PASSIVE REPLICATION SCHEMES IN AQUA

Yansong (Jennifer) Ren*, Paul Rubel**, Mouna Seri****,
Michel Cukier***, William H. Sanders****, and Tod Courtney****

*Bell Laboratories, Holmdel, New Jersey, reny@lucent.com

** BBN Technologies, Cambridge, Massachusetts, prubel@bbn.com

*** Center for Reliability Engineering, Department of Materials and Nuclear Engineering,
University of Maryland at College Park, Maryland, mcukier@eng.umd.edu

****Department of Electrical and Computer Engineering and Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign, Urbana, Illinois,
{seri, tod, whs}@crhc.uiuc.edu

Abstract

Building large-scale distributed object-oriented systems that provide multidimensional Quality of Service (QoS) in terms of fault tolerance, scalability, and performance is challenging. In order to meet this challenge, we need an architecture that can ensure that applications' requirements can be met while providing reusable technologies and software solutions. This paper describes techniques, based on the AQuA architecture, that enhance the applications' dependability and scalability by introducing two types of group members and a novel passive replication scheme. In addition, we describe how to make the management structure itself dependable by using the passive replication scheme. Finally, we provide performance measurements for the passive replication scheme.

1. Introduction

There have been several attempts to build distributed object-oriented systems that provide tunable fault tolerance, scalability, and performance. Notable work includes the Delta-4 project [Pow91], Aurora [AMW], and Chameleon [Bag98], among others. Providing fault tolerance at the process level through the use of the group communication paradigm has also been studied. Work in this area includes Ensemble [Hay98], Cactus [Bha97], and Rampart [Rei95]. There have also been many attempts to provide fault tolerance to distributed CORBA applications by using group communication. Those projects include Maestro [Vay98], OpenDREAMS [Fel96] and Eternal [Nar99]. The Object Management Group (OMG), which designed CORBA, has also recently provided a Fault Tolerant CORBA standard [OMG99]. Readers are referred to [Ren01a] for a detailed review of the related work.

The AQuA architecture [Ren01a, Ren01b] is a framework for building dependable, distributed, object-oriented systems that support adaptation to both faults and changes in an application's dependability requirements. It provides the types of fault tolerance specified by the CORBA fault tolerance standard. In the previous AQuA work, active replication schemes were developed. In active replication, all object replicas independently execute the method invocations. This technique requires that CORBA ORB and object replicas be deterministic to ensure that incoming messages are dispatched from the ORB to the replicas in the same order, and that the replicas process the messages in the same order. However, distributed applications do not include only deterministic operations; some of them may contain non-deterministic operations. For example, an application could contain multiple threads. For an application with non-deterministic operations, although the active replication mechanism can ensure that all incoming messages are delivered to the replicated objects in a totally ordered sequence, the non-deterministic operations will cause the replicas in different states to generate conflicting results. Since active replication schemes are not suitable in these cases, we developed the AQuA passive replication schemes to handle non-deterministic applications. The passive replication scheme, in which only the leader of the replicas processes messages and makes decisions, has the ability to provide fault tolerance to the non-deterministic applications.

The remainder of this paper is organized as follows. In Section 2, we will give a brief review of the AQuA architecture in order to help readers understand the rest of the paper. Section 3 introduces two types of group members and the methods of reliable communication. Section 4 details the implementation of the passive replication scheme. Section 5 focuses on how to provide dependability to the dependability manager using the passive replication schemes. Section 6 provides detailed performance measurements for the passive replication scheme. Finally, Section 7 concludes the paper.

2. AQuA Overview

This section briefly reviews the AQuA architecture. Readers are referred to [Cuk98, Ren01a] for details. In AQuA, fault tolerance is achieved through replication of objects. The AQuA infrastructure consists of a replicated dependability manager, a set of object factories, and gateway handlers. The dependability manager determines system configurations at runtime so that dependability requests are satisfied and system resources are used in a cost-effective manner. An object factory that resides on each host is used to create and kill objects. The replicated objects form a group. Messages communicated among different objects are sent through groups. A group communication system (Maestro/Ensemble [Vay98, Hay98]) is used to provide reliable message multicast, totally ordered message delivery, and group membership maintenance.

