# Integration of QoS-enabled Distributed Object Computing Middleware for Developing Next-generation Distributed Applications

Yamuna Krishnamurthy, Vishal Kachroo
{yamuna,vishal}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO, USA

David A. Karr, Craig Rodrigues,
Joseph P. Loyall and Richard Schantz
{dkarr,crodrigu,jloyall,schantz}@bbn.com
BBN Corporation
Cambridge, MA, USA

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering Department
University of California
Irvine, CA, USA

## Abstract

*This paper describes the integration of QoS-enabled distributed object computing (DOC) middleware for developing next-generation distributed applications. QoS-enabled DOC middleware, facilitates ease of development and deployment of applications that can leverage the underlying networking technology or end-system QoS architecture. This paper also describes the development of a demonstration application utilizing QoS-enabled middleware to control the dissemination of Unmanned Air Vehicle (UAV) data throughout a ship.*

**Keywords**: Distributed object computing, QoS-enabled Middleware, QuO middleware, CORBA-based Multimedia Streaming

## 1 Introduction

DOC middleware has evolved tremendously over the past few years. Initially, it facilitated the seamless inter-operation of various applications over a variety of heterogeneous environments. Its growing acceptance has opened its applicability to a broader variety of applications, whose demands are far beyond inter-operability. This is especially true for next-generation applications such as e-commerce, autonomous process control, and global event notification systems.

These advanced applications require a wide range of quality of service (QoS) support, where resources are managed both prior to and during run-time. For example in mission critical telecommunication systems and distributed electronic medical imaging systems, a failure to meet certain deadlines can result in significant loss of property and even life. Therefore, these systems must be analyzed and monitored both off-line and on-line to ensure that resources are properly allocated and managed. Also, these applications must be able to (1) autonomously reflect upon situational factors as they arise in the run-time environment and (2) adapt to these factors while preserving the integrity of key mission-critical activities.

These requirements have driven R&D efforts in DOC middleware to develop QoS-enabled DOC middleware which simplifies the development of advanced applications that can leverage the advances in networks and end-systems end-to-end. Two major research efforts in this area have been:

**Quality Objects (QuO):** This is a DOC middleware extension, which supports adaptive QoS specification, measurement, and control [1].

**ACE QoS API (AQoSA):** This is a unified QoS API in the Adaptive Communication Environment (ACE) [2] that abstracts various network QoS protocols like RSVP and DiffServ.

**Paper organization:** The remainder of this paper is organized as follows: Section 2 gives an overview of the QuO middleware framework; Section 3 presents a case study of AQoSA, a unified QoS API and describes how this API has been used to provide qos to multimedia services that span the network and the ORB end-system layers;Section 4 describes the integration of the QoS-enabled CORBA Audio Video Streaming Service and QuO into the Unmanned Air Vehicle video dissemination application; and Section 5 summarizes concluding remarks.

## 2 Overview of the Adaptive QuO Middleware

Figure 1 illustrates a client-to-object logical method call. In a traditional CORBA application, a client makes a logical method call to a remote object. A local ORB proxy (i.e., a stub) marshals the argument data, which the local ORB then transmits across the network. The ORB on the server side receives the message call, and a remote proxy (i.e., a skeleton) then unmarshals the data and delivers it to the remote servant. Upon method return, the process is reversed.

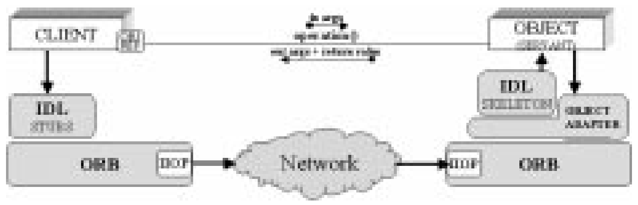

Figure 1: CORBA DOC model

Quality Objects (QuO) is a distributed object computing (DOC) framework designed to develop distributed applications that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time. By providing these features, QuO opens up distributed object implementations [3] to control an application's functional aspects and implementation strategies that are encapsulated within its functional interfaces.

A method call in the QuO framework is a superset of a traditional DOC call, and includes the following components, illustrated in Figure 2:

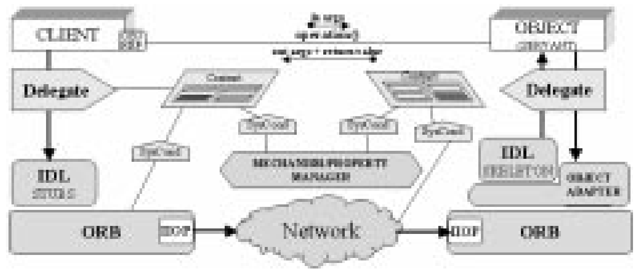

Figure 2: QuO CORBA model

- Contracts specify the level of service desired by a client, the level of service an object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
- Delegates act as local proxies for remote objects. Each delegate provides an interface similar to that of the remote object stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.
- System condition objects provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO contracts.

