NetQoPE: A Middleware-based Network QoS Provisioning Engine for Enterprise Distributed Real-time and Embedded Systems

Jaiganesh Balasubramanian,[*] **Sumant Tambe**[†] **Aniruddha Gokhale**[†] **and Douglas C. Schmidt**
EECS Dept., Vanderbilt University
Nashville, TN, USA

Shrirang Gadgil[†]**, Frederick Porter**[†] **and Balakrishnan Dasarathy**[†]
Telcordia Technologies
Piscataway, NJ, USA

Nanbor Wang
Tech-X Corporation
Boulder, CO, USA

Contact author: jai@dre.vanderbilt.edu

Abstract

Standards-based quality of service (QoS)-enabled component middleware is increasingly used as a platform for enterprise distributed, real-time and embedded (DRE) systems, where applications require assured QoS even when network resources are scarce or congestion is present. Traditional approaches to building QoS-sensitive DRE systems, such as Internet telephony and streaming multimedia applications, have relied on low-level proprietary command line interfaces (such as CISCO IOS) and/or low-level management interfaces (such as SNMP). Although QoS-enabled component middleware offers many desirable features, until recently it has lacked the ability to simplify and coordinate application-level services to leverage advances in end-to-end network QoS management.

This paper describes a declarative framework called NetQoPE that integrates modeling and provisioning of network QoS with a QoS-enabled component middleware for enterprise DRE systems. NetQoPE enhances prior work that predominantly used host resources to provision end-system QoS by integrating a Bandwidth Broker to provide network QoS into our QoS-enabled middleware. NetQoPE's modeling capabilities allow users to (1) specify application QoS requirements in terms of network resource utilization needs, (2) automatically generate deployment plans that account for the network resources, and (3) provision network and host elements to enable and enforce the network QoS decisions.

We demonstrate and evaluate the effectiveness of NetQoPE in the context of a representative enterprise DRE system with a single Layer-3/Layer-2 network that supports different types of traffic. Our empirical results show that the capabilities provided by NetQoPE yield a predictable and efficient system for QoS sensitive applications even in the face of changing workloads and resource availability in the network.

Keywords: QoS-enabled Component Middleware, Bandwidth Broker, Network QoS Provisioning, Model-Driven Engineering, Deployment and Configuration, Enterprise DRE systems, DiffServ Networks

1 Introduction

Enterprise distributed real-time and embedded (DRE) systems provide the runtime environment of many mission-critical applications. Examples of enterprise DRE systems include shipboard computing; power grid control systems; and intelligence, surveillance and reconnaissance systems. Such systems must collaborate with multiple sensors, provide on-demand browsing and actuation capabilities for human operators, and possess a wide range of non-functional attributes including predictable performance, security and fault tolerance.

[*]Work done while on summer internship at Tech-X Corporation
[†]This work is supported in part or whole by DARPA Adaptive and Reflective Middleware Systems Program Contract NBCH-C-03-0132

QoS-enabled component middleware, such as CIAO [33], PRiSM [28], and Qedo [27], are increasingly used to develop and deploy next-generation enterprise DRE systems. QoS-enabled component middleware leverages and extends conventional component middleware (*e.g.*, J2EE, .NET, CCM) capabilities that include: (1) standardized interfaces for application component interaction, (2) standards-based mechanisms with clear separation of concerns for the different lifecycle stages of applications, including developing, installing, initializing, configuring and deploying application components, and (3) declarative (as opposed to imperative) approaches to lifecycle management activities, such as assembly, configuration, and deployment. QoS-enabled component middleware helps overcome shortcomings of conventional component middleware by explicitly separating QoS provisioning aspects of applications from their functionality, thereby yielding enterprise DRE systems that are less brittle and costly to develop, maintain, and extend [14, 33].

Advances in QoS-enabled component middleware to date, however, have largely focused on managing CPU and memory resources in the operating system. Although standard mechanisms, such as integrated services (IntServ) [18] and differentiated services (DiffServ) [2], exist to support network QoS, there are limited capabilities in QoS-enabled component middleware to (1) specify, enable, and enforce network QoS at the level of individual flow communication (even between the same two components different QoS may be required at different times to satisfy different needs, *e.g.*, voice and video communications have different latency and jitter requirements), (2) configure network elements (*e.g.*, routers and switches of different types) in a consistent manner, regardless of their low-level provisioning and monitoring interfaces, and (3) enforce QoS with functions (such as policing for usage compliance) so all flows receive their required treatment.

To overcome these limitations, we have developed a QoS-enabled component middleware-based network QoS provisioning engine, called *NetQoPE*, that provides enterprise DRE systems with network QoS provisioning capabilities. NetQoPE includes novel model-driven mechanisms for (1) declaratively specifying network QoS requirements (in terms of priority, reliability, bandwidth required, latency and jitter) of different communication flows in a DRE system, (2) assuring network resources for communications within a DRE system based on the specified QoS, especially for critical communications, (3) automatically marking application packets for classification at the entrance to the network, and (4) policing usage compliance at the ingress points of the network so that QoS is assured for all admitted traffic.

