PROTEUS: A SOFTWARE INFRASTRUCTURE PROVIDING DEPENDABILITY FOR CORBA APPLICATIONS

BY

BRIJBHUSHAN SHRIKANT SABNIS

B.S., University of Illinois, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

To my parents

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Professor William H. Sanders. He gave me the opportunity to be a research assistant despite my lack of prior experience in distributed systems. He has always been willing to give advice and offer his technical ingenuity. He has managed to keep me focused on the objectives of the project while still allowing me to pursue the research interests that I enjoyed.

I would like to thank Dr. Michel Cukier for always providing a different perspective and supporting my work at the same time. He has been a model of intelligence and integrity. I could not have completed my work without his contributions. Dr. Dave Bakken and Dr. David Karr provided excellent technical contributions to the design and gave me a solid implementation base.

Jennifer Ren and Ayesha Ibrahim, my research partners, helped me in designing and implementing several facets of the work. Jessica Pistole and David Henke have been generous with their expertise and have expedited the solution of a myriad of obstacles. Paul Rubel has allowed me to leave with confidence that my work will be employed. Alex Williamson has provided system administration support beyond my understanding. Jenny Applequist has helped with technical editing and figure creation. I would also like to thank Dan Deavours, Jay Doyle, Gerard Kavanaugh, Doug Obal, John Sowder, and Patrick Webster for their help.

I would like to thank the Defense Advanced Research Projects Agency Information Technology Office for funding this research under contracts F30602-C-0315 and F30602-97-C-0276.

TABLE OF CONTENTS

Page

1. INTRODUCTION....1
1.1. CORBA....1
1.2. Fault Tolerance....1
1.3. AQuA....3
2. AQUA OVERVIEW....6
2.1. Maestro/Ensemble....6
2.2. Quality Objects....9
2.3. Proteus....10
3. PROTEUS....12
3.1. Application Model....12
3.1.1. Distributed Application Features....12
3.1.2. Application Requirements....13
3.2. Dependability Manager....14
3.2.1. Protocol Coordinator....16
3.2.2. Advisor....18
3.3. Object Factory....19
4. GROUPS....22
4.1. Group Structure....22
4.2. Communication....25
4.2.1. Active Replication Scheme....26
4.2.2. Proteus Communication Scheme....30
5. GATEWAY....33
5.1. Gateway Overview....33
5.2. Active Replication Scheme....37
5.2.1. Buffers....38
5.2.2. Voters....40
5.2.3. Active Replication Scheme Handler....41
5.2.4. Replication Group Member....42
5.2.5. Connection Group Member....45
5.3. Proteus Communication Scheme....46
5.3.1. Dependability Manager Handler....47
5.3.2. Factory/QuO Handler....48
5.3.3. PCS Group Member....48
5.3.4. Point-to-Point Group Member....51
5.4. Additional Features....52
5.4.1. State Transfer....52
5.4.2. Reporting View Changes & Leaving Groups....53
5.4.3. Synchronization Queue....54
5.4.4. Atomic Behavior....56
6. RESULTS, CONCLUSIONS, AND FUTURE WORK....58
6.1. Experimental Results....58

6.1.1. Dependability Manager and Object Factory 58
6.1.2. AQuA Applications 60
6.2. Conclusions and Future Work 63
APPENDIX A: STARTUP PARAMETERS.....69
REFERENCES 71

LIST OF FIGURES

Figure Page

Figure 1. AQuA Architecture....6
Figure 2. Example of Proteus Groups....23
Figure 3. Pass First Communication Scheme....27
Figure 4. Expanded View of Gateway Architecture....34
Figure 5. OMT Diagram for Active Replication Scheme....38
Figure 6. Active Replication Handler: Send Message Algorithm....42
Figure 7. Active Replication Group Member: Receive Sent Message Algorithm....43
Figure 8. Active Replication Group Member: Receive Cast Message Algorithm....44
Figure 9. Active Replication Group Member: View Change Algorithm....45
Figure 10. Active Connection Group Member: Receive Cast Message Algorithm....46
Figure 11. OMT Diagram for Proteus Communication Scheme....47
Figure 12. PCS Group Member: Gateway Message Cast Algorithm....49
Figure 13. PCS Group Member: View Change Algorithm....50
Figure 14. PCS Group Member: Receive Cast Algorithm....51
Figure 15. Dependability Manager GUI....59
Figure 16. Castle-Guarding Application....62

1. INTRODUCTION

1.1. CORBA

Many modern applications are distributed. As distributed computing has grown, standards have emerged. Among these is the Common Object Request Broker Architecture (CORBA) [1]. CORBA provides application developers with an interface to build distributed object-oriented applications. The Object Management Group (OMG), the developers of CORBA, were able to solve or mollify several common problems that occurred during distributed application development. Their achievements include an interface definition language that is both programming-language- and platform-independent. They also provide a dynamic invocation interface to allow distributed communication interfaces to be executed and discovered at run-time. CORBA also sets a standard for services to the application such as naming, event, and life cycle services. However, CORBA does not provide a simple architecture to allow distributed applications to be fault tolerant.

1.2. Fault Tolerance

Many of the concepts introduced in this section are taken from the Delta-4 project [2]. A system *failure* is occurs when the service delivered by the system no longer complies with its specification. An *error* is that part of the system state that is liable to lead to failure. The cause of an error is a *fault*. An error is thus the manifestation of a fault in the system, and a failure is the effect of an error on the service.

Fault tolerance is the ability to provide service complying with the specification in spite of faults. Fault tolerance is achieved by error processing and by fault treatment. *Error*

processing is aimed at removing errors from the computational state, preferably before a failure occurs. Error processing may be carried out either by error detection and recovery or by error compensation. *Fault treatment* is aimed at preventing faults from being activated again. Whereas error processing is aimed at preventing errors from becoming visible to the user, fault treatment is necessary to prevent faults from causing further errors. Fault treatment entails fault diagnosis, fault passivation, and, if possible, system reconfiguration to restore the level of redundancy so that the system is able to tolerate further faults. *Fault diagnosis* is the process of determining the cause of observed errors. *Fault passivation* is preventing diagnosed faults from being activated again.

When designing a fault-tolerant system, it is important to define clearly what types of faults the system is intended to tolerate and the assumed behavior (or failure modes) of faulty components. If faulty components behave differently from what the system's error processing and fault treatment facilities can cope with, then the system will fail. In distributed systems, the behavior of a node can be defined in terms of the messages that faulty nodes send or do not send. The most common assumption about node failures is that nodes are *fail-silent* (also called *crash failures*). A crash failure occurs when a node stops sending out messages and when the internal state is lost. *Value faults* occur when the message arrives in time but contains the wrong content. A *time fault* includes delay and omission faults. A *delay fault* occurs when the message has the right content but arrives late. An *omission fault* occurs when no message is received.

1.3. AQuA

The Adaptive Quality of Service for Availability (AQuA) [3] project aims to provide fault tolerance to CORBA applications through software replication. *Software replication* means that a software process is instantiated in multiple locations throughout the system to tolerate faults in one or more of the instances (*replicas*). AQuA provides mechanisms to process errors and treat different types of faults. For many applications, fault tolerance is handled in an application-specific way. This often leads to embedding the code providing the fault tolerance within the code that achieves the functional requirements. As a result, there is an increase in development costs and development time. This is because software engineers have to develop code that is cognizant of the fact that it is replicated. The software engineers therefore have to be experienced in fault tolerance, an uncommon skill, and have to develop more code.

A CORBA Object Request Broker (ORB) simplifies development of distributed applications by automatically generating the code for communicating between remote objects based on a specification in the interface definition language (IDL). Similarly, AQuA aims to provide the code for achieving fault tolerance so that the application developer does not have to develop code that is cognizant of its fault tolerance. Since different applications need to tolerate different types and numbers of faults, AQuA provides the application developer a simple interface to specify the fault tolerance desired for the application.

Many distributed applications have dynamic requirements. Consequently, the services provided to distributed applications also need to be dynamic. Therefore, another goal of AQuA is to provide fault tolerance that can be reconfigured dynamically at run-time. Additionally, it is not possible to assume that the system always has the resources available to

provide the desired level of fault tolerance. Therefore, the AQuA system aims to adapt to changes in the system configuration while providing fault tolerance.

The AQuA architecture is layered. The four layers, from top to bottom, are the CORBA application itself, Quality Objects (QuO) [4, 5], Proteus, and Maestro/Ensemble [6, 7]. Each layer only communicates with the layer immediately above or below it. The QuO layer provides the CORBA application with an interface to specify its fault tolerance requirements at a high level. Proteus aims to translate the high-level requirements of the application into a system configuration, provide fault tolerance, and adapt to changes inputted by the QuO layer and the Maestro/Ensemble layer. Maestro/Ensemble is a group communication system that provides services to Proteus to manage the system.

Other work has been done to provide dependability to distributed CORBA applications. The OpenDREAMS project [8] aims to provide software replication as a true CORBA service. The OpenDREAMS approach has the potential to become an official CORBA service, because it was built using CORBA communication and is therefore interoperable and heterogeneous. The approach can easily be used with any CORBA 2.0-compliant ORB to provide reliability and high availability. The Eternal project [9] takes the interception approach to providing dependable communication to CORBA applications. With the interception approach, CORBA invocations are intercepted before they leave a host. The intercepted messages are sent using reliable group communication and then delivered as CORBA invocations to the correct CORBA objects. Eternal uses a different underlying group communication system from AQuA (Totem) and has a goal of completely hiding fault tolerance from the application. Electra [10] is a specialized CORBA ORB that provides

availability to CORBA applications. One drawback to this approach is that it does not allow the application developer to choose any CORBA ORB.

AQuA takes an approach similar to Eternal, but has a different emphasis. The AQuA project aims to allow the application to specify dynamic QoS requirements and to provide adaptation mechanisms when the requirements cannot be met. Distributed resource management for dependability and translation of high-level dependability specifications into system configurations are among the cutting-edge research goals addressed by the AQuA project.

The design and implementation of Proteus is a first step in addressing these research areas. This thesis describes the design and implementation of an initial version of Proteus. The research goals addressed are:

- Design and implementation of a dependability manager to treat crash failures by restarting replicas (Chapter 3).
- Design and implementation of an object factory to start and kill processes on a host and provide information about a host to the dependability manager (Chapter 3).
- Design and implementation of a scalable and reliable group structure for active replication and communication with a dependability manager (Chapters 4 and 5).
- Design and implementation of a gateway process to implement reliable communication independent of functional behavior for a CORBA object (Chapter 5).
- Design of a flexible infrastructure that can be easily extended to provide additional features (Chapter 6).

2. AQUA OVERVIEW

This chapter presents an overview of AQuA. The AQuA architecture provides three components to a CORBA application: Maestro/Ensemble, QuO, and Proteus. Maestro/Ensemble provides reliable group communication over a Local Area Network (LAN). QuO provides a CORBA application with an interface for dynamically specifying dependability requirements. Proteus provides fault tolerance to the CORBA application. Proteus is composed of a dependability manager, an object factory, and a gateway. Figure 1 shows the AQuA architecture.

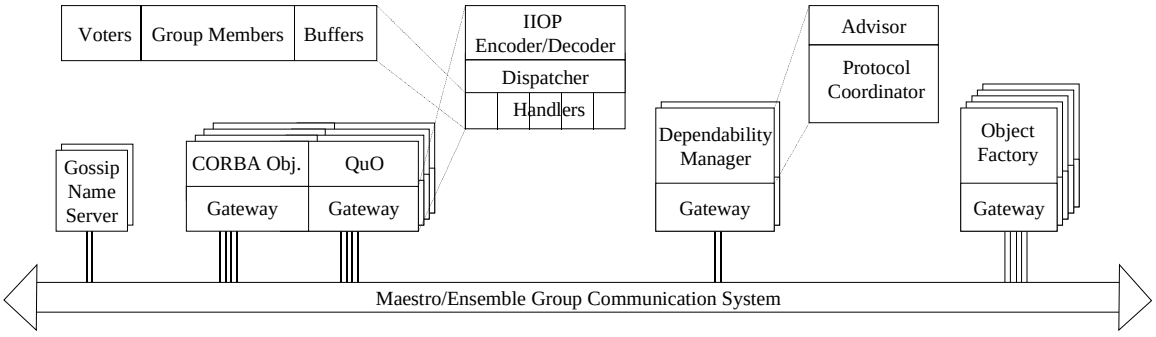

Figure 1. AQuA Architecture

2.1. Maestro/Ensemble

Ensemble [6] is a group communication system developed at Cornell University. It provides services to an application developer for reliable distributed communication. Ensemble focuses on communication of processes within groups. In each process group, there is a set of members. An Ensemble group member is a running process that has the Ensemble communication libraries linked in with its application-specific code.

