Developing Next-generation Distributed Applications with QoS-enabled DPE Middleware

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

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

An abridged version of this paper appeared in the IEEE Communications magazine, edited by Abdi Modarressi and Sheshadri Mohan, Vol 17, No. 10, October, 2000.

Abstract

This paper describes how recent advances in distributed object computing (DOC) middleware are enabling the creation of common quality-of-service (QoS) capabilities that support next-generation distributed applications. DOC middleware helps to simplify and coordinate applications in order to leverage the underlying network and endsystem QoS architectures more effectively. This paper also describes a QoS-enabled middleware framework used to customize the CORBA Audio/Video Streaming Service for applications on multiple operating system platforms.

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

1 Introduction

Current R&D trends: The successful commercialization of today's Internet is motivating new R&D on hardware and software infrastructure support for next-generation distributed applications, such as e-commerce, autonomous vehicle control, and global event notification systems. Many R&D activities are focusing on how to scale the Internet to accommodate traffic from advanced applications requiring a wide range of capabilities that support various quality-of-service (QoS) requirements, such as predictable performance, secure communications, and fault tolerance. There are also efforts to develop adaptive applications, such as Internet telephony and streaming video, that require QoS guarantees, but can adjust dynamically to changing user demands and available resources [1].

In recent years, R&D efforts have progressed along the following dimensions:

1: Providing scalable high-performance core networking technologies, such as Gigabit Ethernet and terabit IP/ATM routers.

2: Defining network protocols and endsystem operating system (OS) architectures that enforce QoS specifications provided by applications.

3: Developing QoS-enabled distributed object computing (DOC) middleware, which simplifies and coordinates application-level services to leverage the advances in networks and endsystems from end-to-end.

Other articles in this special issue focus on TINA, Megaco, SIP, H.323, and NGN control. This article describes R&D activities on QoS-enabled DOC middleware frameworks for next-generation distributed applications. DOC middleware is distributed processing environment (DPE) software that resides between applications and the underlying operating systems, protocol stacks, and hardware devices to simplify and coordinate how these components are connected and interoperate. As shown in Figure 1, DOC middleware is commonly

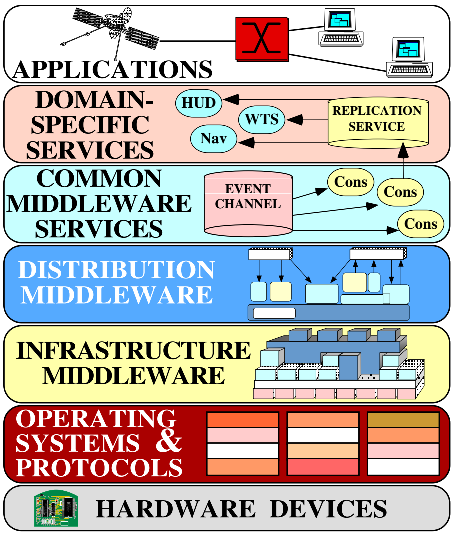

Figure 1: Layers of Distributed Object Computing (DOC) Middleware

*This work was supported in part by ATD, BBN, Boeing, Cisco, DARPA contract 9701516, Motorola Commercial Government and Industrial Solutions Sector, Motorola Laboratories, Siemens, and Sprint.

decomposed into the following layers:

• **Infrastructure middleware:** This layer encapsulates core OS communication and concurrency services to eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications via low-level network programming mechanisms, such as sockets. Widely-used examples of infrastructure middleware include Java virtual machines (JVMs) and the ADAPTIVE Communication Environment (ACE) [2].

• **Distribution middleware:** This layer builds upon lower-level infrastructure middleware and allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [3]. At the heart of distribution middleware are *Object Request Brokers* (ORBs), such as the OMG's CORBA, Microsoft's DCOM (DCOM), and Sun's Java RMI.

• **Common middleware services:** This layer augment the distribution middleware by defining domain-independent services, such as event notifications, logging, multimedia streaming, persistence, security, transactions, fault tolerance, and distributed concurrency control. Applications can reuse these services to perform common distribution tasks that would otherwise be implemented manually.

• **Domain-specific services:** Unlike the other three middleware layers, domain-specific services are not generally reusable, but instead are tailored to the requirements of particular domains, such as telecommunications, e-commerce, health-care, or process automation. Domain-specific services are the least mature of the standardized middleware layers today. Since they embody domain-specific knowledge, however, they have the most potential to increase system quality and decrease the cycle-time and effort required to develop particular types of distributed applications.

Together, these DOC middleware layers provide the following benefits:

1. Strategic focus: They elevate application developer focus away from a preoccupation with low-level operating system mechanisms and networking protocols. While it is important to have a solid grasp of these topics, they are relatively tactical in scope. Therefore, these protocols and mechanisms should be placed in the proper strategic context within a broader software architecture.

2. Effective reuse: They amortize software life-cycle effort by leveraging previous development expertise and reifying implementations of key patterns [4, 5] into reusable middleware frameworks [2]. Most distributed applications in the future will be built by integrating and scripting domain-specific and common "pluggable" middleware service components, rather than being programmed entirely from scratch.

3. Open standards: They provide a standard [3] set of software components that help to direct the focus of developers towards higher-level software application architecture and design concerns, such as the reuse of suitable security, QoS resource management, and fault tolerance services and components.

By providing these benefits, DOC middleware helps to minimize the impact of vexing inherent and accidental complexities [4], such as partial failures, distributed deadlock, and non-portable programming APIs, that have historically complicated the development of distributed applications. An increasingly important role is being played by "commercial-off-the-shelf" (COTS) DOC middleware, such as CORBA and Java, which is readily available for purchase or open-source acquisition. COTS middleware has become essential in today's software development organizations, which face many time- and effort-related constraints arising from global competitive pressures.