In addition, QuO applications may use property managers and specialized ORBs. Property managers are responsible for managing a given QoS property (such as the availability property via replication management [4] or controlled throughput via RSVP reservation management [5]) for a set of QuO-enabled server objects on behalf of the QuO clients using those server objects. In some cases, the managed property requires mechanisms at lower levels in the protocol stack. To support this, QuO includes a gateway mechanism [6], which enables special purpose transport protocols and adaptation below the ORB.

In addition to traditional application developers (who develop the client and object implementations) and mechanism developers (who develop the ORBs, property managers, and other distributed resource control infrastructure), QuO applications involve another group of developers, namely QoS developers. QoS developers are responsible for defining QuO contracts, system condition objects, callback mechanisms, and object delegate behavior. To support the added role of QoS developer, we are developing a QuO toolkit, described in earlier papers such as [7, 8], and consisting of the following components:

- Quality Description Languages (QDL) for describing the QoS aspects of QuO applications, such as QoS contracts (specified by the Contract Description Language, CDL) and the adaptive behavior of objects and delegates (specified by the Structure Description Language, SDL). CDL and SDL are described in [9, 8].
- The QuO runtime kernel, which coordinates evaluation of contracts and monitoring of system condition objects. The QuO kernel and its runtime architecture are described in detail in [10].
- Code generators that weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program. Runtime integration of QDL specifications is discussed in [8].

## 3 Unified Middleware-centric QoS API

Application developers need standardized interfaces to allow QoS specification and to receive guarantees from the underlying network and QoS infrastructure. As the different QoS protocols become more and more mature, the need for a QoS API that applications can use as a unified interface to the underlying QoS protocols has grown tremendously. Firstly, the applications would like to shield themselves from the protocol specific details. Secondly, although the QoS protocols provide a mechanism for allocating resources between two end systems, they are not sufficient to address the translation required from the application level QoS parameters to the network level QoS parameters. Thirdly, the adaptive applications require a uniform mechanism to get notified of changes to the available resources so they can re-negotiate the QoS. All these considerations motivate the design of a unified QoS API that addresses these issues in a platform/protocol independent way.

Once the unified API is developed, it can be exposed to the applications through middleware like CORBA. This allows the applications to get QoS guarantees through standard middleware APIs and at the same time leverage all the usual features of a middleware. The API can also be integrated with higher level middleware services. At this level, the API would have added functionality like binding QoS to specific application data flows or translating standard QoS flow requirements to network QoS.

With this motivation we have designed and implemented a unified QoS API in the Adaptive Communication Environment (ACE) [2] to abstract the two separate implementations of IntServ available. They are, the GQoS implementation on Windows 2000 from Microsoft and the RSVP API (RAPI) implementation on UNIX from Sun. The interface provided by these to the application developers is totally disparate from each other.

Our effort, in this work, has been to abstract out the common functionality from these implementations from an application developer's perspective and to design APIs driven by these abstractions. The unified ACE QoS API (AQoSA) enables the users to QoS enable their applications without bothering about the underlying platform or QoS protocol implementation. These APIs are further exposed to the ORB, thereby, empowering the ORB with network level QoS guarantees. The APIs will also be used by the ORB services like the Audio/Video service to provide QoS to applications that make use of such services, such as the tele-immersion application,

## 3.1 Overview of the ACE QoS API (AQoSA)

AQoSA was designed by inductively identifying common patterns [11, 12] used to program to existing QoS APIs. Below, we describe the features provided by the AQoSA implementation addresses key design requirements.

**Portability:** AQoSA encapsulates applications from the details of platform-dependent GQoS and RAPI IntServ implementations in the underlying endsystem platform. AQoSA encapsulates the functions, data structures, and macros used to represent various QoS parameters in these two IntServ implementations. Thus, applications and higher-level DOC middleware can access IntServ capabilities via a convenient and portable QoS programming interface.

**QoS Event Notification for Adaptivity:** AQoSA provides applications with a platform-independent API for receiving notifications when the underlying network QoS changes. ACE applications often use an event handling model based on the Reactor pattern [11], which allows servers to decouple event demultiplexing/dispatching from their application-specific event handling. In RSVP, the notifications are carried in *RSVP events*.