This paper describes how NetQoPE extends our earlier work on declarative QoS provisioning frameworks, which includes: (1) a model-driven engineering (MDE) tool suite called CoSMIC (`www.dre.vanderbilt.edu/cosmic`)), which alleviates many accidental complexities associated with developing, deploying, and configuring QoS-enabled component-based enterprise DRE systems, (2) a deployment and configuration engine (DAnCE) [10] that provides standards-based deployment and configuration mechanisms to map software components onto hardware nodes using the CIAO [33] lightweight CORBA Component Model (LwCCM) [23] implementation, (3) a dynamic resource allocation and control engine called RACE [31] that manages enterprise DRE system resources, and (4) reflective middleware techniques within CIAO to support runtime adaptive CPU and memory QoS management [33]. Our new contributions described in this paper include:

- Extending CoSMIC to model network QoS requirements of DRE applications. These new QoS modeling capabilities are integrated with existing model-driven capabilities for specifying the interfaces, communication, and assembly details of component-based enterprise DRE systems.

- Extending RACE to integrate with the Bandwidth Broker [6, 7] developed by Telcordia. The Bandwidth Broker assures network resources for flows by using an admission control process that tracks network capacity against bandwidth commitments. It also provisions ingress routers to police traffic at the level of a flow.

- Extending CIAO and DAnCE to configure application communications with network QoS settings determined by the Bandwidth Broker so that the applications and network elements have the same view on the importance of a flow.

The remainder of the paper is organized as follows: Section 2 uses a representative case study to motivate key challenges in provisioning network QoS for enterprise DRE systems; Section 3 describes the novel solutions we have developed to extend our existing QoS provisioning framework; Section 4 provides experimental validation of our approach in the context of the case study; Section 5 compares our work on NetQoPE with related research; and Section 6 presents concluding remarks, lessons learned, and future work.

2 Case Study to Motivate Network QoS Provisioning Requirements

This section describes (1) the structure/-functionality of a representative enterprise DRE system case study, (2) the key network traffic and QoS requirements in the case study, and (3) the challenges NetQoPE was designed to address to provision the network QoS requirements for these types of DRE systems using QoS-enabled component middleware.

2.1 Enterprise DRE System Case Study

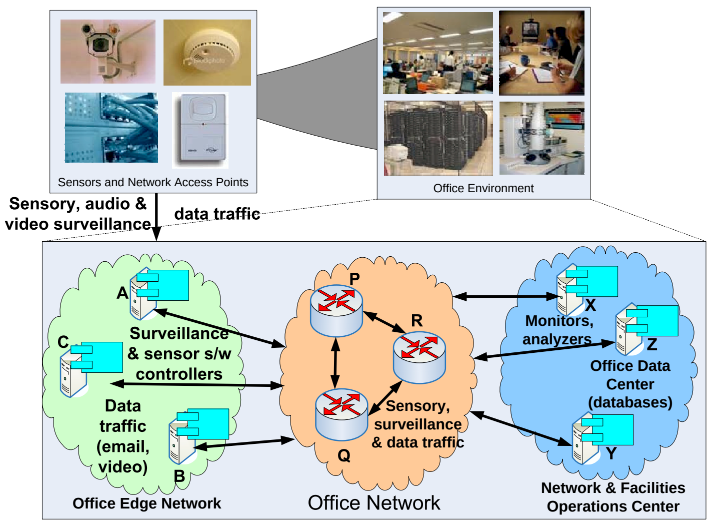

Sensors and Network Access Points
Office Environment
Sensory, audio & video surveillance
data traffic
A
C
B
Surveillance & sensor s/w controllers
Data traffic (email, video)
Office Edge Network
P
R
Q
Sensory, surveillance & data traffic
Office Network
X
Z
Y
Monitors, analyzers
Office Data Center (databases)
Network & Facilities Operations Center

Figure 1. A Corporate Environment and Network Setup

Figure 1 shows a representative enterprise DRE system case study based on a modern corporate enterprise consisting of (1) data centers hosting computing facilities, (2) desktops throughout the enterprise accessing data and compute cycles in the computing facilities, (3) videoconferencing facilities throughout the enterprise, (4) laboratories hosting expensive and sensitive instruments, and (5) safety/security related hardware throughout the premises.

The security and safety-related hardware artifacts of the network and facilities operations include sensors (such as fire and smoke sensors, audio sensors, and video surveillance cameras) that monitor sensitive parts of the corporate premises to detect intruders and unauthorized accesses, and monitor safety violations (e.g., fire). These sensors are connected to the corporate network where the data is streamed across a smaller number of subnets to the networking and facilities operation center.

Each sensor is associated with a software controller that is physically connected to the sensor via cables. The sensory information sent by the hardware sensors is filtered and aggregated by their software controllers, and relayed to the monitoring systems at the network and facilities operations center via the corporate network. These monitoring systems consist of a wide variety of hardware, such as displays, monitors, and analyzers. These sensor hardware units and monitoring systems are all driven by software controllers, which are implemented as CIAO [33] components. There is one Layer-3 IP network for all the network traffic, including email, videoconferencing, as well as the sensory traffic. The underlying network technology is itself a typical enterprise Layer-2 technology that is IP-aware, such as Fast Ethernet CISCO 6500 switches.

2.1.1 Network Traffic and QoS Requirements in Enterprise Scenario

The traffic across the corporate network is diverse and has specific QoS requirements. For example, the traffic between sensors that monitor smoke or fire send periodic events of small- or moderate-sized data and require low and predictable latencies. Conversely, video surveillance cameras send a constant stream of images to the monitoring system. These video streams consume more bandwidth and while latency may not be critical, reliability of data is a critical factor. The normal business-related operations use both TCP and UDP. For example, email traffic using TCP requires best-effort QoS, while applications, such as videoconferencing and other multimedia applications using UDP require appropriate latency, jitter, and bandwidth assurance, though they may tolerate some packet losses. These requirements demonstrate the need for differentiated QoS over different transport protocols for different applications.