Ensemble provides various types of reliable communication for groups. In particular, it provides totally ordered and causally ordered reliable multicast [11]. *Totally ordering* means that messages delivered to members of a group are delivered in the same order to each member of the group. *Causal ordering* means that a message is delivered to a group member only when all messages upon which it depends have been delivered. Ensemble also provides group membership services. In particular, each group member has a *view* of what other members are in the group. Ensemble ensures that all group members receive the same view of the group when group membership changes. Each Ensemble group has a group leader that is used in the group's communication protocols. Ensemble also provides a leader election service to ensure that each group has exactly one leader.

The model of communication used for AQuA applications is the virtual synchrony model [12]. The *virtual synchrony model* supports dynamic group membership, state transfer to joining processes, reliable multicasts, and view-synchronous multicast delivery. Ensemble provides the ability to use the virtual synchrony model of group communication. This allows the developer the luxury of knowing that messages will be delivered to the other members of the group within the same view in which they were sent.

All communication is done through an Ensemble protocol stack. The application-specific code sends messages down the Ensemble protocol stack, and receives messages after they are sent up the protocol stack. The protocol stack is composed of layers. Each layer provides a specific functional property that is useful in group communication. The system thus provides flexibility by allowing the application developer to choose the layers for each group's stack. Some of the layers provided include the point-to-point communication, group membership, and total ordering layers.

Ensemble uses a naming service called *gossip*. When a group member wishes to join a group, it contacts the gossip name server. The gossip name server connects it to the group, if the group exists. The gossip name server can be replicated. Therefore, the name server is not a single point of failure.

Maestro [7], also developed at Cornell, provides an object-oriented C++ interface to the Ensemble group communication system. Maestro also provides higher-level support for mechanisms such as state transfer and message creation. A simple method to specify the characteristics of different group types is also provided.

A Maestro application developer defines group members by deriving application-specific classes from the *Maestro_CSX* base class. The class provides base methods for group membership and message transmission. The application developer is responsible for defining callbacks. Each callback defines application-specific functions to perform in the case of a particular event. Maestro automatically invokes callbacks when the event corresponding to the callback occurs. The following callbacks are used in AQuA:

- *receive cast* – This callback is called when a multicast message is delivered to the group member.
- *receive sent* – This callback is called when a point-to-point message is delivered to the group member.
- *view change* – This callback is called when a new view of group membership is delivered to the group member.
- *ask state* – This callback is called at an existing group member when a new group member joins the group. It is required to get its state and then call *send state* to send its state to the new group member.

- *state transfer* – This callback is called at a new group member before it has completed joining the group. It asks for the state of an existing member using a blocking *get state* call. After receiving the state of another member, this callback sets the state of the new member.

2.2. Quality Objects

The goal of Quality Objects (QuO) [4], developed at BBN Technologies, is to provide distributed applications with a simple object-oriented interface to specify their Quality of Service (QoS) requirements. In the case of AQuA, the dependability of a remote CORBA application is the QoS being specified. QuO works something like a CORBA ORB. The application developer specifies QoS regions and their associated requirements through a Contract Definition Language (CDL) [5]. Based on this specified contract, a QuO runtime process handles the dynamic QoS requirements of the application. In AQuA, a CORBA application requests a region of dependability to the QuO runtime, instead of binding to remote CORBA objects directly through the naming service. When the QoS for the region is met, a callback is communicated to the CORBA application from the QuO runtime.

Like other modern distributed communication tools, QuO is dynamic. The application may request a new region of dependability at runtime, and the QuO runtime will attempt to satisfy this. Additionally, if the system resources change such that the dependability of the remote objects is not within the specified region, a callback is made to the application. At this point the application may adapt to the current region, make a request for another region, or perform other application-specific actions. In AQuA, the inputs to the QuO runtime concerning the remote object's QoS come from Proteus. Proteus communicates changes in

the specified dependability of the remote objects to the QuO runtime. In turn, if the region requested by the application is no longer valid, a callback is made to the application by QuO.

QuO defines contracts through the CDL. A contract consists of negotiated regions and reality regions. Negotiated regions specify the expected behavior of the local and remote objects, and are defined by predicates on the state of system condition objects, which provide a view of the state of the distributed system. Reality regions are defined within each negotiated region, and specify measured or observed conditions of interest in the distributed system. Reality regions are specified by predicates on a portion of the state of the distributed system itself, as viewed through system condition objects. In the case of AQuA, system condition objects provide information to the QuO contract from the dependability manager concerning whether requested dependability requirements are being met.

2.3. Proteus

Proteus provides adaptive fault tolerance via software replication for distributed CORBA applications. From the perspective of the application developer, it can be thought of as a sophisticated CORBA dependability service. Proteus replaces the need for the naming and life cycle services, since object binding and object creation must be managed directly. However, Proteus should not be considered a CORBA service. Proteus places communication in between CORBA method invocations, unlike the traditional framework for a CORBA service. Additionally, the object does not interface with Proteus directly, but instead interfaces with the QuO runtime.

Fault tolerance via software replication can be used to tolerate a variety of faults. Two common schemes that can be used for software replication are active replication and passive

replication. *Active replication* means that all replicas of an application process input messages. An application must be deterministic to be actively replicated. Active replication can be used to tolerate crash, value, and time faults. *Passive replication* means that all replicas are delivered messages, but only the leader processes them. Passive replication allows an application to be non-deterministic, but can only tolerate crash failures. Proteus is designed to support both replication schemes.

Proteus is composed of dependability managers, object factories, and gateways. A dependability manager is the component of Proteus that communicates with the QuO runtime. The dependability manager decides how to configure the system and then attempts to carry out its decisions. The object factories are primarily responsible for starting and killing replicas, based on instructions from the dependability manager. They also provide additional information to the dependability manager about a host in the system. The dependability manager uses this information to aid in the selection of hosts on which to place replicas. The gateway is a process that is atomically associated with a CORBA object. The gateway intercepts any message sent by its CORBA object and then communicates the message via Maestro/Ensemble to other gateways. The other gateways then pass up the message to their applications. Gateways are the implementation of replication schemes. Gateways are not only used by CORBA applications using the AQuA architecture (*AQuA applications*), but are also used by the dependability manager, object factories, and QuO runtime. A gateway and its CORBA object are collectively referred to as an *AQuA object*.

3. PROTEUS

This chapter describes the design and current implementation of the Proteus dependability manager and object factory. The application requirements and the type of AQuA applications that are currently supported by Proteus are also described. The gateway, also a component of Proteus, is described separately in Chapter 5.

3.1. Application Model

This section describes the type of applications that are supported by the current implementation of Proteus.

3.1.1. Distributed Application Features

AQuA applications using the current implementation of Proteus may exhibit the following properties:

- Any CORBA object in the application may act as a client and as a server.
- Any CORBA object in the application may communicate with multiple applications.
- Any CORBA object in the application may have state.
- The application may use synchronous or deferred synchronous communication. *Synchronous communication* means that an object making a request to another object blocks until a reply has been received from the other object. When an object does not block after making a request to another object but maintains a request-reply structure, this type of communication is referred to as *deferred synchronous* communication in this thesis.

- The application may make hierarchical method invocations. For example, suppose object A makes a request to object B. Before responding to object A, object B may make a request to object C.
- An application may be developed independent of the AQuA architecture. There is a minimal amount of integration into AQuA because the application is written so that it is unaware that it may be replicated. Integrating an application to use Proteus requires only the definition of two extra CORBA methods for each object. These methods are used to transfer an object's state. If an object does not need to be replicated, there are no changes that need to be made to that object.
- The application may be developed with any CORBA 2.0-compliant ORB (tested with VisiBroker 3.1).

3.1.2. Application Requirements

This section lists the requirements imposed on an AQuA application. Each requirement is listed as being either a design restriction or an implementation restriction. Section 6.2 discusses the removal of the restrictions imposed by the current implementation.

- The application cannot use the CORBA Life Cycle or Naming services. The Life Cycle service may not be used because Proteus must manage object creation. Object binding may not be done with the Naming service, but rather through the direct use of Internet Object References (IORs). This is usually implemented by having the application read in a file that contains the IOR it needs to use. These requirements are imposed by the design of AQuA.

- The application cannot use general asynchronous communication. Only synchronous and deferred synchronous communication are supported. This requirement is imposed by the current implementation of Proteus.
- If an object needs to be replicated, it must be deterministic. Determinism means that if two replicas are delivered an ordered sequence of messages, both replicas will have the same state after all the messages have been processed. This generally requires that an object be prohibited from using non-deterministic features such as threads, random number generators, and time stamps. It also requires that all communication done by the object be done through the AQuA architecture. This requirement is imposed because the current implementation of Proteus only supports the active replication scheme for AQuA applications.
- An object that is developed using a threaded ORB with non-deterministic scheduling, deferred synchronous communication, and communication with multiple objects cannot be replicated. This requirement is imposed by the implementation of the synchronization queue discussed in Section 5.4.3.
- The application must use Linux (tested with Kernel 2.0.32). This requirement is imposed by the current implementation of Proteus.

3.2. Dependability Manager

The Proteus dependability manager translates QoS requirements into a system configuration. The dependability manager is the component of Proteus that interfaces with QuO. Through QuO, the dependability manager is notified of the application's desired QoS. The dependability manager makes a callback to QuO when the desired QoS has successfully

been achieved or when the desired QoS can no longer be maintained. The dependability manager also provides fault treatment for AQuA applications. Based on errors detected by other components, the dependability manager evaluates the system configuration and either treats the fault or makes a callback to QuO. The dependability manager is composed of a gateway process and a CORBA object process. This section describes the functional interface to the dependability manager and the implementation of the CORBA object process. The dependability manager's gateway is described in Chapter 5.

The current implementation of the dependability manager does not allow the dependability manager to be replicated. Therefore, the dependability manager is a single point of failure in the current implementation of the AQuA architecture. The dependability manager only manages the system configuration. No application-level messages are transmitted through the dependability manager. Therefore, if the dependability manager crashes, any AQuA applications will continue to communicate despite the lack of a dependability manager. Because of that, if a fault in the application occurs, the fault will not be treated and no callback will be given to QuO. In the current implementation, the dependability manager cannot be restarted in the case of failure, since it will not be aware of the state of the dependability manager prior to the crash.

Errors are detected by object factories and AQuA application gateways. If an application cannot be started or killed, the error is reported by the object factory. If an AQuA application crashes, the error is detected by the AQuA application gateways. The dependability manager uses these error detection mechanisms to treat faults.

The dependability manager can be used without a QuO runtime by directly specifying the QoS requirements from an application itself, or from an input window on the

dependability manager. The input window allows a user to create, edit, and delete QuO regions as if QuO were making the calls itself. This is useful for testing the dependability manager independently of QuO.

The internal structure of the dependability manager's CORBA application process is divided into a protocol coordinator and an advisor, as seen in Figure 1. The protocol coordinator is a CORBA object that is used to interface with the other system components. The advisor makes decisions about how to configure the system.

3.2.1. Protocol Coordinator

The protocol coordinator is the CORBA object used to communicate with QuO, object factories, and application gateways. The protocol coordinator also manages the data structures in the dependability manager that keep track of the system configuration. If an unexpected event occurs in the system configuration, the protocol coordinator calls the appropriate method in the advisor. The protocol coordinator's CORBA interface consists of the following methods:

- *view change* – This method is called by the AQuA application gateways to report Maestro/Ensemble view changes. Given view change information, the protocol coordinator determines whether the view change is the result of a crash failure, a configuration change requested by Proteus, or both. If the view change is the result of a crash, the advisor is called to take action for each replica that has crashed.
- *register* – This method is called by an object factory to register itself. After setting up the internal data structures to handle a new host in the system, the dependability manager returns a reply to inform the factory that it has registered itself successfully.

- *start reply* – This method is called by an object factory to report the status of a request made to start an application.
- *kill reply* – This method is called by an object factory to report the status of a request made to kill an application.
- *load reply* – This method is called by an object factory to report the load of its host. This may be the result of a request for the load from the dependability manager or be part of the periodic update of the load performed by the object factory.
- *new expected region* – This is the method by which the dependability manager receives input from QuO. This method is called when the QoS of the remote application is first specified or has changed due to adaptation.

The protocol coordinator also carries out the decisions of the advisor. In the current implementation of the dependability manager, the two methods within the protocol coordinator used by the advisor are *start* and *kill*. The protocol coordinator calls *start* or *kill* on the correct object factory, based on the decision of the advisor. The reason for this extra level of abstraction for starting and killing replicas is that in future implementations of the protocol coordinator, mechanisms for dynamically reconfiguring AQuA application gateways will reside within the protocol coordinator.

