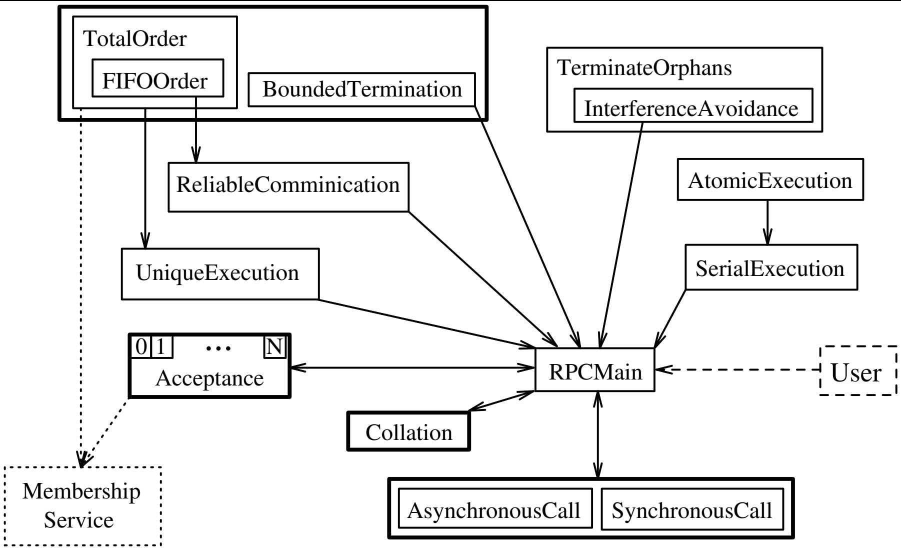


Figure 7.8: Group RPC Configuration Graph

Note that for a group of N servers there are $N + 1$ possible acceptance policies and, in principle, an unbounded number of possible collation policies. Given these facts and the above graph, it is easy to see that hundreds of different group RPC services can be configured from the above micro-protocols and even a small number of collation policies. Given even just one collation and acceptance policy, the total number of functional configurations is still 198.

To illustrate how a specific instance of a group RPC service might be configured, consider a simple group RPC designed to provide quick response time to read-only requests. To achieve this, the system is configured with "at least once" semantics—that is, no unique or atomic execution properties—acceptance one—that is, only one response required—synchronous call semantics, and bounded termination time. Furthermore, we choose to implement reliability directly in the RPC service rather than relying on the underlying transport. This combination of semantics can be realized using micro-protocols **RPCMain**, **SynchronousCall**, **ReliableCommunication**, **Acceptance(1)**, **BoundedTermination**, and **Collation(fid)**, where **fid** is a function that returns the latest response.

It is also easy to configure RPC services that realize the same set of properties as existing services. For example, [BN84] corresponds to a service configured with **RPC-Main**, **SynchronousCall**, **Acceptance(1)**, **Collation(id)**, **ReliableCommunication**, and **UniqueExecution**. Rajdoot [PS88] corresponds to the same set, plus **BoundedTermination** and **TerminateOrphans**. Among group RPC services, the one-to-many RPC described in [Coo85] corresponds to the set **RPCMain**, **SynchronousCall**, **Acceptance(N)**, **Collation** with a function consisting of identity and comparison to detect inconsistencies at the server processes, **ReliableCommunication**, **UniqueExecution**, and **TotalOrder**. As an example of a very simple group RPC, *lookup RPC* [WZZ93] corresponds to **RPCMain**, **SynchronousCall**, **Acceptance(1)**, and **Collation(fid)**.

Finally, note that the configuration graph does not map directly to the dependency graph of the properties given in Figure 7.1. One cause for differences are the typical transformations of dependencies into inclusion relations. Another difference is that Figure 7.8 contains extra dependencies that simplify the implementation rather than being inherent to the properties themselves. For example, there is an edge from **TotalOrder** to **UniqueExecution** since our implementation of **TotalOrder** assumes that any request is received at the server only once. Furthermore, we introduce an additional micro-protocol **SerialOrder** that, although not required, makes it easier to implement **AtomicExecution**. Finally, **BoundedTermination** conflicts with the ordering micro-protocols because it operates by terminating an unfinished call when the timelimit is reached. This causes **ReliableCommunication** to stop retransmitting the call, which means that a server may never receive the call. Note that this conflict does not exist between the corresponding properties, since returning the call to the user as unsuccessful does not require that retransmission of the call to the server be stopped.

7.5 Conclusions

This chapter has illustrated our approach to constructing configurable services in the context of a group RPC service. The resulting service can provide hundreds of different configurations to match different application requirements, and in particular, can be configured to match the properties of many existing RPC services. Although, the presentation was not as detailed as in chapters 4 and 5, this chapter shows again the general design steps and application of the event-driven execution model described in chapter 3.

Other researchers have also proposed modular implementations of RPC. For example, in [HPOA89] a modular implementation of RPC service based on the *x*-kernel is described. In contrast with our emphasis on configurability and modularization based on abstract properties, however, that paper describes a modularization of an RPC service implementing one chosen semantics where the modules are syntactic components rather than implementations of abstract properties. The work on an agent-synthesis system for Cross-RPC communication in [HR94] is relatively closely related to our goals. Although its primary goal is to allow heterogeneous RPC systems to communicate with one another, the system also offers the possibility for designing and prototyping new variants of RPC. Specifically, the authors divide RPC semantics into three components: call semantics (synchronous versus asynchronous), failure semantics, and RPC topology (one server versus multicast RPC). An RPC agent is synthesized from a specification written in Cicero, an event-driven specification language. Our approach to building composite protocols from micro-protocols provides more structuring support, however, and promotes a style in which RPC services are configured from a collection of already-written micro-protocols rather than generated from specifications.

CHAPTER 8

ADAPTIVE SYSTEMS

Configurability can be used to match service properties to application requirements, as was done in chapter 5, or it can be used to match the implementation of a service to the underlying execution environment, both to improve performance and to allow continued operation in the face of changed conditions. The potential gains in these areas are maximized if changes can be made during system execution. In this case, we say that the system *dynamically reconfigures* or *adapts* to changes. Thus, an *adaptive computing system* is one that modifies its behavior based on changes in the environment. These changes could be, for example, changes in communication patterns, frequency, or failure rates, but also changes such as processor or link failures, or changed user requirements. Therefore, adaptive techniques are not only useful for improving performance, but can also be used as the mechanism for reacting to failures and responding to changes in user requirements.

Numerous examples of adaptive techniques can be found in existing systems. A simple example of an adaptive algorithm is the Ethernet protocol, which may increase or decrease the interval after which it tries to resend the message based on the collisions on the broadcast medium. The ability to adapt in this way reduces the number of collisions and thereby improves the overall throughput of the system. The Transmission Control Protocol (TCP) of the Internet protocol suite uses adaptive mechanisms for flow control, retransmission, and congestion control [Jac88]. Other examples include concurrency control of database transactions [BFHR90], real-time parallel systems [BS91, SBB87], operating systems [MS96], and high-speed communication protocols [SBS93]. Furthermore, as noted above, adaptive systems are important in the area of dependable computing [GGL93]. An example in this area is the SCOP (Self-Configuring Optimistic Programming) scheme [BDGX93], an adaptive version of N-version programming [Avi85]. In this scheme, multiple alternative algorithms are executed, with the actual number of alternatives used being determined at runtime based on the level of confidence desired. Adaptive algorithms have also been used to diagnose faulty processors in distributed systems [KH83, BB91, LYS93].

This chapter focuses on the adaptive aspect of configurability and use of the event-driven execution model in this context. We first describe a general model for adaptive systems. This model divides the adaptation process into three different phases—change detection, agreement, and action—that can be used as a common means for describing various algorithms. The use of the general model is then demonstrated by applying it to different adaptive situations. The examples are classified based on the reason for adaptation: performance, failure, or change of the expected failure model. Finally, we demonstrate how adaptive systems structured using the general model can be implemented using the event-driven model. Here, micro-protocols are not used to encapsulate the implementations of abstract properties, but rather to implement the different phases of the adaptive process.

8.1 A General Model for Adaptive Systems

8.1.1 Phases of Adaptation

An adaptive system built on a distributed computing platform can be modeled as responding to changes with the following three phases:

1. **Change Detection**. Monitoring for a possible change in the environment and deciding when to suspect that the change has actually occurred.

2. **Agreement**. Reaching agreement among all sites that adaptation is required.

3. **Action**. Changing the behavior of the system.

The change detection phase can take various forms depending on what type of system or change is being dealt with. For example, in a distributed system, it might involve monitoring message flow between sites or sending control/test messages to the change detection processes on other sites. The change could also be initiated by the user or the application if the service needed from the underlying system is changing.

The agreement phase is often some kind of distributed agreement algorithm. Depending on the situation, it may be a majority vote, agreement on maximum or minimum value, or something similar. In some cases, an expensive distributed agreement protocol is unnecessary. For example, if a centralized algorithm is used, only the decision of the central entity may be required. Similarly, each site can sometimes make a decision independently of others, in which case the entire agreement phase can be omitted.

The action phase can take various forms depending on the type of adaptation. Perhaps the simplest case is changing some execution parameters, such as modifying the timeout period for a network protocol. Another possibility is reassigning roles, such as changing a centralized coordinator or reassigning roles in a primary/backup fault-tolerance scheme. Slightly more involved would be taking some corrective action, such as recomputing routing tables, aborting deadlocked computations, or regenerating a lost token. In the extreme case, the action might consist of actually changing the program that provides the service.

8.1.2 Adaptation Policies

The execution of each of the three phases is governed by policies, as follows:

1. **Change Detection**.
   - *Detection Policy*: Specifies the condition under which a change is suspected.
2. **Agreement**.
   - *Response Policy*: Specifies the response of a site when the agreement starts, for example, if the site agrees or disagrees that a change has happened.
   - *Voting Policy*: Dictates under what conditions the result of the agreement is positive ("change has occurred") and negative ("change has not occurred"). If the agreement is quantitative, the voting policy also describes how to combine the responses.
3. **Action**:
   - *Action Policy*: Specifies the action to be taken as a result of the (agreed) change.
   - *Timing Policy*: Specifies when the action is to be taken.

8.1.3 Model

Putting these two aspects of adaptation together yields the general model depicted in Figure 8.1. The adaptation process sometimes needs more than one round of agreement and action phases, which is represented in the figure by the dashed arrow.

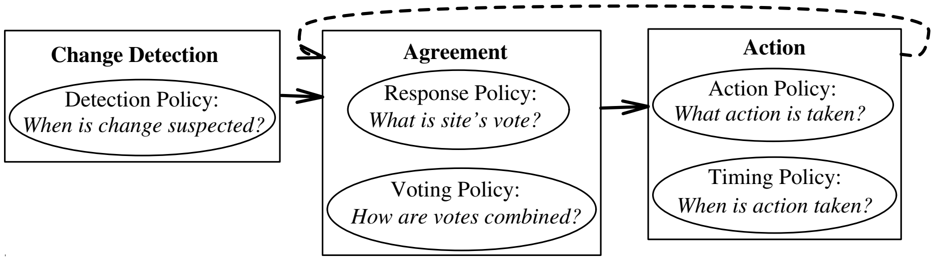


Figure 8.1: General Model of Adaptive Systems

The normal operation of the adaptive system or algorithm often continues in parallel with the change detection phase and sometimes even with the agreement phase. An adaptive system may also adapt multiple times, either because the system handles different types of changes concurrently, or because one change leads to the start of new change detection and agreement algorithm that adapts to subsequent changes. The latter case can also be used to cover adaptation back to the original algorithm if it turns out that the change in the environment was only temporary.

