An Object-level Gateway Supporting Integrated-Property Quality of Service

Richard Schantz, John Zinky, David Karr, David Bakken, James Megquier, Joseph Loyall
BBN Technologies/GTE Internetworking
10 Moulton Street; Cambridge, Mass. 02138 USA
{schantz,jzinky,dkarr,dbakken,jmegq,jloyall}@bbn.com

Abstract

As networks and the use of communication within applications continue to grow and find more uses, so too does the demand for more control and manageability of various "system properties" through middleware. An important component supporting an integrated property architecture is the concept of an object gateway, which is a quality-of-service (QoS) aware element transparently inserted at the transport layer between clients and objects to provide the managed communication behavior for the particular property being supported. In this paper, we introduce the concept of a QoS-oriented gateway to integrate a variety of QoS enforcement and implementation mechanisms controlling the underlying distributed interactions. We discuss the functions performed by such a component in achieving the desired overall end-to-end QoS, and the design considerations underlying our current implementation. We conclude with experiences to date with two variations of the gateway: one controlling managed latency and throughput using bandwidth allocation, and one controlling dependability through the coordination of object replicas.

1. Introduction

Middleware has emerged as the answer to making it easier to develop distributed applications. It does so by inserting higher levels of abstraction between the network and the applications to provide system software oriented toward managing common communication and resource management tasks from the perspective of the application developer. As the network and the use of communication within applications continues to grow and find more uses, so too does the demand for more control and manageability of those resources through its middleware interface. In fact, the recent popularity of Web interactions has focused attention on the diverse and dynamically varying computer and communications environments which may be available at any given time for network based applications, and managing the tradeoffs in application implementation that these variations demand.

The software underlying Quality Objects (*QuO*) [18, 9] is advanced, reusable middleware enabling a new generation of flexible distributed applications which have more explicit control over their resource management strategies, as well as being both easily reconfigurable and dynamically adapting to changes in network and computing environments. In one view, QuO is an extensible software development framework built on a distributed object middleware base, which makes it easy to support dynamic runtime adaptation to changing configurations, requirements or availability of resources. In a complementary view, QuO is an evolving architecture with a growing base of components and mechanisms filling out this architecture to support an integrated quality-of-service (QoS) concept for managing collections of "system properties" and the tradeoffs among these properties to support varying operating objectives. The "system properties" that are under investigation in current integration activities are managed communication bandwidth, dependability, real-time behavior, and security.

QuO was designed to help distributed application programmers to more effectively manage the "how well" of the client-object interaction, by providing a flexible environment for handling and integrating QoS attributes. A key element required for middleware to support QoS, but almost totally absent from today's commercial and research products, is to provide a relatively simple way to organize, collect, disseminate, integrate, and act upon an expanding set of dispersed QoS-related information. Convenient access to such information will enable better decision making and adaptation. In addition, there are a variety of enforcement mechanisms either currently available or under investigation, each typically with its own operating characteristics and narrow areas of applicability.

It is important that the QuO infrastructure be independent of any particular QoS property, and be easily extensible. Toward accomplishing this technical challenge, we had a number of related objectives. Two of these objectives, the ones most closely associated with the design and implemen-

Accepted for publication in Proceedings of ISORC '99, The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing, May 2-5, 1999, Palais du Grand Large 35 407 Saint-Malo, FRANCE.

tation of the QuO object gateway which is the focus of this paper, are highlighted here:

- Providing a Common Platform for Integrated QoS Dimensions

  Regularize the implementations for specification, monitoring, reserving, and adapting to changes, as applied to the various QoS dimensions. This avoids the single-point solutions which are usually tied closely only with a particular QoS dimension or enforcement algorithm, making it difficult or impossible to use more than one simultaneously. It also serves as a common interface to an integrated and combined QoS capability, which can take into account the synergies and conflicts among the various QoS dimensions.

- Facilitating Integration of New and Alternative Control Mechanisms and Policies

  It is important that the QuO infrastructure support additional information collection and additional mechanisms and policies to accommodate new innovation and extend the narrow range of behavior moderated by current mechanisms.

In this paper, we introduce an additional component, that of a QoS-oriented Object Gateway for integrating with a variety of QoS enforcement and implementation mechanisms controlling the underlying distributed interactions. A QuO object gateway is a QoS-aware element inserted at the transport layer between clients and objects/servers to provide managed communication behavior for the particular QoS property being supported. The two main problems addressed by the gateway design are

1. how to insert the QoS managed behavior into existing functional end-to-end distributed object pathways, and

2. how to enable the integration of a variety of specific enforcement mechanisms into a common framework, with minimal additional mechanism developer implementation effort, allowing both substitution of specific mechanisms and integration across properties.

In order to understand the role and operation of the QuO Object Gateway, it is necessary to establish some of the underlying basics of what QuO is, and why it is what it is. To do this, we first describe some background concepts from distributed object computing in general, and CORBA specifically, that are needed to understand QuO at a very high level. After describing these, we discuss the functions performed by an object gateway component in achieving the desired overall end to end QoS, and the design considerations underlying our current implementation. We then provide a more detailed view of two variations of the object gateway — one controlling managed latency and throughput between clients and objects using bandwidth allocation, and the other controlling degrees of dependability through the coordination of object replicas. We conclude with experiences to date and future evolution of the object gateway concept within QuO.

2. Brief Overview of Distributed Objects, CORBA, and QuO

2.1. Distributed Objects

Years of experience in the field of software engineering has proven that organizing the software development process around a coherent underlying model is essential for effectively developing systems of any complexity. While there is still considerable debate about which development paradigm is most effective for the large scale, network-based systems envisioned for QuO, the distributed object paradigm is the most advanced, mature, flexible context available today and was selected for this work. At the core of that model, software is broken up into collections of clients and objects dispersed throughout the network, and clients invoke operations on different types of objects to accomplish the interactions needed to bind the pieces together into a running system. Within this middleware context many of the problems of distributed computing such as remote location interoperability, heterogeneity, transparency, common services, synchronization, etc., are coherently delivered through the distributed object model.

