AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects[1]

Michel Cukier, Jennifer Ren, Chetan Sabnis, David Henke, Jessica Pistole, and William H. Sanders

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

David E. Bakken, Mark E. Berman, David A. Karr, and Richard E. Schantz

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

Abstract— **Dependable distributed systems are difficult to build. This is particularly true if they have dependability requirements that change during the execution of an application, and are built with commercial off-the-shelf hardware. In that case, fault tolerance must be achieved using middleware software, and mechanisms must be provided to communicate the dependability requirements of a distributed application to the system and to adapt the system's configuration to try to achieve the desired dependability. The AQuA architecture allows distributed applications to request a desired level of availability using the Quality Objects (QuO) framework and includes a dependability manager that attempts to meet requested availability levels by configuring the system in response to outside requests and changes in system resources due to faults. The AQuA architecture uses the QuO runtime to process and invoke availability requests, the Proteus dependability manager to configure the system in response to faults and availability requests, and the Ensemble protocol stack to provide group communication services. Furthermore, a CORBA interface is provided to application objects using the AQuA gateway. The gateway provides a mechanism to translate between process-level communication, as supported by Ensemble, and IIOP messages, understood by Object Request Brokers. Both active and passive replication are supported, and the replication type to use is chosen based on the performance and dependability requirements of particular distributed applications.**

1. Introduction

Many modern applications are distributed. By their very nature, such applications are difficult to build and maintain [Zin97]. Distributed object middleware, such as CORBA [OMG96], has contributed to the prevalence of distributed applications by simplifying their development and maintenance. It does so by hiding implementation details behind functional interfaces. However, many applications also have non-functional requirements such as dependability and performance. These requirements may change during an execution and are not explicit in implementations based on current middleware, making them difficult or impossible to achieve by current approaches.

There have been several efforts to add support for specifying non-functional requirements in distributed applications (e.g., [Koi97], among others). Quality Objects (QuO) [Zin97, Loy98], one such effort, allows distributed applications to specify quality of service (QoS) requirements at the application level using the notion of a "contract," which is a finite state machine specifying actions to be taken based on the state of the distributed system and the desired requirements of the application. The QuO framework provides an environment in which a programmer can specify contract states in terms of high-level QoS measures, the system elements that need to be monitored to determine the QoS that is being received, and the adaptation mechanisms that are used to try to achieve the desired QoS. In this framework, "property managers" are used to try to achieve desired QoS requirements, based on the desires of one or more QuO contracts.

This paper describes a use of the QuO framework called AQuA (Adaptive Quality of Service Availability). The goal of AQuA is to allow distributed applications to request and obtain a desired level of availability using a QuO contract through a property manager. In an attempt to meet the requested availability levels, the property manager configures the system in response to those requests and to changes in system resources due to faults. The AQuA framework uses the QuO runtime [Loy98] to process and make availability requests, the Proteus dependability manager to configure the system in response to faults and availability requests, and Ensemble [Hay98] to provide group communication services. In addition, a CORBA interface is provided to application objects using the AQuA gateway. The gateway translates between process-level communication, as supported by Ensemble, and IIOP messages, understood by Object Request Brokers (ORBs), in CORBA. The gateway also supports a variety of "handlers," which are used by Proteus to tolerate crash failures, value faults, and time faults.

Fault tolerance in AQuA is provided by Proteus, which dynamically manages the replication of distributed objects to make them dependable. The first function of Proteus is to decide, based on the desired availability of the application, how to provide fault tolerance. The choice of how to provide fault tolerance involves choosing a style of replication (active or passive), the type (algorithm and location) of voting to use, the degree of replication, the type of faults to tolerate (crash, value, or time), the location of the replicas, and other possible factors. This function is achieved by translating high-level availability requirements transmitted by the application through QuO into a system configuration with an adequate level of fault tolerance, if possible. The second function of Proteus is to implement the chosen fault tolerance scheme. After the system configuration has been chosen, Proteus takes action based on the decision by dynamically modifying the current configuration. This is done by starting/killing replicas and activating/deactivating fault tolerance mechanisms such as voters and monitors.

The AQuA architecture is not the only approach that tries to make distributed objects dependable. (A survey of closely

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

related work is given in Section 6.) It does, however, offer significant advantages over traditional ways of building dependable distributed systems. First, it raises the level of abstraction at which a programmer thinks about fault tolerance much higher, relative to existing group communication systems, allowing an application programmer high-level control over the type of faults that should be tolerated and the level of availability desired from a distributed object. Second, it dynamically adapts the configuration of a system at runtime, in response to faults that occur, to try to maintain a desired level of availability. Third, it recognizes that applications will require different levels of availability during different phases of their executions, and supports system reconfiguration in response to changing application requirements. In short, it provides a highly flexible approach for building dependable, distributed, object-oriented systems that support adaptation due to both faults and changes in an application's availability requirements.

The remainder of this paper is organized as follows. First, Section 2 presents an overview of the AQuA architecture, reviewing the core technologies that are used, and showing how they fit together. Section 3 describes how groups are used in AQuA to achieve different types of object replication and how reliable communication is achieved between groups of the same or different replication type. Section 4 describes the types of faults that can be tolerated, and how errors are detected and the corresponding faults treated. Section 5 then presents the architecture chosen to implement the described design. Section 6 reviews other approaches to achieving dependable distributed objects, putting the AQuA approach in context. Finally, Section 7 concludes this paper and offers suggestions for future work.