The sensor traffic also has different importance levels. For example, a fire or smoke alarm has higher priority over sensors that track air conditioning temperature control within the corporate environment. Certain traffic flows within the corporate network therefore must take preference over others.

The corporate computing scenario described here presents a wide variety of characteristics that are representative of many enterprise DRE systems, particularly from the network QoS requirements perspective. The applications in these systems demand different degrees of reliability, timeliness/latency, jitter, and importance/priority.[1] Even within a process, these characteristics typically vary for different flows, as illustrated by our corporate computing case study.

2.2 Challenges Provisioning Network QoS in QoS-enabled Component Middleware

As shown in Figure 1, a modern corporate environment could have many different data flows between the same or different sets of components. Moreover, these communications can have different network QoS requirements. To support this scenario,

[1]Although higher priority translates to lower latency, to ensure a specifi c latency the network utilization must be kept quite low, *i.e.*, priority and latency are non-overlapping dimensions of network QoS.

a corporate environment requires a QoS-enabled component middleware that can support the following requirements:

1: Specifying network QoS requirements for component interactions. Component-based applications require specific levels of QoS (*e.g.*, bandwidth amount, desired latency, and priority) to be honored for the inter-component communications. For example, different flows within the corporate environment have varied network QoS requirements. Bandwidth amount is often specified as the tuple $<$rate, burst size$>$. In Layer-3 networks, the network protocol (*e.g.*, UDP, TCP) is often a parameter used to distinguish a flow. Such network QoS requirements must be collected for all dataflows involved in the system. Section 3.2 describes how NetQoPE supports this requirement.

2: Allocating resources to meet inter-component flow QoS requirements. Satisfying the network QoS requirements of flows between components involves: (1) identifying which hardware nodes the application components are deployed onto, (2) determining how much bandwidth is available in every network connection between the hardware nodes hosting the application components, (3) allocating network resources for every network link between the two nodes, and (4) configuring the switches and routers with appropriate policing and mark down behaviors for the flows. Determining how much bandwidth is available on each link for a flow between two components requires discovery of the path the flow will take. Section 3.3 describes how NetQoPE supports this requirement.

3: Configuring applications and their flows to meet network-level QoS requirements. Once the desired network-level resources for various flows are reserved, the edge routers are configured for appropriate policing and mark down behaviors for the flows. Likewise, application communications must be performed with appropriate QoS settings so routers can provide the right packet forwarding behavior for flows traversing through those devices. QoS settings for achieving network QoS are usually added to IP packets, when an application communicates with another application. In a QoS-enabled component middleware, such low-level details for working with IP packet headers are hidden by the middleware, which provides mechanisms for interacting with applications to configure them with appropriate QoS settings. These QoS settings can then be conveyed to the underlying network subsystem, which uses them to provision network QoS for those applications. Section 3.4 describes how NetQoPE supports this requirement.

3 Declarative Network QoS Provisioning Capabilities in Component Middleware

This section describes the novel enhancements we made to our QoS provisioning framework for component-based enterprise DRE systems. First, we describe of our existing QoS provisioning framework. We then present our enhancements to this framework to enable network-level QoS provisioning.

3.1 Overview of Enabling Technologies

This section provides an overview of the enabling technologies we have developed in prior work, which we leverage and significantly enhance in our current work on NetQoPE.

3.1.1 Model-driven Component Middleware

The DOC Group at Vanderbilt University has created a framework for developing and managing the operational lifecycle of component-based enterprise DRE systems. Figure 2 shows the elements in our open-source[2] tool chain, which uses model-driven engineering (MDE) tools to capture application structural and behavioral characteristics and convert them into metadata that is used by planners to determine resource allocations and component deployments. This information is subsequently used by deployment and configuration tools to host the application components in a QoS-enabled component middleware framework. We briefly describe each artifact in our existing framework below. Subsequent sections then describe how we enhanced each artifact in this tool chain to meet the network QoS provisioning requirements summarized in Section 2.2.

The Component Synthesis with Model Integrated Computing (CoSMIC) MDE toolsuite. To simplify the development of component-based applications, we developed CoSMIC, which is an open-source set of MDE tools that support the deployment, configuration, and validation of component-based enterprise DRE systems. CoSMIC supports the specification and implementation of *domain-specific modeling languages* (DSMLs), which define the concepts, relationships, and constraints used to express domain entities [16].

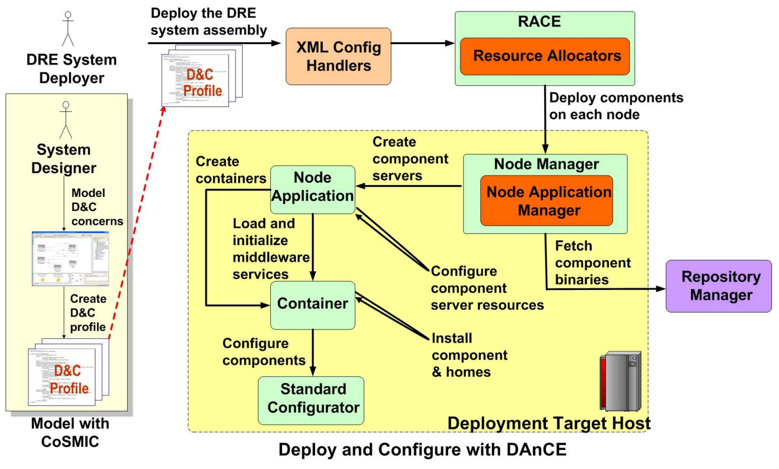


Figure 2. Model-driven Component Middleware Framework

A key CoSMIC DSML is the *Platform Independent Component Modeling Language* (PICML) [1], which enables graphical manipulation of modeling elements, such as component ports and attributes. PICML also performs various types of generative actions, such as synthesizing XML-based deployment plan descriptors defined in the OMG Deployment and Configuration (D&C) specification [25]. CoSMIC has been developed using a DSML metaprogramming environment called the *Generic Modeling Environment* (GME) [19].