Today, there are a number of distinct, sometimes disjoint, sometimes converging instantiations of these ideas, chief among them (in chronological order) CORBA, DCOM and Java RMI. They vary by scope, constituency, maturity, technical focus, and assumptions about the past, present and future. However, the underlying approaches to many of the fundamental technical problems are similar, and the QuO concepts can apply equally well to any of them. The implementation, however, must be bound to a particular context, and we have chosen CORBA as the initial implementation vehicle, with DCOM and hybrid CORBA/DCOM versions planned for next year. While we support Java as a programming language and in fact built some of QuO using Java, we do not at this time have plans for a Java-RMI based QuO, as the Java internals seem at this moment to be converging with CORBA. The version of QuO described in this paper is completely CORBA-based.

In any distributed object computing scheme, the client's *functional path* is the flow of information between a client's invocation to a remote object and back. While getting the information between the remote entities is the job of the middleware utilizing the underlying network capabilities, the information itself is application-specific and de-

termined solely by the functionality being provided (hence the term "functional path"). The functional path deals with the "what" of the client-object interaction from the perspective of the application (e.g., the object runs a complex simulation for the client). QuO also adds a *system path* (or *QoS path*), which involves issues regarding "how well" the functional interaction appears to work (for example, the resources committed to the interaction (and possibly subsequent interactions), proper behavior when ideal resources aren't available, the degree of security needed, the recovery strategy for detected faults, etc.). Much of the job of QuO is to facilitate the collection, organization, and dissemination of the information required to manage "how well" the functional interaction occurs, and enable the decision making and adaptation needed under changing conditions for these "how well" properties. We call these properties collectively *quality-of-service attributes* and we call such information *QoS Meta-Data*.

2.2. CORBA Components

See Figure 1 for an overview of the software components in a simple CORBA environment. In summary, a client first obtains a *reference* to the object it wishes to invoke by calling the bind method on the object's class. This creates a local *proxy* object and returns a pointer to it; the client uses this pointer just like it would a pointer to a local object (e.g., one it created with the **new** operator in C++). When the client makes an invocation to a remote object, it does so using the object reference the Object Request Broker (ORB) passed back from the bind call. This sends the invocation to the proxy, which is code generated in the client's programming language based on the IDL interface for the object and the IDL mapping for that programming language (e.g., C++, Ada95, Java). The proxy creates a request object of class CORBA::Request and passes this on to the ORB. The ORB sends this request to the computer on which the server resides. There the ORB makes an upcall (via something roughly equivalent to the client-side proxy) to the code which implements the method the client has called. When this returns, any return value or **out** or **inout** parameters are placed in a CORBA::reply object and sent back to the ORB for delivery to the client. There, the request is passed back to the client via the ORB. The client's proxy then uses the values in this reply to set any out and inout variables in the client's program, then returns the return value, and in doing so of course returns control to the client's code.

2.3. QuO Components

QuO augments this view in a number of ways to better support the "how" as well as the adaptive behavior mentioned above, as shown in Figure 2. The components added

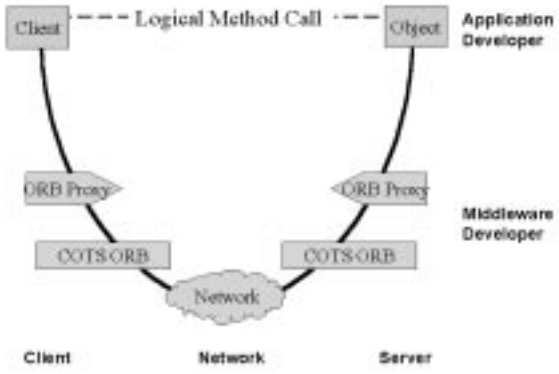

Figure 1. CORBA Architectural Components

with this configuration will be discussed in the remainder of this section.

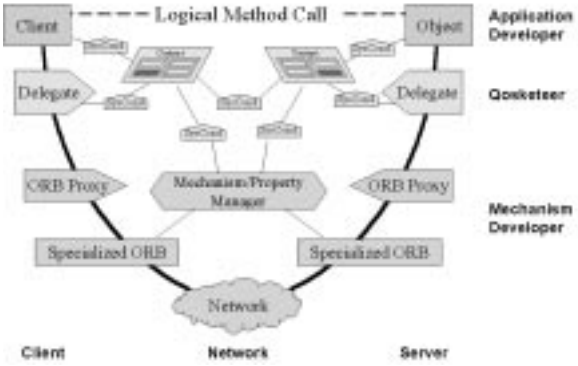

Figure 2. QuO Architectural Components

There are three complementary parts to QuO. The first part deals with the constructs needed to introduce the concepts for predictable, adaptable behavior into the application program development environment, including flexible specification of desired levels of QoS properties. The second part deals with providing runtime middleware to ensure appropriate behavior, including collecting information and coordinating any needed changes in behavior. The third part deals with the inserting the mechanisms for achieving and controlling each particular aspect of QoS which is to be managed, including aggregate allocation and control policies.

QuO's functional path is a superset of the CORBA functional path described above. QuO interposes a *delegate* component in the client's functional path, whose purpose is to do the middleware level QoS decision making. In the current version of QuO, instead of calling the ORB's bind call as with a current CORBA application, the QuO client calls QuO's connect call, which has a superset of the arguments for bind. Connect creates the delegate (which may itself call an ORB bind, saving the object reference) and returns to the

client a pointer to the delegate. The delegate implements the exact same API as the client's proxy, and is automatically generated by the QuO code generators based on the IDL and the QuO QDL (Quality Description Languages) which have been developed to capture the relevant QoS oriented information. At this time, QDL consists of two languages, CDL which describes QuO contracts, and SDL which describes selection information. Figure 3 represents the role of QDL in enhancing the standard IDL interfaces currently provided with the distributed object computing paradigm. Thus, to use QuO a client only has to modify its bind calls, not all its object invocations. Interposing the delegate in this relatively transparent manner allows the client's functional path to be instrumented and controlled in ways described below.