The protocol coordinator uses timers to keep track of the system configuration. The advisor is notified of a start failure if a factory response to a start request does not arrive within a specified amount of time or if the view change expected from starting a replica does not arrive within a specified amount of time. Note that using timers makes the dependability manager non-deterministic.

3.2.2. Advisor

The advisor, implemented by Jennifer Ren, is called by the protocol coordinator if an unexpected event occurs in the system. The advisor treats faults and processes QuO requests to change the system configuration. The advisor's interface consists of the following methods:

- *crash failure* – This method is called if a replica is removed from a view unexpectedly.
- *start failed* – This method is called if an object factory responds with a failure to a requested start of a replica, if it does not respond in time to a requested start of a replica, or if the replica fails to join its replication group in time.
- *new region* – This method is called if there is a new QoS region for an application.
- *kill region* – This method is called if the current region for an application is to be removed entirely.
- *unexpected replica join* – This method is called if a replica unexpectedly joins a replication group.
- *unexpected replica start* – This method is called if a factory unexpectedly sends a start reply.
- *unexpected replica kill* – This method is called if a factory unexpectedly sends a kill reply.

The current implementation of the advisor has methods defined for *crash failure*, *start failed*, *new region*, and *kill region*. The *crash failure* method chooses to start a replica on the least-loaded host in the system that does not have a replica of the same application already running. The *start failed* method also chooses to start a replica on the least-loaded host in the system that does not have a replica of the same application already running, except that the method will not choose the host where the replica start failed.

The *new region* method defines the QoS in terms of the minimum number of crash failures to tolerate (*n*). If no replicas of the application are running, the advisor decides to start *n + 1* new replicas. If replicas are already running, then the advisor may decide to start additional replicas or kill existing replicas so that *n + 1* replicas are running. In the case where more than one region requests a QoS for an application, the advisor chooses to maintain the maximum *n + 1* replicas between the regions. The *kill region* method decides to kill all replicas of an application unless the QoS for the application is specified by another region. In this case, the advisor chooses to maintain the maximum *n + 1* replicas between the remaining regions.

If a *start failed* or *new region* call results in an inability to meet QoS requirements, a callback is made to QuO. A call to *start failed* will result in a QuO callback if all available hosts, except for the host where the start failed, have an application replica running. A call to *new region* will result in a QuO callback if the specified minimum number of faults to tolerate is greater than or equal to the number of hosts in the system.

3.3. Object Factory

One object factory resides on each host in the system that is managed by the dependability manager. The factory's main purpose is to start and kill processes. Like the dependability manager, a factory is composed of a CORBA object process and a gateway process. This section details the factory's CORBA object process, which was written using VisiBroker 3.1 for Java. The factory's gateway is detailed in Chapter 5.

When a factory is started, it reads in a file that is specific to its host. This file specifies which applications a factory can start. For each application in the file there is an application

name and an application path. The application name is the name by which the dependability manager will make a request to start or kill an application. The application path contains the executable name and startup parameters to use to start the application. The information in this file is stored into memory upon startup.

After reading in the file of applications, a factory registers itself with the dependability manager by calling the dependability manager's synchronous *register* method. The factory registers itself to let the dependability manager know that the factory's host is available to start replicas. The factory also reports the load of its host in the registration message. After receiving a reply from the dependability manager, the factory is ready to start and kill processes.

When the dependability manager sends an asynchronous *start* request to the factory, the factory attempts to start the application specified by the dependability manager. If an exception is generated while starting the application, a start failure is asynchronously replied to the dependability manager; if no exception is generated, the factory adds the application to a list of running applications, and a successful start is asynchronously replied to the dependability manager.

Requests from the dependability manager to *kill* an application are handled in the same manner. If an exception is generated while attempting to kill the application, a kill failure is asynchronously replied to the dependability manager. If no exception is generated, the factory removes the application from the list of running applications and a successful kill is asynchronously replied to the dependability manager.

The dependability manager also notifies the factory if a replica on the factory's host fails through the *replica crashed* method. This is done so that the factory has the correct state

of its host. This method simply removes the crashed replica from the list of running applications.

The factory is also responsible for providing information to the dependability manager about its host. In the current implementation of the factory, the factory periodically sends the load of its host to the dependability manager. The dependability manager uses this information to decide how to assign replicas to hosts. In the current implementation, a factory failure can only be tolerated if it is the consequence of a host crash. A factory may be restarted once the host has successfully restarted.

4. GROUPS

This chapter describes how process groups are used to provide reliable communication between Proteus components. A goal of the group structure is to define a common framework that can be used for a variety of replication schemes. Section 4.1 describes the design of the group structure. Section 4.2 details the design of the current implementation from the perspective of a message flowing through the system. Chapter 5 details the current implementation from the perspective of a gateway.

4.1. Group Structure

A general object model is used rather than a client/server model. Objects can both initiate requests and respond to requests. In AQuA, the four types of objects are AQuA applications, QuO runtimes, factories, and dependability managers. Each of these objects is composed of a CORBA application and a gateway. The CORBA application for each object type implements its application-specific code using a CORBA ORB. The gateway for each object type provides the reliable communication needed by that object type. When an object is said to be in a group, it is actually the object's gateway that is a member of the group. A gateway uses different communication mechanisms depending on which object type it is a part of.

Four types of groups are used in Proteus: replication groups, connection groups, point-to-point groups, and the Proteus Communication Service (PCS) group. Figure 2 shows example replication groups, example connection groups, and possible states of the PCS group. Replication groups and connection groups are used by AQuA applications to reliably communicate with one another. The PCS group is used by a factory, QuO, or AQuA

application to cast messages to the dependability managers in the system. The point-to-point groups are used to send messages from a dependability manager gateway to factory and QuO gateways.

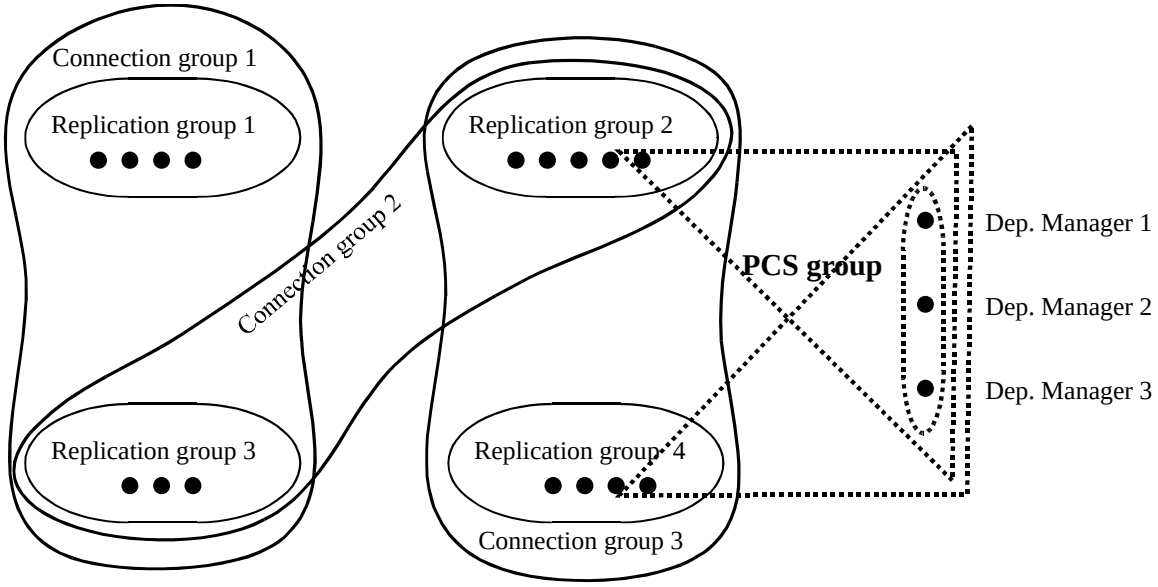

Figure 2. Example of Proteus Groups

A replication group is composed of one or more replicas of an AQuA application, as seen in Figure 2. From the perspective of a CORBA application developer, each replication group can be viewed as a reliable object. A replication group has one object that is designated as its leader. All objects in the group have the capacity to be the object group leader, and a protocol is provided to make sure that a new leader is elected when the current leader fails. The leader may perform special functions depending on the replication scheme used. To maintain a group, Maestro/Ensemble uses protocols that use group leaders. For implementation simplicity, the object whose gateway process is the Ensemble group leader is designated the leader of the replication group. This allows Proteus to use the Ensemble leader election service to elect a new leader if the leader object fails.

Communication between replication groups is performed using connection groups. A connection group is a group consisting of the members of two replication groups that wish to communicate, as seen in Figure 2. A message is multicast within a connection group in order to send a message from one replication group to another replication group. A connection group can define different communication schemes. The replication group sending the message must communicate according to the communication scheme specified by the connection group. For each message, the sending replication group chooses which communication scheme to use based on which connection group a message is destined for.

Reliable multicast is also needed for communication with the dependability manager. In AQuA, this is achieved using the Proteus Communication Service (PCS) group. The PCS group consists of all the dependability manager replicas in the system. The PCS group also has transient members. These transient members are factory, AQuA application, or QuO objects that want to cast messages to the dependability manager replicas. AQuA applications provide notification of view changes through the PCS group. QuO makes requests for QoS through the PCS group. A factory responds to start and kill commands and provides host load updates through the PCS group. After a multicast to the PCS group, the transient member will leave the group. Figure 2 shows the PCS group in three possible membership states.

Since communication involving the PCS group is fairly infrequent, the overhead in joining and leaving the PCS group is small relative to the cost of maintaining a group that consists of all objects in the system. Group communication is used to communicate with the replicated dependability manager to ensure that all dependability manager replicas receive the same input.

A point-to-point group is used to send messages from a dependability manager to a factory or QuO. This group is necessary in order to be consistent in the form of communication used between AQuA objects. By using this group for communication of messages from a dependability manager to other objects, all communication between any two objects in the AQuA architecture can be monitored at the gateways. Note that not all dependability manager replicas are involved in joining the group. That is because there is higher overhead in Ensemble if multiple objects join a group simultaneously. The overhead is great enough that it is advisable to avoid, when possible, having multiple objects simultaneously join a group.

This group structure is scalable because it uses many small groups. Protocols used to provide group membership and total ordering do not scale well to large groups. Therefore, it is better to use multiple groups of smaller size if scalability is necessary. The groups in Proteus do not get larger as the number of objects in the system increases. Instead, the number of Proteus groups increases. Proteus groups will only get larger as the number of faults to tolerate increases. Additionally, if a single large group is used, replica crashes can cause view changes to affect the entire system. By using the Proteus group structure, replica crashes will result in view changes only affecting the objects involved.

4.2. Communication

This section details the protocols of the active replication scheme and the Proteus communication scheme. The active replication scheme is the only scheme currently implemented in Proteus for AQuA applications. Other communication protocols will be

implemented in the future. Communication is described from the perspective of a message traveling within the system. The implementation of these protocols is described in Chapter 5.

4.2.1. Active Replication Scheme

In the active replication scheme, all replicas of a replication group process delivered messages. Connection groups are used to transmit messages reliably from one replication group to another replication group. One of two communication schemes is used to send messages: *pass first* or *leader only*. This section will first describe how communication takes place when no faults occur, and then describe how faults are tolerated in each phase of communication.

To specify precisely how this is done for the *pass first* communication scheme, it is helpful to introduce some notation. In particular, let $O_{i,k}$ be object k of replication group i, and let object $O_{i,0}$ be the leader of the group. Furthermore, let $\{O_i\}$ be a replication group i of size no_i, composed of the objects $O_{i,k}$ ($k=0, \ldots, no_i-1$). Using this notation, Figure 3 shows the communication within a connection group made up of replication groups $\{O_i\}$ and $\{O_j\}$. To illustrate how communication takes place, suppose that replication group i is the sender group and group j the receiver group. Labels (1), (2), and (3) correspond to the order of steps used in the message transmission process.