Component Integrated ACE ORB (CIAO). CIAO is an open-source implementation of the OMG Lightweight CORBA Component Model (LwCCM) [23] and Real-time CORBA [24] specifications built atop *The ACE ORB* (TAO). CIAO's architecture is designed using patterns for composing component-based middleware and reflective middleware techniques

[2]All elements of our tool chain are available for download from `www.dre.vanderbilt.edu`.

to enable mechanisms within the component-based middleware to support different QoS aspects, such as CPU scheduling priority [33].

Deployment and Configuration Engine (DAnCE). Enterprise DRE system application component assemblies developed using CIAO are deployed and configured via DAnCE [10], which implements the OMG Deployment and Configuration (D&C) specification [25]. DAnCE manages the mapping of DRE application components onto nodes in a target domain. The information about the component assemblies and the target domain in which the components will be deployed are captured in the form of standard XML assembly descriptors and deployment plans generated by various MDE tools and planners. DAnCE's runtime framework parses these descriptors to extract connection and deployment information, deploy the assemblies onto the CIAO component middleware platform, and establish connections between component ports.

Resource and Control Engine (RACE). RACE [31] is an adaptive resource management framework that extends CIAO and DAnCE to provide (1) *resource monitor* components that track utilization of various system resources, such as CPU, memory, and network bandwidth, (2) *QoS monitor* components that track application QoS, such as end-to-end delay, (3) *resource allocator* components that allocate resources to components based on their resource requirements and current availability of system resources, (4) *configuration* components that configure QoS parameters of application components, (5) *controller* components that compute end-to-end adaptation decisions to ensure that QoS requirements of applications are met, and (6) *effector* components that perform controller-recommended adaptations.

3.1.2 Network QoS Provisioning Tools

Telcordia has developed a network management solution for QoS provisioning called the Bandwidth Broker [6], which leverages widely available mechanisms that support Layer-3 DiffServ (Differentiated Services) and Layer-2 Class of Service (CoS) features in commercial routers and switches. DiffServ and CoS have two major QoS functionality/enforcement mechanisms:

- At the ingress of the network, traffic belonging to a flow is classified, based on the 5-tuple (source IP address and port, destination IP address and port, and protocol) and DSCP (assigned by the Bandwidth Broker) or any subset of this information. The classified traffic is marked/re-marked with a DSCP as belonging to a particular class and may be policed or shaped to ensure that traffic does not exceed a certain rate or deviate from a certain profile.

- In the network core, traffic is placed into different classes based on the DSCP marking and provided differentiated, but consistent per-class treatment. Differentiated treatment is achieved by scheduling mechanisms that assign weights or priorities to different traffic classes (such as weighted fair queuing and/or priority queuing), and buffer management techniques that include assigning relative buffer sizes for different classes and packet discard algorithms, such as Random Early Detection (RED) and Weighted Random Early Detection (WRED).

These two features by themselves are insufficient to ensure end-to-end network QoS because the traffic presented to the network must be made to match the network capacity. What is also needed, therefore, is an adaptive admission control entity that ensures there are adequate network resources for a given traffic flow on any given link that the flow may traverse. The admission control entity should be aware of the path being traversed by each flow, track how much bandwidth is being committed on each link for each traffic class, and estimate whether the traffic demands of new flows can be accommodated. In Layer-3 networks, the shortest path used to communicate between any two hosts, so we employ Dijkstra's all-pair shortest path algorithms. In Layer-2 network, we discover the VLAN tree to find the path between any two hosts.

The Bandwidth Broker is responsible for assigning the appropriate traffic class to each flow. It then uses a Flow Provisioner to provision complex parameters for policing and marking, such that a contracted flow obtain end-to-end QoS and makes use of no more resources than allocated to it. The Flow Provisioner translates technology-independent configuration directives generated by the Bandwidth Broker into vendor-specific router and switch commands to classify, mark, and police packets belonging to a flow.

Configuration in each router/switch for scheduling and buffer management, resulting in the same per hop forwarding behavior is done not at deployment time but at the time of network engineering or re-engineering. The Bandwidth Broker is aware of these configuration decisions/restrictions, including the number of traffic classes supported, the QoS semantics of each class, and the capacity in each link. Much of this information is input to the Bandwidth Broker as reference data. The network topology itself is discovered by the Bandwidth Broker.

The Bandwidth Broker provides these functions to support the following types of deployment decisions: (1) *flow admission functions*, which reserve, commit, modify, and delete flows, (2) *queries*, which report bandwidth availability in different classes among pairs of pools and subnets, (3) *bandwidth allocation policy changes*, which support mission-mode changes, and (4) *feedback/monitoring services*, which provide feedback on flow performance using metrics, such as average delay.