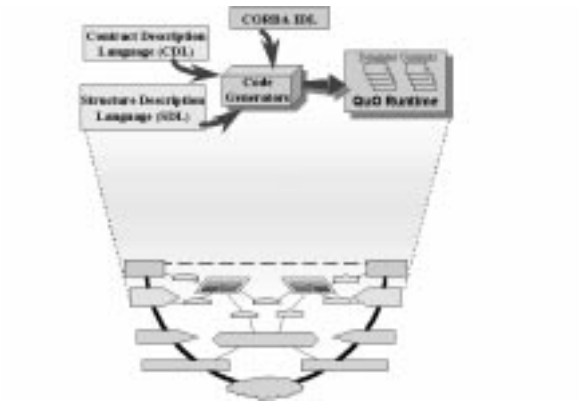

Figure 3. QuO's Quality Description Languages and Generated Code

A QuO *contract* provide a means to specify what the client requires or desires in terms of QoS as well as means for it to be informed of what level of QoS it is actually receiving. This also allows a client to cleanly specify, in an application friendly manner, what to do when what it is actually receiving diverges from its expectations. QuO's Contract Description language is used for writing contracts, and is described below. The contract can specify *callbacks* to the client to alert it to when conditions have changed sufficiently to warrant the client being notified (and possibly adapting on its own behalf). QuO *system conditions* are objects which project a value into a contract. As such, they are a way for the contract to integrate information from different sources, or looked at from another point of view, are the mechanism to connect complex information sources into the QuO context in a relatively simple manner. System Conditions also provide a way for a contract to control or influence the way a property such as bandwidth or replication is managed, by serving as a conduit for passing on the client's requested level of service to the appropriate property manager.

Mechanism Managers (aka *Property Managers*) are responsible for managing a given QoS property (such as the availability property via replication management or controlled throughput property via RSVP reservation management) for a set of QuO-enabled server objects on behalf of the QuO clients using those server objects.

3. An Object Gateway Component

3.1. The Need for an Object Gateway

In order to provide the type of controllable, predictable, and manageable environment we seek with QuO, we need mechanisms to control and enforce resource management and synchronization for the networked entities. An important factor behind the rising level of interest in QoS is the increased embedded use of still largely unmanaged network communication services. The inherent variability in using these services, due to changes in resource configuration, load, relative location and current availability (operational status), make the end to end results delivered to the application highly unpredictable. This has led to the development of a variety of mechanisms and approaches for better manageability as well as to encapsulate some of its complexity. However, these mechanisms are usually at a fairly low level in the protocol stack or closely tied to the transmission mechanisms where they can better control behavior. That makes them difficult to incorporate into the software engineering paradigms close to the application programmer's level of abstraction, for example distributed object computing and QuO. One possible approach would be to develop an enhanced specialized ORB with just the right properties. An alternative that we were seeking would work with a variety of off-the-shelf ORB products, both commercial and experimental.

Our solution, and the one adopted for QuO, is to combine control elements at the object interface level with control elements at the transport level in a translucent manner. That is, to link the desired behavior at the client/object interfaces with the appropriate behavior at the communication interfaces, in a manner which makes visible and controllable (versus transparent) the connection between the two. An important new component supporting the integrated property QuO architecture is the concept of a *QuO object gateway*. An object gateway is the QoS aware element inserted at the transport layer between clients and objects to provide the managed communication behavior for the particular QoS property being supported. In the QuO architecture, it is the job of the QuO object gateway to be superimposed at the transport level to apply the appropriate mechanisms needed to fulfill the obligations incurred at the QuO contract level. Figure 4 illustrates the QuO Object Gateway

concept. Since it is highly desirable that the QuO system

Figure 4. The Role of the QuO Object Gateway

be useable with a variety of ORB products, and the ORB is generally responsible for establishing communication with the designated object, our immediate goal becomes inserting the proper mechanism after ORB processing, but before handing the request off to the network transport subsystem. In CORBA, the open protocols between ORBs needed to support ORB interoperability (Interoperable Internet Operation Protocol, or IIOP), also allows the seamless insertion of a QuO gateway function to provide the appropriate transport level conditioning needed to meet the high level QoS contract.

3.2. Two Simplified Object Gateway Examples

Depending on the QoS property being managed, and depending on the choice of mechanism to provide or control the communications needed to support that property, different specialized protocols are inserted in the control path between clients and objects. For example, for a bandwidth reservation style of communication control, RSVP [17] can be the mechanism of choice to be used by the gateway to "condition" the internetwork connectivity to meet the QoS objectives between objects. Similarly, to deliver certain levels of dependability, a group communication mechanism, such as Ensemble [6] implementing a virtual synchrony form of "multicast" communication discipline, can be the mechanism of choice to be used by the gateway to "condition" the connectivity to meet replicated availability objectives. Other properties, such as real time constraints and security, will also have specific mechanisms which contribute to meeting their QoS objectives.

Before describing the organization and design of the gateway itself, it would be useful to develop the two examples cited above in a bit more detail, to illustrate the multipurpose nature of the mechanisms plugged into the gateway. The DIRM project [2] has used QuO to develop an initial high-level API that allows applications to control bandwidth management QoS for their network communications using resource reservation. This API hides much of the complexity of the underlying reservation control structures and present QoS abstractions in a form natural to application developers.