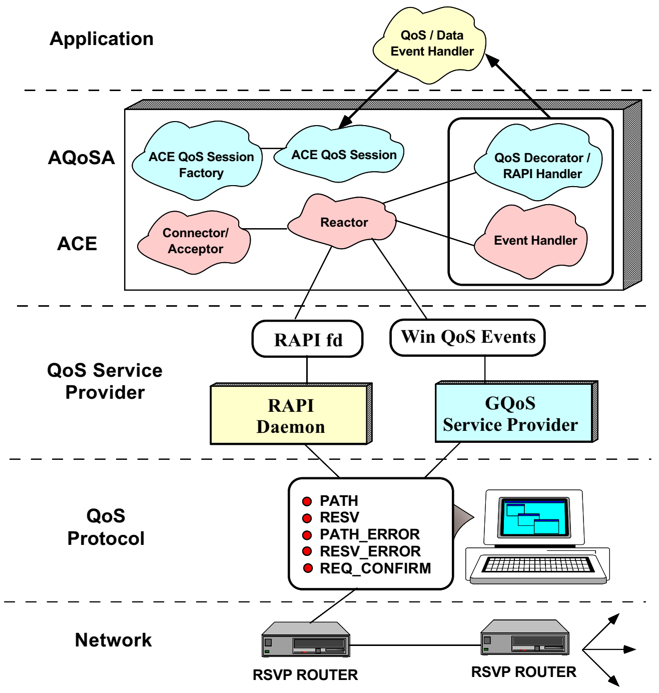

Figure 3: ACE QoS Event Handling

As shown in Figure 3, AQoSA receives and handles RSVP events uniformly for different network-level QoS implementations via the Reactor pattern [11]. In this pattern, a *synchronous event demultiplexer*, such as `select` or `WaitForMultipleObjects`, handles event demultiplexing and an associated reactor notifies previously registered application-specific event handlers so they can adapt to QoS state change events.

**Extensibility:** AQoSA enables new network- and endsystem-level QoS mechanisms to be integrated without tedious refactoring of its public APIs, *e.g.*, it is straightforward to extend AQoSA to support other QoS models, such as DiffServ. To accomplish this, AQoSA extends the existing ACE framework components by introducing new capabilities that allow applications and underlying DOC middleware to manage QoS multicast or unicast sessions. Moreover, AQoSA applies several patterns to ensure that new network-level QoS implementations can be easily integrated without changing applications that use its API.

**Advanced QoS capabilities:** AQoSA binds multicast or unicast flows to reservations via a uniform and portable component called a *QoS session*. A QoS session represents the application's notion of the underlying network-level QoS. Though modeled originally using IntServ RSVP sessions, a AQoSA QoS session can also accommodate other QoS mechanisms, such as DiffServ.

An AQoSA QoS session explicitly separates QoS properties of its sessions from lower-level socket data transfer aspects. Internally, the ACE QoS socket maintains an association between QoS sessions to which an application has subscribed. This separation of concerns also facilitates more advanced QoS functionality, such as QoS event notification.

Figure 4 depicts the UML class diagram for the components in AQoSA. These components allow applications to specify

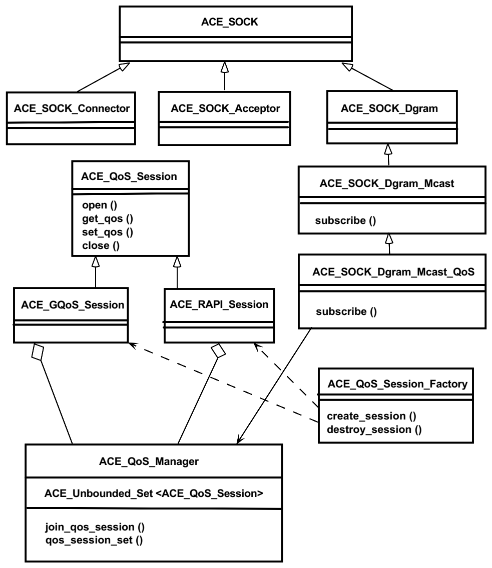

Figure 4: AQoSA QoS Class Design

and query for the QoS configured currently for a particularly unicast or a multicast session. The UML diagram also shows how AQoSA uses the Bridge pattern [12], which provides a uniform interface for different mechanisms implementations, such as RAPI and GQoS. New QoS mechanisms can be added by subclassing implementations of this interface. In addition, the diagram shows how AQoSA applies the Factory Method pattern [12] to create and manage the lifetime of QoS session objects subscribed to by applications.

