from the Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications, San Jose, CA, January 6-8, 1999, pp. 137-156

Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA[1]

Chetan Sabnis, Michel Cukier, Jennifer Ren,
Paul Rubel, and William H. Sanders

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory and
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
{sabnis, cukier, ren, rubel, whs}@crhc.uiuc.edu

David E. Bakken and David A. Karr

BBN Technologies
Cambridge, Massachusetts 02138
{dbakken, dkarr}@bbn.com

Abstract

Building dependable distributed systems from commercial off-the-shelf components is of growing practical importance. For both cost and production reasons, there is interest in approaches and architectures that facilitate building such systems. The AQuA architecture is one such approach; its goal is to provide adaptive fault tolerance to CORBA applications by replicating objects, providing a high-level method for applications to specify their desired dependability, and providing a dependability manager that attempts to reconfigure a system at runtime so that dependability requests are satisfied. This paper describes how dependability is provided in AQuA. In particular, we describe Proteus, the part of AQuA that dynamically manages replicated distributed objects to make them dependable. Given a dependability request, Proteus chooses a fault tolerance approach and reconfigures the system to try to meet the request. The infrastructure of Proteus is described in this paper, along with its use in implementing active replication and a simple dependability policy.

1. Introduction

Providing fault tolerance to distributed applications is a challenging and important goal. In many applications, the cost of a custom hardware solution is prohibitive. Even if custom hardware is used, the flexibility that software can provide makes it a natural choice for implementing a significant portion of the fault tolerance of dependable distributed systems. Furthermore, when the dependability requirements change during the execution of an application, the fault tolerance approach must be *adaptive* in the sense that the mechanisms used to provide fault tolerance may change at runtime. Together, these requirements argue for a software solution that can reconfigure a system based both on the levels of dependability desired by a distributed system during its execution and on the faults that occur.

[1] This research has been supported by DARPA Contracts F30602-96-C-0315 and F30602-97-C-0276.

The AQuA architecture [Cuk98] provides a flexible approach for building dependable, distributed, object-oriented systems that support adaptation due to both faults and changes in an application's dependability requirements. Its goal is to provide a simple high-level way for applications to specify the level of dependability they desire and the type of faults that should be tolerated. A dependability manager configures the system, based on notification of faults that occur and dependability requests from applications, in order to try to achieve a requested level of dependability. It is important to note that the architecture is extensible: it does not prescribe that a particular replication or group communication policy must be used, but instead provides a framework for implementing multiple advisor policies, replication policies, voting policies, and communication protocols.

In order to provide a simple way for application objects to specify the level of dependability they desire, the AQuA architecture uses the *Quality Objects* (QuO) [Zin97, Loy98] framework to process and invoke dependability requests. QuO allows distributed applications to specify quality of service (QoS) requirements at the application level using the notion of a "contract," which specifies actions to be taken based on the state of the distributed system and desired application requirements. The QuO framework provides an environment in which a programmer can specify regions of operation in terms of high-level QoS measures, and translates these measures into specific requests to *Proteus*. QuO also provides ways for multiple QoS properties to be integrated, as well as ways for an application to adapt above the ORB without involving the application, but these topics are beyond the scope of the paper.

Proteus provides fault tolerance in AQuA by dynamically managing replicated distributed objects to make them dependable. It does this by configuring the system in response to faults and changes in desired dependability levels. The choice of how to provide fault tolerance involves choosing the types of faults to tolerate, the styles of replication to use, the types of voting to use, the degrees of replication to use, and the location of the replicas, among other factors. The replication protocols in Proteus assume the existence of an underlying group communication system that provides reliable multicast, total ordering, and virtual synchrony. For our implementation, we have used the *Maestro/Ensemble* [Hay98, Vay98] group communication system. Communication between all architecture components is done using *gateways*, which translate CORBA object invocations into messages that are transmitted via Ensemble, and contain mechanisms to implement a chosen fault tolerance scheme.

This paper describes the detailed design and implementation of Proteus, and shows how Proteus provides a flexible infrastructure for implementing multiple advisor policies, multiple replication protocols, multiple voter types, and multiple group communication schemes. Particular choices among these alternatives depend on the number and type of faults an application wishes to tolerate (crash, value, and time), the performance expected, and the type of communication style (*e.g.*, synchronous vs. asynchronous) desired at the application level. Based on instructions from QuO, these choices can be made during the execution of a distributed application, and hence provide fault tolerance that adapts depending on the needs of one or more applications.

Other researchers have also seen the importance of making CORBA objects reliable. In particular, other approaches that use group communication include Electra [Maf95, Maf97], Eternal [Nar97], Maestro [Vay98], OpenDREAMS [Fel96], and ROMANCE [Rod93]. For a comparison of these approaches with AQuA, see [Cuk98].

The remainder of this paper is organized as follows. Section 2 provides an overview of Proteus, describing how its dependability manager, object factories, and gateways aim to provide a desired level of dependability. Section 3 details the functioning of Proteus's dependability manager and object factories, describing their general architecture and the current policies. Section 4 then describes the gateways, focusing on how voting and replication protocols are built on top of group communication. Details of our current implementation of active replication are given in Section 5. Finally, Section 6 presents a discussion of research directions that we are currently pursuing.

2. Proteus Overview

The organization of Proteus is shown in Figure 1. Proteus makes remote objects dependable by using 1) a replicated dependability manager to make decisions regarding reconfigurations and coordinate changes in system configurations, 2) object factories to kill and start objects and provide information to the dependability manager regarding a host, and 3) gateways that implement particular voting and replication schemes. An interface to the QuO runtime is provided to allow the application to specify the level of dependability.

The *Proteus dependability manager* is composed of two parts: an *advisor,* which makes decisions regarding reconfiguration based on reported faults and dependability requests from QuO, and a *protocol coordinator,* which, together with the gateways implements, the chosen fault tolerance approach. Depending on the choices made by the advisor, Proteus can tolerate and recover from crash failures, time faults, and value faults in application objects and the QuO runtime. Note that we do not aim to tolerate Byzantine faults, value faults in the gateway, or faults in the group communication system itself. If tolerance of more complex fault types is required, one could substitute a more secure group communication protocol (e.g., [Kih98, Rei95]) for Ensemble within the AQuA architecture.