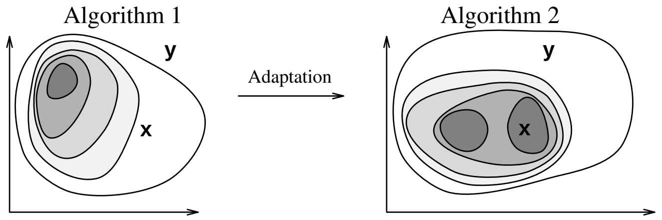


Figure 8.2: Correctness Domains and Performance Profiles

8.1.4 Correctness Domains and Performance Profiles

An adaptive system can be characterized by the *correctness domains* and *performance profiles* of the algorithms it employs during its execution. The correctness domain of an algorithm is the subset of all execution environments in which the algorithm behaves correctly, that is, provides a service according to its specification. Thus, two algorithms can provide identical service in most cases, but still have different correctness domains if one can handle situations the other cannot. A system with a smaller correctness domain is usually easier to construct and faster to operate since it can be based on more simplifying assumptions. However, since it cannot handle certain scenarios, it may become necessary to adapt the system to one that has a larger correctness domain if the execution environment happens to change significantly. Correctness domains are related to *failure models* for fault-tolerant systems, which describe the type of software or hardware failures a system is designed to tolerate.

The performance profile of an algorithm is a function that maps each point of the correctness domain to a performance metric of choice, such as throughput or response time. In this case, if two algorithms implement the same service and have the same correctness domain but their performance profiles differ, it may make sense to adapt from one to the other simply for performance reasons.

Figure 8.2 illustrates these concepts. Here, the correctness domain of each algorithm—represented by the largest enclosed figure–is characterized along two dimensions; these might be, for example, failure rate and transmission time if the algorithms are network protocols. The shading is used to represent the performance profile of each program, with the darker shades representing better performance expressed using some metric of interest. If the environment is in point marked by x, then, it might be desirable to adapt to algorithm 2 to improve the performance of the system even though both are equally correct. However, if the environment is in the point marked by y, the adaptation to algorithm 2 is required since algorithm 1 is no longer able to provide correct service.

8.2 Adapting to Improve Performance

Examples in this section show how adaptive systems can increase the efficiency or decrease the cost of running a system. All these examples can be characterized as being transitions within one correctness domain solely because of the performance profiles of the respective algorithms, as illustrated in Figure 8.2. The changes to which the adaptive system reacts here usually cause the service's performance to deteriorate without actually causing the service to stop or act incorrectly. The adaptive action is taken to attempt to restore the performance. Most of the examples of adaptive systems, protocols, and algorithms in the literature fall into this category.

8.2.1 Adaptive Timeout Protocols

Timeouts are used in distributed systems for a variety of purposes, such as deciding when to retransmit a potentially lost message or reconfigure the system to exclude a potentially failed site. In some cases, the ability to adjust timeout periods dynamically would be advantageous. For example, consider a long-haul network, where the transmission time and failure rate of any given connection may vary over time. Because of such changes in the environment, each site should be able to adjust its timeout period individually based on the particular circumstances at any given time.

This type of adaptation is straightforward to describe in the above model. Assume that sites have local clocks that have approximately the same, approximately constant rate, and that special "ping" messages are used to measure the propagation time between sites. Then the following characterizes the system:

1. **Change Detection**. Send ping messages periodically to all sites; upon reception of such a message, send a response. *Detection Policy*: Compare the time it takes for a response to arrive to the timeout periods stored; if the difference is greater than some value X, suspect change.

2. **Agreement**. Not necessary.

3. **Action**. *Action Policy*: Change the timeout period to be a weighted sum of the old timeout period and the new "ping delay." *Timing Policy*: Immediately.

8.2.2 Adaptive Reliability Protocols for Message Passing

The two most common approaches for implementing reliable message transfer in a distributed system are positive and negative acknowledgments. The positive acknowledgment scheme is based on the receiver sending an acknowledgment message immediately after receiving a message. If such an acknowledgment is not received by the sender within a specified time interval, the message is retransmitted. In contrast, the negative acknowledgment scheme is based on the receiver being able to detect a missing message using, for example, sequence numbers, and then explicitly requesting retransmission of the missing message.

An adaptive version of this type of reliability protocol can change between the two schemes based on current conditions. Since losing a message is relatively uncommon in modern networks, starting with negative acknowledgments is reasonable, assuming that sites exchange messages frequently. If the failure rate becomes unacceptable or sites begin exchanging messages less frequently, the system should adapt to using positive acknowledgments.

In designing such an adaptive reliability protocol, a number of issues must be addressed. One is the technique to use for change detection. This can be done, for example, by monitoring the average message delivery latency, defined as reception time minus sending time. This approach requires globally synchronized clocks, however. Another approach is to monitor the failure rate, i.e., how often a request for a missing message must be made when negative acknowledgments are used. In either case, the detection policy gives a threshold value defining the boundary at which the change must be made. Another issue is whether to make the adaptation on a per message or a per session basis. In the former, the sender of each individual message decides if the message is to be positively or negatively acknowledged, while in the latter, all the processes make a global decision to change from one method to another. Below, we describe both message-based and session-based approaches.

Message-based approach. Assume that the primary method is negative acknowledgments, with positive acknowledgments used only when necessary. To smooth the transition between the two schemes, all messages, including those to be positively acknowledged, carry the sequence numbers needed for negative acknowledgments. In our design, both protocols are actually run all the time, but the positive ack protocol only acts if a "positive ack bit" is set on a given message.

1. **Change Detection**. See above.

2. **Agreement**. Not necessary.

3. **Action**. *Action Policy*: Send each message using positive/negative ack scheme depending on the decision of Detection Policy. Each message carries information of what kind of scheme was used, so the receiver can behave accordingly. *Timing Policy*: Immediately.

Session-based approach. In this approach, all processes change to the new protocol at the same time. Note that this adaptive action has two separate agreement/action phases.

1. **Change Detection**. Same as message-based approach.

2. **Agreement**(1). Not necessary.

3. **Action**(1). *Timing Policy*: When change detection or agreement message arrives. *Action Policy*: Stop sending new messages.

4. **Agreement**(2). *Response Policy*: Define range around the threshold where the answer is yes; outside the range no. Send response to every site. When changing from positive acks to negative acks, send the response only when all messages transmitted by your site have been acked. When changing from negative acks to positive acks include an ack vector with the vote where the ack vector indicates which is the last message received from each site in correct order (no gaps). *Voting Policy*: Must wait for response from everyone; lots of different options for voting policies (e.g., majority vote).

5. **Action**(2). *Timing Policy*: Immediately when agreement reached. *Action Policy*: If decide to change from negative to positive acks each site checks the ack vectors it received from every other site with the vote and starts resending messages starting from the first message that somebody did not receive. Otherwise resume sending new messages using the new algorithm.

Note that both protocols operate on the same pool of messages, so that the new protocol must have access to the messages received using the old one.

8.2.3 Adaptive Concurrency Control Protocol

The problem of concurrency control in database systems is to ensure that two or more concurrent transactions do not leave the database in an inconsistent state, that is, a state that could not have been reached by executing the transactions in some serial order [BHG87]. There are numerous such concurrency control algorithms, which can be broadly classified as pessimistic versus optimistic. The pessimistic algorithms are based on preventing conflicts that can lead to inconsistencies using locking or some other technique. Optimistic algorithms, on the other hand, allow conflicts to occur, with detection and rollback occurring in a final commit step. The tradeoff, of course, is a greater degree of concurrent execution in the optimistic case versus the chance that multiple conflicting transactions will have to be aborted.

By design, optimistic algorithms work well if the system is lightly loaded whereas pessimistic ones are better when the system is heavily loaded. Therefore, it is advantageous to design an adaptive concurrency control protocol that changes between optimistic and pessimistic depending on the load. See, for example, [BFHR90] for more discussion on adaptive concurrency control.

8.3 Adapting to Failures

This section examines examples of adaptive algorithms where the change in the environment is a processor or communication link failure. Changes to be adapted to here typically cause the system to stop until an adaptation is made. This type of adaptation is fairly easy to deal with, since the only negative effect between the time the change occurs and the adaptation is made is denial of service. Examples of this type of adaptations are common in the literature, but are often not characterized as adaptive.

In terms of performance profiles and correctness domains, failures here transform the execution environment to a point outside the original correctness domain of the algorithm. An adaptive mechanism is then invoked to, in essence, expand the correctness domain by taking some corrective action. In other words, the mechanism makes it appear as if the environment is indeed contained in the correctness domain despite the failure. This case is illustrated in Figure 8.3 where the correctness domain of the original program is shown as being within the correctness domain of the program including the adaptive extension. The figure also shows the behavior that results with this type of adaptation. Specifically, if the environment spontaneously transfers to a point marked by *x* due to a failure, the adaptation mechanism compensates and expands the correctness domain of the program to encompass the new environment.

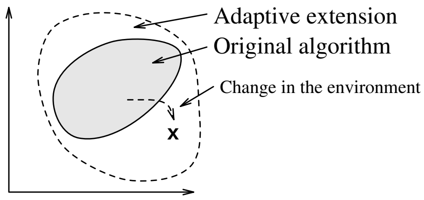


Figure 8.3: Adapting to Failure

8.3.1 Membership Service

The importance of membership service in distributed systems was extensively discussed in chapters 4 and 5, and a configurable implementation of the service was presented in chapter 5. Here, we take a different look at the membership problem, viewing it as an example of adaptive system where the changes in the environment are membership changes such as failures or recoveries. The system has to detect these changes, agree on them, and generate membership messages, a sequence that follows the general adaptive system model quite closely.

As an example, consider a membership protocol like that used in the Consul system described in sections 2.1.2.1 and 4.4.2. This algorithm is based on examining successive *waves* (i.e., levels) of the context graph of causally-ordered messages. A membership change in this scheme can be described using the general model of adaptive systems as follows.

1. **Change Detection**. *Detection Policy*: If no message has arrived from a site within a T-second interval, suspect failure and multicast "failure suspicion" message.

2. **Agreement**. *Response Policy*: If local context graph has no message from site suspected to have failed in the same wave as failure suspicion message, respond yes; otherwise, respond no. *Voting Policy*: If every site excluding those suspected to have failed respond yes, result is positive; otherwise, result is negative.

3. **Action**. *Action Policy*: Remove failed site from membership list. Trigger an event indicating membership change for all interested parties. Send a membership change message to user, if required. *Timing Policy*: Immediately after agreement is reached.

The actions taken by a membership protocol are actually just a small part of those taken by the system as a whole when a failure occurs. In fact, membership acts as detection and agreement mechanism for other protocols that are interested in membership changes. An example of such an adaptation can be seen in the protocol in Consul that implements a consistent total ordering of messages at all sites.

1. **Change Detection**. Provided by membership. *Detection Policy*: If site failure event is triggered by membership.

2. **Agreement**. Not necessary (provided by membership).

3. **Action**. *Action Policy*: Remove failed site from the membership list used to make completeness decisions. *Timing Policy*: When all waves before the one where the failure was observed have been processed.

8.3.2 Adaptive Token Passing

Token passing is a technique used in different application areas to regulate access to a shared resource. The loss of token is a change in the environment to which the system must adapt. Here, we consider the detection of token loss and subsequent regeneration of a new token as an adaptive protocol.

The detection phase can be implemented using any of several strategies. One common approach is to use timeouts, in which case a loss is suspected by a site if some period of time passes without it having received the token. Alternatively, if the token is used to regulate access to some shared resource, the resource itself can be monitored. With this scheme, a token loss is suspected if the resource remains unused for some time even though there are processes waiting access.