Paper organization: The remainder of this paper is organized as follows: Section 2 outlines key properties of next-generation distributed applications to illustrate the requirements being addressed by R&D on QoS-enabled DOC middleware, endsystems, and networks; Section 3 outlines the QoS-related aspects of CORBA 3.0, which is emerging as the industry standard of choice for QoS-enabled distributed applications; Section 4 describes the object-oriented design of a portable QoS API; Section 5 presents a case study that shows how this QoS API is used to provide QoS in multimedia services; and Section 6 summarizes concluding remarks.

2 Key Properties of Next-generation Distributed Applications

Next-generation distributed applications will require end-to-end QoS support where network and system resources must be managed both prior to and during run-time. For example, in mission-critical systems in domains such as telecommunications, global trading, distributed electronic medical imaging, and aviation collision avoidance, failure to meet certain deadlines can result in significant loss of property or even loss of life. These types of systems must therefore be analyzed and monitored off-line *and* on-line to ensure that the resources they require are allocated and managed properly.

In this section, we first present several scenarios that motivate key application requirements that must be addressed by R&D on QoS-enabled DOC middleware. We then generalize from these scenarios and summarize the QoS-related challenges associated with supporting next-generation distributed applications.

2.1 Tele-immersion Application Scenarios

Some of the most demanding types of next-generation QoS-enabled distributed applications involve *tele-immersion* [6], which combines tele-conferencing, tele-presence, and virtual reality. Tele-immersion places stringent demands at multiple levels along the end-to-end path of distributed applications, such as the following:

- **Endsystems:** Tele-immersion requires real-time response and predictable behavior from endsystems to (1) interact with the physical world within specific delay bounds and (2) present images or other stimuli to users in real-time.

- **Networks:** Tele-immersion end-users may be distributed across the Internet or intranets. Thus, applications require predictable *network* performance to provide low-latency and high-bandwidth to applications end-to-end.

Applying tele-immersion to health care: Intensive care medicine is a domain where tele-immersion applications can provide significant benefits. For instance, emergency teams responding to natural disasters or urban terrorism must make critical decisions based on information emerging from a variety of sources at an accelerated tempo. Consultations with remote experts, modeling of physiological processes, and integration of both existing and emerging information often must be performed while in close proximity to patients, as illustrated in Figure 2. To support this scenario, it is essential that

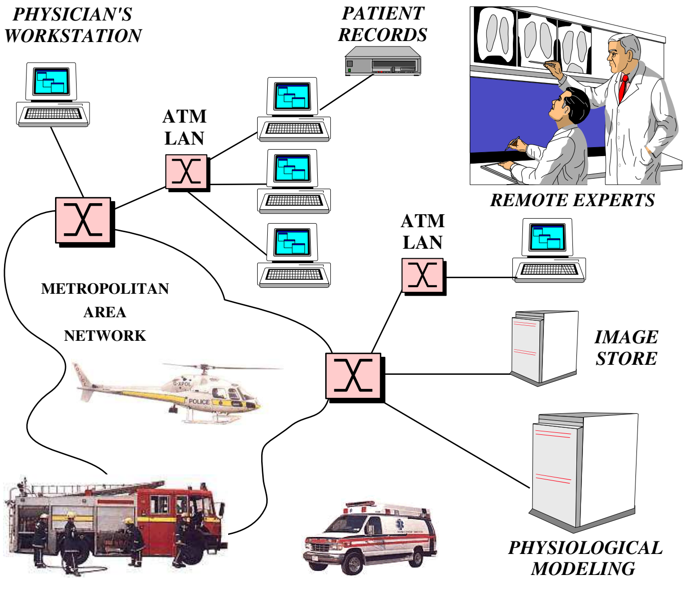

Figure 2: Real-Time Medical Informatics Example

networking and computing technologies perform and adapt in real-time to changing situational requirements, while still maintaining QoS guarantees for critical operations, such as tele-radiology or even tele-surgery.

Applying tele-immersion to aerospace: The aerospace domain will also receive significant benefit from tele-immersion applications. In the battle zone of the future, a distributed web of sensors, weapons, and decision-makers must interact in real-time to gain and preserve tactical advantage. Since battle environments change constantly, systems must adapt rapidly both globally and locally. For instance, multiple unmanned air vehicles (UAVs) can provide surveillance, weapons delivery, and battle damage assessment capabilities at both tactical and strategic scopes.

With support from tele-immersion applications, immediate remote interaction with the physical environment can help maximize, effectiveness at all levels of the system. For example, a swarm of UAVs can (1) share sensor data, post-processed data products, and remote operator requests and (2) exchange information with remote locations, *e.g.*, via satellite, as shown in Figure 3. To respond flexibly to unanticipated

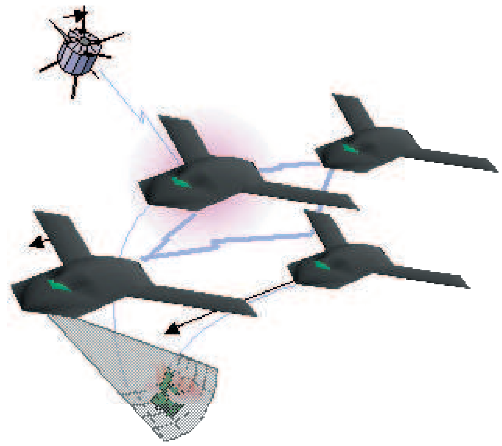

Figure 3: Unmanned Air Vehicle Example

situational factors that arise in its run-time environment, each UAV must collaborate with others in its swarm and interact in real-time with operators manning remote command and control systems.

The ability of tele-immersion systems to preserve the necessary application QoS end-to-end will translate directly into users' perceived worth of next-generation applications and their supporting services. For example, if a tele-medicine system routinely delays the delivery of packets, it will provide relatively low perceived value to its users. Supporting the demanding tele-immersion applications outlined above, therefore, requires a range of QoS support from network and endsystem elements, and the DOC middleware that integrates these elements end-to-end.