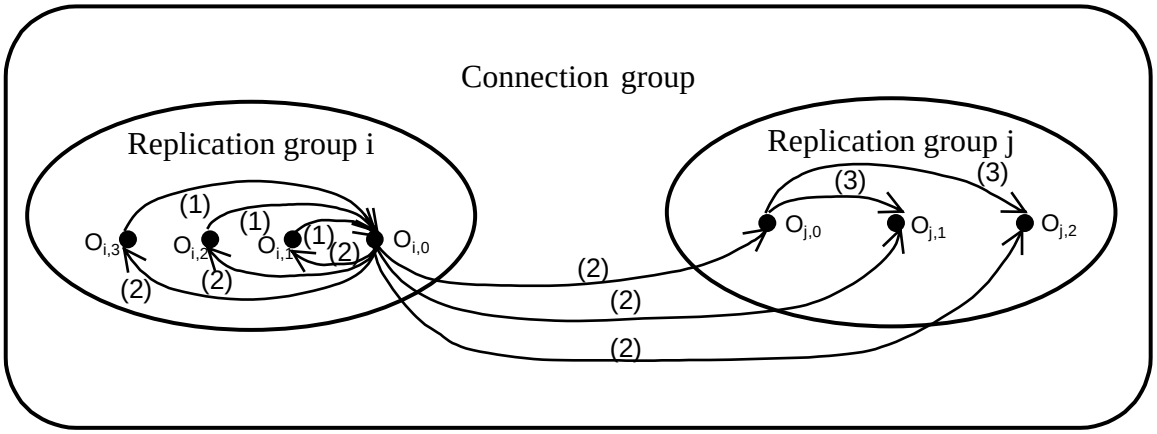

Figure 3. Pass First Communication Scheme

To send a request to the replica objects $O_{j,k}$, ($k=0, \ldots, no_j-1$), all objects $O_{i,k}$ ($k=0, \ldots, no_i-1$) use reliable point-to-point communication to send the request to the leader of the group, $O_{i,0}$ (1). Replica objects $O_{i,k}$ also keep a copy of the request in case it needs to be resent. The leader then multicasts the request in the connection group composed of the replication groups i and j (2). The objects $O_{i,k}$ of replication group i use the multicast to signal that they can delete their local copy of the request. The objects $O_{j,k}$ ($k=0, \ldots, no_j-1$) of replication group j store the multicast on a list of pending rebroadcasts. Since there can be multiple replication groups, in order to maintain total ordering of all messages within the replication group, $O_{j,0}$ multicasts the message again in the replication group j (3). The $O_{j,k}$ use the multicast as a signal that they can deliver the message and delete the previously stored copy from the connection group multicast.

After processing the request, all objects $O_{j,k}$ send the result through a point-to-point communication to the leader $O_{j,0}$. The set of steps used to transmit the request is then used to communicate the reply from replication group j to group i. In this manner, total order is

maintained among messages sent from multiple replication groups to another group, and messages are buffered in a way that tolerates crash failures during any phase of the communication.

The *leader only* communication scheme has the same protocol as the *pass first* communication scheme, except that point-to-point messages are not sent to $O_{i,0}$ from objects $O_{i,k}$ ($k=1, \ldots, no_i-1$) (1). All replicas, however, still keep a copy of the message until it is multicast within the connection group.

The current implementation of each communication scheme only tolerates crash failures. Therefore, the communication scheme chosen depends on what type of performance is best suited for the system. When the *pass first* communication scheme is used, performance may be improved if one replica responds faster than the other replicas. However, when compared to the *leader only* communication scheme, there is increased network traffic for Maestro to handle. It is generally better to use the *pass first* communication scheme with applications that are more computationally intensive. It is generally better to use the *leader only* communication scheme with applications that place higher demands on the speed of the network.

Any number of crash failures may occur in the sending and receiving replication groups during message transmission. Provided that neither replication group loses all of its members, the message will be delivered to all of the correct (non-faulty) replicas of the receiving replication group. The probability of all members crashing is small, since the dependability manager restarts replicas; the replicas would all have to crash before the dependability manager had had a chance to treat any of the faults. Tolerating crash faults in the *pass first* communication scheme is described next.

Tolerating crash failures in non-leader replicas ($O_{i,k}$ ($k=1, \ldots, no_i-1$) and $O_{j,k}$ ($k=1, \ldots, no_j-1$)) is simple using the Proteus group structure, since no message retransmission is necessary. A replica can crash before or after a multicast message (labels (2) and (3) in Figure 3) is delivered. If a non-leader replica crashes after a multicast is delivered there is no need to retransmit the message, because the message was delivered to all the correct replicas. If a non-leader replica crashes after the multicast is sent, but before it is delivered, there is no need to retransmit the message, because Maestro/Ensemble will deliver the multicast to all of the correct replicas. The other cases where a non-leader replica can crash (before and after sending a point-to-point message) also require no action from other replicas.

If $O_{i,0}$, the leader of the sending replication group, crashes before step two in the message transmission process is complete, the message transmission process is restarted once the replication group goes through a view change and elects a new leader. In this case step one is performed again. The replicas of the sending replication group, having kept a copy of the message, resend the message to the new leader of the sending replication group. The new leader of the sending replication group then multicasts the message in the connection group and the message transmission process is resumed. If the original leader dies immediately after sending the multicast, a new leader may be elected before the multicast is delivered. In that case, the new leader will also multicast the message, and this second multicast will be ignored by the members of the connection group. If $O_{i,0}$ crashes in other stages of the communication scheme, no message retransmission is necessary, since $O_{i,0}$ is not responsible for message transmission in those stages.

If $O_{j,0}$, the leader of the receiving replication group, crashes before step three in the message transmission process is complete, step three is performed once the replication group

has gone through a view change and elected a new leader. The new leader, having stored a copy of the message from the connection group multicast, then multicasts the message in the receiving replication group. If the original leader dies immediately after sending the multicast, a new leader may be elected before the multicast is delivered. In that case, the new leader will also multicast the message, and this second multicast of the message will be ignored by the members of the replication group. If $O_{j,0}$ crashes in other stages of the communication scheme, no message retransmission is necessary, since $O_{j,0}$ is not responsible for message transmission in those stages.

4.2.2. Proteus Communication Scheme

The Proteus communication scheme is used for all communication involving the dependability manager. This section describes the current implementation of the Proteus communication scheme, which does not support a replicated dependability manager. The current implementation of the Proteus communication scheme should be seen as a first step to replicating the dependability manager. If the dependability manager was not designed to be replicated in future Proteus implementations, there would be no need to use group communication to communicate with the dependability manager in the current implementation.

Sending messages to the dependability manager is done using the PCS group, of which the dependability manager is always a member. When another object wishes to send a message to the dependability manager, it joins the PCS group. Once the sending object is in the group with the dependability manager, it multicasts its message, waits for an acknowledgement, and then leaves the group. Messages are multicast, rather than sent point-

to-point, so that the sending object gets an acknowledgement that the message has been delivered to the dependability manager, and so that total ordering will be preserved among dependability manager replicas in future implementations.

The PCS group supports membership of multiple objects that wish to send messages to the dependability manager. Therefore, sending objects in the PCS group ignore messages multicast by other sending objects. Additionally, one sending object may generate more than one message that needs to be sent to the dependability manager. If more than one message needs to be sent to the dependability manger, all messages are sent once the dependability manager is in the view of the sending object. Leaving and then rejoining the group is not required.

When using Maestro/Ensemble for group communication, an object that joins a group may receive a partial view of the group before receiving the entire view of the group. In the PCS group, the sending object may initially join the group such that the group membership only includes other sending objects. In that case, if the sending object multicasts its message, the message will not be delivered to the dependability manager. Therefore, it is not sufficient for the sending object to consider itself a member of the PCS group in order to multicast its message. The object must be sure that the dependability manager is in the object's view of the group. Extra communication is required to ensure that a dependability manager is in the view of a sending object. After every view change, the dependability manager multicasts an identification message to all the members of the PCS group. After a sending object joins the PCS group, it waits to receive an identification message from the dependability manager. After receiving an identification message from the dependability manager, the sending object multicasts any messages it has to send.

Point-to-point groups are used to send messages from the dependability manager to the object factories and QuO. Each factory object and QuO object is in its own point-to-point group. When the dependability manager wishes to send a message to the receiving object, the dependability manager joins the group for that object. Once the dependability manager is a member of the group, it multicasts its message, waits for an acknowledgement, and then leaves the group. Since the group is always of size one or two, an identification message is not needed, as it was in the PCS group. If more than one message needs to be sent to the receiving object, all messages are sent once the receiving object is in the view of the dependability manager. Leaving and then rejoining the group is not required.

5. GATEWAY

This chapter presents a detailed description of the design and implementation of the gateway. Section 5.1 gives an overview of the gateway architecture. Sections 5.2 and 5.3 detail the implementations of the active replication scheme and Proteus communication scheme, described earlier in Section 4.2. Additional implementation features necessary to provide fault tolerance to AQuA applications are described in Section 5.4.

5.1. Gateway Overview

A gateway is used for communication in all AQuA objects. AQuA application objects have gateways that provide reliable communication to other AQuA application objects. The dependability manager, factory, and QuO objects have gateways that provide the type of communication needed for these object types.

The gateway is a process that intercepts calls from a CORBA object. A gateway resides on the same host as the CORBA object that it intercepts. A gateway takes IIOP messages generated from its CORBA object and transmits them using the Maestro/Ensemble group communication system to other gateways. A gateway sends messages to its CORBA object when a delivered message is in the last stage of the communication scheme in use.

The gateway is layered into the IIOP encoder/decoder, the dispatcher, and the handlers. Figure 4 shows an expanded view of the gateway architecture.

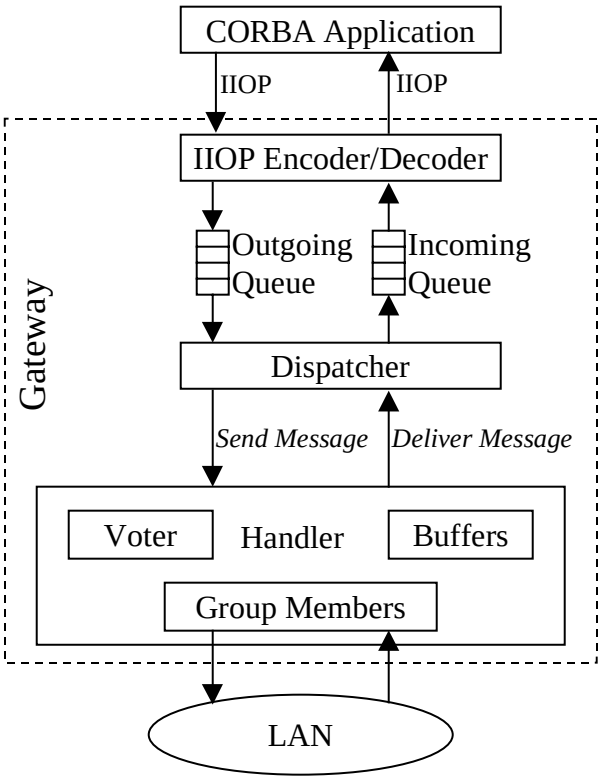

Figure 4. Expanded View of Gateway Architecture

As seen in Figure 4, the gateway communicates with its CORBA object using the Internet Inter-ORB Protocol (IIOP). This protocol is a standard for the communication format used by CORBA ORBs. IIOP thus allows CORBA applications that are developed using ORBs from different vendors to communicate with each other. For the purposes of AQuA, IIOP standardizes the interface between a gateway and a CORBA application. This allows the gateway to be used with any ORB that uses IIOP.

The *IIOP encoder/decoder*, developed at BBN, is used to interface with the CORBA object. When an IIOP stream is intercepted by the IIOP decoder, it parses the message into data structures corresponding to the various data types described in the IIOP protocol itself. It then places the decoded message onto an outgoing queue. The encoder reads messages from an incoming queue, packs the information into an IIOP stream, and sends the message to the

CORBA application. The IIOP encoder/decoder uses sockets to communicate with the CORBA application.

The *dispatcher*, developed jointly with David Karr and Dave Bakken of BBN, interfaces with the IIOP encoder/decoder and provides a set of functional features for using the handlers. The dispatcher's main function is to wrap extra information around an IIOP message dequeued from the outgoing queue and deliver the message to the correct handler. This wrapped message is called a *gateway message*. Additionally, when a handler delivers a gateway message to the dispatcher, the dispatcher strips off the wrapper and enqueues the IIOP message to the incoming queue.

The dispatcher may also store the wrapper of a gateway message delivered to it from a handler. This is done so that the wrapper to an IIOP request can be reassigned to the IIOP reply when it is dequeued from the decoder. Since oneway IIOP requests do not require IIOP replies, the wrapper is not stored if the request is oneway. A *wrapper* contains information that is useful for differentiating between and keeping track of messages. A wrapper contains the following fields of information:

- whether the message is big endian or little endian
- replication group name of the sender
- replication group name of the receiver
- sequence number
- operation code (opcode)

The dispatcher is responsible for assigning the sequence numbers that are used by the handlers. The dispatcher assigns a sequence number to each IIOP request dequeued from the decoder. Sequence number assignment is done as part of constructing the wrapper for the

gateway message. The sequence number is specific to the handler that is responsible for the gateway message. Each gateway message used by a particular handler will be assigned a unique sequence number. However, a gateway message may be assigned the same sequence number as another gateway message if they are processed by different handlers. A handler uses opcodes to keep track of the state of a message in the message transmission process. Sections 5.2 and 5.3 will demonstrate the use of sequence numbers and opcodes.