Object factories are used to kill and start replicated applications, depending on decisions made by the dependability manager, and to provide information regarding the host to the dependability manager.

Gateways provide two functions. First, they provide a standard CORBA interface to applications. CORBA provides application developers with a standard interface to build distributed object-oriented applications, but does not provide a simple approach to allow applications to be fault-tolerant. The gateway provides a standard CORBA interface by translating between process-level communication, as supported by Ensemble, and IIOP messages, which are understood by Object Request Brokers (ORBs) in CORBA. In this way, CORBA-based distributed applications written for the AQuA architecture can use standard, commercially available ORBs. In addition

to providing basic reliable communication services for application objects and the QuO runtime, the gateway also provides fault tolerance using different *voters* and *replication protocols*. These services are located in the *gateway handlers*. Both active and passive replication of "AQuA objects" can be supported. *AQuA objects* are the basic units of replication in the AQuA architecture. Each one consists of a gateway, an "application" object, and a QuO runtime, if QuO is being used to manage the desires of the application object. In this context, the *application object* can be part of the distributed application itself, or part of the AQuA architecture that uses the services of a gateway (such as the dependability manager and the object factories). The architecture of the gateway will be described in Section 4.

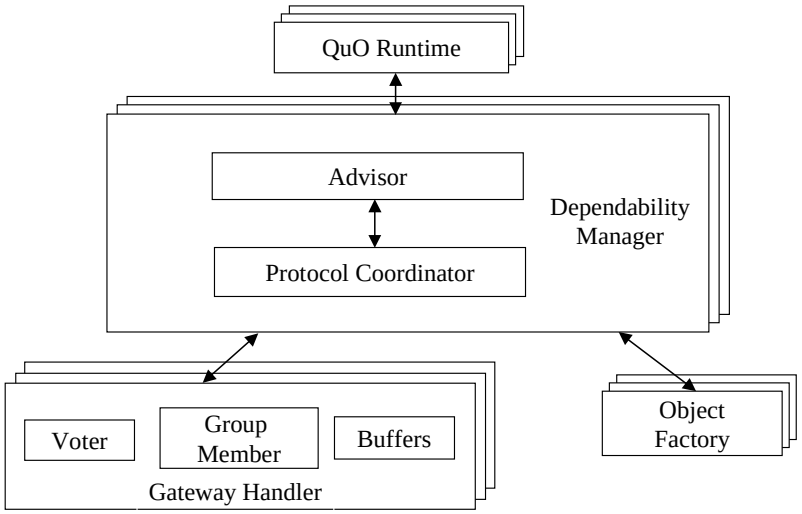

Figure 1: Proteus Architecture

3. Dependability Manager and Object Factories

This section describes Proteus's protocol coordinator, advisor, and object factories. The functional interface between the dependability manager and the gateways, the object factories, and the QuO runtimes is shown in Figure 2 and described in this section. The internal structure of the dependability manager is also described. In Figure 2, arrows are labeled with method calls, using the convention that the arrow indicates the direction of the method invocation.

3.1. Protocol Coordinator

The protocol coordinator is used to communicate with QuO, object factories, and application gateways. The role of the protocol coordinator is to carry out the decisions of the advisor in a consistent way. The protocol coordinator contains the algorithms necessary to execute the decisions and to order the different executions. The particular coordination algorithm used depends on which replication types and fault tolerance mechanisms are supported in a given implementation of the AQuA architecture, but the interface to other AQuA architecture components remains the same in all cases.

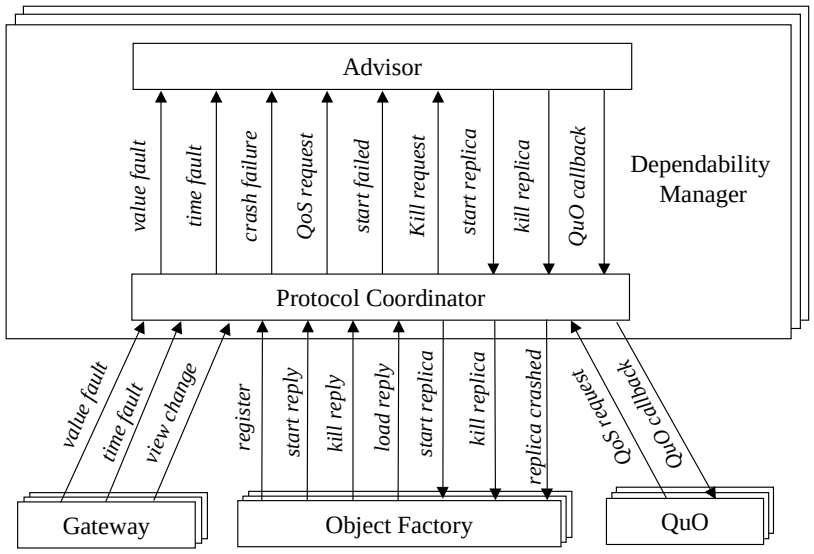

Figure 2: Protocol Coordinator and Advisor Methods

Specifically, the protocol coordinator's CORBA interface consists of several methods. The *view change* method is called by 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. The *register* method is called by an object factory to register itself. The *start reply*, *kill reply*, and *load reply* methods are called by an object factory to report, respectively, the status of a request made to start an application, to kill an application and the load of its host. The *QoS request* method is called by the QuO runtime or an application object to provide a dependability request to the manager. It is used when the desired QoS of a remote object is first specified or has changed during an application's execution. Finally, the *value/time fault* method is called by application gateways if a value/time fault is reported by a gateway. This information is then passed to the advisor for processing.