2.2 Synopsis of QoS-related Challenges for Next-generation Distributed Applications

Many research challenges arise when attempting to support the stringent QoS requirements of next-generation distributed applications, such as the tele-immersion scenarios presented in Section 2.1. Table 1 characterizes the key challenges associated with developing QoS-enabled middleware for next-generation distributed applications. In general, solutions that are emerging to meet the QoS-related challenges outlined in Table 1 possess the following capabilities:

- They offer applications the ability to flexibly configure layered resource management mechanisms needed to control their QoS end-to-end;
- They automatically protect resources needed by certain application-critical operations;
- They promote autonomous or semi-autonomous behavior to respond adaptively and reflectively to changing situational aspects in their run-time environment.

The following section summarizes recent advances in DOC middleware that provide COTS-based implementations for some of these capabilities.

3 Recent Advances in QoS-enabled DOC Middleware

Significant R&D efforts have focused on DOC middleware during the past decade. These efforts have yielded open standards, such as OMG CORBA [3], as well as popular proprietary solutions, such as Microsoft's Distributed Component Object Model (DCOM) and Sun's Remote Method Invocation (RMI). Thus, DOC middleware is now available off-the-shelf that allows clients to invoke operations on objects without concern for object location, programming language, OS platform, communication protocols and interconnects, or hardware [3].

This section outlines the QoS-related aspects of CORBA 3.0, which is emerging as the industry standard for distributed applications that possess a wide range of QoS requirements.[1] In addition, we outline the key open R&D issues related to QoS-enabled DOC middleware.

3.1 Overview of CORBA 3.0 QoS Capabilities

First-generation DOC middleware was not targeted for applications with stringent QoS requirements. Not surprisingly, its efficiency, predictability, scalability, and dependability was problematic. Over the past several years, how-

[1] The complete CORBA 3.0 standard should be available in 2001. However, many key components [7, 8, 3] are already specified and available in COTS ORBs.

Requirement	Description
Diverse inputs	Next-generation applications must simultaneously use diverse sources of information, such as raw sensor data, command & control policies, and operator input, while sustaining real-time behavior.
Diverse outputs	Many next-generation applications must concurrently produce diverse outputs, such as filtered sensor data, mechanical device commands, and imagery, whose resolution quality and timeliness is crucial to other systems and users with whom they interact.
Shared resources	Application-critical and/or time-critical operations must share resources effectively end-to-end with operations that possess less stringent timing or criticality constraints.
Critical operations	QoS management for next-generation applications with hard timing constraints for certain critical operations must insulate these operations from the competing demands of non-critical operations.
High availability	The system infrastructure must react to hardware failures, network topology changes, and feature upgrades, and restore correct real-time operation within a bounded interval after failures/changes.
Diverse resource management goals	There must be a balance between different and sometimes competing resource management goals involving different kinds of resources, such as maximizing utilization of the CPU or sharing link bandwidth fairly between threads at the same priority.
End-to-end requirements	Many next-generation applications may operate in heterogeneous environments and must manage distributed and layered resources to enforce end-to-end QoS requirements.
System configuration	Developers of next-generation applications must be able to control the internal concurrency, resource management, and resource utilization configurations throughout networks, endsystems, middleware and applications, to provide the end-to-end QoS to applications.
System adaptation	Next-generation middleware frameworks and applications must be able to (1) autonomously reflect upon situational factors as they arise in their run-time environment and (2) adapt to these factors while preserving the integrity of key mission-critical activities.
	Operators must be insulated from the programming model for resource management, *e.g.*, via suitable abstractions for communicating operator QoS requirements and monitoring/controlling the received QoS.
Development time & cost management	The time and effort expended to develop, validate, optimize, deploy, maintain, and upgrade next-generation distributed applications must be amortized across product families.

Table 1: QoS-related Challenges for Next-generation Distributed Applications

ever, the use of CORBA-based [3] DOC middleware has increased significantly in aerospace, telecommunications, medical systems, and distributed interactive simulation domains. These domains are characterized by applications with high-performance and real-time QoS requirements. The increased adoption of CORBA middleware in these domains stems largely from the following two factors:

1. The maturation of DOC middleware patterns and frameworks: A number of notable advances in patterns and frameworks for QoS-enabled middleware have occurred recently. For instance, research conducted in the DARPA Quorum program [1, 9] has identified key design and optimization patterns [4], which have been instantiated into high-quality frameworks [2, 10] for QoS-enabled DOC middleware and applications. These patterns and frameworks are now being applied widely in COTS DOC middleware products [11].

2. The maturation of DOC middleware standards: The OMG's suite of CORBA standards has matured considerably over the last several years, particularly with respect to the specification of QoS-enabled components and capabilities. For instance, the forthcoming CORBA 3.0 [3] standard includes the Messaging [8] and Real-time specifications [7]. The CORBA Messaging specification defines asynchronous operation models and allows applications to control many end-to-end ORB QoS policies, such as timeouts, priority queueing order, and message reliability semantics. The Real-time CORBA specification defines standard interfaces and policies for managing ORB processing, communication, and memory resources.

As shown in Figure 4, CORBA 3.0 ORB endsystems consist of network interfaces, operating system I/O subsystems and communication protocols, and CORBA-compliant middleware components and services. CORBA 3.0 identifies capabilities that can be *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, peer-to-peer) integrated and managed by ORB endsystems to ensure end-to-end predictable behavior for operations exchanged between CORBA clients and servers.

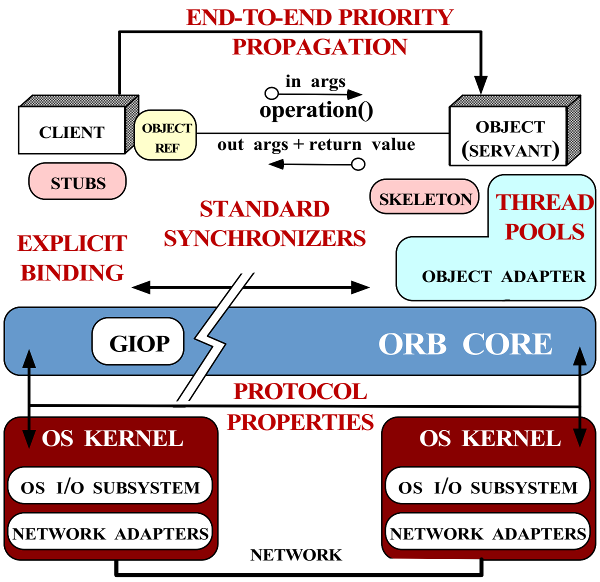

Figure 4: QoS-enabled ORB Endsystem Capabilities in CORBA 3.0

Below, we outline the QoS-related capabilities in CORBA 3.0, starting from the lowest level of abstraction and building up to higher-level common services and applications.