A *handler* is responsible for sending messages via Maestro/Ensemble according to one of the communication schemes described in Section 4.2. A handler is specific to two objects that wish to communicate. Currently, the active replication handler, the dependability manager handler, and the factory handler are implemented in the gateway. The active replication handler manages connection and replication group members. The dependability manager and factory handlers manage point-to-point group members and the PCS group member. Each handler is part of a "scheme." A *scheme* is composed of handlers, group members, and possibly buffers and voters. The *active replication handler* is part of the active replication scheme detailed in Section 5.2. The *dependability manager handler* and *factory handler* are part of the Proteus communication scheme detailed in Section 5.3.

Group members are used within the handlers. Group member classes are derived from the *Gateway Group Member* base class. The *Gateway Group Member* base class is derived from the *Maestro_CSX* base class. Therefore, each group member provides the ability to use a Maestro/Ensemble group. The *Gateway Group Member* base class keeps track of data that is commonly used by derived group member classes in Proteus. The data that are kept track of group-wide are the group's name, state (joining, leaving, member, non-member), size, and leader. The data specific to the group member is the member's Maestro endpoint ID and rank.

The class also defines common methods used by derived group member classes. The class provides accessor functions to the data members listed above. Base class methods are provided for the *view change callback*, *exit group callback*, *join group*, *leave group*, and *group initialization*. These base methods ensure that the base data members are correct when accessed by derived classes.

5.2. Active Replication Scheme

The *active replication scheme* is used for communicating messages between AQuA application objects. The scheme consists of a replication group member, connection group members, and handlers. The active replication handler manages all communication between two replication groups. If the application communicates in a client-server manner, then the active replication handler manages a single connection group member between the two replication groups. If both replication groups can initiate requests (act as clients), the active replication handler manages two connection group members between the replication groups. The handler only defines a method to use when a message is given to it by the dispatcher. The replication group member and connection group member have callbacks that handle the other steps in the message transmission process.

The scheme's replication group member is an implementation of the replication group described in Section 4.2.1. The replication group member manages the buffers used in the communication scheme. There is one replication group member shared by all of the active replication handlers within a gateway. The scheme's connection group member is an implementation of the connection group described in Section 4.2.1. The connection group manages a voter used in its scheme. Figure 5 is an Object Model Template (OMT) diagram

[13] that describes the classes used in the active replication scheme. This diagram is a useful reference for understanding the relationships between different classes in the implementation. Each of the active replication scheme components will be discussed in the following subsections.

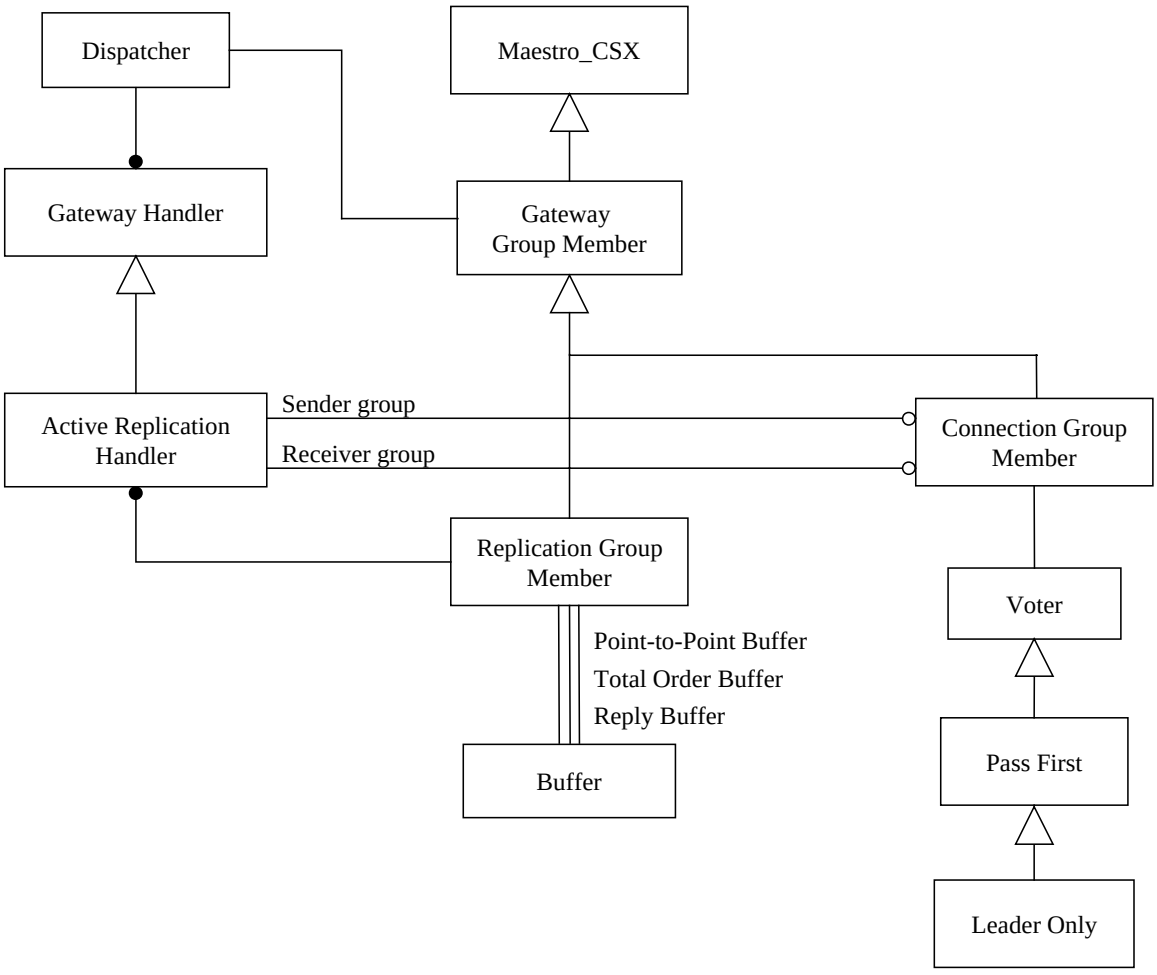

Figure 5. OMT Diagram for Active Replication Scheme

5.2.1. Buffers

Buffers are used in the active replication scheme to provide tolerance to crash failures and to ensure request-reply ordering. A gateway message is stored in a buffer before it is sent or cast. Once there is some form of acknowledgement that the message is no longer needed, it

is removed from the buffer. All the buffers have the same data structure, which could easily be used with other replication schemes.

The buffer data structure is a doubly linked list of gateway messages. Methods are provided to add a message, remove a message, get the buffer size, and iterate through the buffer. There are also methods to get and set the buffer state that are used during state transfer. When messages are added, they are always added to the front of the list. A message is removed from the buffer based on its wrapper. If the wrapper matches, then the message is removed and returned to the calling method. The remove function starts searching at the end of the linked list. That provides an efficient implementation, since the message is more likely to be at the end of the linked list than the beginning, based on its use in the active replication scheme.

There are three buffers in the active replication scheme. The first one, the *point-to-point buffer*, buffers gateway messages generated by the CORBA application associated with the gateway. The message is removed from the buffer after the message has been cast in the connection group. The second buffer, the *total ordering buffer*, buffers messages within the replication group members receiving the message. The message is removed from the buffer once it has been rebroadcast within the replication group to achieve total ordering. The final buffer, the *reply buffer*, buffers messages right before they are delivered to the dispatcher. A message is placed in the reply buffer only if the application is not ready to receive the message because it has not generated the corresponding request. This may happen if the replica lags behind other members of its replication group. In that case, the message is not delivered to the dispatcher until the application has generated messages of its own to place it in the correct state.

5.2.2. Voters

A *voter* is an abstract data type used to specify a communication configuration for a replication scheme. Replicas in the active replication scheme may communicate in a variety of ways. The voter that the active replication scheme uses specifies the communication configuration to use. The name voter may seem inappropriate, since it doesn't appear to encompass all of the data type's purposes. However, if the choice of whether or not to submit a vote and the process by which representatives are elected are considered part of voting, then the name is appropriate.

Note that while a replication scheme is specified for a replication group, a voter is specified for a connection group. This allows different voters to be used for the connection groups associated with a replication group. The voter interface described below, though not designed for other replication schemes, may prove to be sufficient for some of them, such as passive replication.

The abstract voter interface is designed to allow flexibility in the types of voters that may be defined. The methods to the interface are:

- *process* – This method is called at the leader of a replication group after receipt of a sent message. It determines which message to cast in the connection group.
- *reset* – This method is called to reset the voter state.
- *set group size* – This method, not used by either of the current derived voters, will be used by voters that tolerate value faults.
- *non-leader sends* – This method determines whether or not members that are not the leaders of the replication group should send point-to-point messages to the leader.

What these methods do depends on the type of voter they are associated with. In particular, recall that the purpose of the *pass first* voter is to inform its connection group member to cast the first message of a given sequence number. To accomplish this, the voter keeps track of the sequence numbers of the messages it has seen. The *process* method returns the first message of a given sequence number. That indicates that the connection group member should cast the message. When a sequence number is seen again, the message is deleted and no message is returned to the connection group member. That indicates that the connection group member should not perform a cast. The *reset* method simply resets the voter to its initial state of having seen no sequence numbers. The *set group size* method performs no operation and the *non-leader sends* method returns *true*.

Similarly, the *leader only* voter implements the *leader only* communication scheme for synchronous and deferred synchronous communication. *Leader only* communication occurs when replicas are actively replicated, but only the leader sends out messages. This voter's implementation is derived from the *pass first* voter. The only difference is that the *non-leader sends* method is overloaded to return *false*.

5.2.3. Active Replication Scheme Handler

When a message generated by the CORBA object reaches the dispatcher, the message is wrapped and given to the active replication handler specific to the destination replication group. The dispatcher gives the message to the active replication handler by calling the handler's *send message* method. This starts the message transmission process (step one in Figure 3). Figure 6 details the algorithm used by the handler's *send message* method.

```
set message opcode
set last sent sequence number
add to point-to-point buffer

if (voter.non-leader sends) or (replication group leader)
      convert gateway message to Maestro message
      send Maestro message to replication group leader

if matching message in reply buffer
      remove message from reply buffer
      deliver message to dispatcher

if last casted sequence number > message sequence number
      remove message from point-to-point buffer
```

Figure 6. Active Replication Handler: Send Message Algorithm

When a message is given to the handler by the dispatcher, the opcode is set and the message is placed in the point-to-point buffer, as seen in Figure 6. A generated message is sent to the leader of the replication group using point-to-point communication if the communication scheme calls for it. Additionally, if the fact that a message was generated indicates that it is ready to receive a buffered reply message, then that message is sent to it. Finally, if the message has already been cast in the connection group by another replica, the message is removed from the point-to-point buffer. After initiating this first step, the group member callbacks handle the remainder of the message transmission process.

5.2.4. Replication Group Member

The replication group member is the implementation of a group member's endpoint within a replication group. The replication group member defines Maestro callbacks for events in the active replication scheme that are associated with a replication group. In particular, the *receive send* and *receive cast* callbacks are called when a point-to-point message and a multicast message are delivered to a group member. The *view change* callback is used to resend messages if a failure occurs during the message transmission process. There

are also two state transfer callbacks that are used to transfer the state of an existing replica to a new replica. These are detailed separately in Section 5.4.1.

Receive Send

The *receive send* callback is called at a group member when a point-to-point message has been delivered to it. This occurs after a handler's send message method sends a point-to-point message to the leader of the replication group (step one in Figure 3). In the current communication schemes, point-to-point messages are only sent to the leader of a replication group. This callback initiates step two of the message transmission process described in Figure 3. Figure 7 describes the algorithm used when this callback is called.

```
if leader of replication group
      set opcode for message
      message = voter.process
      if message is not blank
            if last casted sequence number < message sequence number
                  convert gateway message to Maestro message
                  cast message within connection group
```

Figure 7. Active Replication Group Member: Receive Sent Message Algorithm

The callback forwards the message to the voter. The voter acts on the message as necessary to the communication scheme defined. If the voter returns a message, it means that this new message should be cast in the connection group. This new message may be the message just passed in to the voter, or could be a message that the voter has stored previously. If no message is returned by the voter, nothing else has to be done. The next callback used in the message transmission process is the connection group member's *receive cast* callback, described in Section 5.2.5.