## 3.2 Applying AQoSA to the CORBA Audio/Video Streaming Service

As shown in Section 3.1, AQoSA shields applications from the non-portable aspects of the underlying operating system and network-level QoS implementations. The infrastructure middleware abstractions provided by AQoSA are sufficient for certain types of applications, such as controlling and managing network switch and router elements [13]. Other types of applications, however, can benefit from higher-level middleware programming models that support a broader range of protocols and common middleware services.

For example, the TAO open-source real-time CORBA ORB [14] provides an implementation of the CORBA A/V Streaming Service [15], which supports multimedia applications, such as video-on-demand and tele-immersion. The QoS requirements of these types of applications depend on the following factors:

• **Application class,** such as interactive vs. non-interactive. Interactive applications require real-time response and hence predictable delivery of application data with bounded end-to-end latencies. In contrast, non-interactive applications have less stringent response requirements, but often possess higher throughput demands.

• **Application media types,** such as audio and video. Depending on the media type, different performance criteria may apply. For example, audio delivery is sensitive to delay, loss, and bandwidth, and hence needs guaranteed QoS. In contrast, video can often be best-effort since it is less sensitive to delay, loss, and bandwidth. Therefore, it can be adapted more readily to the available network QoS.

• **Application adaptation policies,** which may require implicit or explicit adaptations to changes in delivered QoS. Implicit adaptation is transparent to the application layer, *e.g.*, dropping selected portions of a video stream at the transport layer. Conversely, explicit adaptation, such as changing quantization coefficients or application coding algorithms, is not transparent to applications.

To provide acceptable QoS to multimedia applications developed using TAO, we therefore developed a QOS-enabled implementation of the CORBA A/V Streaming Service using AQoSA, as described in this section.

### 3.2.1 Overview of the CORBA A/V Streaming Service

The CORBA A/V Streaming Service controls and manages the creation of streams between two or more media devices. Although the original intent of this service was to transmit audio and video streams, it can be used to send any type of data. Applications control and manage A/V streams using the A/V Streaming Service components shown in Figure 5.

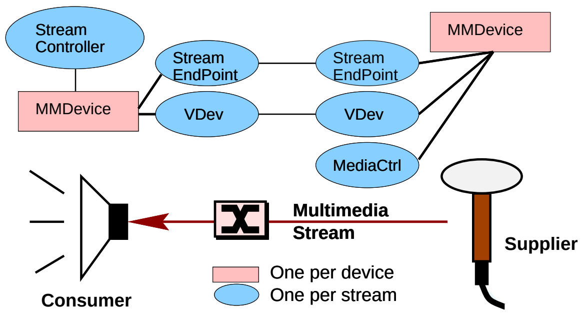

Figure 5: CORBA A/V Streaming Service Components

Streams are terminated by endpoints that can be distributed across networks and are controlled by a stream control interface, which manages the behavior of each stream.

The CORBA A/V Streaming Service combines (1) the flexibility and portability of the CORBA object-oriented programming model with (2) the efficiency of lower-level transport protocols. The stream connection establishment and management is performed via conventional CORBA operations. In contrast, data transfer can be performed directly via more efficient lower-level protocols, such as ATM, UDP, TCP, and RTP. This separation of concerns addresses the needs of developers who want to leverage the language and platform flexibility of CORBA, without incurring the overhead of transferring data via the standard CORBA interoperable inter-ORB protocol (IIOP) operation path through the ORB.

The CORBA A/V Streaming Service specification defines interfaces and policies to allow applications to specify end-to-end QoS parameters, such as video frame rate or audio sample rate, for individual flows within a stream. It also defines a mandatory set of network-level QoS parameters, such as token bucket, peak-bandwidth, and token rate. These QoS parameters are specified as name/value pairs using the CORBA Property Service. Multimedia applications and A/V Streaming Service implementations use these name/value pairs to (1) negotiate QoS between two peer media devices and (2) modify the QoS if there is a violation in the initial QoS or if the specified QoS cannot be met due to run-time environment changes.

### 3.2.2 The TAO A/V Streaming Service QoS Framework

Though the CORBA A/V Streaming Service *specification* provides interfaces to specify and modify QoS, it is the responsibility of *implementations* to enforce the negotiated QoS. For TAO's A/V Streaming Service implementation, we designed a framework based upon the ACE QoS API (AQoSA) described in Section 3.1. This framework provides a middleware interface that encapsulates QoS-specific details within the TAO A/V Streaming Service, rather than in the multimedia applications. To obtain end-to-end QoS therefore, application developers simply specify the QoS they require for each flow in their streams. These specifications are translated, enforced, and modified transparently by the AQoSA-enabled TAO A/V Streaming Service.