In this first example a CORBA client has indicated contract regions, specifying the expectations on bandwidth availability in terms of invocations between the client and object (for this example, think of the object as a data server, an image server, or even a web server). Through its contract, the application requests sufficient bandwidth to retrieve each image within a specified time and monitors the measured latency. The client adapts if it does not receive adequate QoS by switching to an alternate server that delivers smaller images or data payloads. In the absence of any transport level enforcement mechanism, the QuO application merely notes the expectation, requests that the functional invocation be handled and delivered normally by the ORB, and measurement components put in place to signal when expectations are not met. By adding an enforcement mechanism for reserving network bandwidth, we can ensure that at least some of the traffic stays within its preferred region, perhaps at the expense of forcing other traffic out of a better service region. The current capability to reserve network bandwidth is based on the reservation protocol, RSVP, which permits applications to request reserved network bandwidth in an Internet Protocol (IP) network.

Specifically in this case, if the client has appropriate capability, the object gateway halves establish an RSVP mediated path for conveying the object invocations between that client and the designated object established using the standard CORBA binding mechanisms. Once established by the gateway invocation of RSVP connection setup mechanisms, the QuO delegate assures the use of the appropriate channel for meeting the QoS objective for the managed network communication.

In our second example, the QoS property being managed is not communication bandwidth, but rather dependability through object replication strategies which can withstand certain types of failures. The AQuA project [1] has used QuO to develop an initial high-level API that allows applications to control object dependability using group communication and synchronization technology. This API hides much of the complexity of the underlying group synchronization and error recovery, and presents QoS abstractions in a form natural to application developers.

There are two parts to the mechanism solution we seek in order to support this enhanced QoS interface to clients. First, we need to keep the various instances of the target resource/object synchronized with respect to changes so that they each represent a potential source of servicing the next

request; and second, we need a way to transmit a new request to any available instance, so as to withstand the potential failure of other instances without negatively affecting the completion of the request. Group communication and virtual synchrony (e.g., Ensemble, or any of a number of other similar mechanisms) form the basis of the enforcement mechanism used to support a higher degree of dependability.

Within the group communication mechanism framework, provision is made for a dynamically changeable grouping of objects to remain synchronized. This is implemented by having appropriate update messages go to the entire group in the appropriate order, and by establishing a group identity which can be used to reach any of the available instances, thereby providing the abstraction of a more (or less) highly available single resource, depending on the degree of replication. It is the function of the QuO object gateway to allow the easy insertion of the group communication mechanisms needed to support this QoS enforcement within the communication path between client and collection of object instances representing the dependable object. In this second example, the specialized protocol is not an RSVP controlled communication path, but rather a carefully managed virtual multicast (either network enabled or through a series of point to point communications). In either case, however, the specialized protocol is inserted at the communication level (i.e., beyond the IIOP boundary) in synchrony with the QoS objectives at the object interface level.

Currently, we are establishing the gateway and mechanism concept as a separate QuO component so as to achieve the ORB independence mentioned above. In the future, it may prove advantageous to integrate this functionality more tightly with the ORB itself. In either case, the QoS enforcement mechanisms need to be changeable and selected from among a family of implementations, each with their own distinctive footprint.

3.3. Functions of the Object Gateway

The primary function of the QuO Gateway is to allow the easy and convenient insertion of QoS aware transport layer protocols between distributed clients and servers. By transport layer protocols we include not only traditional protocols (e.g., IP/TCP) for moving data but also specialized protocols to support specific QoS enhanced data transport mechanisms providing specific attributes in the areas of real time performance, dependability and security. These specialized protocols contribute various enhanced properties to the transport of message data, ranging from reserving bandwidth capacity to ensure high priority, real time message data traverses the network unimpeded by further delay from other competing traffic, to organized group (multicast) transport distribution to ensure synchronized parallel, redundant transformation updates, to controlling which messages can or can't get through and with or without proper encoding to support overall security objectives.

In its CORBA instantiation, the QuO Gateway piggybacks on the existing IIOP (Interoperable Internet Operations Protocol) invocation/response transport mechanisms, inserting additional protocols which manage QoS properties as noted above. In effect, the QuO Gateway serves to manage enhanced IIOP transport interactions between CORBA clients and objects. While transport level QoS is not equivalent to end-to-end QoS, it does nonetheless address a significant, and perhaps most critical, part of the end-to-end problem. Another step in QoS management is to link the transport level resource management provided by the QuO gateway with other parts of the QuO framework to produce the desired managed, adaptive and integrated behavior. We are still distant from an integrated capability, needing first to instantiate an effective common transport level QoS management capability which can individually support single property QoS.

The functions of the QuO gateway fall into two distinct categories: those that are standardly needed for the general QuO gateway support independent of the property being managed; and those that are specific to the various QoS property protocol mechanisms being inserted into the general gateway structure. In the rest of this subsection we enumerate the general gateway functions supported. Examples of specific property functions will be described later in the sections on case studies of the AQuA dependability gateway and the DIRM controlled throughput gateway.

The standard functions of the object gateway include the following:

- (Approximate) transparent insertion in the IIOP stream of QuO specific management functions (capable of calling appropriate property specific protocols) on both the client and server sides if needed. This involves inserting in the IIOP stream matching gateway halves providing a "shadow" QoS transport server on the client end, and a "shadow" QoS transport client on the server end. In this way, to the client, the QuO gateway looks like an endpoint CORBA server, while to the server, the other half of the QuO gateway looks like an initiating CORBA client. The job of the gateway is to transport the request/reply to its appropriate destinations utilizing the appropriate path selection and mix of QoS mechanisms.

- Request/Reply matching needed to coordinate the two asynchronous activities

- Error handling, including system exceptions acting on behalf of the ORB, as well as other CORBA based

redirection functionality such as “Cancel” and “Locate”.

- Standard Interfaces for inserting and substituting a wide range of specialized transport level QoS mechanisms and transport level QoS handlers in support of these mechanisms.