Communication infrastructure resource management: A CORBA 3.0 endsystem must leverage policies and mechanisms in the communication infrastructure that support resource guarantees. This support can range from (1) determining which connection to use for a particular invocation to (2) exploiting advanced protocol properties, such as controlling the cell pacing rate of ATM virtual circuits.

OS scheduling mechanisms: ORBs exploit OS mechanisms to schedule application-level activities end-to-end. The CORBA 3.0 specification targets fixed-priority real-time systems, where thread priorities are set by applications and only changed by the ORB endsystem to enforce priority inheritance or priority ceiling policies. Thus, these mechanisms correspond to managing OS thread scheduling priorities. CORBA 3.0 focuses on operating systems that allow applications to specify thread scheduling priorities and policies. For example, the real-time extensions in IEEE POSIX 1003.1c define a static priority FIFO thread scheduling policy that meets this requirement.

ORB endsystem: ORBs are responsible for communicating requests between clients and servers transparently. An ORB endsystem must therefore provide standard interfaces that allow applications to specify their resource requirements to an ORB. The policy framework defined in CORBA 3.0 defines standard interfaces and QoS policies that allow applications to configure and control the following resources:

- *Processor resources* via thread pools, priority mechanisms, and intra-process mutexes;
- *Communication resources* via protocol properties and explicit bindings with non-multiplexed connections;
- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

Common middleware services: Having a QoS-enabled ORB that manages endsystem and communication resources does not provide a complete end-to-end solution. Therefore, ORBs must also preserve QoS properties for higher-level common services and application components, such as the following defined in CORBA 3.0:

- A global Scheduling Service [7, 9, 12] that distributed applications can use to manage and schedule distributed resources via fixed-priority analysis and scheduling techniques.

- An Audio/Video (A/V) Streaming Service [13] that facilitates the creation of data, video, and audio streams between two or more media devices.
- A Fault Tolerance service [14] that defines a standard set of interfaces, policies, and components to provide robust support for applications requiring high availability.

These services augment ORBs to provide mechanisms that support the specification and enforcement of end-to-end operation timing, stream synchronization, and dependability. Developers can structure their applications to exploit the reusable capabilities exported by QoS-enabled ORBs and their associated higher-level common services.

3.2 Open R&D Issues

Meeting the QoS requirements of next-generation distributed applications requires an integrated framework architecture that can deliver end-to-end QoS support at multiple levels of abstraction. While DOC middleware based on CORBA 3.0 offers solutions to certain resource management challenges facing researchers and developers, it does not yet provide a complete solution for all types of distributed applications. In particular, the CORBA specification does not yet define standard components, protocols, or APIs that support the following capabilities:

Dynamic resource management: The real-time support in CORBA 3.0 targets applications designed using fixed-priority scheduling. However, an important class of real-time applications encounter dynamic load conditions that can vary significantly at run-time, particularly in interactive telecommunication systems and open-loop control platforms. It is hard for developers of these types of systems to determine the priorities of various operations *a priori* without significantly underutilizing various resources, such as CPU and network bandwidth.

To address these issues, the OMG is attempting to standardize dynamic CORBA scheduling [15] techniques, such as deadline-based [16], value-based [17], and hybrid static/dynamic [12] scheduling. Dynamic scheduling offers relief from certain limitations of static scheduling, such as resource underutilization. It often has a higher run-time cost, however, because certain scheduling operations must be performed on-line. Moreover, operations can be scheduled dynamically that may never be dispatched. Therefore, additional R&D is necessary to determine the most suitable ways to integrate dynamic scheduling into DOC middleware.

Portable networking QoS APIs: Many network-oriented QoS technologies, such as integrated services (IntServ), differentiated services (DiffServ), multi-protocol label switching (MPLS), common open policy service (COPS), and bandwidth brokers (BB), have been defined to enable network-level QoS. Most existing approaches are highly platform/protocol-specific, however. This tight coupling makes it hard to develop and deploy *portable* applications that use these networking QoS capabilities effectively.

Some work has been done to provide developers with standard programming interfaces [18] that can leverage advances in underlying network technology to provide application-level QoS guarantees. However, standard DOC middleware, such as CORBA 3.0, only defines limited protocol property APIs and cannot yet handle sophisticated network-level QoS capabilities, such as IntServ and DiffServ. Section 4 describes a richer QoS API, called *AQoSA*, that can be integrated with CORBA or other DOC middleware so that applications can both (1) benefit from middleware capabilities and (2) utilize network-level QoS support via portable middleware-centric APIs.

Multiple QoS property integration: While emerging network-level QoS mechanisms are essential enabling technologies, they are insufficient in isolation because they only manage network-level QoS. Likewise, although some operating systems now support real-time scheduling of CPU resources, they do not provide integrated end-to-end solutions. In general, conventional QoS solutions tend to focus either on specific network signaling and enforcement mechanisms or single endsystem resource allocation techniques. While these research activities are important building blocks, they often yield point solutions that emphasize relatively fixed, lower-level policies and mechanisms.

Introducing application-level awareness of changes to expected and delivered QoS is a new direction for inserting adaptive behavior into distributed applications. Adaptation can occur at any and all of the various system layers, including customized approaches in the application itself and standard service (re)configurations within the supporting middleware and network infrastructure, such as the following examples:

- **Application-level adaptation:** This type of adaptation might involve moving from full-motion video over high-speed links to audio and still imagery or even text-only interactions over low-speed links.