Receive Cast

The *receive cast* callback is called at a group member when a multicast message has been delivered. A message is cast in the replication group only if its next step is to be delivered to the application (completion of step three in Figure 3). Figure 8 describes the algorithm used in the callback. If the message hasn't already been seen, then it is either delivered to the dispatcher or placed in the reply buffer. If the message is placed in the reply buffer, a new message must be generated by the application before the message can be delivered to the dispatcher.

```
if message sequence number > last delivered sequence number
      set last delivered sequence number
      if correct replica state
            deliver message to dispatcher
      else
            add message to reply buffer
      remove message from total order buffer
```

Figure 8. Active Replication Group Member: Receive Cast Message Algorithm

View Change

The *view change* callback is called at a replication group member when the view of the group membership changes. This occurs either when a new replica joins the group or when a replica that is already within the group crashes. The *view change* algorithm is described in Figure 9. If the replication group has a new leader, then any message in the point-to-point buffer is resent to the new leader if the voting scheme calls for it. If the group member is the leader of the group, then any message in the total order buffer is multicast again within the replication group. If the size of the group has changed, then a view change message is

generated and the dispatcher is scheduled to report it. Scheduling view change reports with the dispatcher is detailed in Section 5.4.2.

```
if new leader in group
      for each message in the point-to-point buffer
            if (voter.non-leader sends) or (group leader)
                  convert gateway message to Maestro message
                  send Maestro message to group leader

if leader of group
      for each message in the total order buffer
            convert gateway message to Maestro message
            cast Maestro message within replication group

if size of group has changed
      for each connection group
            set voter size
      if leader of group
            generate view change message
            schedule dispatcher to report view change
```

Figure 9. Active Replication Group Member: View Change Algorithm

5.2.5. Connection Group Member

The connection group member is the implementation of a group member's endpoint within a connection group. The Maestro callback defined by the connection group member is the *receive cast* callback. This callback is called after the replication group member's *receive send* callback has multicast a message within a connection group (completion of step two in Figure 3).

Receive Cast

Figure 10 contains the algorithm used by the connection group member when the *receive cast* callback is called. When a message is multicast in the group, the action that is taken depends on whether the group member in which the callback occurs is in the sending replication group or the receiving replication group. If the group member is in the sending

replication group, then the message is marked as being cast and is removed from any point-to-point buffers that store it. If the cast is received in the receiving replication group, then the message is placed in the total order buffer. If the group member is the leader of the receiving replication group, then the message is multicast within the receiving replication group. The next step in the message transmission process occurs when the replication group multicast is received in the replication group member's *receive cast* callback.

```
if member of sending replication group
      set last casted sequence number
      if message is in point-to-point buffer
            remove message from point-to-point buffer

else // must be member of receiving replication group
      set message opcode
      add message to total order buffer
      if replication group leader
            convert gateway message to Maestro message
            cast Maestro message within replication group
```

Figure 10. Active Connection Group Member: Receive Cast Message Algorithm

5.3. Proteus Communication Scheme

The Proteus communication scheme is used for communication involving the dependability manager. The scheme consists of the dependability manager handler, the factory handler, the PCS group member, and the point-to-point group member. The dependability manager handler is used by the dependability manager's gateway to communicate with an object factory or QuO. There is one dependability manager handler instance per factory or QuO object in the system. The factory handler is used by the gateways that interface with a factory or QuO object, and is used to communicate with the dependability manager. A PCS group member and the point-to-point group member are used by the

handlers to cast messages reliably. Messages are sent to the dependability manager using the PCS group. The PCS group member is shared among each dependability manager handler in a gateway. When view changes are reported, the dispatcher uses the PCS group member without using a handler. Messages are sent to the factories using the point-to-point groups. Figure 11 shows an OMT diagram [13] of the Proteus communication scheme. Each of the Proteus communication scheme components will be discussed in the following subsections.

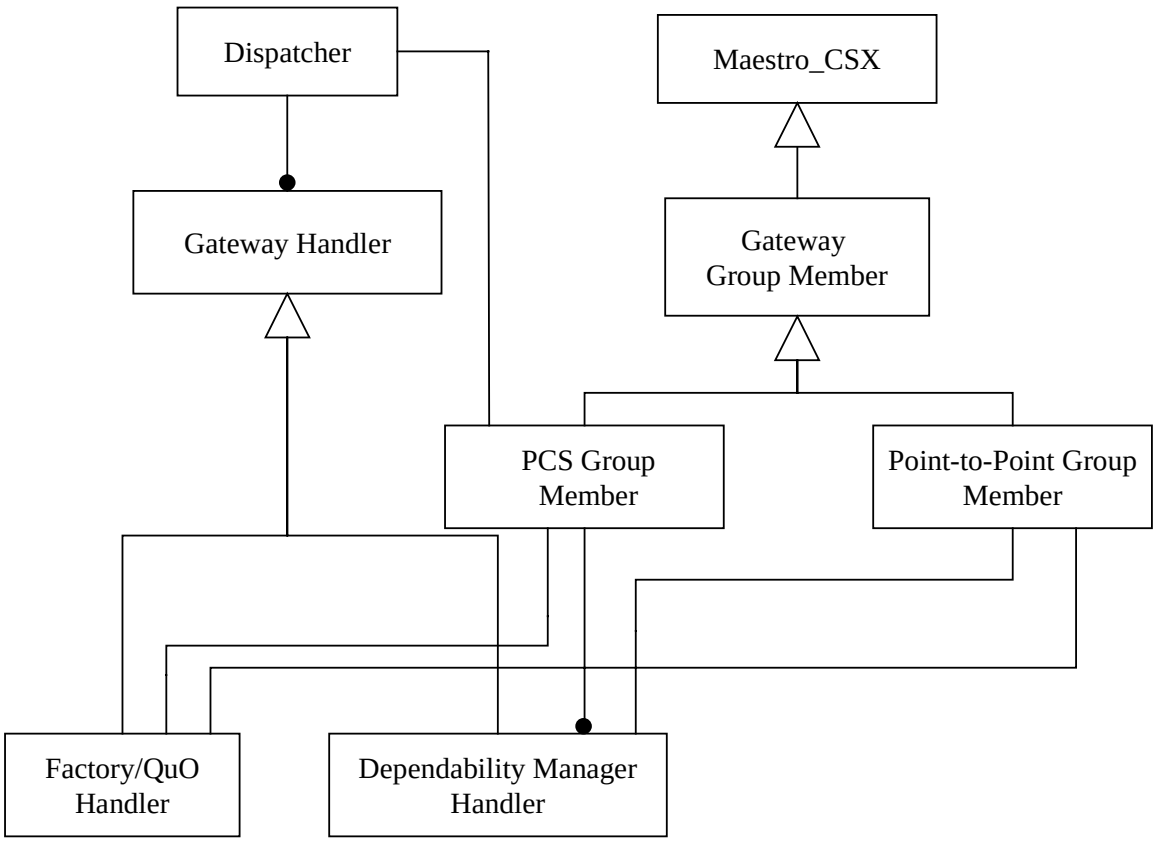

Figure 11. OMT Diagram for Proteus Communication Scheme

5.3.1. Dependability Manager Handler

The dependability manager handler is used in the dependability manager's gateway to communicate with a factory or QuO. The *send message* method for this handler is called

when the dependability manager wants to send a message to a factory or QuO. The method simply sets the appropriate opcode for the message, joins its point-to-point group, and then casts the message via the point-to-point group member. The responsibility of ensuring that the message is delivered is then placed on the point-to-point group.

5.3.2. Factory/QuO Handler

The factory/QuO handler, which is similar to the dependability manager handler, is used in the gateway of a factory or QuO object to communicate with the dependability manager. The main difference is that it uses the PCS group member instead of a point-to-point group member. The *send message* method for this handler is called when a factory or QuO wants to send a message to the dependability manager. The method simply sets the appropriate opcode for the message, joins the PCS group, and then casts the gateway message via the PCS group member. The responsibility of ensuring that the message is delivered is then placed on the PCS group.

5.3.3. PCS Group Member

The PCS group member is the group member used in the gateway to send messages to the dependability manager. The dependability manager is always in the PCS group. Whenever another gateway wishes to send a message to the dependability manager's gateway, it joins the group, casts its message, and leaves the group. The dynamic nature of the group and the diversity of its members require that the communication scheme exhibit special properties. There are two types of message cast within the PCS group. A *functional message* is a gateway message that is to be delivered to the dependability manager. An *identification message* is a message multicast by the dependability manager to identify itself to other group

members. The idea is that after every view change, the dependability manager casts an identification message indicating that it is in the group. When a group member that wishes to communicate with the dependability manager sees this identification message, it casts any messages it is waiting to send to the dependability manager (see Section 4.2.2). The PCS group member is given a gateway message to cast (by a handler or dispatcher) through the *gateway message cast* method. The group member also defines Maestro callbacks for the *receive cast* and *view change* callbacks.

Gateway Message Cast

The *gateway message cast* method is used to cast a gateway message within the PCS group. Messages multicast by the group's callbacks, on the other hand, are Maestro messages. These multicasts are performed by directly using Maestro multicast. Figure 12 describes the algorithm for this callback. If the dependability manager is in the view, then the message is multicast within the group. A counter of the number of messages in transit is used so that this group member knows when to leave the group. If the dependability manager is not in the current view, the message is placed in a queue of pending messages.

```
if dependability manager in current view
      convert gateway message to Maestro message
      set Maestro message type to functional message
      increment number of messages in transit
      cast Maestro message within PCS group
else
      add message to pending message queue
```

Figure 12. PCS Group Member: Gateway Message Cast Algorithm

View Change

The *view change* callback occurs when a group member either joins or leaves the group. Figure 13 describes the algorithm for this callback. If the group member is the dependability manager, then it multicasts an identification message to the entire group. If the group member is not the dependability manager, then it resets its variable that keeps track of whether the dependability manager is in the view.

```
if dependability manager
      create Maestro message
      set Maestro message type to identification message
      cast Maestro message within PCS group

else  // must be factory, application, or QuO gateway
      set dependability manager in current view to false
```

Figure 13. PCS Group Member: View Change Algorithm

Receive Cast

The *receive cast* callback is called when a functional or identification message is received by the group member. Figure 14 describes the algorithm used in this callback. If the group member is the dependability manager, it delivers any functional messages it receives and ignores any identification messages received. If the group member is not the dependability manager and the message is an identification message, the group member promptly multicasts any pending messages within the group. If the group member is not the dependability manager and the message is a functional message, the group member decrements its record of the number of messages in transit (if it sent the message). If, as a result, there are no more messages in transit, the group member schedules the dispatcher to leave the group (see Section 5.4.2).

```
if dependability manager
      if functional message
            deliver message to dispatcher

else // must be factory, application, or QuO gateway
      if functional message
            if message origin is this replica
                  decrement number of messages in transit
                  if messages in transit equals zero
                        schedule dispatcher to leave group

      else // must be identification message
            set dependability manager in current view to true
            add pending queue size to number of messages in transit
            for each message in the pending message queue
                  convert gateway message to Maestro message
                  set Maestro message type to functional message
                  cast Maestro message within PCS group
```

Figure 14. PCS Group Member: Receive Cast Algorithm

5.3.4. Point-to-Point Group Member

The point-to-point group member is the group member used in the gateway to send messages to factory and QuO objects. The point-to-point group member has algorithms similar to the PCS group member. The differences in the communication will be described. Instead of the dependability manager gateway being a member of the group all the time, the factory gateway or QuO gateway is a member of the group all the time. The dependability manager gateway joins the group when it needs to send a message and leaves once the message transmission is complete. That means that there are several groups of size one in the system; when the dependability manager wishes to use one of them, the size of the group becomes two. This is more scalable than a single large group that contains all the possible group members in the system.

Since the group size is at most two, the *cast* callback and *view change* callback use the group size as a test of whether or not the factory or QuO gateway is in the view. This eliminates the need for an identification message, since if the group size is two, the group

must be composed of one factory or QuO gateway and one dependability manager gateway. It also eliminates the need for the dependability manager to check whether or not it originated a cast message.

5.4. Additional Features

This section discusses other functional features that are provided in the current implementation of the gateway. In particular, the dispatcher and active replication group member work together to provide state transfer for AQuA objects (Section 5.4.1). In order to have dynamic group membership, the dispatcher schedules the joining and leaving of groups (Section 5.4.2). A synchronization queue is implemented within the dispatcher to work around non-determinism problems in certain ORBs (Section 5.4.3). Finally, atomic behavior between a gateway and its CORBA object is ensured using signals (Section 5.4.4). Each of these provides important functionality to applications using AQuA.