In AQuA, we developed a unique proxy approach that uses "AQuA gateways" to give a client object the perception that it is talking to a single remote object. The AQuA gateway has a standard IIOP interface to communicate with CORBA applications, and a set of communication schemes and replication protocols to reliably communicate invocations to the replicated remote servers. The communication and replication schemes are located in gateway handlers. Only the handlers are inside the communication groups, as shown in Figure 1, in which "h" represents a handler. The AQuA gateway handler is responsible for finding a set of replicated server objects that can implement a client's request, passing them the parameters, invoking their methods, and returning the results. The replication schemes in the handler provide strong data consistency among the replicated objects. They ensure reliable message communication, no matter when and where server object replicas fail before the client receives the replies.

3. Persistent and transient group members

To achieve scalability, we enhance the AQuA group structure by introducing persistent and transient group members. In AQuA, two types of groups are provided: replication groups and connection groups. A *replication group* is composed of one or more replicas of an AQuA object. These objects may be transient or persistent members of the group. *Persistent members* join the group when they are created, and remain in the group. *Transient members* join a replication group only when they need to multicast a message to the replicas in the group. After sending a message, these objects leave the group. A replication group has one persistent object that is designated as its leader and may perform special functions. A *connection group* consists of the persistent members of two replication groups that wish to communicate.

As specified above, each replicated object is located inside a replication group. AQuA provides two methods of reliable communication between objects that are in two different replication groups. One way is to use a connection group, which is done if the sending object is a persistent member of its replication group, and hence is in a connection group shared by the destination replicated object. In that case, a replicated object that is inside a replication group multicasts messages within a connection group to forward them to the other replicated object. Using that approach, two different objects are able to communicate using both one-way and synchronous remote method invocations. This approach requires that there be a pre-established connection group before objects send messages to each other.

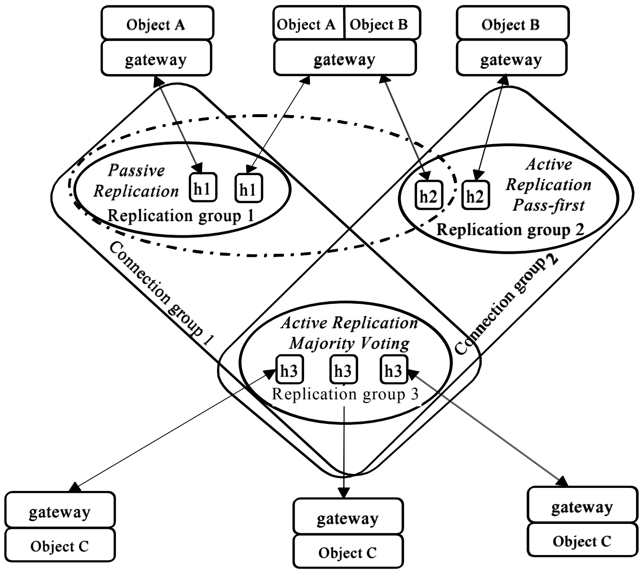


Figure 1: Example Group Structure in AQuA

In the second method of reliable communication, the sending object becomes a transient member of a replication group with which it wishes to communicate. The invocations made by transient members can only be one-way. In addition, only the leader of a sender replication group is allowed to become a transient member of another replication group, and the leader is responsible for making invocations on behalf of the sender replication group. The method is only suitable for situations in which duplicate messages are allowable (at-least-once semantics). Communication through a transient group member is useful in situations in which communication is fairly infrequent. In such cases, the overhead in joining and leaving a replication group is small relative to that of maintaining a connection group between two replication groups.

As a result, the above two types of group members not only provide different reliable communication methods, but also allow an application to choose different types of AQuA groups. For an illustration of the possible use of persistent and transient group members, consider Figure 1. Solid lines define the replication and connection groups. Each replication group implements a type of replication scheme. Interoperability between different replication schemes is supported though the connection group. The dashed oval represents the occurrence of a transient member joining a replication group. The leader of replication group 2 becomes a transient group member of replication group 1 in order to send messages to the replicas in replication group 2.