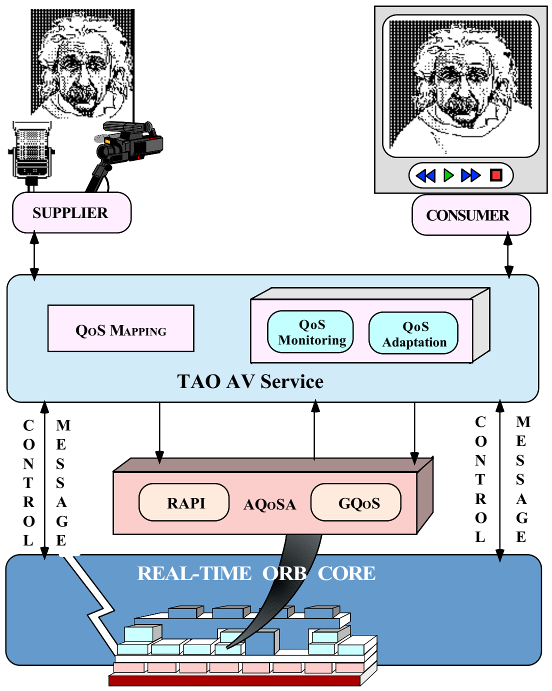

Figure 6: QoS Components in the TAO A/V Streaming Service Framework

### 3.2.3 Components in TAO's A/V Streaming Service Framework

TAO's A/V Streaming Service framework comprises three main components, which are shown in Figure 6 and outlined below.

**1. QoS mapping:** TAO's QoS mapping component translates QoS parameters between the application-level and network-level. This translation process allows application developers to specify QoS as perceptual qualities, *e.g.*, the video quality can be specified by the frame rate for a video flow. The QoS mapping component is then responsible for translating the frame rate into network bandwidth requirements.

**2. QoS monitoring and adaptation:** These two components support applications that require QoS guarantees, but are flexible in their needs, *e.g.*, they can adapt to changing resource availability within specified QoS bounds. The QoS monitoring component, which consists of AQoSA and the higher-level TAO middleware framework, measures end-to-end QoS of application flows over a finite period of time. If there are violations in the reserved QoS the monitoring component notifies the application of actual resources available currently. TAO's CORBA A/V service QoS midleware can then decide if the available QoS is sufficient to meet the requirements specified by an application.

If the available QoS is insufficient, TAO's A/V Streaming Service notifies the application, which in turn can renegotiate the QoS or adapt to the available QoS. Due to the extensible design of TAO's QoS adaptation component, various adaptation algorithms can be configured.

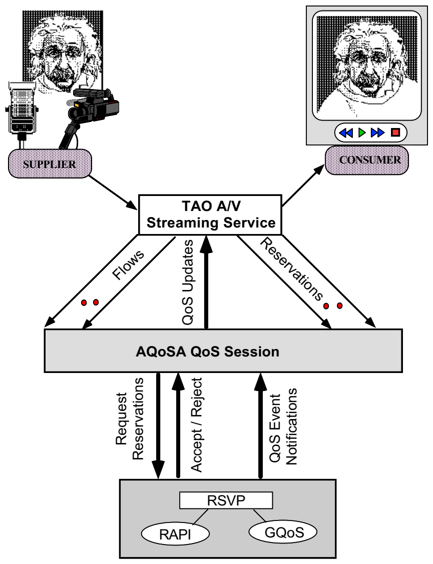

Figure 7: QoS-based Transport API

**3. QoS-Based transport API:** This component is provided by AQoSA, which enforces end-to-end QoS by reserving network resources in accordance with application-level requirements. As shown in Figure 7, the CORBA A/V Streaming Service is layered atop TAO and ACE, which handle flow control processing and media transfer, respectively. The CORBA A/V service uses AQoSA for network-level QoS provisioning, renegotiation and violation notification control, and media transfer. Likewise, application-level end-to-end QoS is

1. Translated from application-level to network-level parameters via TAO's QoS mapping component and
2. Passed through the portable AQoSA interfaces that portably encapsulate the GQoS and RAPI APIs.