The following summarizes the three phases of adaptive change. The agreement phase is especially important here since the number of tokens must be kept under some specified limit (one in most cases).

1. **Change Detection**. If some period of time passes without receipt of token, suspect loss and initiate agreement algorithm.

2. **Agreement**. *Response Policy*: If site suspects token has not been lost, answer no; otherwise, answer yes. *Voting Policy*: Consensus, i.e., if all currently functioning sites agree that token is lost, decide token is lost.

3. **Action**. *Action Policy*: The leader site generates a new token. *Timing Policy*: Immediately.

Note that normal operation—the part of the program that uses the token—can be separated completely from the algorithm that realizes adaptation in the face of changes. This example is treated in more detail later in this chapter.

8.3.3 Centralized Total Ordering

A simple and straightforward way to achieve a consistent total ordering of messages in a distributed system is to use a central ordering server [KTHB89]. In this scheme, messages are first sent directly to the ordering server, which then retransmits them in some total order to all receivers. Alternatively, messages can be multicast directly to the receivers, with the central ordering service only sending ordering messages [AFM92]. Like any centralized service, a total ordering service of this type suffers from the problem of how to handle the failure of the central authority.

This protocol can be structured as a fault-tolerant adaptive system, where the regular total ordering algorithm implements the ordering assuming the central server remains functioning, and the adaptive portion takes care of changing the server when necessary. Without loss of generality, assume that the scheme being used has the central server sending only ordering messages. Assume further that an ordering message is the pair $<$message id,total order number$>$ and that a negative acknowledgment reliability protocol is used for the application messages. The latter ensures that, if any non-empty subset of the sites receives the message, every site will eventually receive the message unless all sites in the subset fail. Finally, assume that a membership service provides notification of membership changes.

Given these assumptions, then, the three phases are as follows.

1. **Change Detection**. Provided by membership. *Detection Policy*: Site failure event triggered by membership.

2. **Agreement**. *Response Policy*: Send the total order number of the latest totally-ordered message at this site. *Voting Policy*: Site sending maximum total order number is elected as new central ordering server, with unique site id used as tie breaker. Must wait for vote from everybody assumed alive.

3. **Action**. *Action Policy*: New central ordering server assumes the role of the central server and sends ordering message starting from the smallest total ordering number received in the agreement phase. Other sites start expecting ordering messages from this new site. *Timing Policy*: Immediately after decision reached.

8.4 Adapting to Change of Failure Model

This section examines adaptive systems where the system actually adapts its actions from operating in one correctness domain to another. Fault-tolerant systems are usually designed to tolerate failures that conform to a particular failure model, as was described in chapter 1. Some of these models, such as fail-stop and omission, are easier to deal with from the perspective of the system designer, but represent a risk since a failure outside the model can lead to unpredictable results. On the other hand, failures in the more inclusive failure models, such as Byzantine, are rare and require expensive protocols. As a result, it would be advantageous to design an adaptive system that initially assumes a benign failure model, but is prepared to change to a less benign one should the environment no longer match the original assumption. This type of adaptation is a special case of configurability where the user reliability requirements are matched by choosing an algorithm with the required failure model, as discussed in chapter 1.

Figure 8.4 illustrates this type of adaptation. Here, two algorithms 1 and 2 are designed for failure models C1 and C2, respectively. If the environment changes to point *x* outside the correctness domain of algorithm 1, an adaptive action is taken to replace 1 by 2.

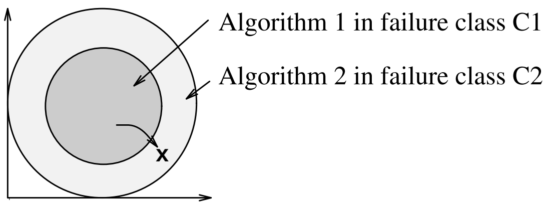


Figure 8.4: Adapting to Change of Failure Model

These types of changes do not typically stop the system execution, but the system may operate incorrectly, possibly corrupting the system state until an adaptation is made. Therefore, reestablishing a correct state is a problem. For example, if the system implements total ordering of messages assuming a fail-stop failure model and a different type of failure occurs, several messages may be delivered out of order before the change is detected and corrective action taken. A well-known, but expensive, solution is to write the system state periodically onto stable storage and then use these checkpoints to roll back the system to an earlier uncorrupted state should such a failure occur.

Note that this approach is only practical if detecting the change of failure model is cheaper than tolerating the more difficult failure model for the whole duration of the system execution.

8.4.1 Point-to-Point Data Transmission Protocol

As a simple example, consider a point-to-point data transmission protocol that assumes no transmission failures (i.e., failure model *none*), combined with an adaptive portion that detects and adapts to possible transmission failures.

1. **Change Detection.** Add sequence numbers to all messages. *Detection Policy*: Monitor reception of messages. Since the channel need not be FIFO to be reliable, receiving messages out of order is not a reason to suspect change. The detection policy could, however, be based on assuming bounded FIFO-ness—messages cannot arrive more than R messages out of order—or bounded transmission time—messages cannot arrive more than T time units after any message that follows it in the order.

2. **Agreement**. Not necessary.

3. **Action**. *Action Policy*: Start negative ack protocol starting from the missing message(s). *Timing Policy*: Immediately.

Note that the solution is not much cheaper than running negative acks to begin with. One advantage, however, is that it reduces the number of unnecessary nacks and retransmissions.

8.4.2 Synchronous Atomic Broadcast

In [CASD85], a set of broadcast protocols for different failure models is described. The assumption is that the underlying network is *synchronous*—that is, that messages are delivered within some time bound—and that each pair of sites is connected by at least k disjoint paths, where k is the number of network failures to be tolerated. These broadcast protocols guarantee a consistent total ordering. as well as atomic delivery of messages to all sites.

The network synchrony and k-connectivity assumptions are difficult to guarantee in practice. As a result, even though a network may satisfy these assumptions with a high probability, there is always some small chance of the assumptions being violated. This can cause one or more sites to enter an inconsistent state relative to other sites, and raises the possibility of *system contamination* should a site in an inconsistent state send a message [GT91].

Adaptivity can be used to address such a scenario and increase the overall dependability by allowing the system to continue providing message ordering despite the change in the environment. To deal with the loss of network synchrony, the system can change to using a total ordering algorithm intended for asynchronous networks, such as one that uses a centralized ordering server. Similarly, to deal with loss of k-connectivity, the system can change to using a reliability protocol for asynchronous networks, such as the use of negative acknowledgments.

Detection of such changes can be implemented using a scheme similar to the one presented in [GT91]. Assume the sender of a message, say site p, includes the sequence of messages delivered to the user at site p, DEL_p, in the header of each message sent.

Now, a site q that receives this message can check its DEL_q against DEL_p. When it comes time to deliver the message from p, DEL_p must be a prefix of DEL_q. If this is not the case, the total order must have been violated. This basic idea can be optimized to cut down the overhead on each message either by using the approach taken in [GT91] of sending message counts instead of the complete history, or by using the fact that stable messages—that is, messages received by every site—can be removed from the message delivery history. With either of these approaches, the space overhead can be cut on average to $O(n)$ per message.

An agreement phase is required for establishing the last message that has been received by every site before the ordering failed. Since the synchronous protocols can no longer be trusted, the agreement must be done using an asynchronous reliability protocol. Agreement messages need not be totally ordered, however, so the central ordering server is not required for this phase.

This series of actions fits within the generalized adaptive system model. One action, changing the protocol, is actually done before the agreement phase. The sequence of phases is therefore: detection, agreement (nil), action (change protocol), agreement (agree on last properly ordered message), action (rollback/recovery). The first agreement is nil since sites are assumed to suffer only crash failures: if one site detects that the total order is corrupted, other sites can trust its judgment.

Several approaches can be taken to deal with sites whose state becomes inconsistent between the occurrence of the change and the corrective action. One is to force such sites to fail and then recover, building their new state from a state transferred from some other site. Another is to use the checkpoint technique outlined above; if the checkpointing and change detection algorithms are coordinated appropriately, at most two checkpoints per site are required.

8.5 Implementation Based on the Event-Driven Model

The event-driven execution model described in chapter 3 has a number of characteristics that make it especially appropriate for implementing adaptive systems based on this general model. For example, the three distinct phases can often be implemented as three separate micro-protocols, with the normal behavior of the system also realized by one or more micro-protocols. Since micro-protocols are designed for configurability, the normal behavior of the system can easily be combined with the micro-protocols required by the adaptation mechanisms.

Events and shared data structures are also a natural way to implement interactions between phases. That is, the change detection phase can trigger an event, say SUSPECT_CHANGE_X, that starts agreement, and agreement can trigger an event, say CHANGE_X, that starts the action phase. Sometimes the interaction and synchronization requirements between phases are more complex; for example, the different phases may need to access a shared data structure. Such requirements are also easy to accommodate in this model using the shared data structures.

Finally, the event-driven model makes it easy to implement different types of adaptive actions. The simplest actions only involve changing execution parameters such as timeout periods or membership, which can be done by updating shared variables within the composite protocol. Some actions require a more sophisticated corrective step such as recreating a token or reassigning the roles of different sites. These steps can be encapsulated in the action phase micro-protocol. The most complicated actions may even require changing the algorithms executed. The model supports this by allowing handler bindings to be changed at runtime.

```
const HIGH = 1; MEDIUM = 10; LOW = 100; % priorities
var   have_token: boolean;
      next_site: site_identifier;

micro-protocol TokenControl() {

   event handler handle_net_msg(var msg: NetMsgType) {
      var new_msg = new(UserMsgType);

      if msg.type = TOKEN then {
         new_msg.type = TOKEN;
         User.Pop(new_msg);
      }
   }
   event handler handle_user_msg(var msg: UserMsgType) {
      var new_msg = new(NetMsgType);

      if msg.type = TOKEN and have_token then {
         new_msg.type = TOKEN;
         Network.Send(new_msg,next_site);
         have_token = false;
      }
   }
   initial {
      register(MSG_FROM_NET,handle_net_msg,LOW);
      register(MSG_FROM_USER,handle_user_msg,LOW); }
}
```

Figure 8.5: **TokenControl** Micro-Protocol

As an example of mapping an adaptive system onto this implementation model, consider the token passing example described in section 8.3.2. In this scheme, a distributed application program uses a token to implement access to a shared resource, with adaptation in the event of token loss being implemented transparently with separate micro-protocols. The token passing is based on a logical ring where each application process knows the identity of its successor in the ring. We assume a point to point communication network accessed using routine `Network.Send`, and that communication is unreliable, but with FIFO ordering guaranteed between pairs of processes. We also assume that the sites involved with the token passing are known in advance, and that if one fails, it either recovers

```
micro-protocol TokenLossDetection(timeout_period: real) {

   event handler handle_msg(var msg: NetMsgType) {
      if msg.type = TOKEN then {
         deregister(TIMEOUT,monitor);
         have_token = true;
         register(TIMEOUT,monitor,timeout_period);
      }
   }
   event handler monitor() {
      trigger(SUSPECT_TOKEN_LOSS);
   }
   initial {
      register(MSG_FROM_NET,handle_msg,HIGH);
      register(TIMEOUT,monitor,timeout_period); }
}
```

Figure 8.6: **TokenLossDetection** Micro-Protocol

quickly or is replaced by a new site with the same address. All these assumptions are not strictly necessary, but simplify the code so that emphasis can be placed on the structuring principles.