2. AQuA Overview

Before describing the AQuA architecture itself, it is helpful to review the technologies that are used in AQuA to support group communication (Ensemble) and quality of service specification (QuO). After doing so, we describe the AQuA architecture at a high level, focusing on the structure of Proteus and the AQuA gateway.

2.1 Ensemble and Maestro

Groups can be used in distributed computing systems to help manage the complexity of large applications or to help provide non-functional properties, such as availability or security. The full benefits of the group concept, however, can be realized only if one knows how to coordinate groups of processes that work together to fulfill a common purpose. To provide fault tolerance at the most basic level, the AQuA system uses the Ensemble group communication system [Hay98] to ensure reliable communication between groups of processes, to ensure atomic delivery of multicasts to groups with changing membership, and to detect and exclude from the group members that fail by crashing.

The Ensemble protocol stacks used in AQuA provide inter-process communication based on the virtual synchrony model [Bir96]. Both total and causal multicast are used in the AQuA group structure, resulting in a total order of delivered messages between different groups of replicated objects.

Ensemble assumes that the process failures are fail-silent (or crash failures), and detects process failures through the use of "I am alive" messages. The AQuA architecture uses this detection mechanism to detect crash failures, and provides input to Proteus to aid in recovery. Value and time failures are not detected by Ensemble, and hence must be detected at a higher level in the AQuA architecture, as will be described in Section 4.

Maestro [Vay97] provides an object-oriented interface (in C++) to Ensemble. Object-oriented applications can thus be written by deriving from Maestro classes that provide reliable communication.

2.2 Quality Objects

QuO [Zin97, Loy98] allows distributed object-oriented applications to specify dynamic QoS requirements. The goal of QuO is to develop a common middleware framework, based on distributed object computing, that can manage and integrate non-functional system properties such as network resource constraints, availability requirements and security needs. In the AQuA approach, QuO is used to transmit applications' availability requirements to Proteus, which attempts to configure the system to achieve the desired availability. QuO also provides an adaptation mechanism that is used when Proteus is unable to provide a specified level of availability.

In particular, contracts in QuO can have multiple availability requirements specified *a priori* to allow for multiple fallback positions, if Proteus fails to provide a requested level of availability. To do this, QuO uses two types of region to summarize conditions of interest. Negotiated regions specify the expected behavior of the local and remote object, and are defined by predicates on the state 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. They are also specified by predicates, this time 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 availability requirements are being met. At any given time, there is one current negotiated and reality region (if more than one predicate is true, one is chosen). A transition between regions occurs when the predicate of the current one becomes false and a new one is true. Transitions can be used to request changed levels of availability to Proteus, or to notify the local object that the desired level of availability could not be met, using a callback to the local object.

In this way, QuO provides mechanisms for distributed applications to inform a system of the level of availability they desire, mechanisms for a system to inform applications of the level of availability that they are receiving, and mechanisms for adaptation under changing fault, workload, and environmental conditions. Applications thus have a simplified awareness of system conditions, without having total responsibility for managing availability, and the system can effectively manage the applications' desires in changing system conditions. From the functional point of view, the use of QuO makes no difference to the application, since QuO provides the same IDL interface to the client.

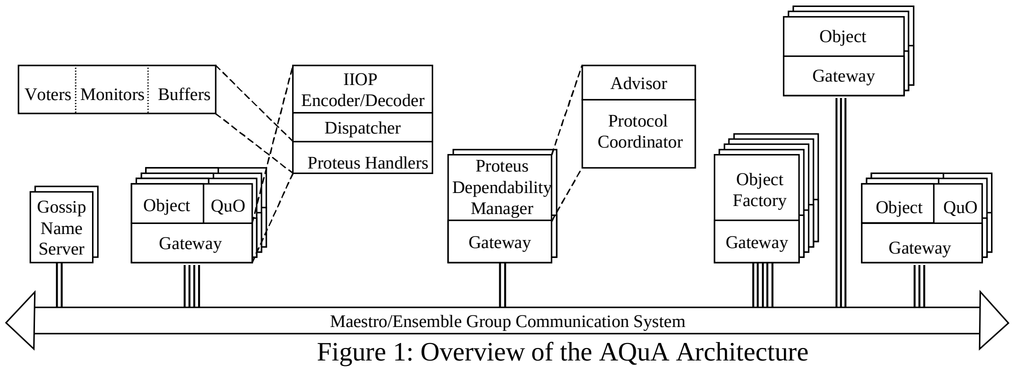

Figure 1: Overview of the AQuA Architecture

2.3 Proteus

Most group communication systems, including Ensemble, are based on the assumption that processes fail by crashing, but no mechanism is implemented to ensure that processes fail only by crashing. Furthermore, recovery by automatically starting new processes on the same or different hosts is not implemented in the protocol stack. Instead, it is left to the application. A fault tolerance framework is thus necessary to tolerate other fault types and provide more sophisticated recovery mechanisms than process exclusion. The framework could be implemented at the process level by implementing further fault tolerance in Ensemble. However, in order to be independent of any particular group communication system and to fully use the features offered by CORBA applications, we have provided additional fault tolerance above the group communication infrastructure. The framework we have developed is able to tolerate crash failures of processes and hosts, as well as value and time faults of CORBA objects. In addition to the fault tolerance mechanisms themselves, two types of replication can be used: active and passive. For each replication scheme, different communication schemes can be used. Different replication schemes and communication schemes enable tolerance of different types of faults, and provide different recovery characteristics.