The Bandwidth Broker admission decision for a flow is not based solely on requested capacity or bandwidth on each link traversed by the flow, but is also based on delay bounds requested for the flow. The delay bounds for new flows must be assured without damaging the delay bounds for previously admitted flows and without redoing the expensive job of readmitting every previously admitted flow. We have developed computational techniques to provide both deterministic and statistical delay-bound assurance [7]. This assurance is based on relatively expensive computations of occupancy or utilization bounds for various classes of traffic, performed only at the time of network configuration/reconfiguration, and relatively inexpensive checking for a violation of these bounds at the time of admission of a new flow.

3.2 Providing Mechanisms for Specifying Network QoS Requirements

Context. Challenge 1 in Section 2.2 described the need to capture the network QoS requirements of all dataflows in an enterprise DRE system to make appropriate network resource reservations for those flows. The enterprise DRE systems of interest to us, such as the corporate computing network described in Section 2.1, are moderate- to large-scale, so the number

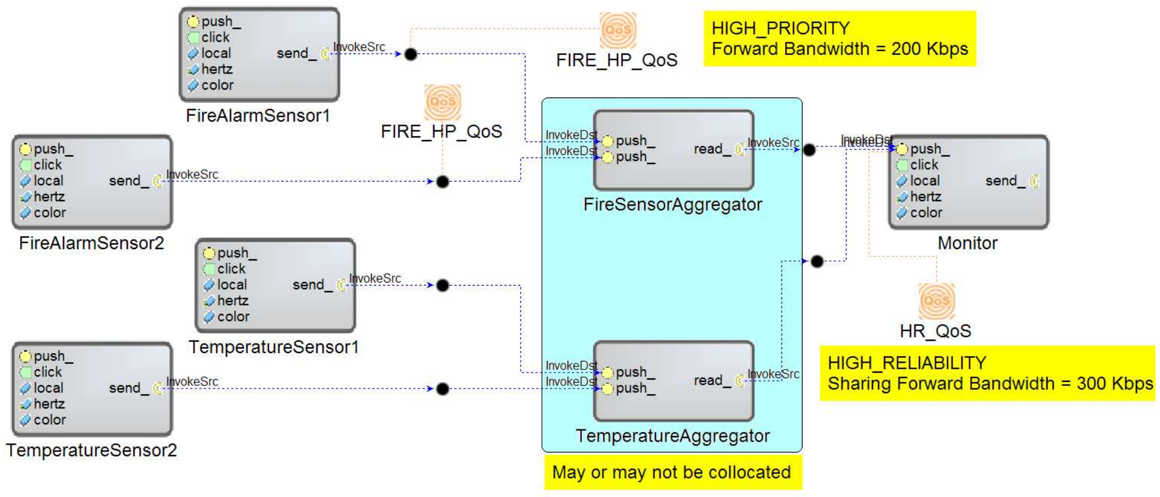


Figure 3. Network QoS Modeling in CQML

of flows to specify is in the order of thousands.

Problem. With thousands of flows, it is infeasible for application developers to manually specify the QoS needs of components and their interactions on a per flow basis. Moreover, QoS specifications are often made on a trial and error basis, *i.e.*, the specification-implementation-(simulation)-testing cycle is commonly repeated many times. A desired feature therefore is an easy to use, fast, declarative, and scalable means of specifying these QoS requirements.

Solution Approach: Domain-specific modeling and generative programming. MDE is a promising approach to capture network QoS requirements for dataflows in a DRE system because it raises the level of the abstraction of system design to a level higher than third-generation programming languages. Modeling network QoS requirements and synthesizing the metadata from the model alleviates many deployment time concerns and eliminates the need for low-level, out-of-band programming. At the heart of our MDE approach to provision network level QoS support for enterprise DRE systems is a platform-specific DSML for LwCCM, called the *Component QoS Modeling Language* (CQML).

Modeling network QoS requirements. CQML allows modelers to annotate the elements modeled with platform-specific details and QoS requirements as shown in Figure 3, in the following manner:

• **Network QoS specification via annotating component connections.** Connections between components and component assemblies in a LwCCM application can be annotated with (1) one of the following network QoS attributes: HIGH PRIORITY (HP), HIGH RELIABILITY (HR), and MULTIMEDIA (MM), and (2) bi-directional bandwidth requirements. The HP class represents the highest importance and lowest latency traffic (*e.g.*, fire and smoke incidence reporting). The HR class represents TCP traffic requiring low packet drop rate (*e.g.*, surveillance data). The MM class involves a large amount of traffic that can tolerate certain amount of packet loss, but requires predictable delays (*e.g.*, video conferencing, news tickers). There is also a default BEST EFFORT (BE) class for applications like email that requires no special QoS treatment.

• **Generation of deployment metadata.** The network QoS requirements must be specified along with other component deployment-specific details (*e.g.*, CPU requirements and connections), so that resource management algorithms can allocate specific hardware nodes to components that can satisfy multiple CPU resource requirements. A CQML model interpreter traverses the multiple views of the system, generates system deployment information, and reconciles network QoS details in the system deployment information, relieving system deployers from these tedious and error-prone tasks.

• **Model scalability in QoS annotations.** Figure 3 specifies how the **FireSensorAggregator** component aggregates fire sensor information from multiple sensors. In our corporate computing environment scenario, there could be multiple fire sensors deployed, where each sensor needs to send the information with the same urgency, and hence the same network QoS. To enable system modelers to associate the same network QoS across multiple connections, we follow a "write once, deploy everywhere" pattern, *i.e.*, we allow modelers to define the QoS (in this case, the property **FIRE_HP_QOS**) once and refer it across multiple connections, as shown in Figure 3. The QoS of additional fire sensors can be captured rapidly, thereby increasing the scalability of the modeling process.