Note that this problem is very closely related to the problem of token regeneration in the **TokenDriver** micro-protocol in chapter 5. The main difference here is that the problem is structured using the general model for adaptive systems and decomposed into a number of micro-protocols instead of being implemented in one micro-protocol as was the case with **TokenDriver**. This decomposition makes each part correspondingly simpler. Naturally, a direct comparison is not possible because both the assumptions and the actual problem here are much simplified.

Figure 8.5 gives an outline of the micro-protocol that controls the passing of the token and implements interaction with the application. The application is notified of token acquisition by passing it a TOKEN message using the `User.Pop` routine. Similarly, the token is released and passed to the next site when a TOKEN message arrives from the application, as signaled by the occurrence of the MSG_FROM_USER event.

The change detection phase shown in Figure 8.6 is based on simply monitoring that the token is received once every specified time period. If the token is not received within this period, its loss is suspected and the event SUSPECT_TOKEN_LOSS is triggered. The micro-protocol detects arriving messages by registering an event handler for the event MSG_FROM_NET, which is triggered by the framework when a message is delivered to the composite protocol by the network.

Figure 8.7 shows the outline of the token loss agreement and action micro-protocols. The agreement portion is based on sending a VOTE message along the same logical ring used to transmit the token. If some site receives the VOTE message while holding the token, it simply drops the VOTE message. If, on the other hand, the sender of the VOTE

```
micro-protocol TokenLossAgreement() {
    var vote_id: int (mod m);    % id of the latest vote started
        mutex: semaphore;

    event handler start_vote() {
        var msg = new(NetMsgType);

        P(mutex);
        if not have_token then {
            msg.type = VOTE;
            msg.sender = my_id;
            vote_id++;
            msg.id = vote_id;
            Network.Send(msg,next_site);
        }
        V(mutex);
    }
    event handler handle_vote(var msg: NetMsgType) {
        P(mutex);
        if msg.type = VOTE then {
            if msg.sender = my_id and msg.id = vote_id then
                trigger(TOKEN_LOST);
            elseif not have_token then
                Network.Send(msg,next_site);
        } elseif msg.type = TOKEN then
            vote_id++;
        V(mutex);
    }
    initial {
        register(SUSPECT_TOKEN_LOSS,start_vote,LOW);
        register(MSG_FROM_NET,handle_vote,MEDIUM);
        vote_id = 1; }
}

micro-protocol TokenLossAction() {

    event handler recreate_token() {
        var msg = new(NetMsgType);

        msg.type = TOKEN;
        Network.Send(msg,next_site);
    }
    initial {
        register(TOKEN_LOST,recreate_token,LOW); }
}
```

Figure 8.7: Agreement and Action Micro-Protocols

message receives that message back before receiving the token, it is assumed that the token has been lost and the event TOKEN_LOST is triggered. Note that the change detection micro-protocol can trigger SUSPECT_TOKEN_LOSS even if agreement is in progress. In fact, this strategy is used to deal with transmission failures during the voting process.

Essentially, a new VOTE message is sent every time the event is triggered, with agreement only reached if such a message reaches the sender before the time arrives to send another. This assumption is reasonable since the timeout period of the change detection micro-protocol would typically be much greater than the time it takes to circulate either the token or the VOTE message.

The action micro-protocol is simple in this case. All that is required is to field the TOKEN_LOST event and then regenerate the token.

Finally, note that agreement is the most complicated part of this example despite the relative simplicity of the application. One way to address this problem is to decompose the agreement phase further into a voting mechanism—based on logical ring in this case—and response and voting policies. Such a structure also enhances configurability by allowing, for example, the same voting mechanism to be used for different policies without modification.

8.6 Conclusions

This chapter has illustrated a special case of customization, namely adaptation, where the system changes occur at runtime instead of compile or link time, and how the event-driven model can be applied in this case. We introduced a general model for adaptive systems and presented examples of how this model can be applied in different scenarios that arise in the context of distributed systems. Based on these examples, our tentative conclusion is that the model has the potential to simplify the design and construction of a wide variety of adaptive systems. It provides a unifying framework for discussing various attributes of such systems, as well as suggesting new strategies to be pursued. It also illustrates a different way of using micro-protocols. As for cost, analytical and simulation studies are needed for any definitive answer, but our initial investigations suggest a considerable variation based on the type of adaptation considered. Most promising are adaptations to enhance the performance of the system in the face of changes in the computational environment, with those done to extend the system's failure coverage to deal with site or network failures close behind. The ability to adapt from tolerating one failure model to another is also intriguing, but more speculative at this point.

CHAPTER 9
CONCLUSION

9.1 Summary

This dissertation has addressed the problem of providing customizable services for fault-tolerant distributed applications. Customizing underlying services, such as membership, multicast, or RPC, is important because no single implementation of a service is a perfect match for every application; either the service implementation provides guarantees that are too weak to ensure correctness, or it provides guarantees that are too strong, thereby slowing execution of the application unnecessarily. Furthermore, the chosen guarantees may have different implementations that perform better in different execution environments.

Prior work on configurability and extensibility, both in the context of fault-tolerant systems and other system-level services, was described in chapter 2. We reviewed a number of systems and showed that the approaches taken by existing systems can be typically be characterized as hierarchical or function-based. Both approaches have their limitations that our work attempts to eliminate.

In chapter 3, we presented our approach to constructing configurable services. Our goal is to provide support for flexible configuration, where the units of configuration correspond to the abstract properties of the service. Thus, the design always starts from identifying the properties. We introduced a new approach to specifying abstract properties that allows all message delivery orders that satisfy required properties of a service to be represented and accommodates graphical illustrations of properties. We also identified and specified three relations between properties—dependency, conflict, and independence—that dictate which combinations of properties are feasible in any service implementation.

The implementation of configurable services built using our approach is based on an event-driven execution model that was also described in chapter 3. In this model, properties are implemented as micro-protocols, which interact through events and shared data structures. These mechanisms provide a degree of indirection that makes it possible to construct highly configurable services. Next, the event-driven model as well as the design steps involved in constructing a configurable service were outlined. Analogously to the relations between properties, we can identify relations between micro-protocols that dictate which combinations of micro-protocols work correctly. Four relations—dependency, inclusion, conflict, and independence—were specified and illustrated using configuration graphs, which are graphical representations corresponding to dependency graphs. Finally, three prototype implementations of the event-driven model were described.

Chapters 4 and 5 illustrated the application of the approach to membership services. In chapter 4, message ordering graphs were used to specify 23 properties of membership services, and dependency graphs that represent the relations between the properties were given. The properties include change detection, agreement, ordering, synchrony, and partition handling properties. The specified properties have thousands of different feasible combinations, which illustrates the great potential for customization. A number of the relations between membership properties were proven to illustrate our proof techniques. Finally, a collection of existing membership services were characterized in terms of the properties specified. In chapter 5, an implementation of a configurable membership service was presented. This implementation is based on the event-driven execution model and is extremely configurable, providing over 1000 legal configurations. The costs of various properties were calculated in terms of number of messages required. The chapter concluded with a discussion about the implementation of the membership service using the C++ prototype.

A problem closely related to membership, system diagnosis, was introduced in chapter 6. Despite their close resemblance, these two problems have been almost always treated separately. In this chapter, we compared the problems and concluded that they are closely enough related to be considered specializations of the same general change detection and reporting problem. Using observations about the differences, we transformed a number of system diagnosis algorithms into typical membership algorithms, resulting in a simple but powerful membership algorithm. A similar transformation in the other direction was used to derive a new system diagnosis algorithm and to augment an existing one with membership-type properties.

Our approach was applied to group RPC services in chapter 7. A number of properties were defined and a configurable implementation presented.

In chapter 8 the flexibility of the event-driven model was illustrated by applying it to adaptive systems. We developed a general model for adaptive systems that divided the adaptation process into three different phases: change detection, agreement, and action. This model was applied to a large number of examples including some not usually considered as adaptive systems. An implementation approach was then presented using the event-driven execution model. Here, micro-protocols were used to encapsulate phases in adaptive process instead of implementations of abstract properties. This demonstrated the versatility and flexibility of the model.

9.2 Future Work

This work can be expanded in many different directions. These include applying the approach to other fault-tolerant distributed services, adding real-time constraints to the model allowing implementation of timeliness properties, expanding the work on adaptive systems, more rigorously treating correctness issues, expanding the prototype implementations, and finding commercial applications.

Although our approach has been successfully applied to membership and RPC services, the work could be expanded to many other interesting fault-tolerant services. For example, the work on multicast services in [HS93, GBB+95] could be expanded. In particular, the properties of multicast services can be specified using message ordering graphs as done in chapter 4 for membership services. Another potential application of the model was mentioned in chapter 7, where we proposed a configurable change detection and reporting service, which is a generalization of membership and system diagnosis services. Furthermore, transactions with their ACID properties have been an example that we have frequently used, but have not yet properly studied. The choice here is not only which of the ACID properties the application requires, but also which algorithms are used to implement each property. The applicability of the approach could also be explored in other service areas, such as time and synchronization services, file systems, and perhaps even operating systems.

An important topic that we have not yet addressed is real time. To guarantee real-time or any timeliness properties, changes are required in the model itself and its implementation, such as scheduling of eligible event handlers using deadlines. The model has features that should make it extendable in this direction. For example, event handlers are typically small, and therefore their execution time is predictable in most cases. Naturally, issues such as synchronization and priority inversion have to be addressed. Related to timeliness are issues about the system model, such as whether the system is synchronous, asynchronous, or something in between, such as timed-asynchronous [Cri96].

The work on adaptive systems presented in chapter 8 is just a beginning. The general model and the implementation using the event-driven approach appear promising, but we have to apply the ideas to real implementations to gain a better understanding of the performance tradeoffs. The addition of real-time considerations to the adaptation process is the next major conceptual issue. An interesting extension of current work would also be combining the property-based design of chapter 5 with the adaptive mechanisms. In this case each property would be realized by a set of micro-protocols, each implementing a different phase of adaptation. The potential benefits would include not only improved performance, but also simplified design of complex micro-protocols.

There are several fundamental issues still to be explored in conjunction with our approach. In particular, the correctness of a configurable service is an important but difficult problem. The two major issues here are formal verification and testing. Formal verification is required to prove that each micro-protocol implements the specified property and that combinations of micro-protocols can cooperate without inadvertent interference. Compared to proving correctness of monolithic non-configurable programs, our approach has the advantage that micro-protocols are typically relatively small modules, but has the disadvantage that there are potentially thousands of different combinations. Testing is similarly complicated by the number of configurations; testing any one configuration is similar to testing a non-configurable program, but testing all of the potentially thousands of configurations is often not feasible.

To date, three different prototype implementations of the event-driven model have been completed. The SR prototype that provided the first experimentation environment is not currently used or maintained, but the other two prototypes are both in active use. The independent development of these two prototypes allowed experimentation with different tradeoffs in the model. After we gain further experience with these systems, an important aspect of future work is to compare results and potentially combine the best features into a single implementation. The next major step for the C++ prototype is a distributed implementation, either based on UDP or distributed computing environments such as DCE [RKF93] or Corba [OMG95a, OMG95b]. The next major step for the *x*-kernel prototype is to port it to a version of the Mach operating system with features for supporting real time [TNR90, TN91]; this would make it feasible to deal with timeliness properties. Porting this prototype to the Scout operating system [MMO+94a, MP96] might further improve performance over the current Mach implementation.

Although many experimental projects have successfully applied configurability in areas such as operating systems and networking, the idea have not yet gained widespread commercial acceptance. As a result, it would be interesting to find commercial applications where our approach can be shown to result in financial benefits. Compared to projects in operating systems and networking, our approach is more easily applied since the event-driven model can be used at the user level and does not require changes to the operating system or other lower-level services. Furthermore, the model is not restricted to middleware and could be used to construct configurable application software.