The protocol coordinator also has methods that are called by the advisor. In the current implementation of the dependability manager, the three methods within the protocol coordinator used by the advisor are *start replica*, *kill replica*, and *QuO callback*. Upon an invocation of the first two methods by the advisor, the protocol coordinator calls *start replica* or *kill replica* on the correct object factory. The protocol coordinator uses timers to limit the time spent waiting for notification of the result of a configuration change request. In particular, the advisor is notified of a start failure if a factory response to a start request, or the view change expected from starting a replica, does not arrive within a specified amount of time. The *QuO callback* method is used to service a request from the advisor to issue a callback to the QuO runtime.

3.2. Advisor

The advisor determines an appropriate system configuration based on requests transmitted through QuO contracts and observations of the system. As with the pro-

tocol coordinator, the policy used by the advisor to make configuration decisions depends on the styles of replication used and the types of faults tolerated. However, the advisor's interface remains the same in all cases. In particular, the *crash failure* method is called if a replica is removed from a view unexpectedly. The *QoS request* method is called if QuO requests a new QoS for an application. The *start failed* method is called if an object factory responds with a failure to a requested start of a replica, if the object factory does not respond in time to a requested start, or if the replica fails to join its replication group in time. The *kill request* method is called if the remote object referenced in a previous QoS request is no longer needed. Finally, the *value/time fault* method is called if a value/time fault notification is received by the protocol coordinator.

A simple policy was developed to support crash failures. In this policy, when the *crash failure* method is invoked, the advisor 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. When a *start failed* method is invoked, the advisor applies the same policy as for the *crash failure* method, except that the method will not choose the host where the replica start failed.

The policy currently implemented also supports adaptation based on dependability requests from QuO. In particular, the *QoS request* 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 on the least-loaded hosts. If replicas are already running, then the advisor may decide to start additional replicas (on the least-loaded hosts) or kill existing replicas (on the most-loaded hosts) so that $n + 1$ replicas are running. If more than one application requests a QoS for a remote object, the advisor chooses to maintain the maximum $n + 1$ replicas among the applications. The *kill request* indicates to the advisor that it may kill all replicas of an application unless the QoS for the application is also requested by another application. In that case, the advisor chooses to maintain the maximum $n + 1$ replicas to support the remaining applications.

If a *crash failure*, *start failed*, or *QoS request* call results in an inability to meet QoS requirements, a callback is made to QuO. In particular, a call to *crash failure* will result in a QuO callback if all available hosts have an application replica running. 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. Finally, a call to *QoS request* 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. This object-based approach to making decisions allows for flexibility in devising advisor policies. In particular, more complicated policies can be programmed and studied without changing the mechanism by which they are executed in the protocol coordinator.

3.3. Object Factory

The function of an object factory, which runs on each host in a system, is to start processes, to kill processes, and to provide information about the host. The object

factory is not replicated, but since it does not contain state that needs to be preserved between failures, it can be restarted after a host failure so that the host can again be used to support AQuA objects.

When a factory is started, it reads in a file that is specific to its host. This file indicates which applications a factory can start. The factory then registers itself with the dependability manager by calling the *register* method. The dependability manager then knows 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 a *start replica* request to the factory, the factory attempts to start the specified application. If an exception is generated while the application is starting, a start failure is reported. If no exception is generated, the factory adds the application to a list of running applications, and a successful start is reported to the advisor.

A request from the dependability manager to kill an application (*kill replica* method) is handled in the same manner. If an exception is generated during an attempt to kill the application, a kill failure is reported. If no exception is generated, the factory removes the application from the list of running applications and a successful kill is reported.

The dependability manager also notifies the factory, through the *replica crashed* method, if a replica on the factory's host fails. 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 through the *load reply* method. The dependability manager uses this information to decide how to assign replicas to hosts.

4. Gateway Overview

The AQuA gateway has several functions: translating between object- (IIOP) and process-level (Ensemble) communication, providing an infrastructure for implementing various replication and voting schemes, and detecting and reporting faults to the dependability manager. This section first describes the physical structure of the gateway, and then describes the group structure used to support Proteus's replication schemes.

4.1. Architecture Overview

The AQuA gateway is implemented as a process, and is one part of an AQuA object. It intercepts IIOP (Internet Inter-ORB Protocol) messages generated from a CORBA object and transmits them, using the Maestro/Ensemble group communication system, to other gateways. As can be seen in Figure 3, the gateway consists of an IIOP encoder/decoder, a dispatcher, handlers, and an interface to Maestro/Ensemble.

Figure 3 shows a high-level view of the gateway architecture, and its interface to the application and QuO runtime processes.

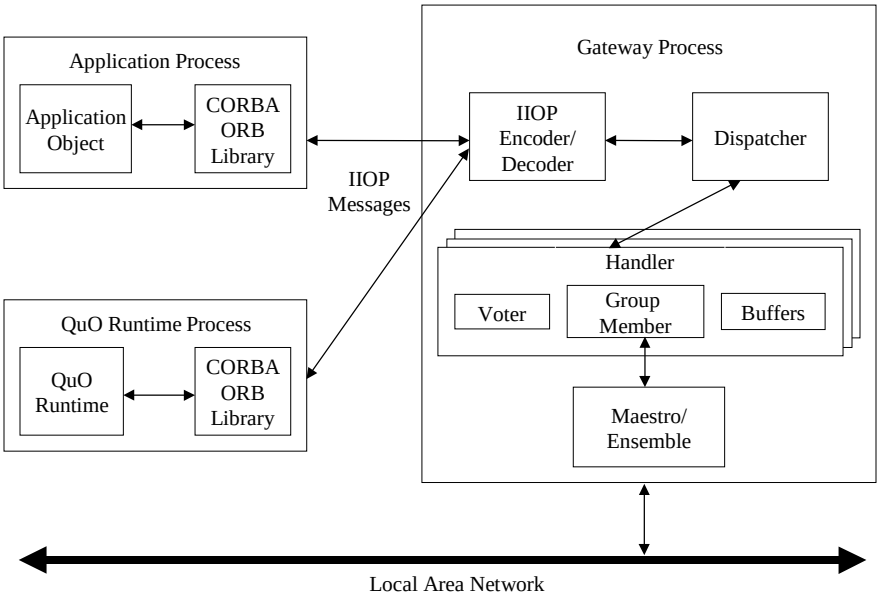

Figure 3: Gateway Architecture

The *IIOP encoder/decoder* is used to interface with a CORBA object. When an IIOP stream is intercepted by the IIOP decoder, it removes the IIOP header and passes the message to the dispatcher, which delivers it according to the protocols in Section 5. The gateway has an incoming and an outgoing queue of messages from the encoder/decoder for use by the dispatcher, which separates the IIOP encoder/decoder from the gateway mechanisms that deal with the messages once they are delivered. The encoder reads messages from the incoming queue, packs the information into an IIOP message, and sends the message to the CORBA application.