AQoSA uses the underlying network-level QoS capabilities to provision the specified QoS to individual application flows. In addition, AQoSA provides mechanisms that are used by TAO's QoS monitoring and adaptation components to detect QoS violations and to notify the A/V Streaming Service middleware so it can renegotiate QoS between peer media devices and application endpoints.

## 4 Shipboard Dissemination of Unmanned Air Vehicle (UAV) video

As part of an activity for the US Navy, we have been developing a demonstration application utilizing QuO to control the dissemination of Unmanned Air Vehicle (UAV) data throughout a ship. Figure 8 illustrates the initial architecture of the demonstration. It is a three-stage pipeline, with an off-board UAV sending MPEG video to an on-board video distribution process. The off-board UAV is simulated in early prototypes by a process that continually reads an MPEG file and sends it to the distribution process. The video distribution process sends the video frames to video display processes throughout the ship, each with their own mission requirements.

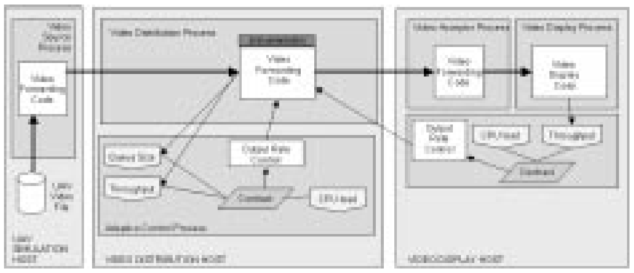

Figure 8: Architecture of the UAV application

The application maintains a data path, across which video frames are sent, and a separate control path, using CORBA IIOP, across which QuO adaptive control is sent. We built and evaluated two versions of this UAV application, one using an ad hoc TCP/IP socket implementation for the data path connection. In the other, we used TAO's A/V Streaming Service to abstract the data path connection from the rest of the application code.

QuO adaptation is used as part of an overall system concept to provide load-invariant performance. The video displays throughout the ship must display the current images observed by the UAV with acceptable fidelity, regardless of the network and host load, in order for the shipboard operators to achieve their missions (e.g., flying the UAV or tracking a target). To accomplish this, system condition objects monitor the frame rate and the host load on the video display hosts. As the frame rate declines and/or the host load exceeds a threshold, they cause region transitions, which trigger the following adaptation:

- The video distribution process is told to drop frames going to the display on the overloaded host.
- The video display on the overloaded host is told to reduce its display frame rate to the rate at which frames are being sent to it.

Simultaneously, system condition objects on the video distribution host are monitoring the host load, the input and output queues, and the frame rate. If the queues fill up or if the host load exceeds a threshold, the contract tells the video distribution process to drop frames to compensate. In this way, the adaptation attempts to maintain the video display processes displaying the current images that the UAV is observing with appropriate fidelity, regardless of the load on the various hosts.

The contracts on each host are simultaneously reporting the current contract region and video distribution metrics (e.g., queue lengths, frame rate, and number of dropped frames) to a system resource manager (RM). When QuO recognizes that the load on the video distribution host has become unacceptable, it notifies the RM. The RM then has the option of starting a video distribution process on another, less loaded host, and hooking it up to the UAV video source process and the video display processes. It then kills the processes on the overloaded host.

We collected data in two areas of performance: performance of the two versions of the application when adapting to abnormal system conditions and performance of the underlying transport protocol under normal conditions.

## 4.1 Adaptation and Performance Under Load

We performed the following experiments to test the effectiveness of our adaptive behavior. The three stages of the UAV were run on three separate hosts. Each host contained a single 200MHz Pentium Pro processor with 128KB of memory, running the RedHat 6.2 distribution of Linux. The application was run and data were collected over a period of five minutes. At approximately one minute into this period, we introduced three competing processes, each one of which would have taken 30% of the CPU on an unloaded machine. This competing load reduced the ability of the distributor process to transfer the MPEG video at the bit rate at which it was recorded. After one more minute, the three competing processes were terminated, and the distributor was given the rest of the five-minute period to recover from any disturbance.

We conducted two experiments using this setup. In the first experiment, we used ad-hoc TCP sockets code for the video transfer. In the second experiment, we used the A/V Streams implementation. Each experiment consisted of two cases:

- Control case: The QuO kernel was not run, and adaptation in the distributor was disabled.

- Experimental case: The QuO kernel was attached to the distributor and was enabled to adapt to the system load by dropping B (bidirectional interpolated) frames, or if necessary both B and P (predictive extrapolated) frames from the MPEG stream.