Proteus consists of a set of monitors, voters, object factories, and a replicated manager. The manager contains both an “advisor” and a “protocol coordinator.” The monitors and voters are in the path of remote object invocations, and handle replies from a set of replicas. Specifically, voters decide which, if any, of the replies from replicas to present to an object, and monitors implement timers to detect delay and omission faults. Object factories start and kill objects on hosts, under the direction of the manager. The manager receives requests from multiple QuO runtimes regarding desired availability of specific remote objects and, using its advisor, makes decisions regarding the type of fault tolerance to provide. The protocol coordinator within the manager then carries out the decisions of the advisor, through communication with the handlers and object factories.

2.4 AQuA Architecture Overview

Figure 1 shows the different components of the AQuA architecture, in one particular configuration. These components can be assigned to hosts in many different ways, depending on the availability that objects desire of remote objects they use.

As pictured in this figure, a QuO runtime is associated with each process that contains objects that make remote object invocations with managed availability. Each QuO runtime manages one or more QuO contracts. The QuO runtime may run either as a separate process, or, if the application is written in Java, in the same process as the application. For simplicity in the discussion that follows, we assume it is a separate process, although either configuration can be used in AQuA. The QuO runtime, like the application, communicates using CORBA, sending requests to Proteus for the use of remote objects with certain availabilities, and receiving information from Proteus regarding the level of availability that is being provided.

An AQuA system also contains one or more Proteus managers, which determine a configuration of the system based on reports of faults and desires of application objects. A Proteus object factory resides on each machine that can support distributed objects and is used to create and destroy objects, as well as to provide load and other information to Proteus managers.

Communication between all architecture components (i.e., applications, the QuO runtime, object factories, and Proteus managers) is done using gateways, which translate CORBA object invocations into messages that are transmitted via Ensemble. Communication between each component and a gateway is via IIOP messages generated by standard ORBs associated with each component. Details about how this is done are given in the next two sections. The gateway is written in C++ using the Maestro interface to Ensemble. In addition to translating messages between IIOP and Ensemble formats, the gateway implements Proteus monitors and voters, using handlers. Finally, the Gossip name server, which is part of Ensemble, is used to provide name service to Ensemble processes.

3. Groups in the AQuA Architecture

This section describes the group structure that is used to achieve reliable multicast and point-to-point communication among multiple groups of replicated objects, object factories, the replicated Proteus manager, and QuO.

In AQuA, we use a general object model, rather than the more restrictive client/server model. The model of computation is thus based on interactions between objects that can be replicated. Objects can initiate requests (acting as clients) and respond to requests (acting as servers). In the AQuA architecture, the basic unit of replication is a two- or three-process pair, consisting of either an application and gateway, or application, gateway and QuO runtime. A QuO runtime is included if an object contained in the application process makes a remote invocation of another object and wishes to specify a quality of service for that object. A basic replication unit may contain one or more distributed objects, but to simplify the following discussion, we refer to it as an "object." Furthermore, when we say that an "object joins a group" we mean that the gateway process of the object joins the group. Mechanisms are provided to ensure that if one of the processes in the object crashes, the others are killed, thus allowing us to consider the object as a single entity that we want to make dependable.

Using this terminology, we can now describe the group structure and mechanisms used for intra-group multicast, inter-replication group communication via connection groups, and point-to-point communication. Four group types are used in the AQuA architecture: "replication groups," "connection groups," "PCS (Proteus Communication Service) group," and "point-to-point groups." By defining multiple replication and connection groups and a dependable method for communication, we can avoid the unacceptable communication overhead associated with large groups. This provides a scalable architecture.

3.1 Replication Groups

A replication group is composed of one or more identical objects. Each 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 performs special functions, as will be described in the next section. For implementation simplicity, we designate the object whose gateway process is the Ensemble group leader of the corresponding process replication group as the leader of the object group, since we can then use the Ensemble leader election service to elect a new leader if the leader object fails.

In the AQuA architecture, two types of replication are performed in replication groups: active and passive. In addition, different communication schemes can be implemented. Each configuration has advantages and disadvantages from performance and dependability perspectives. The type and degree of replication used is determined by the requirements of calling objects, as communicated by the QuO runtimes of each calling object.

In active replication, all replicas process input messages. Different communication schemes can be used: 1) only the leader replica sends out replies; 2) all send out replies, and the first is chosen; or 3) all send replies, and voting occurs. The type of communication that is employed depends on the type of faults tolerated and the requirements of the application. Active replication, as currently implemented in AQuA, thus requires determinism; i.e., for each replica, the same inputs must lead to the same output. Future versions of the architecture may allow non-determinism, by providing high-level hooks that replicas can use to initiate an agreement protocol to synchronize important parts of their states. Tolerating a crash failure of a leader replica is faster in active replication than in passive replication. Furthermore, since all replicas process the requests, total ordering is required, and is provided by the inter-group communication scheme described in the next subsection. Active replication, however, has increased computation requirements, relative to passive replication, since all replicas need to process each request.