Perhaps the most commercially appealing application of configurability would be in the creation of customized products and services. In this case, configurability is only used to facilitate the creation of a customized version of the product for each client, rather than making the configurability itself available to clients. Consider, for example, a company that exports wireless communication systems to a number of different countries. Different countries typically have their own requirements and methods for billing, security, wire tapping, and other additional features. Instead of providing all possible options in one monolithic implementation, the different methods could be implemented as configurable micro-protocols. Naturally, it is impossible to predict all requirements, but services constructed using the event-driven model are easier to extend than monolithic implementations. A similar approach could be taken in embedded systems, where the limited resources of the system could be best utilized by including only the features required for each client.

APPENDIX A

C++ PROTOTYPE IMPLEMENTATION

This appendix describes the key concepts of the C++ prototype and demonstrates how C++ classes can be used to derive specific micro-protocols and composite protocols. However, the prototype does not fully correspond to the event-driven model as described in chapter 3. In particular, some of the latest changes to the abstract model have not yet been implemented in the prototype and a few simplifying assumptions were made.

A.1 Event Handlers

Events handlers are implemented as type `event_handler`, which is a function pointer that takes two character pointer arguments:

```
typedef void (*event_handler)(char *,char *);
```

The first is required to accommodate C++: when a C++ class method is called, the first argument on the stack is expected to be a pointer to the object (*this*). Since the event handlers are called as if they were regular functions, the first argument is used to pass the required object pointer. The second character pointer is used to pass the real arguments to the event handler.

This type definition allows ordinary object methods to be used as event handlers, as illustrated below for an arbitrary `PropertyX` micro-protocol and `handle_eventX` event handler.

```
void PropertyX::handle_eventX(char *args, char *nothing)
{
    ApplMessage *msg; int *site;

    msg  = (ApplMessage *)((PtrPair *)args)->i1;
    site = (int *)((PtrPair *)args)->i2;
}
```

Note that the handler receives the actual arguments in the first argument `args`, since C++ has already stripped out the object pointer. However, the handler has to have two arguments to compile, so a second null argument is used. If a handler requires more than one argument, they are stored in a structure. For example, `args` in this case contains pointers to two arguments, one for a message and the other for a site identifier.

A.2 Event Operations

The event operations for handling an event EVENT_X are illustrated below.

```
cp->Register(EVENT_X,(event_handler *)&(handle_eventX),\
             100,(char *)this);

cp->Deregister(EVENT_X,(event_handler *)&(handle_eventX);

info.i1 = (int *)&msg;
info.i2 = (int *)&site;
cp->Trigger(EVENT_X,(char *)&info);

cp->CancelEvent();
```

In this example, `cp` is a pointer to the composite protocol of the service in question. For the `Register` and `Deregister` operations, the C++ class methods have to be cast into function pointers. Furthermore, the `Register` operation requires the priority argument, in this case `100`, and pointer to the object `this` that enables the runtime system to set the stack correctly when the handlers are invoked. The complexity of these operations could be simplified using a simple macro for both `Register` and `Deregister`. Finally, the example illustrates how several arguments are packed for the `Trigger` operation.

A.3 Micro-Protocols

A.3.1 MicroProtocol Class

The `MicroProtocol` class is the base class for all micro-protocols. Although the class, presented below, is trivial, it enables all micro-protocols to be handled in a uniform fashion.

```
/**************************************************************/
/* FILE : MicroProtocol.h                                     */
/* CLASS: MicroProtocol                                       */
/**************************************************************/
class CompositeProtocol;

class MicroProtocol {
   protected:
      int MyId;          /* unique identifier                 */
      int TraceId;       /* bit mask for tracing the          */
                         /* the execution of this microp      */
   public:
      char *ident;       /* symbolic name of the microp       */

      MicroProtocol(){                /* creation             */
         ident = "MicroProtocol";     /* default identity     */
      };
      ~MicroProtocol();               /* destruction          */
};
```

A.3.2 Custom Micro-Protocols

Specific micro-protocols are derived from `MicroProtocol`, as illustrated below for `TotalOrder`. Here, `cp` is a pointer to a membership composite protocol type `MEM`.

```
/*********************************************************/
/* FILE : TotalOrder.h                                   */
/* CLASS: TotalOrder                                     */
/*********************************************************/
#include "MEM.h"

class TotalOrder: public MicroProtocol {
   private:
      void handle_membmsg(char *,char *);  /* event handler  */

      int previous_mid;      /* previous msg in total order   */
      MEM *cp;               /* pointer to composite protocol */

   public:
      TotalOrder(int, MEM *);
      ~TotalOrder();
};

/*********************************************************/
/* FILE: TotalOrder.cc                                   */
/* CLASS: TotalOrder                                     */
/*********************************************************/
#include "MEM.h"
#include "TotalOrder.h"

void TotalOrder::handle_membmsg(char *args, char *garbage)
{
    ApplMessage *msg;
    PtrPair *info;

    info = (PtrPair *)args;
    msg = (ApplMessage *)info->i1;
    ... code omitted ...
}

TotalOrder::TotalOrder(int id, MEM *comp)
{
   cp = comp;
   TraceId = id;
   MyId = cp->SiteId;
   ident = "TotalOrder";

   cp->Register(MEMBER_MSG,(event_handler *)&(handle_membmsg),\
                100,(char *)this);
}

TotalOrder::~TotalOrder()
{ cp->Deregister(MEMBER_MSG,(event_handler *)&(memb_msg)); }
```

A.4 Composite Protocols

`CompositeProtocol` is the base class for all composite protocols. It implements the event-handling operations `Register`, `Deregister`, `Trigger`, and `CancelEvent`, as well as operations for interfacing with other (composite) protocols above and below: `Push` and `Pop`. The prototype does not currently implement the `Define` operation, so events are specified simply by defining their names as C constants as follows.

```
#define TIMEOUT       0
#define MSG_FROM_NET  1
#define MSG_FROM_USER 2
      ...
```

A.4.1 CompositeProtocol Class

The following code segment lists the type definitions and operations of `CompositeProtocol`. Event handlers registered for events are stored in data structures `TimeoutList` (TIMEOUT event) and `Events` (all other events). For simplicity, all data structures are of fixed size. Note also that the information about each handler contains a pointer to the object that contains the handler (`context`).

```
/*****************************************************************/
/* FILE   : CompositeProtocol.h                                  */
/* CLASS  : CompositeProtocol                                    */
/* - operations: Register, Deregister, Trigger, CancelEvent      */
/*****************************************************************/
#include <std.h>
#include <thread.h>
#include "types.h"
#include "User.h"
#include "MicroProtocol.h"
#include "Mutex.h"
#include "StableStore.h"
#define NUM_HANDLER 20       /* max # of handlers/event          */

class Network;

typedef struct {             /* entry for one handler, includes  */
   int priority;                   /* priority                   */
   event_handler *function;        /* handler function           */
   char *context;                  /* object pointer (this)      */
} Entry;

typedef struct {            /* entry for one event, includes     */
   int count;                   /* how many handlers registered  */
   Entry Handler[NUM_HANDLER];  /* ordered handler entries       */
} Pair;

typedef struct {             /* temp. storage for handler        */
   event_handler *function;       /* handler function            */
   char *context;                 /* object pointer              */
} TempEntry;

typedef struct {             /* entry for one TIMEOUT handler    */
   int delay;                     /* delay from previous         */
   event_handler *function;
   char *context;
} TimeoutEntry;

typedef struct {             /* handlers for TIMEOUT event       */
   int count;                          /* how many               */
   TimeoutEntry Handler[NUM_HANDLER];  /* ordered list           */
```

```
} Timeout_Record;

class CompositeProtocol {
 protected:
   short site_status;          /* FAILED or OK                    */
   Pair Events[NUM_EVENT];     /* registered event handlers       */
   Mutex *TableMutex;          /* control access to Events        */
   Timeout_Record TimeoutList; /* registered TIMEOUT handlers     */
   Mutex *fw_mutex;            /* control access to TimeoutList   */
   Network *myNet;             /* pointer to network protocol     */

   void Sleep_Guardian(void *); /* functions for handling timeout*/
   void Start_Guardian();       /* events                        */
   void Handle_Timeout(event_handler *,int,char *);
   void Cancel_Timeout(event_handler *);
 public:
   User *myUser;               /* pointer to user protocol        */
   int SiteId;                 /* id of the simulated site        */
   CompositeProtocol();
   ~CompositeProtocol();
   void Status();              /* information about handlers      */

   void Register(int,event_handler *,int, char *);
   void Deregister(int, event_handler *);
   void Trigger(int,char *);
   void CancelEvent();
   void push(ApplMessage *);
   void pop(NetMessage *);

   void Start();               /* operations for starting up,     */
   virtual void ShutDown();    /* terminating, and                */
   virtual void ReStart();     /* restarting a composite protocol*/
};
```

Some of the more important operations of `CompositeProtocol.cc` are presented below, although the complete code is not presented here for brevity. The `Register` operation takes the event name (`key`), event handler, priority, and pointer to the object and stores this information either in the `Events` table or in the `TimeoutList` using function `Handle_Timeout`.

```
void CompositeProtocol::Register(int key, event_handler *function,\
                                 int priority, char *context) {
  int next,i;

  if (key < 0 || key >= NUM_EVENT) {
     printf("ERROR: CompositeProtocol Register:");
     printf(" illegal event %d.\n",key);
     exit(-1);
  }

  if (site_status == FAILED) return;

  if (key == TIMEOUT) {
     Handle_Timeout(function,priority,context);
     return;
  }

  TableMutex->Lock();

  if (Events[key].count > (NUM_HANDLER-1)) {
```

```
    printf("ERROR: CompositeProtocol Register:");
    printf(" too many handlers for event %d.\n",key);
    exit(-1);
  }

  next = 0;
  while(next < Events[key].count &&
        Events[key].Handler[next].priority < priority)
    next++;

  for(i=Events[key].count;i>next;i--)
    Events[key].Handler[i] = Events[key].Handler[i-1];

  Events[key].Handler[next].function = function;
  Events[key].Handler[next].priority = priority;
  Events[key].Handler[next].context  = context;
  Events[key].count++;
  TableMutex->Unlock();
}
```

`Trigger` implements triggering of sequential blocking events. Note that `Trigger` first copies the event handlers to be invoked into a temporary data structure. This makes it possible to release the lock on the `Events` table, which in turn allows the handlers to invoke other operations such as `Register` or `Trigger` without deadlock.

```
void CompositeProtocol::Trigger(int key, char *args) {
  int i;
  int prio,count;
  TempEntry Handlers[NUM_HANDLER];
  event_handler ev;

  if (key < 0 || key >= NUM_EVENT) {
    printf("ERROR: CompositeProtocol Trigger:");
    printf(" illegal event %d.\n",key);
    exit(-1);
  }

  if (site_status == FAILED) return;

  TableMutex->Lock();

  count = Events[key].count;
  for(i = 0; i < count; i++) {
    Handlers[i].function = Events[key].Handler[i].function;
    Handlers[i].context  = Events[key].Handler[i].context;
  }

  TableMutex->Unlock();

  for(i = 0; i < count; i++) {
    ev = (event_handler)Handlers[i].function;
    if (site_status != FAILED) ev(Handlers[i].context,args);
  }

  thr_setprio(thr_self(),prio);
}
```

`CancelEvent` currently terminates the thread that executes the operation. So far, this simplified implementation has been adequate, but the operation should be refined to only terminate the execution of event handlers and return control to caller.

```
void CompositeProtocol::CancelEvent() {
   thr_exit(NULL);
}
```