3.4. Architecture, Design, and Implementation of the Object Gateway

The gateway shell provides basic gateway capabilities, based on monitoring Internet Inter-ORB Protocol (IIOP) requests. The gateway inserts itself into the inter-ORB communications stream by terminating the IIOP session at the client side and initiating a new one at the remote site. In between, it applies appropriate QoS measures depending on the current contract region and available mechanisms and resources. This gateway shell is suitable for use as the base for constructing specialized QoS- property object gateways, initially in isolation, but soon in combination.

A specific property object gateway is built by layering the gateway shell on top of underlying QoS implementation mechanisms. For example, we layer the QuO object gateway over QoSME to produce a RSVP gateway. The gateway translates from IIOP to specific requests on the underlying RSVP mechanisms. Similarly, we layer the gateway over Ensemble group communication to provide synchronized message traffic among collections of replicated objects. The gateway translates from the single object invocation to the underlying group communication transport paradigm. Figure 5 illustrates the general architecture of the QuO object gateway shell, including the interaction of its two “gateway halves”.

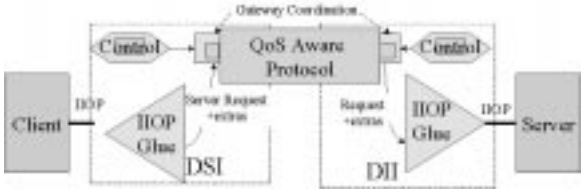

Figure 5. Object Gateway Shell Functional Overview

To the “Client” ORB, the QuO Gateway looks like the object. To the “Server” ORB, the QuO Gateway looks like a client. The ends of the gateway are at minimum on the same LAN as the Client/Object, and may be on the same host. In certain configurations, there may be many gateway instances on a LAN, and even many gateways on a host. CORBA Objects are used to Control QuO Gateway halves by allowing the gateway to be configured remotely, but do not interfere with in-band communication.

DII and DSI are standard CORBA interfaces provided to support dynamic invocation semantics, where the nature of the parameters of the invocation/reply are not known until runtime. We use them as the means of inserting our QuO gateway proxies as object request relays, utilizing the CORBA standard GIOP messages underlying the IIOP implementation. IIOP Glue multiplexes and demultiplexes GIOP messages to the gateway coordinator. The Gateway Coordination module handles flow control and error detection, as well as path selection.

Figures 6 and 7 show the symmetry in the gateway shell halves representing the client side gateway and the server side gateway. Transport interactions between these halves is governed by the appropriate mechanism for the specific QoS property being managed by the gateway within the QuO infrastructure.

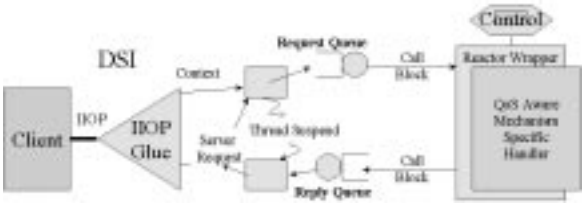

Figure 6. Client-Side Gateway Shell Half in Detail

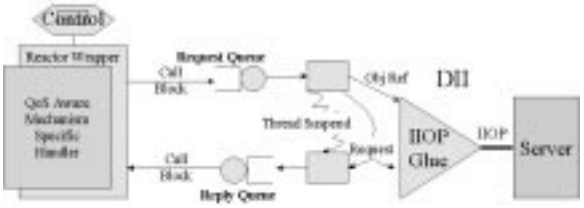

Figure 7. Server-Side Gateway Shell Half in Detail

The Client-Side Interface produces a partially parsed GIOP request to be transported which is bundled into a Call Block and puts it on the Request_Queue for the specialized transport. The Call Block is used for sequencing the transport activities and for matching with an eventual reply. In our current design, there is a separate thread per IIOP transaction, with the suspended processing thread maintained in the Call Block.

The Server-Side Interface essentially does the inverse of the Client-Side, starting from the transported GIOP request. The DII interface sends a Request structure to the object

reference. The return values are bundled into the Call block and put on the Reply_Queue for the specialized transport. Reply messages are forwarded with very little change and are largely opaque to the gateway.

If the transported call encounters an error, an Exception is thrown in the Reply Code. These exceptions are classified as System_Exceptions, because the Gateway is logically an extension of the ORB. Special threading restrictions are needed to maintain the order of arrival of requests, which are important to some QoS mechanisms.

4. Case Studies

4.1. Case Study 1: Assured Bandwidth

The DIRM project is developing techniques for quality assured bandwidth management for the QuO framework. To do this, it needed several software components in addition to those provided already by QuO. The most important of these are the following (also represented in Figure 8):

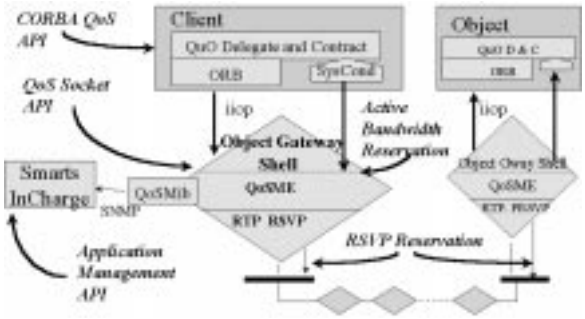

Figure 8. Managed Bandwidth Property Configuration

RSVP Control Module The RSVP control module provides a CORBA interface to control bandwidth reservations. It uses network management interfaces to request RSVP sessions on behalf of DIRM applications. The DIRM project completed and tested an initial RSVP capability in September 1997, shortly before the protocol's official approval by the Internet Engineering Task Force (IETF).

RSVP CORBA Object Gateway This is the generic QuO gateway shell specifically augmented with a mechanism to provide standard CORBA communications protocols with enhancements to utilize reserved bandwidth transport, based on RSVP. Columbia University QoSME, interfaced to RSVP controlled paths, was used as the transport layer between gateway halves. QoSME was extended to include setting up RSVP reservations from the socket level and to measure the QoS for each message sent [4]].