• **Facilitating ease of deployment.** In Figure 3, the **Monitor** component can dictate the bandwidth reservations for all the different aggregator components (*e.g.*, **FireSensorAggregator**) deployed in the corporate environment. For example, the **Monitor** component could receive only upto 300 Kbps of data from all the aggregator components. This bandwidth is shared among the different hosts hosting the aggregator components, as sufficient reservations are needed to send data from the aggregator components to the **Monitor** components. At modeling time, however, the hosts containing the aggregator components are unknown, so system modelers cannot specify the bandwidth reservations for each connection between the aggregator components and the **Monitor** component.

CQML simplifies the modeling of component deployments by defining a *bandwidth pipe* that is shared between the different aggregator components and the **Monitor** component. Information about the *bandwidth pipe* is captured in the deployment metadata using CQML's model interpreter, which then assists the RACE network resource allocator described in Section 3.3 to divide the bandwidth described in the *bandwidth pipe* among the hosts hosting the aggregator components. An optimization is performed in the case of collocated aggregator components. A single pipe of bandwidth is reserved between the host hosting the aggregator components and the host hosting the **Monitor** component.

3.3 Providing Deployment-time Network Resource Reservation and Device Configuration

Context. Challenge 2 in Section 2.2 describes how network QoS provisioning mechanisms keep track of all the available paths and link capacities between any two pair of hosts to provision network resources that satisfy the QoS requirements of component communications. This design allows the network QoS provisioning mechanisms to determine if a network QoS can be provided or not, given the source and destination nodes of communicating components.

Problem. In QoS-enabled component middleware, deployment decisions to identify the hosts where the components will be deployed are usually made using intelligent component placement algorithms [9] based on component CPU resource profiles. The Bandwidth Broker [6, 7] allocates and provisions network resources to satisfy specified QoS on application flows. Network QoS provisioners like Bandwidth Broker must therefore integrate with network allocation and placement algorithms that can retry a component placement decision if the network QoS cannot be met. At deployment time, DRE applications can be provided with required resources at the host and network layer, thereby helping assure end-to-end QoS.

Solution Approach: Network resource allocation planners. Section 3.1.1 described how RACE [31] uses standard LwCCM middleware mechanisms to allocate CPU resources to applications [9] and control enterprise DRE system performance once applications are deployed and running. Since RACE provides mechanisms to plug in a series of resource allocation algorithms, we extended RACE to add a network resource allocator that used the Bandwidth Broker [6].

The resulting NetQoPE framework integrates our QoS-enabled component middleware with network QoS provisioning capabilities that (1) allocate network resources for component communication flows, (2) provide per-flow configurations, specifically for marking/remarking and policing functions at the edges of the network, and (3) communicate with the QoS-enabled component middleware on behalf of the component with desired QoS settings. In particular, NetQoPE's Bandwidth Broker configures CIAO, DAnCE, and RACE with DiffServ codepoints chosen for the various flows originating from the component so that the components and the network elements have the same view of the DSCP markings.

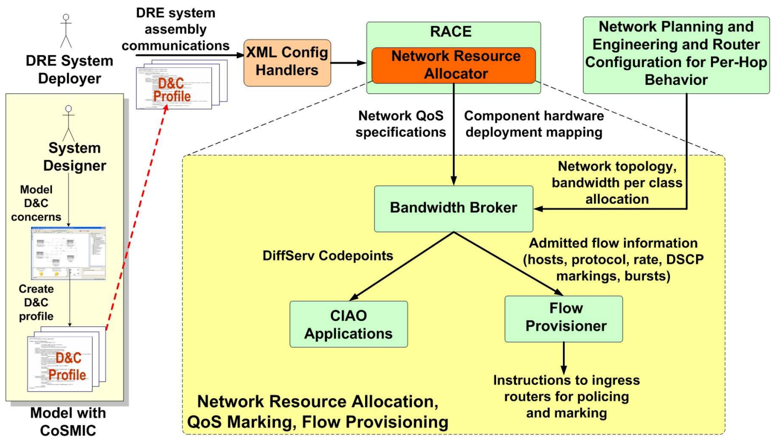

DRE System Deployer
DRE system assembly communications
D&C Profile
XML Config Handlers
RACE
Network Resource Allocator
Network Planning and Engineering and Router Configuration for Per-Hop Behavior
System Designer
Model D&C concerns
Create D&C profile
D&C Profile
Model with CoSMIC
Network QoS specifications
Component hardware deployment mapping
Network topology, bandwidth per class allocation
Bandwidth Broker
DiffServ Codepoints
Admitted flow information (hosts, protocol, rate, DSCP markings, bursts)
CIAO Applications
Flow Provisioner
Instructions to ingress routers for policing and marking
Network Resource Allocation, QoS Marking, Flow Provisioning

Figure 4. NetQoPE's Network Planning and Configuration Capabilities

Bandwidth Broker Integration into RACE Figure 4 illustrates the architecture of the Bandwidth Broker and shows how it is integrated as a RACE *network resource allocator* in NetQoPE. As noted earlier, the Bandwidth Broker leverages widely available mechanisms that support Layer-3 DiffServ (Differentiated Services) and Layer-2 Class of Service (CoS) features in commercial routers and switches. RACE's CPU resource allocators need to determine the hardware nodes onto which the components are deployed, and then check if the determined hardware nodes can also satisfy the network QoS requirements of the various communication flows among components deployed on those hardware nodes. RACE's CPU resource allocator thus makes the hardware node decisions and then invokes the Bandwidth Broker to evaluate the network QoS viability and perform reservations.