- **Service-level adaptation:** This type of adaptation might involve acquiring additional bandwidth by preempting a low-priority application or automatically instantiating additional resource replicas when others become unreachable.

We believe the key to success in these adaptations lies in developing translucent paths through system layers that can integrate multiple QoS properties effectively. These properties must encompass both *performance measures*, such as latency and throughput, and *mission-critical aspects*, such as real-time constraints, dependability, and security. A key R&D challenge is to provide maximum utility to applications and end-users while minimizing interference that can result from a series of independent or transparent actions [1].

4 AQoSA: a Common Middleware-centric QoS API

4.1 Motivation

As network-level QoS protocols and mechanisms mature, developers increasingly require a common interface for (1) specifying the QoS requirements of their distributed applications and (2) receiving notifications from the underlying network and QoS infrastructure when QoS-related conditions change. This common application QoS API is motivated by the following needs:

- **Enhanced portability:** To adapt more readily to new market opportunities and technology innovations, applications must be shielded from non-portable platform- and protocol-specific details, such as representations of QoS parameters and mechanisms for detecting changes in the network-level QoS.
- **Higher-level QoS specification:** Although network-level QoS protocols provide mechanisms for allocating resources between endsystems, they do not address the translation from application-level QoS parameters to network-level QoS parameters.
- **Increased adaptivity:** QoS-enabled distributed applications and higher-level middleware must be notified when available resources change so that they can re-negotiate their QoS specifications.

4.2 Solution Approach → a Common Middleware-centric QoS API

All these considerations motivate the need for a common platform- and protocol-independent QoS API that can expose the underlying network-level QoS protocols to applications via higher-level DOC middleware, as shown in Figure 5. The

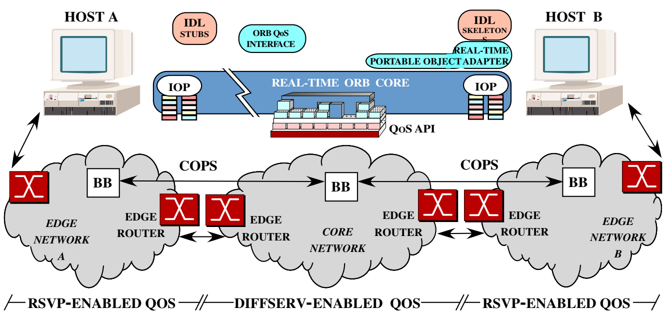

Figure 5: End-to-end QoS Architecture

end-to-end QoS architecture shown in Figure 5 allows applications to obtain network-level QoS guarantees via standard distribution middleware and middleware services, such as CORBA ORBs and the CORBA A/V Streaming Service [13]. Moreover, this architecture simultaneously leverages other important middleware benefits, such as platform- and protocol-independence. In addition, a middleware-centric QoS API can provide additional functionality, such as coordinating the binding of QoS to designated application media streams and translating standard QoS flow requirements to network-level QoS properties.

We have designed a common QoS API based on the architecture shown in Figure 5 and implemented it within ACE [2]. ACE is a widely-used open-source infrastructure middleware framework that implements key patterns [4] for high-performance and real-time communication systems. The *ACE QoS API* (AQoSA) provides a portable C++ encapsulation of two separate implementations of the IntServ resource reservation setup protocol (RSVP): the *GQoS* implementation on Windows 2000 and the *RSVP API* (RAPI) implementation on UNIX.

RSVP is designed for IntServ networks and provides QoS for particular flows via three major components: a *packet classifier*, an *admission controller*, and a *packet scheduler*, as shown in Figure 6. These QoS components facilitate the creation and management of distributed reservation state across a variety of multicast or unicast delivery paths. RSVP defines a *QoS session* as a flow with a particular destination and transport-layer protocol.

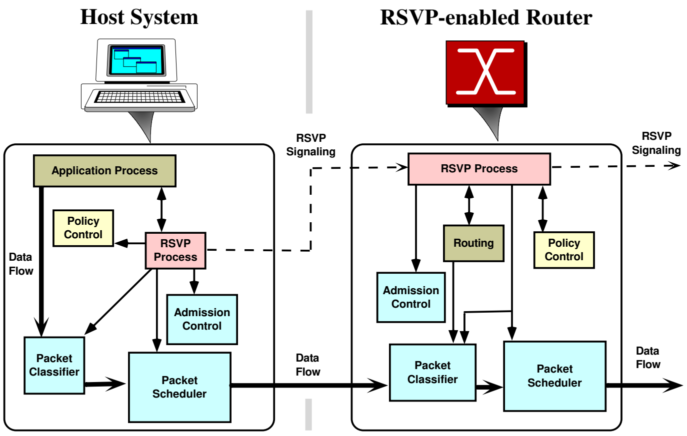

Figure 6: Components in RSVP-enabled Endsystems and Network Elements

Unfortunately, the APIs provided by the GQoS and RAPI IntServ implementations are non-portable and differ along various dimensions, such as the following:

- **Socket characteristics:** GQoS closely couples each QoS session to a socket endpoint by passing QoS session parameters to the socket calls. Conversely, RAPI handles the QoS specification separately from sockets, so that a QoS session can be specified independently of a socket.

• **Operating system and event integration:** The RAPI implementation runs in a separate process address space and notifies an application of QoS-related events via a UNIX-domain socket. Thus, the application must listen on this socket, as well as its usual data-mode sockets. Conversely, GQoS is integrated into the Windows 2000 kernel and does not require an application to listen on a different socket. Instead, it supports QoS event notifications that distinguish between a QoS and other events at the socket level.

Given these differences, it is hard to develop portable applications that are programmed directly using GQoS and RAPI. Yet, developers can benefit greatly if changes to the underlying QoS implementation does not entail changing their application design and implementation. To allow developers to create QoS-enabled applications that are independent of the underlying endsystem platform or IntServ protocol implementation, therefore, AQoSA factors out common functionality and exports an infrastructure middleware API.