The results are shown in Table 1. These figures are based on the timing of the I frames, which occur 2 times per second and (in the ideal) are not biased between non-adaptive and adaptive cases since our adaptations never drop I frames. (We also obtained numbers for all frames, and they are not much different from these.) The lateness of each frame is the difference between when the frame was processed by the viewer's proxy, and when the frame should have been processed assuming that a new I frame should arrive every 1/2 second. The figures for

| Protocol | Adaptation | I frames delivered | Mean lateness (sec) | Max lateness (sec) |
|---|---|---|---|---|
| Ad-hoc TCP | No | 600 | 5.400 | 32.696 |
| Ad-hoc TCP | Yes | 600 | 0.067 | 1.930 |
| A/V streams | No | 541 | 22.085 | 29.936 |
| A/V streams | Yes | 598 | 0.834 | 2.035 |

Table 1: Late frames in UAV real-time application

mean and maximum lateness are much better for QuO than for the non-adaptive cases, showing that the application satisfied the real-time performance requirements with QuO than without. (Clearly, both mean and maximum lateness could have been reduced, perhaps even to zero, by buffering a sufficient number of frames at the player and showing them after a delay of some seconds, but this would violate the requirement that the frames be displayed as near as possible to the time that the video camera captured them.)

These figures also show some room for improvement that can be addressed during the current work in progress. First, the figures for mean and maximum lateness with QuO could potentially be further improved by providing QuO with more complete and timely information about the frame rate achievable at any given time, so that it can more rapidly reduce the number of frames sent under load conditions, and increase the number of frames sent under no-load conditions. Second, the A/V Streams implementation is being improved, which ultimately should bring its performance figures closer to the ad-hoc TCP figures.

## 4.2 Unloaded Throughput and Jitter

In order to better define the baseline for the performance of the adaptive application, We have collected preliminary measurements of the end-to-end quality-of-service of the three-stage pipeline in the absence of load in two separate implementations.

Table 2 describes the quality-of-service observed in the UAV MPEG streaming video implementation over standard TCP sockets code provided by the ACE libraries. Measurements are taken at three points in the UAV application,

- uav: When the UAV (stage 1) sends data to the distributor.
- distrib: When the distributor sends data to the proxy
- proxy: When the proxy is ready to give the data to the viewer.

Table 2 describes the quality-of-service observed in the UAV MPEG streaming video application when the TCP sockets were replaced by A/V Streams. In both cases, the video was sent by the UAV source process at a rate of 30 frames per second and observed for approximately 30 seconds. The three stages of the UAV were run on three separate hosts, each containing a single 200MHz Pentium Pro processor with 128KB of memory and running the RedHat 6.2 distribution of Linux.

| Stage | uav | distrib | proxy |
|---|---|---|---|
| Samples | 900 | 900 | 899 |
| Mean | 33.3 ms | 33.3 ms | 33.3 ms |
| standard dev. | 22.4 ms | 25.2 ms | 24.5 ms |

Table 2: Throughput and jitter in UAV over TCP

| Stage | uav | distrib | proxy |
|---|---|---|---|
| Samples | 899 | 900 | 899 |
| Mean | 33.4 ms | 33.4 ms | 33.4 ms |
| standard dev. | 38.1 ms | 41.7 ms | 40.8 ms |

Table 3: Throughput and jitter in UAV over A/V Streams

In both implementations, the mean times are essentially equal to 1/30 second close to 33.3 ms, meaning the pipeline had completely adequate throughput for the test video (data rate 1 Mbit per second).

In addition, in the A/V Streams implementation we observed several "peaks" or clusters of interframe times whose modal values are recorded in Table 4 The maximum interframe time observed was 212 ms. The TCP implementation times were clustered in an approximately bell-shaped curve around 33 ms, with a few outliers ranging up to 129 ms.

| interframe time | number of occurrences |
|---|---|
| 2 ms | 132 |
| 30 ms | 48 |
| 40 ms | 34 |
| 61 ms | 27 |
| 70 ms | 17 |

Table 4: Throughput and jitter in UAV over A/V Streams

Clearly, frame rates varied considerably during the 30 seconds even though no unusual load was placed on the system. Measuring the rate over more than two consecutive frames tends to reduce the variation in rate because one or two slow frames will cause the system to try to "catch up" with later frames. The number of 2 ms interframe times in the A/V Streams implementation, for example, is attributable to prior frames whose arrivals at the proxy were delayed at least 1/30 second past the time they were due, with the result that the following frames could be sent to the viewer immediately afterward. Since the maximum bit rate achievable by the A/V Streams transport was much higher than the bit rate of these frames, it was able to send them in a fraction of the normal interframe time.