When voting is done, replication groups that use active replication can tolerate three classes of faults: crash failures, value faults and time faults. If one of the first two communication mechanisms is used, value failures cannot be tolerated. However, these communication approaches have performance advantages over active replication with voting. In particular, if only one replica sends replies, only it needs to process each message immediately upon receipt (allowing requests within other objects to be scheduled in a less urgent fashion), since only the leader provides replies. Furthermore, if all replicas send replies, but the first is used, remote invocations can complete more quickly than if a vote takes place.

In passive replication, only the primary replica processes input messages and sends out the replies. In the absence of faults, other replicas, called standby replicas, do not process the input messages and do not send out the replies. The state of the standby replicas must therefore be regularly updated. Since only one replica processes requests, this method is not able to tolerate value failures. On the other hand, unlike active replication, determinism is not required. Moreover, total ordering of requests and replies is not required (causal ordering can be used). Finally, passive replication has low computation overhead, relative to active replication, but leads to increased communication overhead, because of the regular state transfers required.

3.2 Connection Groups

Communication between replication groups is done using connection groups. Such communication must be done in a way that supports the ordering needed for the type(s) of replication used in the constituent replication groups, and that allows for recovery from faults that occur in any phase of communication between two groups of replicas. Informally, a connection group is a group consisting of the members of two replication groups that wish to communicate. The goal in defining connection groups is to provide the properties of group communication (atomic multicast with total or causal ordering) when communicating between replication groups, while avoiding the overhead associated with single, large groups.

This section will describe the example, for active replication in both replication groups, of a synchronous communication scheme where all replicas in the replication groups reply and the first reply is chosen. To specify precisely how this is done, 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 2 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.

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 $O_{i,0}$ (arrows labeled "1" in the figure). The 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 (arrows labeled "2"). The objects $O_{i,k}$ use the multicast to signal that they can delete their local copy of the request. The objects $O_{j,k}$ ($k=0, \ldots, no_j-1$) 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 (the arrows labeled "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 between 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.

3.3 The Proteus Communication Service Group

Reliable multicast is also needed for communication with the replicated Proteus manager. In AQuA, this is achieved using the Proteus Communication Service (PCS) group. The PCS group consists of the Proteus replicas and objects that desire to communicate with the Proteus manager. In particular, objects complaining about an object value or time fault, providing notification of a view change, or communicating a QuO request to the Proteus manager will join the PCS group when they wish to communicate. After having communicated the desired information, the communicator will leave the PCS group. Since communication with the Proteus manager is fairly infrequent, the overhead in joining and leaving the PCS group is small relative to maintaining a group that consists of all objects that may want to communicate with the Proteus manager. By using a group structure to communicate with the replicated Proteus manager, we ensure that all replica managers receive the same information.

3.4 The Point-to-Point Groups

A point-to-point group is used to send messages from a Proteus manager to an object factory. Each factory object is in its own point-to-point group. When the Proteus manager wishes to send a message to the receiving object, the Proteus manager joins the group for that object. Once the Proteus manager is a member of the group, it multicasts its message, waits for an acknowledgement, and then leaves the group.
By using this group for communication of messages from a Proteus manager to other objects, all communication between any two objects in the AQuA architecture can be monitored at the level of the gateways. Note that not all Proteus manager replicas are involved in joining the group. That is because there is higher overhead in Ensemble if multiple objects join a group simultaneously.

For an illustration of replication groups, connection groups, and the PCS group, consider Figure 3. Solid lines define the replication and connection groups. Dashed lines represent the PCS group. We see in Figure 3 that even though a connection group is composed of two replication groups, a replication group can be included in several connection groups. The structure of the PCS group in the figure shows that the leaders of the replication groups 2 and 4 are communicating with Proteus. Note that, most of the time, only replication group leaders will join the PCS group. However, in

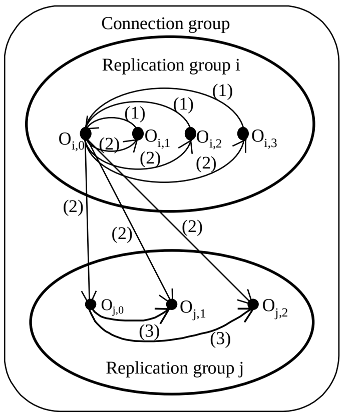

Figure 2: Communication in AQuA via Connection Groups

case of time faults, any member of a replication group may join the PCS group to complain about the time fault. In the next section, we describe how this group structure is used to tolerate and treat crash failures, and value and time faults.

4. Fault Tolerance in AQuA

Most group communication systems, including Ensemble, assume that processes fail only by crashing, and do not automatically restart replicas after a failure occurs. Since no mechanism is implemented in Ensemble to ensure that only crash failures occur, additional fault tolerance is needed to tolerate value and time faults. This section explains how this is done using the group organization described in the previous section, voters and monitors in gateways, and the Proteus manager. Before explaining how fault tolerance is implemented, it is important to precisely define the class of faults we consider.