RSVP Monitor The RSVP monitor (not shown in figure) keeps track of the status of RSVP sessions in real-time. This status information is exported to the QuO kernel via QuO system conditions. These system conditions are then available to QuO applications and contracts.

Collectively, these software components combined with QuO concepts to achieve the construction of an "RSVP-aware ORB." This ORB, which consists of a commercial off-the-shelf (COTS) ORB, composed with a QuO gateway and appropriate control and feedback mechanisms, provides a DOC environment enhanced with managed bandwidth capabilities.

The QuO Gateway for bandwidth management used RSVP to reserve bandwidth between the client-side and server-side gateways (see Figures 9 and 10). Several simul-

Figure 9. Using RSVP to Reserve Network Bandwidth

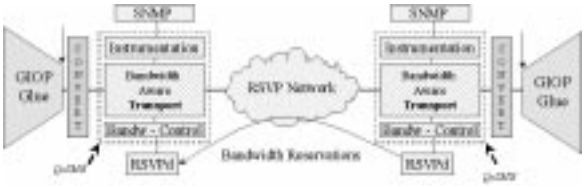

Figure 10. Bandwidth Management Aware Protocols in DIRM

taneous paths through the network are possible, each with a different QoS. If a reserved bandwidth path was selected, then the request and reply were sent over a TCP session which had an RSVP reservation. Thus for long request or reply messages, the delay for the message could be dramatically reduced because the effective bandwidth for that message would be greater going through the gateway, than competing with best-effort traffic for limited bandwidth without using the gateway.

The first implementation of the QuO RSVP Gateway (July 98) used a static configuration for its bandwidth reservations. When the client-side and server-side gateways had both started up, they would reserve a fixed amount of bandwidth for each "well-known" port between them. The

RSVP API would maintain the RSVP session as long as both gateways were alive. An RSVP flow-spec indicated the amount of bandwidth, the burst size for a specific TCP session (IP endpoints, protocol and port). CORBA calls from several clients to several different servers could be multiplexed over the same reserved TCP session. There was no attempt in this incarnation of the gateway to match the resulting flow with the RSVP reservation. The natural TCP flow control mechanism was used to regulate the session throughput. Thus in this case, RSVP was used as a way to give certain TCP sessions priority over other sessions, but with an upper limit on bandwidth. As expected, when the network was lightly loaded the RSVP reservation had little or no effect on the reserved or best effort traffic, i.e., the weighted fair queuing worked adequately as is.

Path selection was also static in the first QuO RSVP gateway implementation. We used a form of source routing by storing the path to the destination server in the CORBA object-key parameter. The request message traveled from the client through client-side and server-side gateways to the server. When the request arrived at a gateway, the object-key field contained the location independent Interoperable Object Reference (IOR) for the next embedded component. A CORBA IOR contains the server IP address and port, and the object key for the remote object. Thus, the IOR used by the client has the address and port for the client-side gateway with the object-key having the stringified version of the IOR to reference the server-side gateway. The IOR used by the client was created by a program that first encapsulates the server IOR in the server-side gateway IOR, second encapsulates the server-side gateway IOR inside the client-side gateway IOR, and finally gives to the client the client side gateway IOR to use for accessing the remote object though the gateways. The disadvantage of this scheme is the size of the IOR grows exponentially with each embedded component, (because we used the stingified version of the IOR instead of the RAW CDR encoding). However, the advantage was that the gateways did not have to implement any path selection algorithm.

4.2. Case Study 2: Dependability

4.2.1. Context

The AQuA project is providing dependability mechanisms and policies for the QuO framework [1]. One of the major goals for the AQuA project is to provide support for a wide spectrum of dependable systems. The interposition of a gateway supports the development of many different operating points trading off performance, dependability, security, and survivability, in a highly portably manner. Specifically, the AQuA gateway architecture supports choices across the following interdependent dimensions:

- Active replication (all replicas process invocations and send replies) vs. semi-active (all replicas process but only one sends a reply) vs. passive (only one replica at a time processes invocations).
- If multiple replicas generate the same request or reply, when, where, and how (e.g., select first copy or majority voting) a single copy is selected for delivery to each destination.
- Whether the replicas of an object have a *leader* or are implemented symmetrically.
- How the messages implementing invocations or replies are ordered and which object replicas receive them.

In the architecture depicted in Figure 11, an operating point selected from the above dimensions is a *replication scheme* and is implemented in one of several "handlers" in the QuO gateway. The "IIOP glue," on receiving a CORBA IIOP message from the application object, hands this message off to the "dispatcher" to be forwarded to the destination object via group communication. The dispatcher in turn selects the correct handler which will actually send this message, depending on the replication scheme selected. The object receiving the message has a similarly configured QuO gateway whose handler and dispatcher will hand the message back to the IIOP glue to be forwarded via CORBA to the application.

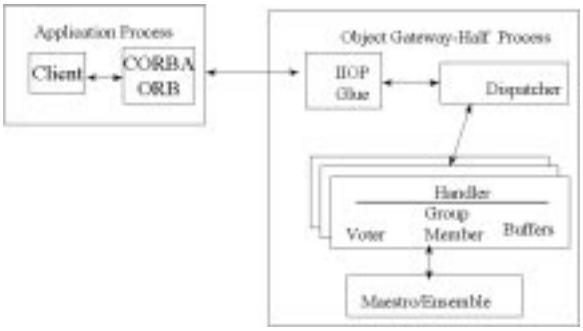

Figure 11. Dependability Gateway Client-Side Components

Because the details of the replication mechanisms are implemented in the handlers transparently to the ORB, it is easy to use and reuse different replication schemes across different application objects. Currently, a client's QuO contract specifies high-level dependability requirements such as number of host failures that must be survived. This is translated into a configuration involving the above dimensions, as well as the number and location of the replicas, by the Proteus dependability manager [12].