The *dispatcher* interfaces with the IIOP encoder/decoder and provides a set of functional features for delivering messages. 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, which takes care of sending and receiving messages. This wrapped message is called a "gateway message." The dispatcher also stores the wrappers of requests it receives from handlers. This is done so that the wrapper of an IIOP request can be reassigned to the IIOP reply when the reply is dequeued from the decoder.

Gateway messages encapsulate IIOP messages that will ultimately be delivered to some ORB, and have a header, which contains information used by replication schemes to process and deliver messages correctly. Specifically, a header contains *SenderReplicationGroup*, *ReceiverReplicationGroup*, *SequenceNumber*, *Opcode*, and *Endian* fields. Replication and connection groups will be described in the next subsection, but it is important to note here that the terms "sender" and "receiver" as used here refer to the sender and receiver of the CORBA *request* that is made, regardless of the sender or receiver of the message during a particular transmission.

The sequence number is determined by the sending and receiving replication groups (unique when tagged with both group names), and the opcode is used to distinguish the message's purpose at a given step in the communication scheme. The "endianess" of the message is also stored, and is used by the destination ORB.

Handlers are responsible for sending and receiving messages. A handler is created in the gateway for each pair of replicated objects that wish to communicate. Currently, three types of handlers are supported in the gateway: the active replication handler, the dependability manager handler, and the factory handler. Different handler types are used depending on the type of group the object is communicating in.

Each gateway handler maintains certain state information to ensure correct the delivery of messages. In particular, the gateway keeps track of the last request sent (*ConnectionGroup.LastSent*), the last request delivered (*ConnectionGroup.LastDeliveredRequest*), the last reply delivered (*ConnectionGroup.LastDeliveredReply*), the last request multicast (*ConnectionGroup.LastMulticastRequest*), and the last reply multicast (*ConnectionGroup.LastMulticastReply*). Each gateway handler also contains buffers that hold messages whose status is not yet known, a *point to point buffer*, a *reply buffer*, and a *total order buffer*, to permit recovery from failures.

4.2. Group Structure

A key part of Proteus's design is the way it uses group communication. A simple approach would be to put all objects that need to communicate in one group, so that all messages would be totally ordered with respect to each other and view changes. While this approach would work for small systems, it scales poorly, and would not be practical for systems with a large number of hosts. To avoid these scalability problems, we have devised a scheme that uses multiple small groups of multiple types.

The group structure used in Proteus was introduced in [Cuk98]; we briefly review it here in order to describe how the communication schemes that we have implemented use this group structure. Four types of groups are used in Proteus: "replication groups," "connection groups," "point-to-point groups," and the "Proteus Communication Service (PCS) group."

A *replication group* is composed of one or more replicas of an AQuA application object. A replication group has one object that is designated as its leader, and may perform special functions, depending on the replication algorithm being implemented. Each object in the group has the capacity to become the object group leader, and a protocol is provided to make sure that a new leader is elected when the current leader fails. 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 object leader fails.

A *connection group* is a group consisting of the members of two replication groups that wish to communicate. A message is multicast within a connection group in order to send a message from one replication group to another replication group. A con-

nection 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 to the dependability manager 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 multicast messages to the dependability manager replicas. Through the PCS group, AQuA applications provide notification of view changes; QuO makes requests for QoS; and factories respond to start and kill commands and provide host load updates. After a multicast to the PCS group, the transient member will leave the group.

Finally, *point-to-point groups* are used to send messages from a dependability manager to a factory.

5. Active Replication Scheme

Multiple communication schemes can be developed, using the above group structure, depending on the communication style (e.g., synchronous, asynchronous), the replication type (e.g., active, passive), and the voter type. An active replication scheme using the *leader pass first* voter policy to send messages has been implemented. This section will describe the structure of this scheme, the algorithms to implement it, and how faults are tolerated using these algorithms.

5.1. Groups in Active Replication

We first review the steps taken to make a remote CORBA call (request/reply) when active replication is used [Cuk98]. Let $O_{i,k}$ be replica k of replication group i, and let object $O_{i,0}$ be the leader of the group. Suppose that replication group i is the sender group and group j the receiver group. To send a request to the object replicas $O_{j,k}$, as shown in Figure 4, all objects $O_{i,k}$ first use reliable point-to-point communication to send the request to $O_{i,0}$. Object replicas $O_{i,k}$ also keep a copy of the request in case it needs to be resent (step 1 in Figure 4). The leader then multicasts the request in the connection group. The objects $O_{i,k}$ use the multicast to signal that they can delete their local copy of the request. The objects $O_{j,k}$ store the multicast on a list of pending rebroadcasts (step 2). 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. The objects $O_{j,k}$ use the multicast as a signal that they can deliver the message and delete the previously stored copy from the connection group multicast (step 3). After processing the request, all objects $O_{j,k}$ send the result through a point-to-point communication to $O_{j,0}$ (step 4). The same set of steps used to transmit the request is then used to communicate the reply from replication group j to group i. Steps (5) and (6), which are responsible for transmitting the reply, are similar to steps (2) and (1) respectively.