5.4.1. State Transfer

To properly implement state transfer, the gateway needs to be able to transfer not only its state, but also the state of the CORBA object. To transfer its state, the gateway must provide methods to get and set the state of the dispatcher and the handlers. In order to transfer the state of the object, the object must add *get state* and *set state* methods to its interface. The dispatcher will call these methods to transfer an object's state.

State transfer is performed when the *state transfer* replication group member callback is called in a new group member. It asks for the state of an existing member using a blocking Maestro *get state* call. Note that the Maestro *get state* call is different from the *get state* method for the object. All communication with the gateways of a replication group is blocked

until state transfer is complete. After receiving the state of another member, *the state transfer* callback sets the state of the new member. Setting the state of the new member requires setting the state of the dispatcher, the handlers, and, if necessary, the object. If the object state needs to be set, an IIOP *set state* request is sent to the object.

The *ask state* callback of the replication group is called in the group leader once a new group member makes a Maestro *get state* call to the replication group. Using the group leader to transfer state is simply an implementation choice, since any group member could be used under the virtual synchrony model. The *ask state* callback gets the current state of its dispatcher and handlers. If the application does not have state, then the *ask state* callback makes a call to Maestro *send state*. This sends the state of the dispatcher and handlers to the gateway that requested the state. If the application has state, then an IIOP message invoking the application's *get state* method is generated and sent to the application through the dispatcher. The application *get state* method is synchronous, and the reply returns the state of the application. The dispatcher intercepts the application's *get state* reply and generates an IIOP message for the application's *set state* method from the state returned in the *get state* reply. At this point, the dispatcher calls Maestro *send state*. This sends the state of the dispatcher, the state of the handlers, and the generated *set state* method to the gateway that requested its state.

5.4.2. Reporting View Changes & Leaving Groups

Maestro requires that calls to join and leave groups be done outside of a group member callback. In C++ terms, they can be thought of as public functions that cannot be called within a group member method. In the case of the gateway, that means that joining and

leaving the PCS group and point-to-point groups cannot be done within a replication, connection, PCS, or point-to-point group callback. The problem is that within these group member callbacks, the gateway may decide it needs to join or leave a group. Therefore, the code within a group member callback needs to be able to schedule the joining or leaving of groups.

The code to schedule dynamic group joins and leaves is placed within the dispatcher, since the dispatcher runs the gateway's main execution loop. When a group member callback wants the gateway to leave a group, it schedules the group to leave by using the dispatcher's *add to leaving* method. The *add to leaving* method simply adds the group to a list of groups waiting to leave. Most of the dynamic group joins are performed in the handlers and therefore do not need to be scheduled with the dispatcher. The one exception occurs when a view change needs to be reported by an active replication group. In that case, the interface to the dispatcher is not an add to joining method, but a *report view* method. This method places a view change message in a buffer so that it can be cast in the PCS group later.

Using this structure, requested group leaves and joins are carried out by the *update groups* method from the main execution loop. The *update groups* method simply calls the *leave* method on any group member that has been scheduled to leave its group. Additionally, if there is a view change message waiting to be sent, the PCS group is joined, and the view change message is cast within the PCS group.

5.4.3. Synchronization Queue

As explained earlier, the active replication scheme requires determinism. A problem with using a threaded ORB such as VisiBroker in AQuA is that it can cause non-deterministic

behavior in the CORBA object. Even if the object is written to these guidelines, the object may behave non-deterministically if it uses a threaded ORB. A threaded ORB will generate a thread to process each message. To ensure determinism, only one message must be processed at a time. One potential way to do this is to set the number of available threads in the ORB to one. If this is done and a second message appears, it is placed in a scheduler and scheduled once the first message has been processed completely. However, if two or more messages are in the ORB's scheduler, the scheduler may pick either message to process next. This will disrupt the total ordering of the messages if schedulers from different replicas pick different orders. Thus, simply setting the number of available threads to one will not work.

A correct solution is to add a synchronization queue within the dispatcher. The synchronization queue works in a simple manner. It queues messages if the object is suspected to be busy processing a message. Once the object is ready to process another message, a message in the queue is dequeued and delivered to the encoder to deliver to the object. Deciding when the object is ready to process another message is the difficult part. In the current implementation, the dispatcher decides that the object is ready to process a new message once the object has generated a message of its own. This prevents the scheduler from ever having two messages to choose from. This works very well for objects that communicate synchronously. It will not, however, work for objects using general asynchronous communication, since requests and replies do not necessarily come in pairs. However, if the object only communicates with one replication group, looking for replies will work for objects using deferred synchronous communication, since requests and replies can be matched. Objects that use deferred synchronous communication with more than one replication group

and use a non-deterministic ORB cannot be replicated in the current implementation. A more general solution is suggested in Section 6.2.

5.4.4. Atomic Behavior

In order to tolerate and detect crash failures properly, it is necessary for the gateway and the CORBA object to behave atomically with respect to crashes. Specifically, if the object process crashes, the gateway should kill itself. Also, if the gateway crashes, the object process should automatically die. This section describes how crash atomicity is achieved in the current implementation.

Crash atomicity is implemented using UNIX-specific features. The CORBA object process is forked off from a gateway process when the gateway is started. That makes the object process a child process and the gateway process its parent. In UNIX, when a child process crashes, a SIGCHLD signal is sent to its parent indicating that the child has died. To ensure that the gateway kills itself upon the crash of the object process, a signal handler is defined for the SIGCHLD signal. If this signal is received, the gateway terminates itself.

Similarly, the gateway process receives a signal before it is terminated. The signal received may be one of many types. These signals are caught by another signal handler in the gateway. If one of these signals is caught, the gateway terminates the object process and then terminates itself. The SIGKILL signal, however, cannot be caught by a signal handler; the object process may continue to exist without its gateway. However, Proteus will still detect a process crash due to the crash of the gateway. The orphan object process will continue to run, occupying virtual memory, but not causing dependability problems, since it will be unable to

communicate without its gateway. Orphan processes should not occur often in practice, since the SIGKILL signal is unlikely to be the cause of the crash if the crash was not intentional.

6. RESULTS, CONCLUSIONS, AND FUTURE WORK

This chapter describes results obtained from testing the current implementation of Proteus. The conclusions from the research, and future research that can be done with Proteus, are also described.

6.1. Experimental Results

All implementations were tested on a 100 megabit/second LAN with a common shared hub. There were five hosts running Red Hat Linux on the LAN. Each host contained a 233 MHz Intel Pentium II processor and had 128 MB of RAM.

6.1.1. Dependability Manager and Object Factory

The dependability manager and object factories were tested by starting from a state of no running application replicas, using the dependability manager's Graphical User Interface (GUI). Figure 15 shows screens from the dependability manager GUI, implemented by Jennifer Ren. Using the region editor window of the dependability manager, QuO regions were defined for the test applications detailed in Section 6.1.2.

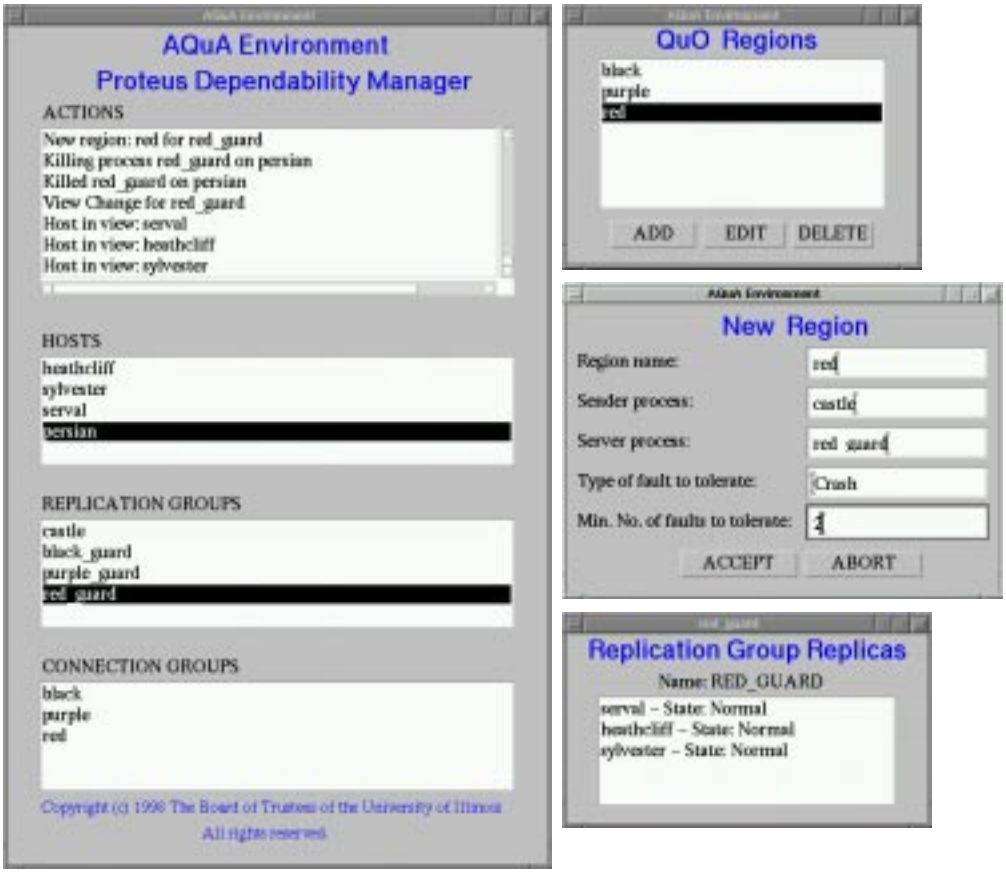

Figure 15. Dependability Manager GUI

The following tests were conducted:

- Adding, editing, and deleting regions by specifying the minimum number of crash failures to tolerate.
- Adding, editing, and deleting regions that share a replication group with other regions.
- Specifying the minimum number of crash failures to tolerate between 1 and 4.
- Inducing crashes and specifying regions in different states of the dependability manager.
- Simultaneously crashing replicas.
- Crashing a single host at a time.
- Restarting an object factory after a host crash.

The following results were observed:

- Dependability manager correctly commanded the starting of new replicas or killing of existing replicas when a region was added, edited, or deleted.
- Dependability manager correctly commanded the starting of new replicas when single and multiple crashes occurred.
- Dependability manager correctly chose the least-loaded hosts to start replicas.
- Dependability manager correctly made a callback to QuO when the minimum number of faults to tolerate could not be satisfied.
- Object factories correctly started or killed replicas when commanded by the dependability manager.
- Object factories correctly reported the host load to the dependability manager at a regular time interval.
- Object factories correctly restarted after a host crash.

The observed results show that the dependability manager and object factory implementations work as detailed in this thesis.

6.1.2. AQuA Applications

Two applications were used to test the ability of Proteus to support the active replication scheme. This section first gives a description of each application and the qualities it has that are useful for testing Proteus. The conducted tests and observed results are then described.

Application Descriptions

The first application is a simple distributed counter application. It is composed of two CORBA objects: *counter client* and *counter server*. The counter server processes requests to *update* its state, an integer variable, and returns the new state of the variable. The *update* method has an integer argument as input. The new value of the variable is the result of adding the argument to the old value of the variable and multiplying that result by the argument. The counter client makes synchronous requests to *update* the state of one or two counter server objects. The counter client prints out the value returned by a request made to a remote counter server.

This application exhibits the following qualities that were necessary to properly stress Proteus:

- Counter client and counter server are deterministic and can be actively replicated.
- Counter server requires state transfer for new replicas.
- Counter server requires total ordering to maintain state consistency between replicas.
- Counter client and counter server can both use multiple connection groups.
- Synchronous communication is used.

The second application is a distributed castle-guarding application developed by Paul Rubel. There are two types of CORBA objects used in the application: *castle* and *guardian*. The castle object displays a game board of a castle, guardians, and raiders. Figure 16 shows the game board. The goal of a raider is to reach the castle, and the goal of a guardian is to prevent raiders from reaching the castle by impeding their paths. The castle object controls all raider movement and each guardian object controls the movement for one guardian. The castle object sends out information to a guardian about the position of the raiders and the other

guardians on the game board. A guardian responds by sending a new position on the board to occupy.