4. Passive replication schemes

In AQuA, passive replication is used to provide dependability to non-deterministic applications to tolerate process crash failures. During fault-free operation, only one member of the object group, the leader, executes the method invoked on the group. The gateways of the other replicas store the sequence of method invocations in a buffer but do not deliver the messages to their applications, and thus do not process the request. Periodically, the state of the leader is transferred to the other members (the backup replicas) or to stable storage. In the presence of a leader crash, a backup member becomes the new leader of the group and updates its state. Two types of passive replication schemes have been implemented: the *passive replication with state cast scheme* and the *passive replication with stable storage scheme*. In the first scheme, the leader multicasts its state to the backup replicas. This scheme provides a way for the backup replicas to access the state immediately during fault recovery. In the second scheme, the leader stores its state in stable storage. When the original leader fails, the new leader will take over from the original leader by getting the state from stable storage. Compared to the state cast scheme, this scheme reduces the number of messages transmitted over the network. However, the recovery time could be longer than in the state cast case, because the new leader needs time to get the state from stable storage.

In both of the passive replication schemes, an application can determine the periodicity of state capture. If the state is captured after more than one outgoing message has occurred (periodic state transfer), the application must be deterministic. Otherwise (for example, if the application is nondeterministic), the replaying of the same input messages will lead the new leader to a different state, which could generate inconsistent output messages. If the state is transferred after each outgoing message (every-message state transfer), the new leader will always receive the state that the original leader had before its last output message, and thus will generate output messages that are consistent with the original leader. In that case, the application does not need to be deterministic.

The challenging issue in designing the passive replication schemes was to guarantee strong data consistency among all replicas (even when replicas crash or generate wrong messages). Strong data consistency includes the ability to (1) ensure reliable transmission of each application message from one replicated object to another, so that messages will not be lost even if replicas crash, (2) guarantee that no duplicate messages are delivered to the replicated objects, and (3) react correctly when the number of replicas in a replication group changes. In order to maintain strong data consistency, each replication scheme contains a communication methodology that has several steps for sending each request and receiving the corresponding reply. By using these steps, the communication scheme can provide reliable message transmission, totally ordered messages across replicas, and automatic recovery from replica crash failures. In the following subsections, we focus on the state cast scheme with every-message state transfer strategy.

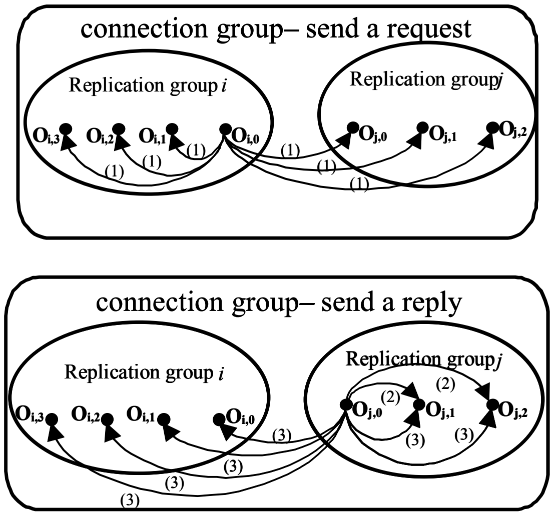


Figure 2: Passive Communication Steps

4.1. Communication steps

We briefly describe the passive communication steps before providing a detailed description in the next section. Suppose that replication group i is the sender group and group j is the receiver group, as shown in Figure 2. The replicas in replication group j are passively replicated. In the first step, the replicas in group j receive a request from the replicated object in group i. Only the leader $O_{\{j,0\}}$ processes the request, and the backup replicas $O_{\{j,k\}}$ keep copies of the request that is waiting to be processed. In the second step, when the leader $O_{\{j,0\}}$ generates a reply, it multicasts the reply together with the list of delivered messages and its gateway state to the backup replicas. The backup replicas keep copies of the reply, remove the request that has been processed by the leader, and update their gateway states. In the third step, the leader forwards the reply to the replicas in group i by multicasting it in the connection group. The backup replicas use the multicast as a signal to remove their copies of the reply.

4. 2. Communication scheme

Figure 3 illustrates the passive communication scheme in detail. When replicas receive messages from other AQuA objects (Figure 3, (1)), the backup replicas store the messages in their *input message buffer*s, which hold the incoming messages and will deliver them to applications when the backup replica becomes the leader. The gateway of the leader delivers incoming messages to its application, and also stores the headers of the delivered incoming messages in a delivered message list. The application does not need to know when its state should be captured. The timing of state captures is determined by an existing interceptor object provided by AQuA. This interceptor object was developed based on the ORB's interceptor technique, which provides access to messages when the messages are sent through the ORB. For the nondeterministic applications, whenever the leader of the replicas sends an output message, (Figure 3, (2) through (5)), the AQuA's interceptor will hold that message and then generate a *GetState* callback to the application. When the state is returned, the interceptor will forward the outgoing message together with the state to the gateway.