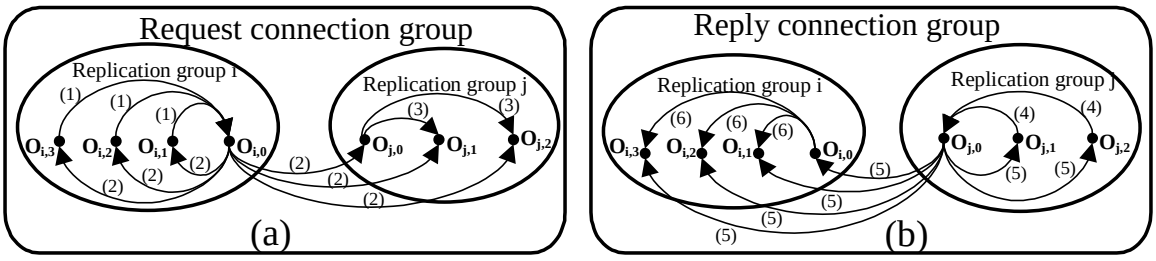

Figure 4: Communication Scheme

5.2. Active Replication Algorithms

The active replication scheme must both 1) deliver requests and replies correctly, and 2) correctly react when the number of replicas in a replication group changes. The active replication communication algorithms are used to correctly communicate CORBA requests and replies between replicated objects using the replication and connection groups that have stable group membership. When the group membership changes, view change algorithms are used to report the change of membership to the dependability manager and, if a new member is added, correctly initialize the new object with the correct state.

5.2.1. Active Replication Communication Algorithms

The algorithms that run in the active replication handler are described in this section. To illustrate their use, we will describe the algorithms in the order in which they would be executed (on various nodes) as a single request/reply is processed, using the "step" nomenclature introduced earlier in this section. More specifically, recall that each message contains an opcode to specify the communication step the message is in. The relationship between opcodes and communication steps is shown in Table 1. By following these steps, we illustrate the algorithms that implement the active replication scheme.

Step	Opcode	Step	Opcode
1	FORWARD_REQUEST_TO_LEADER	4	FORWARD_REPLY_TO_LEADER
2	CONNECTION_GROUP_REQUEST	5	CONNECTION_GROUP_REPLY
3	REPLICATION_GROUP_REQUEST	6	REPLICATION_GROUP_REPLY

Table 1. Communication Step and its Opcode

Step 1: The first step of the communication scheme consists of sending a request to the replication group leader. This is done in two phases. First, each replica on the sender side receives a remote object request, via a CORBA IIOP message to the gateway, that is processed by the IIOP decoder and dispatcher. Note that these requests may not come at the same time, but they will come in the same order (since we assume application objects behave in a deterministic way and receive all external requests in the same order). After processing, the dispatcher calls **SendRequest** (Figure 5). Each message is then tagged with the opcode FORWARD_REQUEST-

_TO_LEADER, so that it is not misinterpreted as a reply to a previous message. The handler variable that keeps track of the last sent sequence number for the group is then set to the sequence number of the message. Next, the reply buffer is checked to see if a reply has already been received for this message. **RemoveReplyFromBuffer** will return the reply if it is in the buffer and a NULL if the reply to this message is not found. The reply is stored in the buffer if another replica has previously forwarded its request to the leader (which is possible, since replicas behave asynchronously) and this replica has already received the reply. If the reply is present, it is delivered to the handler, which in turn calls the IIOP encoder that delivers the message to the application (**DeliverReplyToApp**). If the reply is not contained in the reply buffer, the sequence number is checked to see if the request should be sent to the leader. The message should be sent to the leader if a copy of it has not already been multicast in the connection group (this would occur if another replica in the replication group has forwarded the request to the leader, who then multicasted it to the connection group). If the request should be sent to the leader, it is placed in a point-to-point buffer (so that it can be resent if a failure occurs), and is sent to the leader using a Maestro point-to-point send (**SendToLeader)**.

```
SendRequest( message request )
    request.opcode := FORWARD_REQUEST_TO_LEADER
    ConnectionGroup.LastSent := request.SequenceNumber
    reply :=  RemoveReplyFromReplyBuffer( request )
    if ( reply ≠ NULL ) DeliverReplyToApp( reply )
    else if ( ConnectionGroup.LastMulticastRequest < request.SequenceNumber )
            addToPtPBuffer( request )
            SendToLeader( request )

SendReply( message reply )
    reply.opcode := FORWARD_REPLY_TO_LEADER
    if ( ConnectionGroup.LastMulticastReply < reply.SequenceNumber )
            addToPtPBuffer( reply )
            SendToLeader( reply )
```

Figure 5: **SendRequest** and **SendReply** Algorithms

The second phase begins when the leader receives the request from Ensemble and calls **ReceiveSendToLeader** (Figure 6). As seen in Figure 4, the message sent to the leader can be either a request or a reply, as denoted by the opcode FORWARD-_REQUEST_TO_LEADER or FORWARD_REPLY_TO_LEADER. At this step the message is a request. Non-leaders receiving this message do nothing. The voter then processes the message in the way defined by the communication scheme. The currently implemented voter applies the *leader pass first* policy, and thus returns the message passed to it, unless it has already been multicast (which is determined by the sequence number of the message). Since the message is a request, the opcode is changed from FORWARD_REQUEST_TO_LEADER to CONNECTION-_GROUP_REQUEST. The message is then checked to see whether it should be multicast to the connection group. The *ConnectionGroup.LastMulticastRequest* variable is used to determine this, and keeps the leader from multicasting duplicate re-

quests that it may receive. (A duplicate request may be received if a view change occurs and a replica's point-to-point buffer is not empty). If the request is not a duplicate, it is multicast in the connection group through the **MulticastToConnectionGroup** call, which uses the Maestro multicast facility. Finally, the handler's last multicast request variable is set to the message's sequence number (since the Maestro multicast, as we have used it, does not send the message to the sender of the multicast).

Step 2: The second step of the communication scheme begins with the **MulticastToConnectionGroup** call. When a multicast message is received by a connection group member, Ensemble calls **ReceiveConnectionGroupMulticast** (Figure 6). Since each member of the connection group receives the multicast, **ReceiveConnectionGroupMulticast** needs to determine whether the received multicast is a request or reply message, and whether the original request came from a replica in the same replication group as the receiver, or a different replication group.