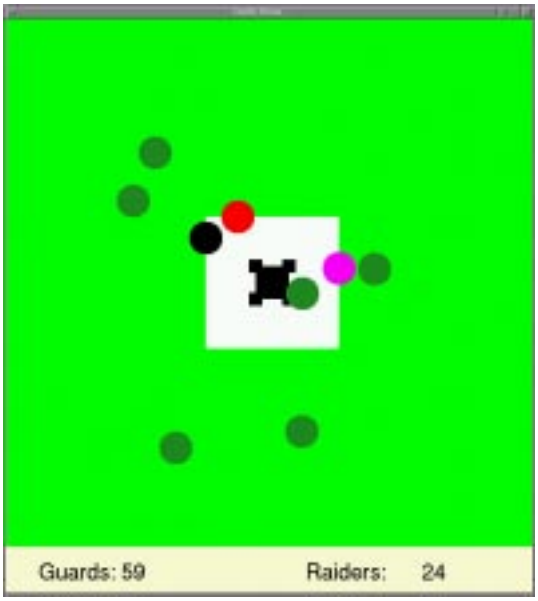

Figure 16. Castle-Guarding Application

This application exhibits the following qualities that were necessary to properly stress Proteus:

- The castle object is non-deterministic.
- The guardian object is deterministic and can be actively replicated.
- The castle object uses multiple connection groups.
- Deferred synchronous communication is used.

Tests and Results

For the counter application, tests were conducted using two counter client replication groups and two counter server replication groups. There was one connection group between

each client-server pair for a total of four connection groups. For the castle-guarding application, tests were conducted using a single replica for the castle object. Guardians were replicated, and there was one connection group per guardian type (purple, red, and black). Since both applications used the active replication scheme, they could be tested in the same manner. The following tests were conducted for each application:

- Using different replication group sizes.
- Inducing crashes and starting new replicas in each stage of the active replication scheme.
- Simultaneously crashing old replicas and creating new replicas.
- Using both the *pass first* and *leader only* communication schemes.
- Running the application with and without a dependability manager.

The following results were observed in both applications:

- Replica state consistency within a replication group.
- Delivery of all sent messages.
- Correct behavior (state consistency and delivery of all messages) with or without a dependability manager.
- New replicas received correct state.

The observed results show that the active replication scheme implementation works as detailed in this thesis.

6.2. Conclusions and Future Work

Proteus provides fault tolerance via software replication to CORBA applications that use the AQuA architecture. Proteus provides an infrastructure for continued operation in spite

of faults, error detection, and fault treatment. It also provides the ability to dynamically specify the desired level of dependability and notify QuO when the desired level of dependability cannot be satisfied. The design is flexible to allow for a variety of functional extensions to be added to the infrastructure.

The Proteus infrastructure is composed of gateways, factories, and dependability managers. Gateways are used to provide reliable communication for AQuA application, QuO, factory, and dependability manager objects. The gateway separates fault tolerance from the functional features of a CORBA object. The dependability manager provides fault treatment and dynamic adaptation to QoS requests from applications. The factories can start and kill processes throughout the system and provide host information to the dependability manager.

The current implementation of Proteus provides toleration to crash failures for AQuA applications. These applications can use synchronous or deferred synchronous communication and can use either the *pass first* or *leader only* communication scheme. The dependability manager provides the ability to treat crash failures, and also communicates with QuO to allow dependability requirements to be specified dynamically. The factory implementation is able to start and kill a process on a host and provides information about a host to the dependability manager.

A scalable group structure is defined to allow Proteus to be used for applications that have many different types of objects. This group structure also provides a dynamic communication scheme to a centralized component that may need to be replicated (currently, the dependability manager). As new replication schemes and communication schemes are defined for AQuA applications (e.g., passive replication and majority voting), they can be implemented using this group structure and still maintain scalability. Additionally, if new

centralized components are added to Proteus, they can be implemented using the dynamic group membership ideas from the communication scheme currently used to communicate with the dependability manager. These centralized components can then be reliably replicated without losing scalability.

The designs of the gateway, dependability manager, and factory are object-oriented and layered. The design allows new components that replace or add on to old components to be added with minimal integration effort. This is because common interfaces are defined and methods that could be useful with other features are already provided. The implementation was developed with particular future goals in mind so that Proteus would not become narrow in its scope. The following additions were considered in the software design when implementing the current version of Proteus:

- In order to tolerate application-level value faults for the active replication scheme, a majority voter can be developed. This voter can use the voter interface described in the implementation of the active replication scheme (Section 5.2.2). Therefore, only a new voter needs to be defined, and none of the other code providing communication needs to be changed. In order for the dependability manager to treat value faults, new methods must be defined in the protocol coordinator and advisor so that value faults can be reported and treated.
- To allow non-deterministic applications to be supported, the passive replication scheme can be implemented. This requires implementation of a new scheme within the gateway. The dispatcher and IIOP encoder/decoder from the current implementation can be used without change. The base classes that have already been defined for handlers and group

members can be abstracted from when the derived classes for the passive replication scheme are being defined.

- Supporting applications that use general asynchronous communication in the active replication scheme will greatly extend the capabilities of Proteus. The current implementation of the active replication scheme does not support general asynchronous communication, because it does not have the requisite complexity to maintain total ordering under those conditions. Adding this capability will require development of more sophisticated data structures to keep track of the sequence numbers assigned to each message, but can be done within the scheme structure described in this thesis.
- A host failure detection mechanism would prove useful to treating faults. The detection mechanism could be based on whether or not a factory has sent an update of the host load recently. With an unreliable host failure detection mechanism (it cannot be known whether a host has crashed or is merely slow), the dependability manager could make pessimistic assumptions about the system to ensure that the desired QoS is met.
- The advisor can be developed independent of the protocol coordinator. This allows more complex policies for diagnosing faults to be integrated into the advisor, while the mechanism by which those decisions are carried out can remain the same.
- Although replication of the dependability manager is not complete, the design and implementation of the Proteus communication scheme is an important step in replicating the dependability manager. The fact that the dependability manager uses a gateway with dynamic group membership ensures that future Proteus implementations in which the dependability manager is replicated will remain scalable. Since the dependability manager

is non-deterministic, it cannot be actively replicated. Therefore, either passive replication or another replication scheme will be needed.

- The current implementation of Proteus assumes that when processes leave a Maestro/Ensemble group, they leave because they have crashed, and that if they join a Maestro/Ensemble group, they do so because they have just been started. These assumptions lead to the broader assumption that groups will not partition spuriously and that the LAN itself is not partitionable. The Proteus implementation can be extended to support group partitions. Maestro/Ensemble already handles groups that might partition and remerge. The code within Proteus that needs to be upgraded is that of the group member callbacks for view changes and state transfer. The other parts of the implementation should not require modification.
- The current implementation of Proteus does not allow for dynamic reconfiguration of the faults to tolerate and the scheme used to tolerate faults. Protocols to switch dynamically between replication schemes and fault types can contribute to research in adaptive fault tolerance. These protocols would also provide additional adaptation mechanisms for AQuA applications that need to dynamically switch the type of faults to tolerate. Proteus currently creates all connection groups possibly needed by a gateway when the gateway starts up. A more dynamic feature would be to have the connection groups be created only when an object makes a request for another object's QoS. Allowing these dynamic capabilities entails using protocols that reconfigure gateways through the dependability manager. The protocol coordinator implementation for switching between different replication schemes can be developed independent of the advisor implementation to decide when to switch between replication schemes.

There are some other features that are future goals of Proteus. Although there are not mechanisms in current software implementation to assist in the development of these features, the layered design clearly defines the points where software integration will be needed. These features include the following:

- In order to tolerate time faults, monitors can be placed in the gateway. These monitors will use time-driven and event-driven entry points to monitor the performance of objects. The use of monitors will have to be coordinated with the use of voters. As in the case of value faults, the protocol coordinator and advisor must define new methods so that time faults can be reported and diagnosed.
- Closer integration between the application and the gateway would provide a better solution to the problem of non-determinism in a threaded ORB. If the application simply sent a message to the gateway after it finished processing an invocation, the gateway would know when the application is ready to receive another message. This would require changes to the CORBA application code, but the changes would not require the application developer to implement fault tolerance. Since the synchronization queue does not work for asynchronous communication, closer integration between the application and gateway will be needed in future implementations.

APPENDIX A: STARTUP PARAMETERS

Table 1. Gateway Startup Parameters

Parameter	Description
-debug	Prints out debug messages. Used with the environment variables AQUA_GW_DBG_FLAGS and IIOP_GW_DBG_FLAGS.
-timeout	Next argument specifies timeout for select statement in IIOP encoder/decoder.
-internal	Next argument should always be “aqua”. (Gateway can also be used for DIRM).
-iiop_port	Next argument specifies the port number to listen on for IIOP requests.
-gw_port	Next argument specifies the port number to use for the IIOP encoder/decoder.
-app_ior	Next argument specifies the ior file for the CORBA object in the application.
-aqua_key	Next argument specifies the name of the replication group.
-aqua_other	Next argument specifies the name of another replication group to set up two connection groups with. This is used when both replication groups may initiate requests.
-aqua_client	Next argument specifies the name of another replication group to set up a connection group with. This is used when the gateway’s application only acts as a server to the other replication group.
-aqua_server	Next argument specifies the name of another replication group to set up a connection group with. This is used when the gateway’s application only acts as a client to the other replication group.
-defaultHandler	Next argument specifies the default handler to use. 0 is the Dependability Manager handler, 1 is the Factory handler, and 2 (default) is the active replication handler for an application.
-hasState	Specifies that application-level state transfer must be done in conjunction with gateway state transfer.
-synchronize	Specifies that synchronizing the messages to the application must be done in order to achieve deterministic behavior in the application. Works for applications that use synchronous communication. Also works for applications that use deferred synchronous communication and connect to just one replication group.
-leaderOnly	Specifies that the leader-only communication scheme should be used. This must be specified after the aqua_other, aqua_client, or aqua_server that this applies to and before the next aqua_other, aqua_client, or aqua_server specified.
-startApp	Next argument specifies the CORBA application process to start up.
-startParam	Next argument specifies a parameter used by the application specified by the startApp parameter. The startParam parameters must be given in the same order as they are used with the application.

Table 2. Dependability Manager & Factory Startup Parameters

Parameter	Description
Dependability Manager	Parameters must be given to the dependability manager in the order listed.
<Port number>	Specifies the port number to use.
<IOR output file>	Optional parameter that specifies an output file to print the dependability manager's IOR.
Object Factory	Parameters must be given to the object factory in the order listed.
<Host name>	Specifies the name of the host that the factory has started on.
<Port number>	Specifies the port number to use.
<DM IOR file>	Specifies the IOR file used by the factory to communicate with the dependability manager.
<Processes file>	Specifies the name of the file listing the location of processes that a factory may start on a host.
<IOR output file>	Optional parameter that specifies an output file to print the factory's IOR.

REFERENCES

[1] Object Management Group, “The common object request broker: architecture and specification,” Revision 2.1, OMG Technical Document PTC/97-09-01, Object Management Group, August 1997.

[2] D. Powell, Ed., Delta-4: A Generic Architecture for Dependable Distributed Computing. New York: Springer-Verlag, 1991.

[3] M. Cukier et al., “AQuA: an adaptive architecture that provides dependable distributed objects,” presented at 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, IN, USA, October 1998.

[4] J.A. Zinky, D. E. Bakken, and R. E. Schantz, “Architectural support for quality of service for CORBA objects,” *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55-73, April 1997.

[5] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, “Specifying and measuring quality of service in distributed object systems,” in *Proceedings of the 1st IEEE International Symposium on Object-oriented Real-time distributed Computing*, Kyoto, Japan, April 1998.

[6] M. G. Hayden, “The Ensemble system,” Ph.D. dissertation, Cornell University, Ithaca, NY, 1998.

[7] A. Vaysburd, “Building reliable interoperable distributed objects with the Maestro tools,” Ph.D. dissertation, Cornell University, Ithaca, NY, 1998.

[8] P. Felber, B. Garbinato, and R. Guerraoui, “The design of a CORBA group communication service,” in *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara on the Lake, Ontario, Canada, October 1996, pp. 150-159.

[9] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, “Replica consistency of CORBA objects in partitionable distributed systems,” *Distributed Systems Engineering*, vol. 4, no. 3, September 1997, pp. 139-150.

[10] S. Maffeis, “Run-time support for object-oriented distributed programming,” Ph.D. dissertation, University of Zurich, Zurich, Switzerland, 1995.

[11] S. Mullender. *Distributed Systems*. New York, NY: Addison-Wesley, 1993.

[12] K. Birman. *Building Secure and Reliable Network Applications*. Greenwich, CT: Manning Publications Co., 1996.

[13] J. Rumbaugh et al., *Object-Oriented Modeling & Design*, Englewood Cliffs, NJ: Prentice Hall, 1991.