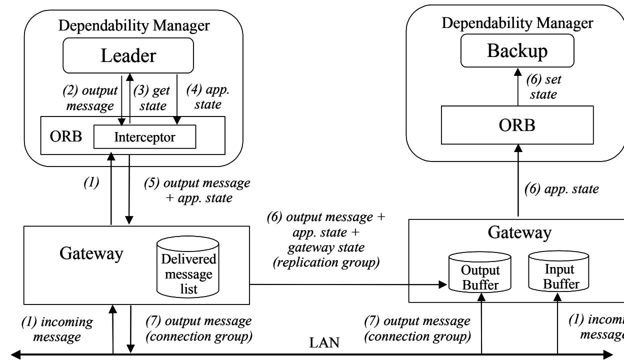
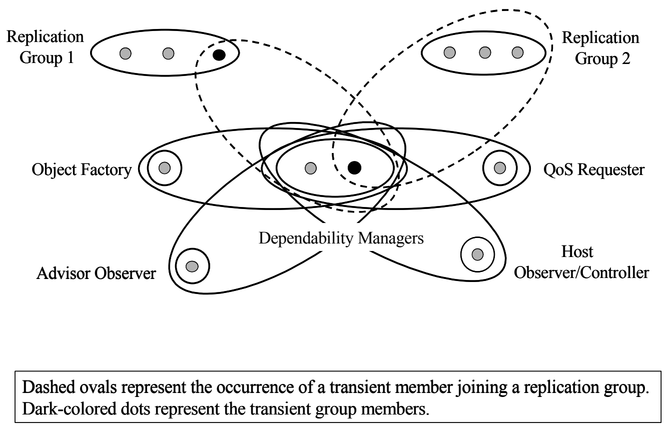


Figure 3: Passive Communication Scheme

When the leader's gateway receives the message (Figure 3, (6)), it will first put the output message, the state, and a list of delivered input messages together, and forward them to the backup replicas by multicasting them in the replication group. Then, the gateway will empty its delivered messages list. When the backup replicas' gateways receive this message (Figure 3, (6)), they place the output message in their *output buffer*s, remove the already delivered messages from their *input message buffer*s, and store the application state. The *output buffer*s are used in case the leader crashes before sending the output messages to the remote replicated objects. If that happens, the backup replica that takes over from the failed leader will get a copy of the messages from the *output buffer*s and then forward the messages to the remote object. Each time a backup replica receives a state transfer message, it stores the state, but will not begin to process the state until the backup replica becomes the new leader. When the backup replica becomes the leader, the gateway of the new leader will first send a notification message (*is_leader*) to its application, and then deliver the messages stored in the *input message buffer* to the application in sequence. The notification message is necessary to let the application start to recover state and process incoming messages. After that (Figure 3, (7)), the leader's gateway multicasts the output message to the connection group. This multicast will forward the message to the remote replicated object. The multicast is also used as a signal to the backup replicas to remove the output message from their *output buffer*s, since the remote objects must have also received copies of the message as a result of this reliable multicast.



Replication Group 1
Replication Group 2
Object Factory
QoS Requester
Dependability Managers
Advisor Observer
Host Observer/Controller
Dashed ovals represent the occurrence of a transient member joining a replication group.
Dark-colored dots represent the transient group members.

Figure 4: The Dependability Manager Group Structure

5. Making the dependability manager dependable

The AQuA dependability manager that provides dependability and resource management to distributed applications is an important component in the system. Preventing it from becoming a single point of failure was an important issue in the design of the AQuA architecture. Since the dependability manager is a multithreaded application, the passive replication state cast scheme with every-message state transfer is used to provide fault tolerance to the dependability manager.

The replicated dependability managers form a replication group, and communicate with the other application components through connection groups and through joining replication groups as transient members, as shown in Figure 4. Connection groups are used to communicate with the object factories, QoS requesters, advisor observers, and host observers/controllers [Ren99, Ren02]. The reason for using the connection groups is that the messages communicated between the dependability managers and the other components contain synchronous CORBA messages. Sending messages to a replicated object by joining its replication group as a transient member is used in three cases: when the leader of a replication group reports persistent group membership changes to the dependability managers, when the leaders report value faults to the dependability managers, and when the leader of the dependability managers changes the other replicas' gateway parameters. Becoming a transient member requires the dynamic joining and leaving of groups. Since the above cases do not happen often, frequent joining and leaving of groups should not occur. An alternative to communicating as a transient member is to communicate via connection groups. However, that approach requires the maintenance of a large number of groups. Using transient members can greatly reduce the number of connection groups kept by the dependability managers. As a consequence, we can avoid the scalability problem.