To distinguish among these cases, the method first checks the opcode of the message to see whether it is a request or a reply. A message received in step 2 will have the opcode CONNECTION_GROUP_REQUEST, marking it as a request multicast by the group leader. The method then checks to see if the receiver of the message is a member of the group specified by the sender replication group field. If it is, then the message may have been buffered by this group member so that it could be resent in case of failure, but since the multicasts are reliable, all other recipients of the message have now also received it. The receiver can thus safely remove this message from its point-to-point buffer, if it is there. The last multicast request variable in the handler is then set to the sequence number of this message, for future reference.

Second, the method checks whether the receiver replication group field of the message is the message receiver's replication group. If this is the case, the opcode (CONNECTION_GROUP_REQUEST) is changed to REPLICATION_GROUP_-REQUEST (initiating step 3). The message is then saved in the total order buffer to permit recovery in case of a replica failure. Then, if this replica is the leader, it multicasts the message to the members of the replication group through the **Multicast-ToMyReplicationGroup** call, again using the Maestro multicast facility.

Step 3: The third step of the communication scheme begins with the **MultcastTo-MyReplicationGroup** call. When a multicast message sent via this call is received by a replication group member, Ensemble calls **ReceiveReplicationGroupMulticast** (Figure 6). In this step, the opcode is REPLICATION_GROUP_REQUEST, so the first block of code in this method is executed. First, the sequence number of the received message is checked (against the last delivered request) to be sure that this message has not already been delivered. This could happen if a leader crashed after sending a multicast, but before the replication group members received it. If this is the case, the message is ignored. Otherwise, the request is delivered to the application after processing by the dispatcher and IIOP encoder using **DeliverRequest**. In either case, the message is removed from the handler's total order buffer, since the message has now been delivered to all replication group members. Finally, the handler's last delivered request variable is set to the message's sequence number.

```
ReceiveSendToLeader( message m )
    message MessageToMulticast := Voter.Process( m )
    if ( Leader and ( MessageToMulticast ≠ NULL )  )
        if ( MessageToMulticast.opcode = FORWARD_REQUEST_TO_LEADER )
            MessageToMulticast.opcode := CONNECTION_GROUP_REQUEST
            if ( ConnectionGroup.LastMulticastRequest < MessageToMulticast.SequenceNumber )
                MulticastToConnectionGroup( MessageToMulticast )
                ConnectionGroup.LastMulticastRequest := MessageToMulticast.SequenceNumber
        if ( MessageToMulticast.opcode = FORWARD_REPLY_TO_LEADER )
            MessageToMulticast.opcode := CONNECTION_GROUP_REPLY
            if ( ConnectionGroup.LastMulticastReply < MessageToMulticast.SequenceNumber )
                MulticastToConnectionGroup( MessageToMulticast )
                ConnectionGroup.LastMulticastReply := MessageToMulticast.SequenceNumber

ReceiveConnectionGroupMulticast( message m )
    if ( m.opcode = CONNECTION_GROUP_REQUEST )
        if ( m.SenderReplicationGroup = myReplicationGroup )
            RemoveFromPtPBuffer( m )
            ConnectionGroup.LastMulticastRequest := m.SequenceNumber
        if ( m.ReceiverReplicationGroup = myReplicationGroup )
            m.opcode := REPLICATION_GROUP_REQUEST
            AddToTotalOrderBuffer( m )
            if ( Leader ) MulticastToMyReplicationGroup( m )
    if ( m.opcode = CONNECTION_GROUP_REPLY )
        if ( m.ReceiverReplicationGroup = myReplicationGroup )
            RemoveFromPtPBuffer( m )
            ConnectionGroup.LastMulticastReply := m.SequenceNumber
        if ( m.SenderReplicationGroup = myReplicationGroup )
            m.opcode = REPLICATION_GROUP_REPLY
            AddToTotalOrderBuffer( m )
            if ( Leader ) MulticastToMyReplicationGroup( m )

ReceiveReplicationGroupMulticast( message m )
    if ( m.opcode = REPLICATION_GROUP_REQUEST )
        if ( m.SequenceNumber > ConnectionGroup.LastDeliveredRequest )
            DeliverRequest( m )
            RemoveFromTotalOrderBuffer( m )
            ConnectionGroup.LastDeliveredRequest := m.SequenceNumber
    if ( m.opcode = REPLICATION_GROUP_REPLY )
        if ( m.SequenceNumber > ConnectionGroup.LastDeliveredRequest )
            if ( ConnnectionGroup.LastSent ≥ m.SequenceNumber) DeliverReplyToApp( m )
            else AddToReplyBuffer( m )
            RemoveFromTotalOrderBuffer( m )
            ConnectionGroup.LastDeliveredReply := m.SequenceNumber
```

Figure 6: **ReceiveSendToLeader**, **ReceiveConnectionGroupMulticast**, and **ReceiveReplicationGroupMulticast** Algorithms

Step 4: Each application object in the receiving replication group processes the request and generates a reply independently. When a reply is ready, the application object generates a standard CORBA reply message and delivers it to the IIOP decoder of its associated gateway. When the gateway receives the message, it decodes it, uses the dispatcher to recover its header (saved when the request was delivered by

the dispatcher), and calls the handler's **SendReply** method to send the reply to the replication group leader. Before the message is sent, its opcode is set to FORWARD-_REPLY_TO_LEADER. The message is only sent if the handler's *Connection-Group.LastSentReply* variable indicates that the reply has not yet been processed by the leader. If the message is sent, it is also added to the point-to-point buffer, so it can be re-sent in case the leader of the replication group fails. There is no need to check for a reply in the reply buffer, as was done in **SendRequest**, since there are no messages dependent on receiving this message. The message is sent to the leader using the **SendToLeader** call, which uses the Maestro point-to-point send. The leader calls **ReceiveSendToLeader** when it receives the message. This method was described in step 1; processing is identical in this step, except that the message's opcode is changed from FORWARD_REPLY_TO_LEADER to CONNECTION_GROUP_-REPLY, and the handler's *ConnectionGroup.LastMulticastReply* variable is updated.