4.3 Overview of the ACE QoS API (AQoSA)

AQoSA was designed by (1) inductively identifying common patterns [4, 5] used to program to existing QoS APIs and (2) developing components that reified these patterns. Below, we describe how our 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.

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 appliations that uses its API.

For example, AQoSA uses the Factory Method pattern [5], which decouples the creation of an object from its use, to relieve applications from managing the lifetime of QoS session objects. Once created, the QoS session object is added to a list of session objects to which a socket has subscribed. AQoSA makes it possible to accommodate new QoS mechanisms via subclassing. To ensure that new mechanisms conform to existing interfaces, AQoSA uses the Adapter pattern [5], which allows interoperability between components that were not designed to work together initially.

QoS Event Notification: 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 [4], which allows servers to decouple event demultiplexing/dispatching from their application-specific event handling. Thus, AQoSA's event notification mechanisms support this common usage.

In addition, AQoSA applies the Decorator pattern [5], which extends an object dynamically by attaching new responsibilities transparently. AQoSA uses this pattern to enhance the existing ACE event handler functionality via *QoS decoration*. For example, as described in Section 4.2, the native RAPI API requires an application to listen on two sockets, whereas the GQoS API requires just one. The AQoSA QoS decorators allow applications to "QoS-enable" themselves dynamically, without requiring any changes to the existing ACE reactive event handling model.

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 7 depicts the UML class diagram for the components in AQoSA. These components allow applications to specify 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 [5], 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 [5] to create and manage the lifetime of QoS session objects subscribed to by applications.

Adaptivity: Applications or higher-level middleware must be notified when changes to the state of their QoS occur. AQoSA notifies an application when (1) a particular QoS state is established for its specified flows and (2) when QoS state is updated, *e.g.*, if there are changes to the existing set of reservations for a particular session. In RSVP, these notifications

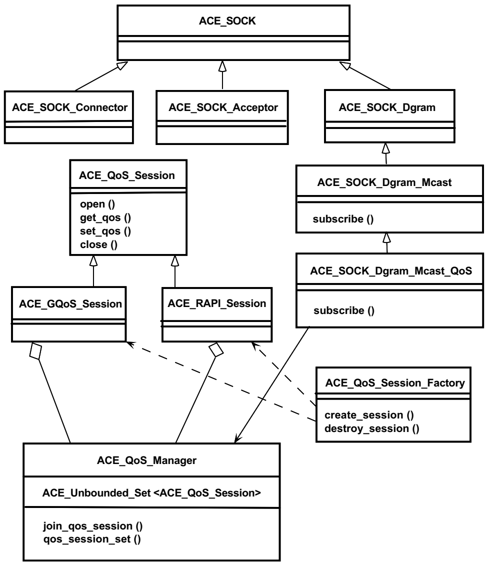

Figure 7: AQoSA QoS Class Design

are carried in *RSVP events*.

As shown in Figure 8, AQoSA receives and handles

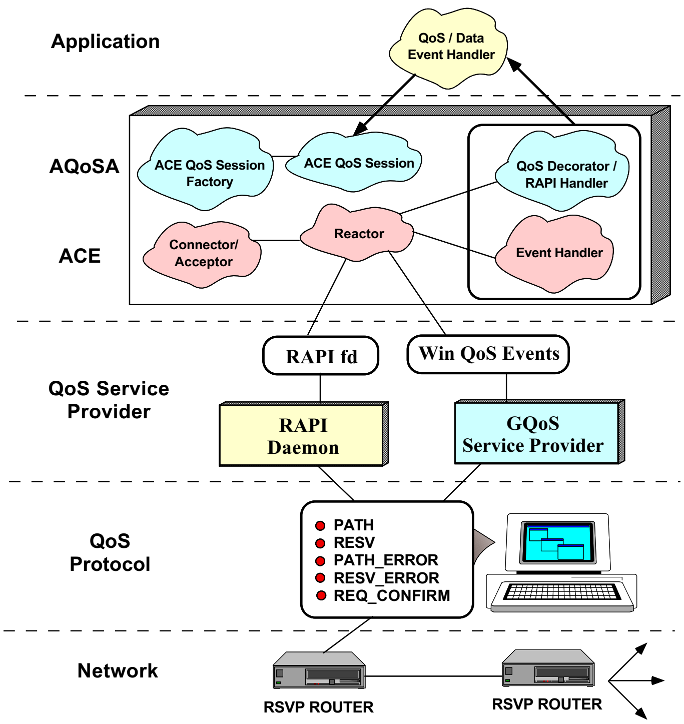

Figure 8: ACE QoS Event Handling

RSVP events uniformly for different network-level QoS implementations via the Reactor pattern [4]. 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.

An AQoSA application can use the Reactor pattern to adapt to QoS state change events, as follows:

1. It instantiates a reactor to listen for various types of events, such as I/O events, timer events, and QoS events.

2. It then registers QoS decorated event handler with a reactor to handle the QoS and I/O events. This handler updates the QoS state of a QoS session by calling the `update_qos` method on a QoS session object. This method internally calls either the `rapi_dispatch` function on UNIX or passes the `SIO_GET_QOS` flag to the Win32 `WSAIoctl` function.

3. When an RSVP QoS event arrives, the reactor notifies the associated QoS session event handler. The application then retrieves the QoS parameters from this handler and adaptively renegotiate its requested QoS.

Applications may choose not to obtain the QoS immediately after a QoS event occurs. Instead, they may defer it and have it dispatched at a later point in the program. They can also run the event handling mechanism in a different thread of control and obtain QoS notifications synchronously. This latter model is motivated by the Half-Sync/Half-Async pattern [4], which decouples synchronous from asynchronous processing in concurrent systems.

5 Case Study: Applying AQoSA to the CORBA Audio/Video Streaming Service

As shown in Section 4.3, 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 [18]. Other types of applications, however, can benefit from higher-level middleware programming models that supports a broader range of protocols and common middleware services.

For example, the TAO open-source real-time CORBA ORB [9] provides an implementation of the CORBA A/V Streaming Service [13], 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.