4.2.2. Group Communication Basics

The AQuA gateway, as used to help manage dependability, is layered on top of a group communication service offering virtual synchrony [15]. Such services provide delivery guarantees that enable a process to determine what messages have been delivered to other processes in an asynchronous environment in the presence of failures. Optionally, they can guarantee that the messages are delivered in a certain order. Using these properties, the handlers in an active replication scheme (for example) can guarantee that if all objects in the system process their invocations in a deterministic fashion, then all replicas of any given object will receive the same invocation stream, so that their states will be kept consistent [14]. In a group communication service, each message is sent by only one process at a time in only one direction to one or several other processes (one-to-one or one-to-many message-passing). The AQuA handlers build on top of this flexible support for two-way (request and reply) object invocations between a replicated client group to a replicated server group.

The AQuA gateway currently uses the Maestro/Ensemble group communication service [6, 16]. A service offering similar semantics could be substituted by using appropriate handlers.

4.2.3. An Example Dependability GW Configuration

An active replication scheme has been implemented, illustrating the use of the QuO dependability gateway. This scheme involves three Ensemble process groups whose members are the QuO gateways. The *client* (resp. *server*) process group is used for coordination of the client (resp. server) replicas, and only the client (resp. server) gateways are members. The *connection* process group is used to communicate between the client and the server, and all gateways on both sides are members.

Figure 12 shows the steps used to send a client's invocation request to the server. "C-Rep1" (replica number 1 of the client group) and "S-Rep2" (replica number 2 of the server group) are leaders of their respective groups. The steps taken by the handlers in this scheme (described in detail in [12]) involve point-to-point messages (step 3), multicast messages (steps 5 and 6), and pass-first-copy voting (step 4). The reply path in this example is symmetric to the request path.

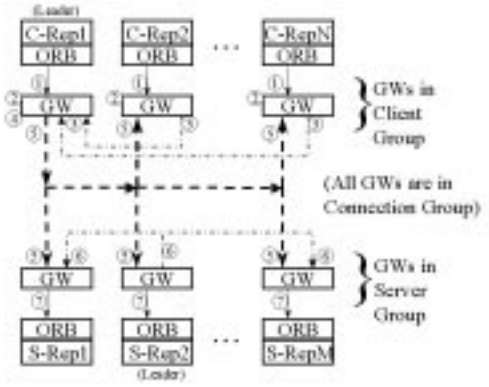

Figure 12. Steps for a CORBA request in Example AQuA Configuration

5. Conclusions

5.1. Experience with the Object Gateways

We have demonstrated the concept that a generic object gateway supporting different QoS properties can be built on a common platform, and in such a way that the strategies for managing the transport level QoS mechanisms can be substitutable. Experience to date with a bandwidth management module and a dependability module have shown significant reuse for the common object gateway concept, and a reimplementation now underway will provide even more. We have not yet tried experiments with a combined QoS property object gateway.

The DIRM Bandwidth Management Gateway has proven effective in sustaining predictable network QoS despite large fluctuations in competing traffic, using emerging RSVP protocols. These protocols are soon to be more widely available, so demonstrating their applicability in an application oriented software engineering context is an important step. We are continuing to work on developing the system level control and utilization modules needed to make a completely useful bandwidth managed capability.

The AQuA Dependability Gateway has proven very flexible. The same code for a given handler runs on both the client and the server side, and it is very modular. The code for a given step above is typically 5–10 lines, and the handler's code is structured with nested blocks of code for each step, bracketed by tests as to whether the replica is leader or not, whether it is on the client or server side of an invocation, and what step in the above protocol is currently underway. Given a few such handlers as examples, we expect it to be relatively straightforward for a programmer to create customized handlers from them.

Several problems where encountered in the first implementation of the common object gateway which are being addressed by the second version. First, the gateways introduced a substantial latency in the CORBA call. Some of the latency was expected due to processing the request and reply in the two additional gateway hops. But a large component of the delay was due to the store and forward nature of the gateway and limitations of CORBA IIOP protocol. For

IIOP 1.0, the whole message must be received at each embedded component before it can be forwarded to the next. This forwarding delay is not an issue when the messages are small, such as at the IP layer, but CORBA messages can be arbitrarily long, such as several megabytes. IIOP 1.1 (and beyond) allows for a "cut through" by introducing a GIOP fragment message. This new IIOP functionality permits taking a long CORBA message and breaking it up into several fragments which can be pipelined through the gateways. Fragmentation can have a big impact on the latency of long request and reply messages. Without fragmentation, long messages have to be sent as one IIOP frame for which the length must be know before the message can be started. With fragmentation, partial messages can be transported before they are completely received, and messages from different clients can be interleaved on shared links. IIOP fragments are currently becoming experimentally available as part of IIOP 1.2 implementations.

A second problem was getting the alignment right for the parameter list, if any of the gateway changes the length of any IIOP fields. A CORBA call's parameter list and result are encoded in the IIOP request-body and reply-body fields. These fields are opaque to the gateway, because their structure is defined in the IDL for the object interface, which the gateway does not know. The message body is aligned based on the type of the first argument in the parameter list. Since, an interface can have an arbitrary first argument, the alignment of the IIOP 1.0 body can have arbitrary alignment. The consequence is that if any of the IIOP message fields are changed, they must maintain the 128 bit modulus of the message body. Because the gateway changed the length of the object key field at each hop, we had to be careful to pad the generated IORs to keep the same 128 bit modulus as the original remote-object IOR. A standardized CORBA solution to this type of alignment problem is part of IIOP version 1.3.

5.2. Related Work

The Eternal and Realize projects at U.C. Santa Barbara are developing fault-tolerant and real-time CORBA-compliant systems [10, 11], and are also involved in the fault-tolerance standardization efforts of CORBA's consortium, the Object Management Group (OMG). Eternal and Realize both use an Interceptor modeule which intercepts an IIOP message and then redirects it to the subsystem to provide replication. As such, their Interceptor is totally "below" the ORB in its naming and implementation, in that it supports complete client transparency and normally intercepts a message before it leaves the client's address space (though it can do so on the server side if needed). This provides better performance than AQuA's use of the QuO gateway, but at a cost of less flexibility and portability to and interoperability with other operating systems.