Step 5: Step five consists of an execution of the **MulticastToConnectionGroup** method by the leader of the receiver replication group and an execution of the **ReceiveConnectionGroupMulticast** method by all members of the connection group. These calls execute like those described in step 2, except that the opcode of the message (CONNECTION_GROUP_REPLY) now indicates that it is a reply, so a member of the sending group will set its last multicast reply (not request) variable, and a member of the receiving group will change the message opcode to REPLICATION-_GROUP_REPLY (initiating step 6) before buffering or multicasting it to the replication group.

Step 6: Finally, the **MulticastToMyReplicationGroup** and **ReceiveReplicationGroupMulticast** methods are performed by the leader of the requesting replication group and the members of the requesting replication group, respectively. These execute like those described in step 3, except that the opcode is REPLICATION-_GROUP_REPLY, and the handling of the messages in **ReceiveReplicationGroup-Multicast** is more complicated. As before, if the message has already been delivered (this time, compare the sequence number to the last delivered reply), it is ignored. Then, if the corresponding request message has not yet been generated by the application, the message is placed in the reply buffer to be held until the replica produces the corresponding request. Otherwise, the message is delivered to the application (after processing by the dispatcher and IIOP encoder) using **DeliverReplyToApp**. As in step 3, the message is removed from the total order buffer, but this time the last delivered reply variable is set. Completion of these steps results in the delivery of the reply to all members of the requesting replication group.

5.2.2. View Change Algorithms

When the membership of a group changes, a view change occurs. Replication group view changes signal changes that must be accounted for to maintain the correct replication group state and structure. A replication group view change will occur if a new member joins the replication group, a member crashes, or a member is killed by an object factory. When a new member enters a replication group, a state transfer is needed to set the new member's state to a state that is correct, with respect to the

other replicas in its replication group. The state transfer is implemented with the help of Maestro state transfer calls. The state transfer occurs as part of Maestro view change processing, so it is atomic with respect to the processing of other messages in the group.

More specifically, Maestro initiates a state transfer when a new member joins a group by calling the **GetState** method (Figure 7) of the handler of an existing group member. This method collects the state information needed by the new replica in a "transfer message." The transfer message holds the state information needed to construct a new replica and is passed to the new replica specified by a requestor. The transfer message consists of three parts: the connection group state, the replication group state, and the application object state. The connection and replication group state (as defined in Section 4.1) is added to the transfer message using the **AddConnectionGroupState** and **AddBufferState** methods, respectively. The application object state is added to the transfer message using the **AddApplicationState** method. To get the state of the application, **AddApplicationState** executes a **GetApplicationState** method on the associated application object. This method, which must be implemented by the application object, packages up the needed state so that it can be placed in the transfer message and, ultimately, delivered to the new application object.

```
GetState( Requestor r )
    TransferMessage t
    AddConnectionGroupState( t )
    AddBufferState( t )
    AddApplicationState( t )

SetState( TransferMessage t )
    SetConnectionGroupState( t )
    SetBufferState( t )
    SetApplicationState( t )

ViewChange( view newView )
   if ( NewLeader() )
     for each message m in PtPBuffer
        SendToLeader( m )
       if ( Leader )
          for each message m in TotalOrderBuffer
             MulticastToReplicationGroup( m )
    if ( Leader ) SendToPCSGroup( newView )
```

Figure 7: **GetState**, **SetState**, and **ViewChange** Algorithms

Once the transfer message is received by Maestro in the new replica, the replica calls the new replica handler's **SetState** method (Figure 7) to set the state of the replica to that prescribed by the transfer message. More specifically, the state of the new replica's handler is set via the **SetConnectionGroupState** and **SetApplicationState** calls. The state of the new application object is set by invoking the **SetApplicationState** method on the new application object. This method, implemented by the application object, unpacks the message to obtain the state information needed to set the

new application object state to that obtained from an existing replica. Once the new member is integrated into the group, Maestro invokes each group member's **View-Change** method, to notify each replication group member's handler that a view change has occurred.

The first step of **ViewChange** (Figure 7) is executed only if the view has changed because the (old) leader left the replication group. The existence of a new leader is determined via a call to method **NewLeader**. If the group has a new leader, all messages in each group member's handler's point-to-point buffer need to be sent to the (new) leader, since the buffer contains messages that were sent to the leader, but not yet multicast to the connection group. Likewise, the leader multicasts all messages in its total order buffer to its replication group, since these are messages that were received by the replication group from an associated connection group, but not yet successfully multicast by the leader to the group. Finally, the leader informs the dependability manager of the membership of the changed group, using the PCS group structure described in Section 4.2.

5.3. Dependability Manager Communication

Algorithms also exist for communication from the gateways, QuO runtime, and object factories to the dependability manager (via the PCS group), and from the dependability manager to a factory (via point-to-point groups). These algorithms are much simpler than those for active replication, since reliable multicasts between multiple groups are not needed. The details of these algorithms can be found in [Sab98]; they are omitted here due to space limitations.

5.4. Algorithm's Response to Faults

We now show how crash failures are tolerated using these algorithms. In particular, we consider all the points at which a crash can occur, for both the leader and non-leader replicas, in both the sender and receiver replication groups. Tolerating failures of non-leader replicas is simple using the Proteus group structure, since no message retransmission is necessary. In particular, a replica can crash before or after a multicast message 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 non-failed replicas. The other cases in which a non-leader replica can crash (before and after sending a point-to-point message) also require no action from other replicas.

If the leader of the sender replication group crashes before **ReceiveSend-ToLeader** was complete, the message transmission process is restarted by using **ViewChange** once the replication group has gone through a view change and elected a new leader. In that case, **SendToLeader** is performed again. Specifically, the replicas in the sender replication group, having kept a copy of the message in the point-to-point buffer, resend the message to the new leader of the sender replication group.