A.4.2 Custom Composite Protocols

Customized composite protocols are created as derived classes of `CompositeProtocol` by defining custom shared data structures, procedures, events, and micro-protocols. This is illustrated below for the membership service prototype described in chapter 5.

```
/**************************************************************/
/* FILE: MEM.h                                                */
/* Defines Membership service composite protocol.             */
/**************************************************************/
#include "CompositeProtocol.h"
#include "OrderingGraph.h"
#include "Network.h"

class MEM: public CompositeProtocol {
 private:
   MicroProtocol *MP[30];           /* pointers to activated mps */
 public:
   User          *Application;
   Network       *Net;
   int           Membership[NUMSITE];
   int           ParList[NUMSITE];
   TokenType     token;
   int           SuspectList[NUMSITE];
   ... other variables omitted ....

   void          Start();           /* start membership service  */
   void          ShutDown();        /* shutdown membership serv. */
   ... other procedure omitted ....

   MEM(int, Network *);
   ~MEM();
};
```

An important part of a customized composite protocol is the initialization of global data and creation of the required micro-protocols objects with their proper arguments, as illustrated below.

```
/**************************************************************/
/* FILE: MEM.cc                                               */
/* Implements Membership service composite protocol.          */
/**************************************************************/
#include "MEM.h"
#include "Network.h"
#include "CommService.h"
#include "MessageDriver.h"
#include "TokenDriver.h"
#include "StartUp.h"
#include "Recovery.h"
#include "SimpleMembershipDriver.h"
#include "MembershipDriver.h"
#include "LiveFailureDetection.h"
#include "TotalOrder.h"
... other micro-protocols omitted ....
```

```
void MEM::Start()
{
/* Initialize data structures here                              */

   MsgGraph = new OrderingGraph(1,SiteId);

/* Start all micro-protocols here                               */

   MP[0]  = new CommService(1,this);
   MP[1]  = new MessageDriver(2,this);
   MP[2]  = new TokenDriver(4,this,4,70,limit+1);
   MP[3]  = new StartUp(8,this,localMship);
   MP[4]  = new Recovery(16,this);
   MP[5]  = new LiveFailureDetection(32,this,60,limit);
   MP[6]  = new AccurateRecoveryDetection(32,this);
   MP[7]  = new MembershipDriver(64,this);
   MP[8]  = new TotalOrder(128,this);
   MP[9]  = new PartitionDetection(512,this,localMship,100);
   MP[10] = new CollectiveJoin(1024,this);
   ... other micro-protocols omitted ...

   Net->Members[SiteId] = TRUE;
}

MEM::MEM(int id, Network *net)
{
   SiteId = id;
   Net = net;
   Start();
}

MEM::~MEM()
{
   int i;
   for(i=0;i<30;i++) delete MP[i];
}
```

A.5 Simulation Driver

The execution of the prototype is coordinated by a simulation driver that creates the network object and the simulated sites, and generates system events such as failures, recoveries, token loss, partitions, and partition joins. Below is an example driver used to test the various membership service configurations. Note that the driver starts up and shuts down the simulated sites by sending special messages using a reliable communication method `rel_push`.

```
/***************************************************************/
/* FILE: driver.cc                                             */
/* The main program of the simulation.                         */
/***************************************************************/
#include <stdio.h>
#include "types.h"
#include "Network.h"
#include "MEM.h"
#include <time.h>

static Network *Net;
```

```
void DropToken()
{
   sleep(80);
   printf("DRIVER: drop token.\n");
   System->DropToken = TRUE;
   sleep(400);
}

void Partition()
{
   sleep(80);
   printf("DRIVER: create partition.\n");
   Net->Partition[0] = 1;
   Net->Partition[2] = 1;
   sleep(400);
   printf("DRIVER: join partitions.\n");
   Net->Partition[0] = 0;
   Net->Partition[2] = 0;
   sleep(600);
}

void FailNRec(int many)
{
  int VICTIM1,VICTIM2,VICTIM3;

  VICTIM1 = 0;
  VICTIM2 = 2;
  VICTIM3 = 2;

  sleep(80);
  printf("DRIVER: kill site %d.\n",VICTIM1);
  Net->ShutDown(VICTIM1);

  if (many > 1) {
     sleep(40);
     printf("DRIVER: kill site %d.\n",VICTIM2);
     Net->ShutDown(VICTIM2);

     sleep(50);

     printf("DRIVER: recover site %d.\n",VICTIM2);
     Net->site[VICTIM2]->ReStart();
     printf("DRIVER: recover site %d - done.\n",VICTIM2);
  }

  sleep(20);
  printf("DRIVER: recover site %d.\n",VICTIM1);
  Net->site[VICTIM1]->ReStart();
  printf("DRIVER: recover site %d - done.\n",VICTIM1);

  if (many > 2) {
     sleep(150);
     printf("DRIVER: kill site %d.\n",VICTIM3);
     Net->ShutDown(VICTIM3);

     sleep(50);

     printf("DRIVER: recover site %d.\n",VICTIM3);
     Net->site[VICTIM3]->ReStart();
  }
  sleep(400);
```

```
}

main()
{
   NetMessage *msg;
   long seed;
   int i,option;

   printf("Enter seed:\n");
   scanf("%d",&seed);
   srand48(seed);

   printf("Choose type of test run:\n");
   printf("0 : an event-less run.\n");
   printf("1 : kill site with token. \n");
   printf("2 : kill and recover one site. \n");
   printf("3 : kill and recover two sites. \n");
   printf("4 : kill and recover three sites. \n");
   printf("5 : create and fix partition. \n\n");
   scanf("%d",&option);
/*                                                                   */
/* Start system                                                      */
/*                                                                   */
   Net = new Network;
   Net->FAILURE_RATE = 0.10;           /* set comm. failure rate */
   for(i=0;i<NUMSITE;i++) Net->Members[i] = FALSE;

   for(i=0;i<NUMSITE;i++)              /* create sites           */
      Net->site[i] = new MEM(i,Net);

   msg = new NetMessage;               /* start site execution   */
   msg->type = SYSTEM_START;
   Net->rel_push(BROADCAST,(char *)msg);
   delete msg;
/*                                                                   */
/* Choose operation here                                             */
/*                                                                   */
   if(option == 0)      sleep(400);
   else if(option == 1) DropToken();
   else if(option == 2) FailNRec(1);
   else if(option == 3) FailNRec(2);
   else if(option == 4) FailNRec(3);
   else if(option == 5) Partition();
/*                                                                   */
/* Shutdown system                                                   */
/*                                                                   */
   msg = new NetMessage;
   msg->type = SYSTEM_STOP;
   Net->rel_push(BROADCAST,(char *)msg);
   delete msg;
}
```

REFERENCES

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR95-1534, Department of Computer Science, Cornell University, Aug 1995.

[AD76] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, Oct 1976.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms (Lecture Notes in Computer Science 647)*, pages 292–312, Haifa, Israel, Nov 1992.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, Jul 1992.

[AFM92] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. Request for Comments (Informational) RFC 1301, Internet Engineering Task Force, Feb 1992.

[AGH$^+$91] H-R. Aschmann, N. Giger, E. Hoepli, P. Janak, and H. Kirrmann. Alphorn: A remote procedure call environment for fault-tolerant, heterogeneous, distributed systems. *IEEE Micro*, 11(5):16–19,60–67, Oct 1991.

[AMMS$^+$93] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[AMMS$^+$95] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, Nov 1995.

[AO93] G. Andrews and R. Olsson. *The SR programming language:Concurrency in Practice*. The Benjamin/Cummings Publishing Company, 1993.

[AOC$^+$88] G. Andrews, R. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation.

ACM Transactions on Programming Languages and Systems, 10(1):51–86, Jan 1988.

[AOG92] D. P. Anderson, Y. Osawa, and R. Govindan. A file system for continuous media. *ACM Transactions on Computer Systems*, 10(4):311–337, Nov 1992.

[Apo89] Apollo Computer Inc. Network computing system (NCS) reference. Technical report, Apollo Computer Inc., 1989.

[ATK91] A.L. Ananda, B.H. Tay, and E.K. Koh. ASTRA — An asynchronous remote procedure call facility. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 172–179, Arlington, Texas, May 1991.

[Avi85] A. Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, Dec 1985.

[BALL90] B. Bershad, T. Anderson, E. Lazokska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 6(1):37–55, Feb 1990.

[BB91] R. Bianchini and R. Buskens. An adaptive distributed system-level diagnosis algorithm and its implementation. In *Proceedings of the 21st Symposium on Fault-Tolerant Computing*, pages 222–229, Jun 1991.

[BB93] R. Buskens and R. Bianchini. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proceedings of the 23rd Symposium on Fault-Tolerant Computing*, pages 470–479, Jun 1993.

[BBG+88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, SE-14(11):1711–1729, Nov 1988.

[BC91] K. Birman and R. Cooper. The Isis project: Real experience with a fault-tolerant programming system. *Operating Systems Review*, 25(2):103—107, Apr 1991.

[BCE+94] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. Spin – an extensible microkernel for application specific operating system services. Technical Report 94-03-03, University of Washington, Feb 1994.

[BDGX93] A. Bondavalli, F. Di Giandomenico, and J. Xu. A cost-effective and flexible scheme for software fault tolerance. *Journal of Computer Systems Science and Engineering*, 8:234–244, 1993.

[BDM95] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Technical Report UBLCS-95-18, Department of Computer Sciences, University of Bologna, Bologna, Italy, Nov 1995.

[Bec94] T. Becker. Application-transparent fault tolerance in distributed systems. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, pages 36–45, Pittsburgh, PE, 1994.

[Ber96] P. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, Feb 1996.

[BFG+85] J. Banino, J. Fabre, M. Guillemont, G. Morisset, and M. Rozier. Some fault-tolerant aspects of the Chorus distributed system. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 430–437, May 1985.

[BFHR90] B. Bhargava, K. Friesen, A. Helal, and J. Riedl. Adaptability experiments in the RAID distributed database system. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, pages 76–85, 1990.

[BG93] K. Birman and B. Glade. Consistent failure reporting in reliable communication systems. Technical Report 93-1349, Department of Computer Science, Cornell University, May 1993.

[BGN90] R. Bianchini, K. Goodwin, and D. Nydick. Practical application and implementation of distributed system-level diagnosis theory. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing*, pages 332–339, Jun 1990.

[Bha96] N. T. Bhatti. *A System for Constructing Configurable High-Level Protocols*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, 1996. In preparation.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[BHV+90] P. Barrett, A. Hilborne, P. Verissimo, L. Rodrigues, P. Bond, D. Seaton, and N. Speirs. The Delta-4 extra performance architecture (XPA). In *Proceedings of the Twentieth Symposium on Fault Tolerant Computing*, pages 481–488, Newcastle-upon-tyne, Jun 1990.

[Bir85a] K. Birman. Replication and fault-tolerance in the Isis system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 79–86, Orcas Island, WA, Dec 1985.

[Bir85b] A. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, Feb 1985.

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[BJRA85] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, Jun 1985.

[Bla91] A. Black. Understanding transactions in an operating system context. *ACM Operating Systems Review*, 20(1):73–76, Jan 1991.

[Blo77] M. Blount. Probabilistic treatment of diagnosis in digital systems. In *Proceedings of the 7th Symposium on Fault-Tolerant Computing*, pages 72 – 77, 1977.

[BM86] B. Bose and J. Metzner. Coding theory for fault-tolerant systems. In D. Pradham, editor, *Fault-Tolerant Computing: Theory and Techniques, Volume I*, pages 265 – 335. Prentice-Hall, 1986.