4.1 Fault Model

Proteus handles object faults of three types: crash failures, value faults, and time faults. In doing so, we do not explicitly consider link faults, and note that they may exhibit themselves as one or more (possibly correlated) object faults, which can be detected using the methods described in this section. Furthermore, we assume that Proteus only needs to be concerned with value faults that occur within objects themselves, since faults that occur on links can be detected using conventional coding techniques. More precisely, we tolerate value faults that occur in the content of messages transmitted from an application or QuO runtime.

A crash failure occurs when an object stops sending out messages and when the internal state is lost. In the AQuA architecture, the crash failure of an object is due to the crash failure of at least one element composing the object. The software is thus implemented in a way that causes the crash of one process to cause the others to be killed. In particular, if the QuO runtime and/or the application appear to have crashed, the gateway process kills the non-failed process before killing itself. When a gateway crashes, the other (non-failed) processes in the object are also killed, using Unix signals.

A value fault occurs when the message arrives in time but contains the wrong content. In Proteus, we assume that the gateway does not fail by sending out wrong messages. Value faults of an object are thus due to value faults of the application and/or the QuO runtime. Proteus contains mechanisms to tolerate value faults originating in both places.

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. The handler in each gateway possesses mechanisms to tolerate such time faults. In particular, two timeouts for delay and omission faults are introduced. Delay faults occur when messages arrive after the timeout defining a delay fault but before the timeout associated with an omission fault. An omission fault is thus a message that is received after the timeout defining omission faults, or that is not received.

4.2 Error Detection

Proteus detects single and multiple process crash failures (a host failure is a well-known example of a multiple process crash failure). The detection mechanism used by Proteus to detect crash failures is based on the detection mechanism implemented in Ensemble. Among the elements composing the object, only the gateway process is an Ensemble process. However, since the crash of the application process or the QuO runtime process leads directly to the crash of the gateway process (cf. Section 4.1), Ensemble can detect the crash of any element of an object. The detection mechanism in Ensemble requires that each Ensemble group member (gateway process) regularly broadcast “I am alive” messages inside the group. The frequency of the messages can be based on the needs of the application. If several messages are not received from a given replica, the replica is considered to have failed. The leader of the group then excludes the presumed failed replica by initiating a view change of the group composition. This view change is communicated to the Proteus manager through the PCS group. The comparison between the old structure of the group and the new composition allows the dependability manager to detect the crash failure.

In the gateway handler, voters detect value faults that occur in the QuO runtime and/or in the application part of each object. A voter is implemented in the gateway part of each object, but only the voter present in the leader of the replication group is active. When an object on the client side sends out a request (the request is sent from either the QuO runtime or the application through the gateway), it sends it to the leader of its replication group. Then the voter of the leader votes on the requests. If some requests differ from the majority, a (single or multiple) value fault has occurred. In this case, the leader gateway process of the replication group joins the PCS group to notify the Proteus manager about the value fault. Equivalently, on the server side, after a request has been processed by the different replicas of the replication group, all replicas send back their reply to the leader of the server replication group. The voter of the leader then votes on the different replies. A value fault has occurred if one or more replies differ from the majority. The leader gateway process then joins the PCS group to complain about the value fault. Proteus thus detects a value fault by the communication of the complaint when the leader joins the PCS group.

Time errors are detected by monitors that record information regarding various times and omissions. Where and how the timers operate depends on the type of faults that are being tolerated. A monitor is implemented in the gateway part of each object. When tolerance to time faults is required,

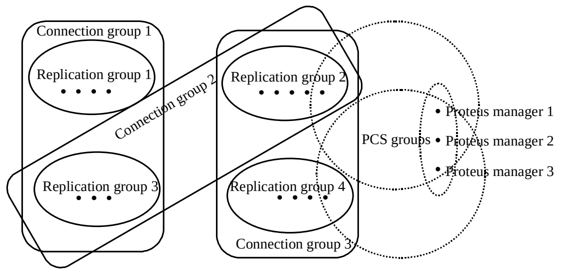

Figure 3: Example Group Structure in AQuA

all monitors of the object members of the replication groups where tolerance is desired are activated. Since several times need to be recorded, several different types of monitors are used. In each case, the recorded time depends on whether the concerned object is the leader of a replication group, and on whether the object is on the sender or the receiver side.

Time faults are communicated to the Proteus manager using the PCS group structure described earlier, i.e., the gateway of the object that detects the fault joins the PCS group, multicasts the complaint to the Proteus managers, and then leaves the PCS group. These times are used to diagnose the cause of the time fault, so treatment can take place.

4.3 Fault Treatment

Decisions regarding fault treatment are made by the Proteus manager advisor, using fault information communicated from the gateways. After a decision is reached, the object factories and gateways make the configuration change, under control of the protocol coordinator. In this section, we describe how crash failures, value faults, and time faults are treated using these Proteus components.

Specifically, in the case of a crash failure, since the number of replicas may need to be maintained, a new object may be started either on the same host or on another host. When and where the new object is started is decided by the advisor. The new object joins the replication group from which the replica crashed and the state of the leader is transferred to the new replica. The state of the gateway of the leader object is transferred using Maestro. The states of the QuO runtime and the application are transferred from the corresponding elements of the leader object by using a state transfer mechanism independent of the group communication framework. State transfer is performed by invoking the **get_state** method on an existing replica and then invoking the **set_state** method on the new replica. **Get_state** and **set_state** are CORBA-style methods that must be specified by an application developer in order to use AQuA with the application.