6. Performance measurement

We conducted some performance measurements using the passive replication with state-cast scheme. We developed the "deet" application, which uses the Visibroker ORB for Java, to help test the replication scheme. The measurement includes the round-trip time recorded in the application, and the round-trip time spent on handlers in the client gateway, the server gateway, and the group communication subsystem. The *application round-trip time* is the time from when an application sends an invocation until it receives a reply, as shown in Figure 5. The *handler round-trip time* is the time recorded from when the gateway ORB (TAO ORB) passes a request to the gateway handler until the handler receives the reply from the group communication subsystem and forwards it to TAO, minus the time needed by TAO in the server gateway and the server application to process the message. The time spent by TAO in the server gateway and the server application is measured from when the server gateway forwards the request to TAO until it receives the reply from TAO. By measuring delays in that way, we can determine the overhead in delay caused by the developed replication scheme on both the client and server gateways, and by the group communication subsystem, independent of the processing time taken to execute the remote method itself and the time spent in the server application's ORB. This measure can thus give us a good indication of the overall overhead added by making an application dependable, and the additional overhead added by the particular replication schemes. In the test, a single (unreliable) client was used to make requests. The server, which implemented the remote method described above, was replicated, with each replica running on a different host. In each study, we ran the instrumented code fifty times to generate each data point. The client was placed on the host that was the leader of the replicated servers, so that we could collect the data in the client and the gateway of the leader of the replicated servers without requiring clock synchronization between the client and server machines.

First, we studied performance for the passive replication scheme in terms of different numbers of replicas both with and without the application's state. The number of replicas, which reflects the level of dependability requirements, was increased from 1 to 11, and the application message length was 100 bytes. The average round-trip times over the fifty runs are shown in Figure 6. With application state (100 bytes), the leader of the replicated server transferred the application's state together with the gateway state to the backup replicas whenever it sent an output message. Without application state, the leader of the replicated server transferred its gateway state to the backup replicas only when it sent an output message. For example, when there were 3 replicas, with application state, the average handler round-trip time was 7.8 ms over 50 runs. Without application state, the handler round-trip time was 5.2 ms. From the figure, we can also see that as the number of replicas increases, the handler round-trip time increases slightly. This performance decrease is caused by the group communication system, since in order to ensure reliable and totally ordered message delivery, the cost of group communication grows with the size of the group.

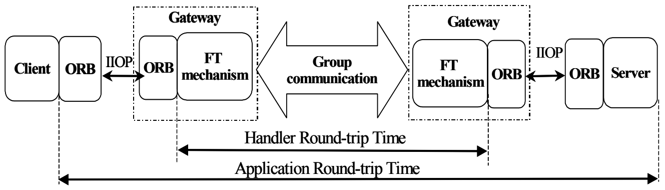


Figure 5: Application and Handler Round-trip Times

We then compared the application round-trip times and the handler round-trip times for the passive replication schemes with various message lengths (Figure 7). The message length is the length of the string that is sent from the client to the replicated server, and is returned back from the server to the client. In the comparison, the message size varies from 1000 to 10,000 bytes. As expected, the round-trip times spent on the application and on the handler increase with message size. In particular, for both cases, the application round-trip time increases greatly. Since the handler round-trip time is about 9% to 12% of the total application round-trip time, the overhead caused by the replication schemes and the group communication subsystem (handler time) is much smaller than the overhead caused by the TAO ORBs and the communication between the TAO and VisiBroker ORBs.