[BMD93] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Survey*, 25(2):171–220, Jun 1993.

[BMST92] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 187–198. Springer-Verlag, Wien, 1992.

[BN84] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb 1984.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology,*, 1(4):355–398, Oct 1992.

[BP90a] P. Berman and A. Pelc. Distributed probabilistic fault-diagnosis for multiprocessor systems. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing*, pages 340 – 346, Jun 1990.

[BP90b] D. Blough and A. Pelc. Reliable diagnosis and repair in constant-degree multiprocessor systems. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing*, pages 316 – 323, Jun 1990.

[BR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[BS91] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transaction on Computer Systems*, 9(2):143–174, May 1991.

[BS95] N. T. Bhatti and R. D. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of SIGCOMM '95*, pages 138–150, Cambridge, MA, Aug 1995.

[BSM89] D. Blough, G. Sullivan, and G. Masson. Fault diagnosis for sparsely interconnected multiprocessor systems. In *Proceedings of the 19th Symposium on Fault-Tolerant Computing*, pages 62 – 69, Jun 1989.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[BSS+95] B. Bershad, P. Savage, S.and Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, Colorado, Dec 1995.

[BY87] F. Bastani and I. Yen. A fault-tolerant replicated storage system. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 449–454, Los Angeles, CA, Feb 1987.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[CD87] M. Carey and D. DeWitt. An overview of the EXODUS project. *Database Engineering*, 6, 1987.

[CD94] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193, Nov 1994.

[CDD90] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the Advanced Automation System. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing*, pages 6–17, Newcastle-upon-Tyne, UK, Jun 1990.

[CFL94] P. Cao, E. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–178, Nov 1994.

[CGR88] R. Cmelik, N. Gehani, and W. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings of the 18th Symposium on Fault-Tolerant Computing*, pages 55–61, Tokyo, June 1988.

[CIM92] R. Campbell, N. Islam, and P. Madany. *Choices*, frameworks and refinements. *Computing Systems*, 5(3):217–257, 1992.

[CJK+87] R. Campbell, G. Johnston, K. Kenny, G. Murakami, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). In *Proceedings of the 4th Workshop on Real-Time Operating Systems*, pages 12–18, Jul 1987.

[CL95] W. Cheung and A. Loong. Exploring issues of operating systems structuring: from microkernels to extensible systems. *Operating Systems Review*, 29(4):4–16, Oct 1995.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[CM95] F. Cristian and S. Mishra. The Pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 215–221, Phoenix, AZ, 1995.

[Coo85] E. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, Orcas Island, WA, 1985.

[Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[Cri96] F. Cristian. Synchronous and asynchronous group communication. *Communications of the ACM*, 39(4):88–97, Apr 1996.

[CS95] F. Cristian and F. Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, University of California, San Diego, 1995.

[CT91] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug 1991.

[CZ85] D. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 2(3):77–107, May 1985.

[DFF+90] Y. Deswarte, J.-C. Fabre, J.-M. Fray, D. Powell, and P.-G. Ranea. Saturne: A distributed computing system which tolerates faults and intrusions. In *Proceedings of the Workshop on Future Trends of Distributed Computing Systems*, pages 329–338, Hong Kong, Sep 1990.

[DHW88] D. Detlefs, M. Herlihy, and J. Wing. Inheritance of synchronization and recovery properties in Avalon/c++. *IEEE Computer*, 21(12):57–69, Dec 1988.

[Dij68] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[DLA88] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds distributed operating system: Functional description, implementation details and related work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9, Jun 1988.

[DLAR91] P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, Nov 1991.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Apr 1996.

[DMS94] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Mar 1994.

[DMS95] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[DSK87] A. Dahbura, K. Sabnani, and L. King. The comparison approach to multiprocessor fault diagnosis. *IEEE Transactions on Computers*, C-36(3):373 – 378, Mar 1987.

[DZ83] J. Day and H. Zimmermann. The OSI reference model. In *Proceedings of the IEEE*, volume 71, pages 1334–1340, Dec 1983.

[EKO94a] D. Engler, M. Kaashoek, and J. O'Toole. The exokernel approach to extensibility. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, page 198, Nov 1994.

[EKO94b] D. Engler, M. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the 6th SIGOPS European Workshop*, 1994.

[EKO95] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, Dec 1995.

[EL90] P. Ezhilchelvan and R. Lemos. A robust group membership algorithm for distributed real-time system. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 173–179, Lake Buena Vista, Florida, Dec 1990.

[EL95] K. Echtle and M. Leu. Fault-detecting network membership protocols for unknown topologies. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications 4*, pages 69–90. Springer-Verlag, Wien, 1995.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[Fon94] H. Fonseca. Support environments for the modularization, implementation, and execution of communication protocols. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, Jun 1994. In Portuguese.

[GBB+95] D. O. Guedes, D. E. Bakken, N. T. Bhatti, M. A. Hiltunen, and R. D. Schlichting. A customized communication subsystem for FT-Linda. In *Proceedings of the 13th Brazilian Symposium on Computer Networks*, pages 319–338, May 1995.

[GGL93] J. Goldberg, I. Greenberg, and T. Lawrence. Adaptive fault tolerance. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132, Princeton, NJ, Oct 1993.

[Gib87] P. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, Jan 1987.

[GJS92] N. Gehani, H. Jagadish, and O. Shumeli. Event specification in an active object-oriented database. In *Proceedings of the 19th ACM SIGMOD Conference on the Management of Data*, San Diego, CA, Jun 1992.

[GMSS94] A. Gheith, B. Mukherjee, D. Silva, and K. Schwan. KTK: Kernel support for configurable objects and invocations. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, pages 92–103, Pittsburgh, PE, 1994.

[Gol92] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Dept of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, Dec 1992.

[Gra86] J. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, Jun 1986.

[GT91] A. Gopal and S. Toueg. Inconsistency and contamination. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 257–272, 1991.

[GT92] R. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California, Santa Cruz, May 1992.

[Gut89] R. Guting. Gral: An extensible relational database system for geometric applications. In *Proceedings of the 15th Conference on Very Large Databases*, Aug 1989.

[HA74] S. Hakimi and A. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, C-23(1):86–88, Jan 1974.

[HC92] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 187–199, Oct 1992.

[Her89] A. Herbert. ANSA Project and Standards. In S. Mullender, editor, *Distributed Systems*, chapter 17., pages 391–438. Academic Press, 1989.

[Her94] A. Herbert. An ANSA overview. *IEEE Network*, 8(1), Jan 1994.

[Hil95] M. A. Hiltunen. Membership and system diagnosis. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pages 208–217, Bad Neuenahr, Germany, Sept 1995.

[HKR84] S. Hosseini, J. Kuhl, and S. Reddy. A diagnosis algorithm for distributed computing systems with dynamic failure and repair. *IEEE Transactions on Computers*, C-33(3):223–233, Mar 1984.

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

[HP94] J. Heidemann and G. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb 1994.

[HP95] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 127–142, Copper Mountain Resort, Colorado, Dec 1995.

[HPOA89] N. Hutchinson, L. Peterson, S. O'Malley, and M. Abbott. RPC in the *x*-kernel: Evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, Dec 1989.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, Dec 1983.

[HR94] Y-M. Huang and C. Ravishankar. Designing an agent synthesis system for cross-RPC communication. *IEEE Transactions on Software Engineering*, 19(3):188–198, Mar 1994.

[HS87] R. Hayes and R.D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transaction on Software Engineering*, 13(12):1254–1264, Dec 1987.

[HS93] M. A. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, Oct 1993.

[HS94a] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. Technical Report 94-37, Department of Computer Science, University of Arizona, Tucson, AZ, Dec 1994.

[HS94b] M. A. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. Technical Report 94-28, Department of Computer Science, University of Arizona, Tucson, AZ, Oct 1994.

[HS95] M. A. Hiltunen and R. D. Schlichting. Properties of membership services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, Phoenix, AZ, Apr 1995.

[IM84] H. Ihara and M. Mori. Autonomous decentralized computer control systems. *IEEE Computer*, 17(8):57–66, Aug 1984.

[IY94] J-I. Itoh and Y. Yokote. Concurrent object-oriented device driver programming in Apertos operating system. Technical Report SCSL-TR-94-005, Sony Computer Science Laboratory Inc., Jun 1994.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–332, Aug 1988.

[JZ93] D. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software Practice & Experience*, 23(2):201–222, 1993.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, Feb 1989.

[Kea90] H. Kopetz and et al. Tolerating transient faults in Mars. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing*, pages 466–473, Jun 1990.

[Kec91] D. Kececioglu. *Reliability Engineering Handbook*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[KG94] H. Kopetz and G. Grunsteidl. TTP - A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan 1994.

[KGR91] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, Wien, 1991.

[KH83] S. Kreutzer and S. Hakimi. Adaptive fault identification in two new diagnostic models. In *Proceedings of the 21st Allerton Conference on Communication, Control, and Computing*, pages 353–362, 1983.

[KLVA93] K. Krueger, D. Loftessness, A. Vahdat, and T. Anderson. Tools for development of application-specific virtual memory management. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications 1993*, pages 48–64, Oct 1993.

[KM85] H. Kopetz and W. Merker. The architecture of Mars. In *Proceedings of the 15th Symposium on Fault-Tolerant Computing*, pages 274–279, Ann Arbor, MI, Jun 1985.

[KN93] Y. Khalidi and M. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th Symposium on Operating Systems Principles, Asheville, NC*, Dec 1993.

[KNKH89] S. Kawakami, T. Nakayama, K. Kashiwabara, and S. Hikita. REAM: An SQL based and extensible relational database management system. In *Proceedings of the 1st International Symposium on Database Systems for Advanced Applications*, Apr 1989.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed, real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, Aug 1987.

[KT91] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, Arlington, TX, May 1991.

[KTHB89] M. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, Oct 1989.

[LAKS93] B. Lindgren, M. Ammar, B. Krupczak, and K. Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. Technical Report GIT-CC-93/22, College of Computing, Georgia Institute of Technology, Atlanta, GE, Mar 1993.

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

[Lam81] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[Lap92] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1992.

[LE90] R. Lemos and P. Ezhilchelvan. Agreement on the group membership in synchronous distributed systems. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, Otranto, Italy, Sep 1990.

[Lew96] T. Lewis. The next $10,000_2$ years: Part II. *Computer*, 29(5):78–86, May 1996.

[LG85] K. Lin and J. Gannon. Atomic remote procedure call. *IEEE Transactions on Software Engineering*, SE-11(10):1126–1135, Oct 1985.

[LH94] Y. Liu and D. Hoang. OSI RPC model and protocol. *Computer communications*, 17(1):53–66, Jan 1994.

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

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, Mar 1988.

[LMJ91] L. Laranjeira, M. Malek, and R. Jenevein. On tolerating faults in naturally redundant algorithms. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 118–127, Sep 1991.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, Jul 1983.

[LSM82] L. Lamport, R. Shostak, and Pease M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, Jul 1982.

[LT91] H. Levy and E. Tempero. Modules, objects and distributed programming: Issues in RPC and remote object invocation. *Software Practice & Experience*, 21(1):77–90, Jan 1991.

[LYS93] J. Lee, H. Youn, and A. Singh. Adaptive voting for faulty (VFF) node scheme for distributed self-diagnosis. In *Proceedings of the 23rd Symposium on Fault-Tolerant Computing*, pages 480–489, Jun 1993.

[Maf94] S. Maffeis. Design and implementation of a configurable mixed-media file system. *Operating Systems Review*, 28(4):4–10, Oct 1994.