For value and time faults, the fault treatment consists of two phases. First, the source of the fault is determined, based on information provided to the advisor. Note that for value and time faults, the fault may not be in an object that reports a fault. Using complaints from the various object monitors and voters, the advisor decides whether to kill suspected replicas, whether to start new replicas, and where to start new replicas. Second, the replicas for which a value or a time fault has been detected may be killed (the three elements composing the failed objects are killed) and new replicas be started, if mandated by the advisor, in order to maintain the global number of replicas. They can be started on the hosts on which the replicas have been killed or on other hosts. The newly created objects then join the replication groups from which objects have been killed, and the leaders of these replication groups transfer their state to the new objects. Specifically, the state of the gateway of the leader object is then transferred to the new objects using Maestro, and the states of the QuO runtime and the application are transferred using the same method as described for crash failures.

5. AQuA Architecture

From a structural point of view, AQuA consists of one or more possibly replicated objects (containing a gateway, and possibly, a QuO runtime) and Proteus. Each of these constituent pieces is described in more detail below.

5.1 QuO Runtime

The AQuA architecture is designed to support adaptation at many different levels to changing conditions that may affect the dependability of the entire system. The overall philosophy is to isolate adaptation at the lowest level capable of adequate response. Thus, Proteus bears responsibility for configuring the system to ensure dependability and for responding to individual failures. However, there are cases where application availability requirement change or Proteus is not able to deliver the requested level of availability. In these cases, adaptation requires cooperation between the application and the Proteus dependability manager. The QuO runtime provides services to facilitate such cooperation. The most important of these services are contracts, system condition objects, and delegates. As mentioned previously, QuO contracts provide a high-level summary of the level of availability requested by an application and the level of availability actually being delivered. The contract is accessed through QuO system condition objects and delegates.

Within the QuO runtime, system condition objects provide the primary windows into the dependability management capability provided by Proteus. System condition objects present a very simple interface to the application programmer, typically exporting only **set_value** and **get_value** methods. A contract receives its input information by monitoring the values of specific QuO system conditions. This monitoring is achieved by indicating in the contract's description that the contract "subscribes" to some collection of system conditions. For instance, AQuA availability contracts subscribe to system condition objects set by the application to specify the desired level of availability. In addition, they subscribe to system condition objects that are regularly updated to indicate Proteus's best current estimate of delivered availability. Whenever any of these subscribed objects is updated, the contract reevaluates the current contract regions to determine whether any changes require adaptive response from the application. This response may be implemented either in-band (synchronously) with object group method invocations or out-of-band (asynchronously). Out-of-band response is identified in the contract description and triggered by transitions between contract regions. For instance, if Proteus reports reduced availability for a particular object group, a contract reevaluation might invoke a designated adaptation procedure within the affected application. The application then adapts to compensate, perhaps by using an alternative implementation that does not rely on the questionable object group.

In-band application-level adaptation is facilitated by QuO delegates. A QuO delegate is a software component that provides for QoS-adaptive behavior while exhibiting the functional behavior of an ORB proxy object. That is, a QuO delegate has the same IDL interface as the remote object on which the client program is performing a remote method invocation. However, the delegate contains code that checks the current contract regions and changes behavior appropriately. All method invocations on QuO objects are routed through a corresponding QuO delegate. In most cases, the delegate checks the current contract regions and acts as a pass-through, simply passing the method invocation on to the remote object group. However, in cases where the current contract status indicates a non-standard condition, the delegate may provide for different behavior.

5.2 Proteus

Structurally, Proteus consists of a dependability manager, handlers (which implement voters and monitors in the gateway), and object factories. The dependability manager is replicated, and consists of an advisor and a protocol coordinator. The advisor makes decisions on how to reconfigure a system, based on faults that occur and the aggregate availability requirements of applications as communicated via the QuO runtime. The protocol coordinator then takes actions based on these decisions.

Voters and monitors are implemented in gateway handlers, described more precisely in the next subsection. Two types of voters are currently implemented in the gateway. The first one forwards the first reply that arrives (without a vote). Note the new use of the term "voter" here; in previous sections we only called the function "voting" if a vote took place, but now, from an implementation point of view, forwarding the first reply is done by a voter. The second type of voter sends out a reply only when a majority of replies are identical (*2k+1* replicas are needed to tolerate *k* value faults; the majority is then reached when *k+1* messages are identical). This voter is used with active replication when value faults must be tolerated. When active replication is used but value faults do not need to be tolerated, the first type of voter is used. Monitors are used to detect time errors by setting timers as needed to detect when delays or omissions occur.

An object factory is implemented on each host. The function of each object factory is to kill processes, to start processes, and to measure the host load, in order to provide information to the advisor to be used in deciding which hosts to start objects on.

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. In particular, the advisor tells the protocol coordinator

- where to start objects,
- where to kill objects,
- what the replication type of a replication group is,
- which voter type to set,
- which monitor to set, and
- which host load to check.

The algorithms implemented in the protocol coordinator manage, by interacting with handlers and object factories, the starting and killing of objects, the changing of voter and monitor types, and checking of host loads. Furthermore, the protocol coordinator implements the type of replication for each replication group and coordinates the level of fault detection (setting of voter and monitor types), the fault treatment (killing and starting of replicas) and the load checking.