The Bandwidth Broker in turn is responsible for assigning the appropriate traffic class to each flow. As shown in Figure 4, it returns a DSCP marking for each flow, which CIAO then uses to mark the IP header of each packet whenever such a flow is initiated by the source component. RACE updates the component deployment metadata to capture the DSCP markings returned by Bandwidth Broker as part of the component connection descriptions. DAnCE can therefore configure the applications with those QoS settings at deployment time.

3.4 Providing Mechanisms to Configure Application QoS Settings

Context: Challenge 3 in Section 2.2 motivates the need to configure low-level network QoS setting in component-based applications. To enhance the reusability of application components, it is important to separate QoS provisioning support from component application logic. A policy framework allows the installation and use of component implementations used under different contexts by configuring the QoS settings of components transparently at runtime using the underlying middleware.

As discussed in Section 3.2 and Section 3.3, the CQML modeling tool allows developers to specify the bandwidth requirements for various components, including forward bandwidth, reverse bandwidth, or both. The RACE network allocator uses these modeled bandwidth requirements to make network resource allocation decisions in the form of QoS settings (*i.e.*, DSCP markings) for various component connections. Components must then be configured with these QoS settings during deployment at various locations throughout an enterprise DRE system.

Problem: QoS-enabled component middleware can be extended to support configuring QoS-related policies of component instances via XML deployment descriptors. As shown in our earlier work on CIAO [33], components in a DRE application can be configured to run with different execution priorities by simply modifying the extended XML deployment descriptors. These CIAO capabilities apply various Real-time CORBA [24] policies. Unfortunately, there is currently no standard CORBA policy for configuring the ORB protocols to set DSCP markings when invoking a request or replying to a client. Moreover, the initial Real-time CORBA capabilities originally supported by CIAO could only configure server-side policies.

Solution approach: CIAO network QoS policy framework. The address the limitations with CIAO, we first extended TAO's Real-time CORBA's policies [30] to support network QoS configurations. We then extended CIAO's real-time extensions to support these new network policies. Similar to Real-time CORBA's priority models, the actual network QoS can be configured on either the client-side or server-side, depending on the usage scenario. For example, some components may dictate how they answer queries, and hence must define the bandwidth they need to receive and service requests. Likewise, clients using those services need to obtain that utilization information before making requests.

To support these scenarios we therefore enhanced TAO to support the following network QoS policy models:

- **Client propagated network policy**, which allows clients to declare their forward and reverse bandwidth requirements by setting their DSCP markings. A TAO server will honor the client-specified reverse bandwidth requirements by using the DSCP markings when sending the replies back to clients.

- **Server declared network policy**, which allows servers to declare their forward and reverse bandwidth capabilities by setting their DSCP marking. Clients will honor the server-declared policy by using the DSCP markings when sending the invocation requests to servers.

Based on the TAO enhancements outlined above, we enhanced CIAO's policy configuration to standard deployment descriptors [25] to parse and set these extended network policies, which are generated by RACE's resource allocator.

Moreover, to accommodate different usage scenarios, CIAO has been enhanced to support configuration of both server-side and client-side policies. Client-side policies are applied when a component invokes methods on other services using policy overrides [30]. For a client to override a server's network policy with its own forward and reverse bandwidth requirements, a client's own network policies must be applied on the object reference pointing to the server.

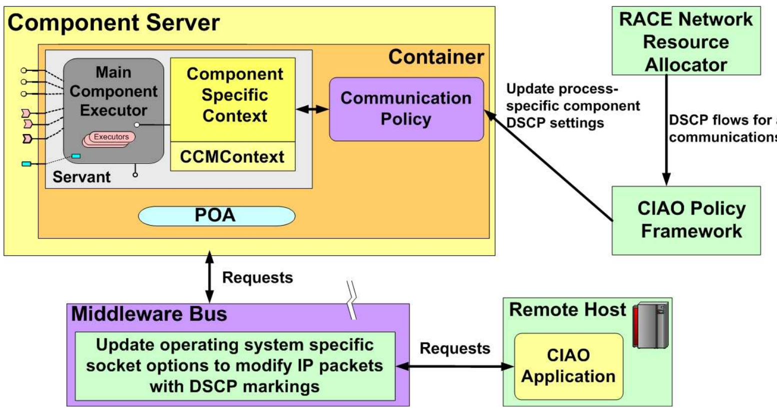

Component Server
Container
Main Component Executor
Executors
Component Specific Context
CCMContext
Servant
Communication Policy
POA
RACE Network Resource Allocator
Update process-specific component DSCP settings
DSCP flows for all communications
CIAO Policy Framework
Requests
Middleware Bus
Update operating system specific socket options to modify IP packets with DSCP markings
Requests
Remote Host
CIAO Application

Figure 5. CIAO Network QoS Policy Framework

In component middleware, object references are shielded from components via a container programming model, which provides this information to components via **Context** objects initialized at deployment time based on the connections a component has. To configure client-side policies, we enhanced the CIAO container programming model as shown in Figure 5 to (1) keep track of the policies associated with each component hosted within the container, (2) automatically override the object references to enforce the appropriate policies when a particular component wants to make a communication, and (3) return the overridden object references to the components using their associated **Context** objects.