[Mal80] M. Malek. A comparison connection assignment for diagnosis of multiprocessor systems. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 31–36, 1980.

[MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, Jun 1994.

[Mar89] P. Martin. Remote procedure call facility for a PC environment. *Computer communications*, 12(1):31–38, Feb 1989.

[Mat89] F. Mattern. Time and global states in distributed system. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, North-Holland, 1989.

[MH76] S. Maheshwari and S. Hakimi. On models for diagnosable systems and probabilistic fault diagnosis. *IEEE Transactions on Computers*, C-25:228 – 236, Mar 1976.

[MMO+94a] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: a communications-oriented operating system. In *Proceedings of the 1st Symposium on Operating Design and Implementation (OSDI)*, page 200, Nov 1994.

[MMO+94b] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: a communications-oriented operating system. Technical Report 94-20, Department of Computer Science, University of Arizona, Tucson, AZ, Jun 1994.

[MMSA+96] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Apr 1996.

[Mos86] J.E.B. Moss. Getting the operating system out of the way. *IEEE Data Engineering*, 5, 1986.

[MP96] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. Technical Report 96-05, Department of Computer Science, University of Arizona, Tucson, AZ, May 1996.

[MPS89] S. Mishra, L. Peterson, and R. D. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 42–52, Seattle, Washington, Oct 1989.

[MPS92] S. Mishra, L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Wien, 1992.

[MPS93a] S. Mishra, L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed System Engineering*, 1:87–103, Dec 1993.

[MPS93b] S. Mishra, L. Peterson, and R. D. Schlichting. Experience with modularity in Consul. *Software Practice & Experience*, 23(10):1059–1075, Oct 1993.

[MS96] B. Mukherjee and K. Schwan. Evaluation of the adaptation techniques in kernel tool kit (KTK). In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 228–235, Annapolis, Maryland, May 1996.

[MSMA94] P. Melliar-Smith, L. Moser, and V. Agarwala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[MSS96] M. Mansouri-Samani and M. Sloman. A configurable event service for distributed systems. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 210–217, Annapolis, Maryland, May 1996.

[Ne94] NeXT Computer, Inc. *OpenStep Specification*, Oct 1994.

[Nel81] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[OIOP93] H. Orman, E. Menze III, S. O'Malley, and L. Peterson. A fast and general implementation of Mach IPC in a network. In *Proceedings of the 3rd Usenix Mach Conference*, pages 75–88, Apr 1993.

[OMG95a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995.

[OMG95b] Object Management Group. *CORBAservices: Common Object Services Specification*, 1995.

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

[Ous96] J. Ousterhout. Why threads are a bad idea (for most purposes). In *1996 USENIX Technical Conference*, Jan 1996. Invited Talk.

[PA88] M. Pucci and J. Alberi. Optimized communication in an extended remote procedure call model. *Computer architecture news*, 16(4):37–44, Sep 1988.

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

[Pel93] A. Pelc. Efficient distributed diagnosis in the presence of random faults. In *Proceedings of the 23rd Symposium on Fault-Tolerant Computing*, pages 462 – 469, Jun 1993.

[PMC67] F. Preparata, G. Metze, and R. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computer*, EC-16(6):848–854, Dec 1967.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

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

[Pow92] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd IEEE Symposium on Fault-Tolerant Computing*, pages 386–395, 1992.

[PS88] F. Panzieri and S. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, SE-14(1):30–37, Jan 1988.

[PSB+88] D. Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the Eighteenth Symposium on Fault-Tolerant Computing*, Tokyo, Jun 1988.

[RB91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal, Quebec, Canada, Aug 1991.

[RB95] R. van Renesse and K. Birman. Protocol composition in Horus. Technical Report TR95-1505, Department of Computer Science, Cornell University, Mar 1995.

[RBG+95] R. van Renesse, K. Birman, B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Department of Computer Science, Cornell University, 1995.

[RBM96] R. van Renesse, K. Birman, and S Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[Rei96] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, Jan 1996.

[RFJ93] R. Rajkumar, S. Fakhouri, and F. Jahanian. Processor group membership protocols: Specification, design, and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 2–11, Princeton, NJ, Oct 1993.

[RHB95] R. van Renesse, T. Hickey, and K. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, Department of Computer Science, Cornell University, Aug 1995.

[RKF93] W. Rosenberry, D. Kenney, and G. Fisher. *OSF Distributed Computing Environment: Understanding DCE*. O'Reilly, 1993.

[RM89] B. Rajagopalan and P. McKinley. A token-based protocol for reliable, ordered multicast communication. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 84–93, Seattle, WA, Oct 1989.

[RSB90] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, Oct 1990.

[RST89] R. van Renesse, H. van Staveren, and A. Tanenbaum. Performance of the Amoeba distributed operating system. *Software Practice & Experience*, 19:223–234, Mar 1989.

[RSV94] L. Rodrigues, E. Siegel, and P. Verissimo. A replication-transparent remote invocation protocol. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 160–169, Dana Point, CA, Oct 1994.

[RV91] L. Rodrigues and P. Verissimo. *x*AMp: A multi-primitive group communications service. Technical report, INESC, Lisboa, Portugal, Sep 1991.

[RV92] L. Rodrigues and P. Verissimo. xAMP: a multi-primitive group communication service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, TX, Oct 1992.

[SA89] A. Somani and V. Agarwal. Distributed syndrome decoding for regular interconnected structures. In *Proceedings of the 19th Symposium on Fault-Tolerant Computing*, pages 70–77, Jun 1989.

[SB90] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 6(1):1–17, Feb 1990.

[SB95] M. Spezialetti and S. Bernberg. EVEREST: An event recognition testbed. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 377–385, Vancouver, BC, Canada, May 1995.

[SBB87] K. Schwan, T. Bihari, and B. Blake. Adaptive, reliable software for distributed and parallel real-time systems. In *Proceedings of the 6th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 32–42, Mar 1987.

[SBS93] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[SCA94] P. van der Stok, M. Claessen, and D. Alstein. A hierarchical membership protocol for synchronous distributed systems. In K. Echtle, D. Hammer, and D. Powell, editors, *Proceedings of the 1st European Dependable Computing Conference (Lecture Notes in Computer Science 852)*, pages 599–616, Berlin, Germany, Oct 1994.

[SCF+86] P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 85–93, Asilomar, CA, Sep 1986.

[Sch82] F. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, Apr 1982.

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

[SDP89] S. Shrivastava, G. Dixon, and G. Parrington. An overview of Arjuna: A programming system for reliable distributed computing. Technical Report 298, Computing Laboratory, University of Newcastle upon Tyne, Nov 1989.

[SDP91] S. Shrivastava, G. Dixon, and G. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, Jan 1991.

[Shr83] S. Shrivastava. On the treatment of orphans in a distributed system. In *Proceedings of 3rd Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, Florida, Oct 1983.

[SM94] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 138–147, Dana Point, CA, Oct 1994.

[SMK+94] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, Feb 1994.

[Spe82] A. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(17):246–260, Apr 1982.

[SR86] M. Stonebraker and L. Rowe. The design of Postgres. In *Proceedings, ACM SIGMOD International Conference on Management of Data*, pages 340–355, 1986.

[SR93] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd Conference on Fault-Tolerant Computing*, pages 534–543, Jun 1993.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, Aug 1983.

[SS94] D. Schmidt and T. Suda. The service configurator framework: An extensible architecture for dynamically configuring concurrent, multi-service network daemons. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, pages 190–201, Pittsburgh, PE, 1994.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 14(7), Jul 1981.

[Sto94] A. Stoyenko. SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls. *Software Practice & Experience*, 24(1):27–49, Jan 1994.

[Sun88] Sun Microsystems. RPC: Remote procedure call protocol specification. Technical Report RFC-1057, Sun Microsystems, Jun 1988.

[TA90] B. Tay and A. Ananda. A survey of remote procedure calls. *ACM Operating Systems Review*, 24(3):68–79, Jul 1990.

[Tal96] Taligent, Inc. CommonPoint Application System 1.0. http://www.-taligent.com/cpappsys/cpappsys.html, 1996.

[Tan88] A. Tanenbaum. *Computer Networks*. Prentice Hall, New Jersey, 1988.

[TB90] Y. Tham and S. Bhonsle. Retargetable stub generator for a remote procedure call facility. *Computer communications*, 13(6):323–330, Jul 1990.

[TB95] J. Thomas and D. Batory. P2: An extensible lightweight DBMS. Technical Report UTEXAS.CS//CS-TR-95-04, The University of Texas at Austin, Department of Computer Sciences, Feb 1995.

[Tea91] ISA Project Core Team. ANSA: Assumptions, principles, and structures. In *Proceedings of the Conference on Software Engineering Environments*, University College of Wales, Aberystwyth, Wales, Mar 1991.

[TL94] C. Thekkath and H. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 110–121, Oct 1994.

[TN91] H. Tokuda and T. Nakajima. Evaluation of real-time synchronization in Real-Time Mach. In *Proceedings of the USENIX 1991 Mach Workshop*, Oct 1991.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards predictable real-time systems. In *Proceedings of the USENIX 1990 Mach Workshop*, Oct 1990.

[Toh86] Y. Tohma. Coding techniques in fault-tolerant, self-checking, and fail-safe circuits. In D. Pradham, editor, *Fault-Tolerant Computing: Theory and Techniques, Volume I*, pages 336–416. Prentice-Hall, 1986.

[TYT92] T. Tenma, Y. Yokote, and M. Tokoro. Implementing persistent objects in the Apertos operating system. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, Sep 1992.

[VH96] A. Veitch and N Hutchinson. Kea – a dynamically extensible and configurable operating system kernel. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 236–242, Annapolis, Maryland, May 1996.

[VM90] P. Verissimo and J. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, AL, Oct 1990.

[VR92] P. Verissimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In *Proceedings of the 22nd Symposium on Fault-Tolerant Computing*, pages 527–536, Boston, MA, Jul 1992.

[VRB89] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.

[Whe89] S. Wheater. *Constructing Reliable Distributed Applications using Actions and Objects*. PhD thesis, The University of Newcastle upon Tyne Computing Laboratory, Newcastle upon Tyne, England, Sept 1989.

[WHS95] C. Walter, M. Hugue, and N. Suri. Continual on-line diagnosis of hybrid faults. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable*

Computing for Critical Applications 4, pages 233–249. Springer-Verlag, Wien, 1995.

[WL88] J. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[WMK95] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In K. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)*, pages 33–57. Springer-Verlag, 1995.

[WSG91] Y.-H. Wei, A. Stoyenko, and G. Goldszmidt. The design of a stub generator for heterogeneous RPC systems. *Journal of Parallel and Distributed Computing*, 11(3):188–197, Mar 1991.

[WZZ93] X. Wang, H. Zhao, and J. Zhu. GRPC: A communication cooperation mechanism in distributed systems. *Operating Systems Review*, 27(3):75–86, Jul 1993.

[Xer81] Xerox. Courier: The remote procedure call protocol. Technical Report XSIS 038112, Xerox System Integration Standard, Stamford, CT, Dec 1981.

[YJT88] K. Yap, P. Jalote, and S. Tripathi. Fault tolerant remote procedure call. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 48–54, Jun 1988.

[Yok92] Y. Yokote. The Apertos reflective operating system: The concepts and its implementation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications 1992*, Vancouver, BC, Oct 1992.

[Yok93] Y. Yokote. Kernel structuring for object-oriented operating systems: The Apertos approach. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*, Nov 1993.

[Yu96] N. Yu. The AWT tutorial. http://ugweb.cs.ualberta.ca/ nelson/java/-AWT.Tutorial.html, 1996.