The advisor determines an appropriate system configuration based on requests transmitted through QuO contracts and observations of the system. The QuO contract contains requirements concerning the fault tolerance and the performance of the system. Among other things, a QuO contract in AQuA specifies

- the fault types (crash failures and/or value faults and/or time faults) to tolerate for each replication group,
- the number of simultaneous faults of each type to tolerate,
- the timeouts defining a delay fault and an omission fault,
- the minimum probability of proper service for a given time interval, and
- the duration during which Proteus can try to recover transparently from a failure resulting in an unacceptable configuration before having to send a callback to the application (via QuO) indicating that the configuration is not acceptable.

For each replication group that uses QuO, this information is communicated to the advisor, which uses it, in conjunction with information regarding host loads and faults and failures that have occurred, to determine the configuration the system should be in. All decisions of the advisor resulting in actions are communicated to the protocol coordinator, which carries them out.

5.3 Gateway

The AQuA gateway translates between CORBA object invocations generated by application objects and the QuO runtime and messages that are multicast by Ensemble in a way that allows for flexible and extensible handling and the filtering of duplicate requests and replies. Figure 4 illustrates the different parts of a gateway: the IIOP gateway, the dispatcher, and the handler. The IIOP gateway decomposes IIOP messages and places them on a queue for the dispatcher. The dispatcher provides bookkeeping information across multiple handler instances (e.g., if an object has multiple contracts involving multiple application groups). The handler contains voters, monitors, and buffers that are controlled by the Proteus advisor, and is the interface to Maestro and the group communication services.

The translation between CORBA request/replies and Ensemble messages is done as follows. Specifically, as shown in Figure 4, an invocation from a CORBA object goes through the IIOP gateway, the dispatcher, and the handler before reaching Maestro and Ensemble. In the handler, a message will go through buffers and possibly monitors and voters. A reply, which comes in as a message at the Maestro-Ensemble level, goes through the handler, the dispatcher and the IIOP gateway before reaching the CORBA ORB. As with the outbound request, the message may pass through the voter and/or monitor, depending on the type of faults that are being tolerated. In addition, both the request and reply are in the handler until rebroadcast by the replication group leader (as described in Section 3) to tolerate crash failures.

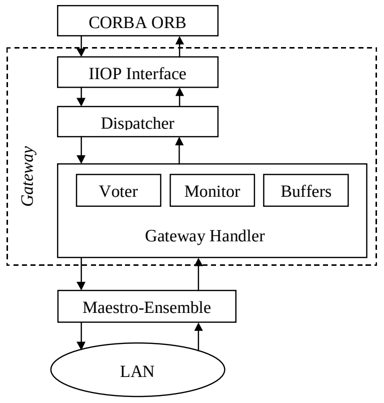

Figure 4: AQuA Gateway Structure

6. Related Work

There has been much work on building reliable distributed systems; however, most of this work has focused on the process level, and has not used CORBA or a high-level quality of service specification method such as QuO to specify an application's desired availability at a high level.

Specifically, one main thrust has been to provide fault tolerance at the process level through the use of the group communication paradigm. Work in this area has been extensive, and includes ISIS [Bir94], Horus [Ren96], Maestro/Ensemble [Vay97, Hay98], Totem [Mos95], Transis [Dol96], ROMANCE [Rod93], Cactus [Bha97], and Rampart [Rei96], among others. Among these, Maestro/Ensemble is unique in that it provides a CORBA invocation interface but does not provide the support for replication provided by the gateway handlers, and Rampart is unique in that it provides support for tolerating malicious intrusions.

In addition, there has been work with the explicit goal of building fault tolerant systems. In particular, the Delta-4 project [Pow91, Pow94] aimed to provide fault tolerance through the use of an atomic multicast protocol, specialized hardware to ensure crash failure of processes, and support for active, semi-active, and passive replication. Also notable are Chameleon [Bag98], the FRIENDS system [Fab98], MARS [Kop88], the Sun Enterprise Cluster [Sun97], and Wolfpack [Wolf97]. All of these provide explicit support for building fault tolerant applications. However, they do not permit the specification of, and adaptation based on, high-level availability requirements as is possible using AQuA.

Several systems have also been developed to provide fault tolerance for CORBA applications. Three systems with goals similar to AQuA are the Eternal system [Nar97], OpenDREAMS [Fel96] and Piranha-Electra [Maf95, Maf97].

The Eternal system adds fault tolerance to CORBA applications by object replication. Replica consistency is maintained by total ordering of multicast operations, detection of duplicate invocations and responses, transfer of state between replicas, consistent scheduling of concurrent operations, and fulfillment operations for restoring a consistent state after network partitioning and remerging. The Eternal system contains a "translator" called the Interceptor, which maps between CORBA objects and the group communication (Totem [Mos95]) processes.

The Eternal system is close in spirit to the AQuA architecture; however, two major points differentiate the two projects. First, in the Eternal system, fault tolerance is developed at the level of the group communication system. In the AQuA system, significant fault tolerance is implemented in the gateway, above the group communication system. Second, Eternal does not support dynamic system configuration changes in response to changing application requirements, as AQuA does. In AQuA, this functionality is achieved by communication between the application, the QuO runtime, and the Proteus manager.