5.1 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 9. Streams

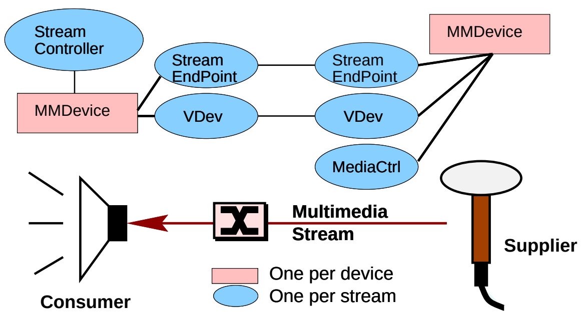

Figure 9: CORBA A/V Streaming Service Components

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.

5.2 Implementing the TAO A/V Streaming Service with AQoSA

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 4.3. 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.

5.2.1 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 10 and outlined below.

1. QoS mapping: TAO's QoS mapping component translates QoS parameters between the application-level and network-level. QoS mapping can be performed both during resource allocation and during renegotiations. 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.

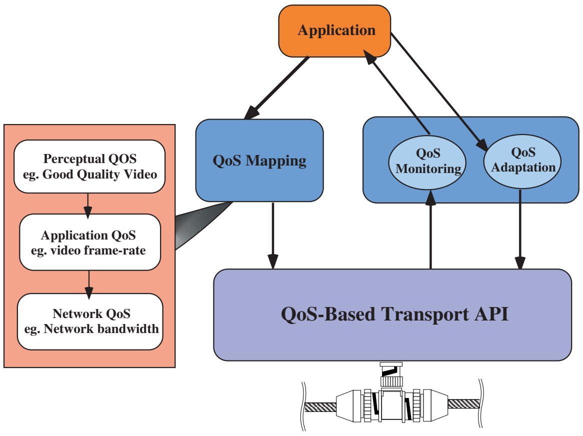

Figure 10: QoS Components in the TAO A/V Streaming Service Framework

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. Adaptation can occur at various levels of abstraction, ranging from the transport (*e.g.*, flow control), to the application (*e.g.*, MPEG-II coding rate adaptation), to middleware signaling (*e.g.*, QoS renegotiation). Due to the extensible design of TAO's QoS adaptation component, various adaptation algorithms can be configured. TAO's QoS adaptation component is accessible by (1) the application, which performs application-level adaptation and (2) the distribution middleware, which coordinates the transport-level adaptation.

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 11, 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.

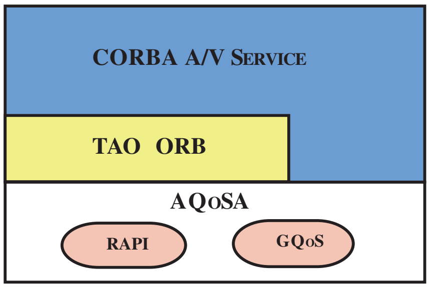

Figure 11: QoS-based Transport API

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.

5.2.2 Meeting Design Requirements

As with AQoSA, TAO's A/V Streaming Service framework was designed by (1) inductively identifying common patterns [4, 5] used to program to existing QoS APIs and (2) developing a framework that reified these patterns. Below, we describe our TAO's A/V Streaming Service framework implementation addressed key design requirements.

Generic and extensible QoS mapping: The CORBA A/V Streaming specification allows application developers to specify the QoS for any data stream via perceptual quality parameters. These parameters characterize application performance and provide a convenient configuration model for developers. To enforce the specified QoS, however, the application-level QoS parameters must be translated to network-level QoS that are used to transport the flows. Hence, a QoS translation component is required. The key design challenges involve (1) providing a generic application-level Qos to network-level QoS parameter mapping that is independent of the codec and network and (2) making it easy to change mapping schemes.

To address these challenges for video streams, we identified a set of application-level QOS parameters: *sharpness*, *color*, and *rate*. The sharpness of the video stream resolution is mapped to luminance quality; color is mapped to the color depth; and rate is mapped to frame rate where luminance, color depth, and frame rate are network-level QoS parameters for a video stream. A single quality factor in the range [0..1] is used to specify the required quality of the displayed video, where 0 is the best and 1 is the worst quality. The relative preference of these perceptual quality parameters can also be specified by assigning them weights.

TAO's A/V Streaming Service defines generic mapping functions to map application-level quality factors and relative weights specified by application developers into the network-level QoS parameters, such as the peak bandwidth of the video. The specified mappings from perceptual quality to system and network parameters are independent of the codec and network. However, the values of the relative weights may vary across codecs. TAO's A/V Streaming Service uses wavelet transformations [19] to encode video streams. Alternative mapping schemes can be configured via the *Strategy* Pattern [5], which defines a family of interchangeable algorithms.

Specific QoS parameter monitoring: After applications specify their required QoS, AQoSA uses reservations to help enforce these specifications. At the time of the reservation the application may receive the desired QoS. If network conditions change over time and the desired QoS is no longer be available, however, application performance may be affected adversely. Hence, multimedia applications must be notified when the current QoS changes so that appropriate steps can be taken to either modify the QoS requirements or terminate the flows.

As described in Section 4.3, AQoSA propagates certain network-level QoS state changes to higher-level middleware and applications. However, other types of QoS changes are not detected by AQoSA. For example, certain QoS parameters, such as late frames, are not detected by the AQoSA network-level QoS event notifier. Likewise, if a receiver adapts to available QoS resources it must be notified when changes occur to specific QoS parameters, such as jitter, so it can then selectively accept or reject the sender's data. To facilitate specific receiver adaptations we have added the following monitoring components to TAO's A/V Streaming Service:

- **Bandwidth monitor:** This component uses AQoSA's notification mechanisms to determine changes in the bandwidth over a period of time. Applications can then adapt appropriately by scaling the flows either up or down.
- **Late frame monitor:** This component checks the arrival times of the packets to determine if they are delayed beyond an expected time and should therefore be dropped.
- **Jitter monitor:** This component measures packet delays that are indicative of a congested network. If congestion is detected, the receiver can notify the sender to decrease the frame rate.