We are currently working on improvements in this underlying behavior that will make measurements of the UAV application taken on short time scales a more reliable measure of the application's reaction to mean system loads. The goal is that such improvements will enable quicker and more accurate QuO reactions, and enable improvements in the performance figures shown in the previous section.

### 4.3 Integrating AQoSA into the UAV Application

Section 4 describes how the QuO framework allows the application to monitor the system conditions, like CPU load, and adapt to these changes. In order to monitor and adapt to network conditions like change in network bandwidth we will be using the AQoSA middleware. The integration of AQoSA with the UAV is being conducted in two stages. In the first stage, reflected in the above results, we have integrated the TAO A/V Streams Service into the UAV application. The TAO A/V Streaming Service initiates the connection establishment between the different components of the UAV application and then streams data between them. In the second stage, we will use AQoSA to control the bandwidth allocated to the TAO A/V Streams, thereby making delivery of the video more predictable in the presence of adverse network conditions.

For example, using AQoSA to control bandwidth, the distributor, or to be more precise, the QuO contract attached to the distributor, could actually request sufficient bandwidth to send each of the viewers a video stream with the desired bit rate of data. AQoSA would inform the QuO contract whether the desired bandwidth is available, or if not, how much, if any, was actually reserved; and if the bandwidth is reduced at a later time, AQoSA would inform the contract of this fact via a callback. If no bandwidth is reserved, the QuO contract would assume that bandwidth is available only on a "best-effort" basis.

The QuO contract could then adapt each channel/stream to the bandwidth received. When the full requested bandwidth is available, the distributor can send the whole desired video stream. When only a smaller amount of bandwidth could be reserved, the distributor might instead send a reduced-bit-rate version of the video, for example by dropping frames. In the case of "best-effort" bandwidth, the actual available bandwidth could be guessed by observing the actual transmission rates achieved, and again the contract could adjust to any reductions in available bandwidth by reducing the bit rate of the video, much as in the example in section 4.2 of adapting to CPU availability.

## 5 Concluding Remarks

The maturation of the QoS-enabled DOC middleware described in this paper is helping to decrease the cycle-time and effort required to develop high-quality systems with stringent QoS requirements. Distributed applications are increasingly being composed out of flexible and modular reusable software components and services, instead of being programmed entirely from scratch via lower-level, proprietary tools. Moreover, standards-based DOC middleware, such as CORBA 3.0 and the Java virtual machine, enables applications to run portably on multiple configuration and operating platforms.

The case study described in Section 4 is representative of the emerging class of multimedia applications whose resource requirements can vary dynamically at run-time. The QoS-enabled CORBA ORB and Audio/Video Streaming Service middleware developed using ACE and TAO and the QuO middleware help to simplify and coordinate such applications. These capabilities provide a cost-effective strategy for improving the quality of service received by end-users. This, in turn, helps to reduce decision/action times for time-critical applications and generally improves overall system response in dynamically changing environments.

## References

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

[2] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[3] G. Kiczales, "Beyond the black box: Open implementation," *IEEE Software*, 1996.

[4] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, , W. Sanders, D. Bakken, M. Berman, D. Karr, and R. E. Schantz, "Aqua: An adaptive architecture that provides dependable distributed objects," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pp. 245–253, October 1998.

[5] BBN Distributed Systems Research Group, DIRM project team, "Dirm technical overview." Internet URL http://www.dist-systems.bbn.com/projects/DIRM, 1998.

[6] R. E. Schantz, J. A. Zinky, D. A. Karr, D. E. Bakken, J. Megquier, and J. P. Loyall, "An object-level gateway supporting integrated-property quality of service," in *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.

[7] P. Pal, J. Loyall, R. Schantz, J. Zinky, , R. Shapiro, and J. Megquier, "Using qdl to specify qos aware distributed (quo) application configuration," in *Proceedings of The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 00)*, to appear March 2000.

[8] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. Karr, R. Vanegas, and K. R. Anderson, "Qos aspect languages and their runtime integration," *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components*, May 1998.

[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 *Proceedings of The 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98)*, April 1998.

[10] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken, "Quo's runtime support for quality of service in distributed objects," *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2.* New York, NY: Wiley & Sons, 2000.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[13] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A Survey of QoS Architectures," *ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture*, vol. 6, pp. 138–151, May 1998.

[14] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[15] Object Management Group, *Control and Management of Audio/Video Streams: OMG RFP Submission*, 1.2 ed., Mar. 1997.