Electra [Maf95] provides fault tolerance to CORBA by building a specialized ORB. The Electra ORB adds several properties of group communication systems to a common ORB. In particular, Electra allows dynamic replication of important object implementations. Moreover, a failure detection service is provided to detect and report failed objects consistently.

Piranha can be seen as a basic version of Proteus, managing only crash failures and replacing the advisor by a GUI. However, since Electra uses a non-standard ORB to provide group communication services, it is incompatible with other ORBs if the fault tolerant features are used. In contrast, the AQuA architecture uses a gateway that translates IIOP messages understood by standard CORBA ORBs into messages for the processes of the group communication system.

The OpenDREAMS research project has focused on the design and the implementation of an Object Group Service (OGS), which provides facilities for CORBA object group communication. The mechanisms used to build the group framework are group multicast, dynamic group membership, view change and state transfer. This object-level group communication system is simpler and less flexible than Ensemble but has the advantage that it is implemented on top of CORBA objects as a new CORBA service. This approach is promising, and has the potential to provide group services to CORBA objects; however, it requires that the application developers be aware of and explicitly make use of the OGS.

7. Conclusions

This paper presents an overview of the AQuA architecture, which provides a flexible and extensible approach to building dependable, object-oriented distributed systems. Systems built using the AQuA architecture support adaptation due to both faults in the environment and changes in an application's availability requirements. Within the AQuA architecture, the QuO runtime supports generation of an application's availability requests, the Proteus dependability manager supports configuration of the system in response to faults and availability requests, and Ensemble supports group communication services. In addition, the AQuA architecture offers a standard CORBA interface to applications, providing all the advantages of CORBA in developing and maintaining distributed applications.

Acknowledgements

We would like to thank several other members of the AQuA and QuO teams, namely Ayesha Ibrahim, Joe Loyall, Paul Rubel and John Zinky, for support and discussions. We would also like to thank several members of the Ensemble team, Ken Birman, Tim Clark, Mark Hayden, and Alexey Vaysburd, for their help in using Maestro and Ensemble. Finally, we would like to thank Silvano Maffeis for discussions regarding the use of CORBA with group communication systems.

References

[Bag98] S. Bagchi, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," To appear in Proc. of the 17th IEEE Symposium on Reliable Distributed Systems, Purdue University, West Lafayette, IN, USA, October 1998.

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

[Bir94] K. P. Birman and R. van Renesse (Eds.), "Reliable Distributed Computing with the Isis Toolkit," Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Bir96] K. P. Birman, "Building Secure and Reliable Network Applications," Greenwich, CT: Manning Publications, 1996.

[Dol96] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication," Comm. of the ACM, vol. 39, no. 4, 1996, pp. 64-70.

[Fab98] J-Ch. Fabre and T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach," IEEE Trans. on Computers, vol. 47, no. 1, 1998, pp. 78-95.

[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.

[Kar97] D. A. Karr, "Specification, Composition, and Automated Verification of Layered Communication Protocols," Ph.D. thesis, Cornell University, 1997.

[Koi97] J. Koistinen, "Dimensions for Reliability Contracts in Distributed Object Systems," HP Laboratories Technical Report, October 1997

[Kop88] H. Kopetz, et al., "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," IEEE Micro, vol. 9, no. 1, pp. 25-40.

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

[Lap92] J.-C. Laprie, ed., "Dependability: Basic Concepts and Terminology," Springer-Verlag, Vienna, 1992.

[Loy98] J. P. Loyall, R. E. Schantz, J. A. Zinky, D. E. Bakken, "Specifying and Measuring Quality of Service in Distributed Object Systems," To appear 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.

[Mos95] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. Lingley-Papadopoulos, T. P. Archambault, "The Totem System," Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing, pp. 61-66, Pasadena, CA, June 1995.

[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.

[OMG96] Object Management Group, "CORBA 2.0, July 96 revision," OMG Document 96-08-04, July 1996.

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

[Pow94] D. Powell, "Lessons Learned from Delta-4," IEEE Micro, vol. 14, no. 4, 1994, pp.36-47.

[Rei96] M. K. Reiter, "Distributing Trust with the Rampart Toolkit," Comm. of the ACM, vol. 39, no. 4, 1996, pp. 71-74.

[Rém98] D. Rémy, J. Vouillon, "Objective ML: An Effective Object-oriented Extension to ML," To appear in Theory And Practice of Objects Systems, 1998.

[Ren96] R. van Renesse, K. P. Birman, S. Maffeis, "Horus: A Flexible Group Communication System," Comm. of the ACM, vol. 39, no. 4, 1996, pp. 76-83.

[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.

[Sun97] Sun RAS Solutions for Mission-critical Computing, White Paper, October 1997, http://www. sun.com/cluster/wp-ras/

[Vay97] A. Vaysburd, K. P. Birman, "Building Reliable Adaptive Distributed Objects with the Maestro Tools," Proc. of Workshop on Dependable Distributed Object Systems, OOPSLA'97, Atlanta, Georgia, October 1997.

[Wolf97] Microsoft Clustering Architecture "Wolfpack," White Paper, May 1997, http://www.microsoft.com/ntserver/info/ wolfpack.htm.

[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, pp. 55-73, April 1997.