All three monitors use the Reactor pattern [4] and TAO's internal reactor instance to notify the application of changes in the corresponding QoS parameters. Applications can register event handlers with TAO's reactor for these monitor events.

Media flow adaptation: Network conditions may change over time. Thus, applications may no longer receive the QoS they specified originally. AQoSA's QoS notification mechanisms can inform the application of changes to the currently available QOS. Applications can decide to terminate the corresponding flow(s), modify their QoS requirements, or adapt to the available QoS. To modify QoS requirements, the applications can then use AQoSA to renegotiate their QoS parameters, within the bounds of the available QoS.

Applications may require either implicit or explicit adaptation, where the former is transparent to the application and the latter is not. Both types of adaptations must be supported for effective dynamic QoS management. Appropriate filter and adaptor selection mechanisms are also required.

To address these requirements, TAO's A/V Streaming Service contains *QoS Adaptor* and *QoS Filter* components that enable applications to adapt to changes in available QoS, as follows:

- **QoS Adaptors:** These components provide explicit adaptation by manipulating the codec or changing the video playout time.
- **QoS Filters:** These components are present both at senders and receivers and reside between the application and the network and provide implicit adaptation. Senders use *shaping filters* to tune the data flow in accordance with available network resources, such as buffers or packet transmission rates. Receivers use *selection filters* to deliver parts of the data stream to the application as dictated by application QoS requirements. For example, video streams encoded using wavelet transforms can drop low frequency image frames when the specified QoS does not require high resolution.

To facilitate the selection and addition of filters, TAO's A/V Streaming Service applies the Chain of Responsibility Pattern [5]. This pattern avoids the coupling of the sender of a request from the receiver by giving more than one object a chance to handle the request. TAO's QoS Adaptors and Filters are the receiving objects and they pass the request along the chain until one of them handles it as dictated by the QoS policy.

To facilitate the selection of the appropriate adaptation and filter components, TAO's A/V Streaming Service defines a QoS Policy object through which application developers can specify the required QoS policies, such as which adaptation(s) and filter(s) to apply. The QoS policy helps the receiving objects in the chain of adaptors and filters decide if they must process the data or forward it to the next receiving object in the chain.

QoS management in multicast scenarios: TAO's CORBA A/V Streaming Service can be used to develop multicast applications, such as videoconferencing. The AQoSA layer provides a multicast QOS session management that such applications can leverage to enforce QoS in multicast scenarios and to be notified when QoS changes. However, conventional adaptation to changes in QoS is not practical and scalable in the multicast scenarios. For example, if more than one receiver sends a change in QoS request to a sender, *and* if the requests are not identical, it may be hard for the sender to satisfy all the requests.

To handle multicast applications, TAO's A/V Streaming Service uses a receiver-centric QoS adaptation approach. This QoS adaptation is achieved by splitting data streams into substreams, each with different bandwidth requirements. All substreams are then transmitted, as follows:

- **One flow:** In this scenario, the receiver selection filter can choose the packets of a particular substream as allowed by the available bandwidth.
- **Three separate flows:** In this scenario, each substream can subscribe to a separate multicast group and the receiver can tune in to all or any group as allowed by the available bandwidth.

When the receivers require a change in their QoS requests, the RESV messages sent from these receivers are merged so that the sender receives a single request. The merging can be done by the network filters in the data path or by a filter at the sender.

6 Concluding Remarks

Advances in core hardware technologies and protocols are enabling the convergence of data and voice networks into a single communication infrastructure that provides a range of multimedia services. The success of the Internet has motivated the development of next-generation distributed applications that will use the emerging communication infrastructure to provide novel tele-immersion functionality, such as distance learning, tele-medicine, and even remotely controlled medical surgical procedures. These advanced applications and communication infrastructures will enable the concentration of R&D expertise to reduce development effort, while expanding the services delivered to geographically distributed locations.

In theory, it will be possible to develop these next-generation applications by writing directly to low-level network programming APIs, such as sockets. However, contemporary economic and organizational constraints, as well as competitive pressures, make it increasingly implausible to do so *in practice*. Thus, distributed object computing (DOC) middleware has emerged as an enabling technology that allows researchers and developers to compete more effectively in markets where deregulation and global competition motivate the need for increased software productivity, quality, and cost-effectiveness.

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. 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. Thus, they can be adapted more readily to new market opportunities, technology innovations, and dynamic changes in their run-time environments.

The case study described in Section 5 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 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.

ACE, AQoSA, TAO, and TAO's A/V Streaming Service have been applied to a range of real-time applications, including many telecommunication systems, aerospace systems, financial systems, medical systems, and manufacturing process control systems. The source code and documentation for ACE and TAO are freely available from URL `www.cs.wustl.edu/~schmidt/TAO.html`.

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, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[3] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, vol. 41, pp. 44–52, October 1998.

[4] 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.

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

[6] J. Lanier, "Tele-Immersion: The Ultimate QoS-Critical Application," in *First Internet2 Joint Applications/ Engineering QoS Workshop*, May 1998.

[7] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[8] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[9] 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.

[10] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[11] R. Callison, M. Goo, and D. Butler, "Real-time CORBA Trade Study," Tech. Rep. D204-31159, The Boeing Company, 1999.

[12] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, to appear 2000.

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

[14] Object Management Group, *Fault Tolerant CORBA Specification*, OMG Document orbos/99-12-08 ed., December 1999.

[15] Object Management Group, *Dynamic Scheduling*, OMG Document orbos/99-03-32 ed., March 1999.

[16] Y.-C. Wang and K.-J. Lin, "Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *IEEE Real-Time Systems Symposium*, pp. 246–255, IEEE, December 1999.

[17] E. D. Jensen, "Eliminating the Hard/Soft Real-Time Dichotomy," *Embedded Systems Programming*, vol. 7, Oct. 1994.

[18] 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.

[19] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image Coding Using Wavelet Transform," *IEEE Transactions on Image Processing*, vol. 1, pp. 205–220, Apr. 1992.