The CORBA group communication service approach described in [3] uses an explicit CORBA service to provide fault tolerance; it is entirely "above" the ORB in both its naming and implementation. As such, the concept of a group is visible to the client program. The AQuA use of the QuO gateway is "above" the ORB in terms of naming, because it uses the ORB to deliver the IIOP message to the gateway by "pointing" the ORB proxy to the gateway. AQuA's gateway is "below" the ORB in terms of implementation, in the sense that its handlers are operating on the level of abstraction of a marshalled IIOP message, not an object and a request to one of its methods.

Electra and Orbix+Isis modify the ORB to make it aware of multicast primitives and explicitly maintain an object group membership [8].

The TAO project has developed a high-performance, open source, real-time ORB [5, 13]. It does not have a flexible and reusable component such as QuO's gateway handlers as part of it. However, it does have all the system hooks — portable interceptors and pluggable transports — required for interting such handlers, and in fact TAO has more mature implementations of these than any CORBA vendor. Ongoing collaboration is exploring the use of these interception mechanisms with QuO's gateway handlers (and also the replication strategies provided by Eternal and Realize). TAO researchers are very involved with the real-time efforts of the OMG.

The Time-Triggered, Message-Triggered Objects project [7] augments traditional CORBA *service methods* (those declared in CORBA IDL — *message-triggered* ones) with *time-triggered methods*, which are activated when a system's real-time clock reaches specific points in time. It uses Orbix's filters mechanism, which is similar to what is being standardized in the ongoing CORBA portable interceptors effort.

5.3. Future Directions

In the future we plan to change the static path selection with a more dynamic version based on the name resolution mechanisms found in CORBA. Instead of the object-key containing the path to the remote object, instead it will contain an object name and QoS properties. At each gateway object key will be given to a Path Selection Module that will determine the best path forward. Path selection could involve sending the call over a reserved bandwidth TCP session or queuing it for later transmission. Each call would resolve the name, so that calls to the same remote object and QoS properties can actually take different paths across the network.

We also plan to setup the bandwidth reservations dynamically, based on configuration commands from the re-

source management system. A management station will monitor the gateways, and can add or remove reservations to match the overall demand and external constraints. Thus, the topology between gateways will be configurable, much the same way as a Virtual Private Network (VPN) is configurable. Also, the path selection will pick the best path at the moment, much the same way as routers route IP messages in the Internet.

The dependability mechanisms which are part of the current configuration already have many degrees of flexibility as noted previously. A next step is to more rigorously evaluate and validate how these degrees of flexibility can combine with each other to achieve and move between different effective operating points. As they exist now, the mechanisms are largely heavy duty, and need to be selectively used. Expanding the set with more light duty alternatives will make it easier to introduce dependability into many more configurations.

Another next step is to experiment with the combined effect of a dependability gateway over bandwidth managed network communication to prove their compatibility, and to improve performance of the complex dependability protocols over wide area configurations.

Acknowledgements

This work is sponsored by the U.S. Defense Advanced Research Projects Agency under Contracts No. N66001-96-C08529 monitored by NRaD, and No. F30602-96-C-0315 monitored by US Air Force Research Laboratory as part of the Quorum program managed by Gary Koob. We would also like to acknowledge the contribution to this work by participating colleagues in the Distributed Computing Groups at BBN (Mark Berman, Partha Pal, Rodrigo Vanegas, Frank Bronzo), at the University of Illinois Champagne/Urbana (Bill Sanders, Michel Cukier, Chetan Sabnis,and Jennifer Ren), at Columbia University (Yechiam Yemini and Danillo Florissi) and at Washington University in St. Louis (Doug Schmidt, Irfan Pyarali and Carlos O'Ryan).

References

[1] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. Berman, D. A. Karr, and R. E. Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, Oct. 1998.

[2] DIRM project technical overview, 1998. `<http://www.dist-systems.bbn.com/projects/DIRM/>`.

[3] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS-15)*, pages 150–159, Niagara-on-the-Lake, Canada, Oct. 1996.

[4] P. Florissi. QoSME: Quality of service management environment, 1998. `<http://www.cs.columbia.edu/dcc/qosockets/>`.

[5] A. Gokhale, I. Pyarali, C. O'Ryan, D. Schmidt, V. Kachroo, A. Arulanthu, and N. Wang. Design considerations and performance optimizations for real-time ORBs. In *Proc. of the Fifth USENIX Conference on OO Technologies and Systems (COOTS '99)*, May 1999. to appear.

[6] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Ithaca, NY, Jan. 1998. Technical Report TR98-1662.

[7] K. H. Kim. Object structures for real-time systems and simulators. *IEEE Computer*, pages 62–70, Aug. 1997.

[8] S. Landis and S. Maffeis. Building reliable distributed systems with corba. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[9] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proc. of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Apr. 1998.

[10] P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki, and P. Narasimhan. The realize middleware for replication and resource management. In *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 123–138, Sept. 1998.

[11] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[12] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr. Proteus: A flexible infrastructure to implement fault tolerance in AQuA. In *Proc. of Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, Jan. 1999.

[13] D. Schmidt, D. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), Apr. 1998.

[14] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[15] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Commun. ACM*, Apr. 1996. See also other articles in this special issue on group communications.

[16] A. Vaysburd. *Building Reliable Interoperable Distributed Objects with the Maestro Tools*. PhD thesis, Cornell University, May 1998. Technical Report TR98-1678.

[17] L. Zhang et al. RSVP: a new resource protocol. *IEEE Network*, 7(6):8–18, Sept. 1993.

[18] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, Apr. 1997.