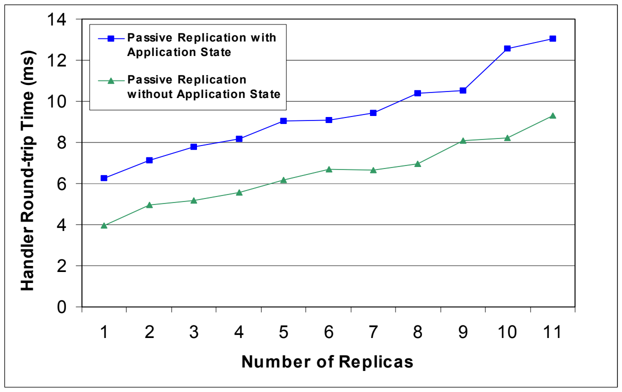


Figure 6: Handler Round-trip Time vs. Number of Replicas (Message Length = 100 bytes, Application State Size=100 bytes)

7. Conclusions

In this paper, we introduced two persistent and transient group members in AQuA to enhance applications' performance and scalability. Based on our unique group organization, the passive replication schemes were developed. In order to achieve strong data consistency among replicated objects, the passive replication schemes uses several steps to send each request and receive the corresponding reply. By using those steps, the passive replication schemes can provide reliable message transmission and automatic recovery from replica crash failures. In addition, this paper described how to make the dependability manager dependable by using the passive replication scheme. Finally, performance measurements were conducted. The passive state-cast replication scheme with different numbers of replicas and various message lengths was studied.

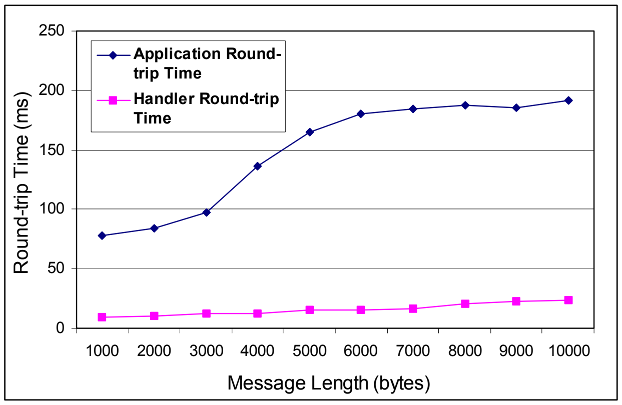


Figure 7: Round-trip Time vs. Message Length (Number of Replicas = 3, Application State = 100 bytes)

Acknowledgments

We would like to thank Jenny Applequist for her editorial comments, which greatly improved the paper. This research has been supported by DARPA Contract F30602-98-C-0187.

References

[AMW] R. Buskens, A. Siddlqui, and Y. Ren, "AURORA Management Workbench," Bell Laboratories, *http://www.bell-labs.com/project/aurora/motiv.html.*

[Bag98] S. Bagchi, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *Proc. of the 17th IEEE Sym. on Reliable Distributed Systems*, West Lafayette, IN, USA, Oct. 1998, pp. 261-267.

[Bha97] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote: A System for Constructing Fine-Grain Configurable Communication Services," Tech. Report TR97-12, Dep. of Computer Science, University of Arizona, Jul. 1997.

[Cuk98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, etc., "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *Proc. 17th IEEE Sym. on Reliable Distributed Systems*, West Lafayette, IN, USA, Oct. 1998, pp. 245-253.

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

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

[Nar99] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Proving Support for Survivable CORBA with the Immune System," *Proc. of the IEEE 19th Int. Conf. on Distributed Computing Systems*, Austin, TX, May 1999, pp. 507-516.

[OMG99] Object Management Group, "The common object request broker: architecture and specification," OMG technical document pts/99-06-0, Object Management Group, June 1999.

[Pow91] D. Powell, ed., *Delta-4: A Generic Architecture for Dependable Distributed Computing*, *ESPRIT Research Reports*, vol. 1, Springer-Verlag, 1991.

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

[Ren99] Y. Ren, M. Cukier, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, "Building Dependable Distributed Applications Using AQuA," *Proc. of the 4th IEEE Sym. on High Assurance Systems Engineering*, Washington D.C., Nov. 17-19, 1999, pp. 189-196.

[Ren01a] Y. Ren, M. Cukier, and W. H. Sanders, "An Adaptive Algorithm for Tolerating Value Faults and Crash Failures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 2, Feb. 2001, pp. 173-192.

[Ren01b] Y. Ren, "AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications," Ph.D. thesis, University of Illinois at Urbana-Champaign, 2001.

[Ren02] Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *IEEE Trans. on Computers*, VOL. 52, NO. 1, Jan. 2003.

[Vay98] A. Vaysburd and 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, pp. 73-80, 1998.