The new leader then multicasts each message in the connection group via **Receive-SendToLeader** and the message transmission process is resumed. If the original leader crashes immediately after sending the multicast, a new leader may be elected before the multicast is delivered. This case uses the same sequence calls as above. The new leader will also multicast the message, and this second multicast will be ignored by the members of the connection group. If the leader crashes in other stages of the communication scheme, no message retransmission is necessary, since the sender replication group leader is not responsible for message transmission in these stages.

If the leader of the receiver replication group crashes before **ReceiveConnectionGroupMulticast** is complete, the message transmission process is performed via **ViewChange** 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 in total order buffer, multicasts the message in the receiver replication group via **MulticastToReplicationGroup**. If the old leader crashes immediately after sending the multicast in the receiver replication group, a new leader may be elected before the multicast is delivered. In that case, the same sequence calls are used. The new leader will multicast the message in the receiver replication group, and this second multicast of the message will be ignored by the members of the replication group. If the leader crashes in other stages of the communication scheme, no message retransmission is necessary, since the receiver replication group leader is not responsible for message transmission in these stages.

6. Results and Conclusions

This paper has described Proteus, a flexible infrastructure for providing adaptive fault tolerance to CORBA applications. Our design permits an application to change the level of dependability that it requires, including the type of faults that should be tolerated dynamically during its execution. In order to make this possible, we have designed Proteus in a modular way, developing a scalable group structure, and a set of communication algorithms that preserve needed communication properties during intergroup communication. Gateways were designed that make use of this group structure, and support multiple communication schemes through the use of different handlers. In addition, the gateways provide the translation facilities needed to convert IIOP messages, understood by standard CORBA ORBs, into a series of group communication multicasts, to achieve reliable communication between replicated CORBA objects. Furthermore, the gateways were designed to be configurable during the execution of an application, accepting requests from the dependability manager to use particular handlers and detect particular fault types. Finally, we designed a dependability manager that makes decisions on how to configure a system based both on high-level, application-driven requests for particular levels of dependability, and on faults and failures that occur. In addition, the dependability manager provides a protocol coordinator that carries out the decisions of the advisor by starting, killing, and reconfiguring object replicas as needed.

Our first implementation of Proteus is complete, and has been successfully tested on several applications (see [Sab98], for details). In each case, the application was first developed as a standard (non-fault-tolerant) distributed CORBA application, and then ported to the AQuA architecture with almost no effort. In fact, the only changes to the applications that were needed were the addition of a **SetApplicationState** and a **GetApplicationState** method to each object that was to be replicated. The implementation has fully functioning versions of all Proteus components, including a dependability manager with advisor and protocol coordinator, object factories that can start and kill replicas and report load information to the manager, and a gateway with several types of handlers that support the active replication and PCS communication schemes.

The implementation includes support for crash failures, an advisor policy that permits changing the degree of replication and placement of replicas during execution based on the dependability desires of an application, and a standard CORBA interface to distributed applications. In addition, a graphical user interface for the dependability manager and object factories was developed to allow the functioning of Proteus to be monitored as it responds to dependability requests from applications and faults that occur. A user can monitor changes in membership that occur in replication and connection groups and in assignment of objects to hosts. This interface was extremely helpful in developing the active replication and PCS communication schemes, and in observing the results of fault injection experiments we used to validate the implementation.

While the current implementation of Proteus can provide significant dependability enhancement for standard CORBA applications, much more can be done. In particular, we are currently implementing voters in gateways to support detection of value faults, and are developing an advisor policy that can determine how to treat such faults. We are also implementing the handlers necessary to replicate the dependability manger, which was not replicated in our original implementation, and to support passive replication.

Acknowledgment

We would like to acknowledge the other members of the AQuA team, namely Mark Berman, David Henke, Jessica Pistole, and Rick Schantz, for many constructive discussions.

References

[Cuk98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, R. E. Schantz, “AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects,” Proc. of the 17th IEEE Symposium on Reliable Distributed Systems, pp. 245-253, West Lafayette, IN, USA, October 1998.

[Fel96] P. Felber, B. Garbinato, R. Guerraoui, “The Design of a CORBA Group Communication Service,” Proc. of the 15th IEEE Symposium on Reliable Distributed Systems, pp. 150-159, Niagara on the Lake, Ontario, Canada, October 1996.

[Hay98] M. G. Hayden, “The Ensemble System,” Ph.D. thesis, Cornell University, 1998.

[Kih98] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," Proc. of the IEEE 31st Annual Hawaii International Conference on System Sciences, vol. 3, pp. 317-326, Kona, Hawaii, January 1998.

[Lan97] S. Landis, S. Maffeis, "Building Reliable Distributed Systems with CORBA," in *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997, pp. 31-43.

[Loy98] J. P. Loyall, R. E. Schantz, J. A. Zinky, D. E. Bakken, "Specifying and Measuring Quality of Service in Distributed Object Systems," in Proc. of ISORC'98, Kyoto, Japan, April 1998.

[Maf95] S. Maffeis, "Run-Time Support for Object-Oriented Distributed Programming," Ph.D thesis, University of Zurich, 1995.

[Maf97] S. Maffeis, "Piranha: A CORBA Tool for High Availability," *IEEE Computer*, vol. 30, no. 4, 1997, pp. 59-66.

[Nar97] P. Narasimhan, L. E. Moser, 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.

[Rei95] M. K. Reiter. "The Rampart Toolkit for Building High-Integrity Services," *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pp. 99-110, Springer-Verlag, 1995.

[Rod93] L. Rodrigues, P. Verissimo, "Replicated Object Management using Group Technology," Proc. of the Fourth Workshop on Future Trends of Distributed Computing Systems, pp. 54-61, Lisboa, Portugal, September 1993.

[Sab98] C. Sabnis, "Proteus: A Software Infrastructure Providing Dependability for CORBA Applications," M.S. thesis, University of Illinois, Urbana, IL, 1998.

[Vay98] A. Vaysburd, K. P. Birman, "The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles," *Theory and Practice of Object Systems*, vol. 4, no. 2, 1998.

[Zin97] J. A. Zinky , D. E. Bakken, R. E. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, April 1997, pp. 55-73.