4 Empirical Validation of NetQoPE

This section empirically evaluates the network QoS provisioning capabilities provided by the declarative NetQoPE capabilities described in Section 3.

4.1 Hardware/Software Testbed and Experiment Configurations

The empirical evaluation of NetQoPE was conducted at ISISlab (`www.dre.vanderbilt.edu/ISISlab`), which is a testbed for experimenting with enterprise DRE systems consisting of (1) 56 dual-CPU blades (distributed over 4 blade centers each with 14 blades) running 2.8 Gz XEONs with 1 GB memory, 40 GB disks, and 4 NICs per blade, (2) 6 Cisco 3750G switches with 24 10/100/1000 MPS ports per switch that can be provisioned using Network Simulator (NS) scripts to emulate different Layer-3 topologies, and (3) Emulab (www.emulab.net) software to reconfigure experiments via the Web using various versions of Linux, *BSD UNIX, and Windows.

Our experiments were conducted on 16 of those dual CPU blades, with 8 of them running as routers and 8 of them hosting the corporate computing environment scenario applications developed using our NetQoPE QoS-enabled component middleware. All blades ran Fedora Core 4 Linux distribution, which supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. The blades were connected over a 1 Gbps LAN with virtual 100 Mbps links and standard Linux router software was used to configure the policing behavior.

The component middleware infrastructure was based on CIAO version 0.5, which also includes the RACE framework. CoSMIC version 0.4.8 was used to model the network QoS requirements of the corporate environment scenario applications. The benchmarks ran in the POSIX real-time thread scheduling class to enhance the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

Our experiment configuration is shown in Figure 6. Blades *A, B, C, D, E, F, G,* and *H* contain the software components developed using NetQoPE. These components include application software components, specifically software controllers controlling the sensors, monitoring system, video servers, and clients and surveillance data generators. Blade *C* contains RACE and its integrated network resource allocator, which interfaces with Bandwidth Broker and is developed using CIAO. The Bandwidth Broker also runs on blade *C*. Blades *P, Q, R, S, T, U, V,* and *W* contain the router software that is configured by the Bandwidth Broker's Flow Provisioner, which is also hosted on blade *C*.

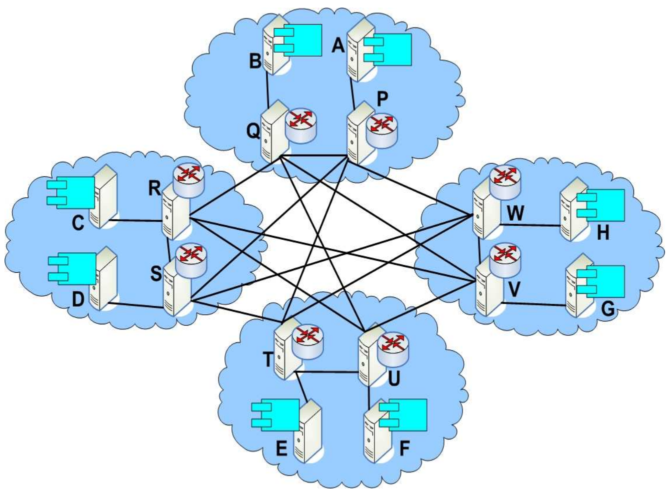


Figure 6. Experimental Setup

Our corporate computing environment case study comprises the following classes of traffic, as outlined in Section 3.2:

- HIGH_PRIORITY (HP), this traffic class represents data flow from the smoke and fire detector sensors to the monitoring subsystem. The essential aspects of a fire or smoke incidence (*e.g.*, location) must be dispatched with utmost importance and quickly, *i.e.*, low latency . The amount of data is typically small.

- HIGH_RELIABILITY (HR), this traffic class represents data flow from surveillance cameras that detect intruders. Since this is a stream of images, the bandwidth requirements are moderate. Moreover, since this image stream is used for security, this class of traffic requires high reliability, *i.e.*, little packet loss. The TCP protocol is used for this class. The latency requirements for this class are not as stringent as those of the HP class.

- MULTIMEDIA (MM), this traffic class represents data flow from applications in the corporate environment scenario that can tolerate some packet losses, but require predictable delays. The UDP protocol is used in this traffic class. Some common applications suited for this class are video conferencing and tickers propagating perishable information, such as latest temperature inside and outside a building.

- BEST_EFFORT (BE) comprises any other network traffic for which no QoS requirements are required.

We configured our network topology described here with the following class capacities on all the links: HP = 20 Mbps, HR = 30 Mbps, and MM = 30 Mbps for all the experiments. The BE class could use the remaining available bandwidth in the network. The HR, MM, and BE classes are supported by a weighted fair queuing discipline at the routers, while the HP class has a priority queue.

Since all network traffic will be sent by software components developed using CIAO, the core benchmarking software was based on the single-threaded version of the "`TestNetQoPE`" performance test distributed with CIAO (`TAO/CIAO/performance-tests/NetQoPE`). This performance test consists of two components, A and B, and creates a session for A to communicate with B. The session permits a configurable number of operation invocations by A on B with a configurable sleep time between the invocations, and a configurable payload sent for each invocation. This test enables us to experiment with different types of traffic by sending different types of payload, thereby varying the bandwidth used for the communications.

We now briefly summarize how the various elements in our tool chain were used to run the experiments:

- **Modeling the QoS requirements** – We used the CQML modeling capabilities described in Section 3.2 to model the QoS requirements of applications whose delivered QoS we were interested in measuring. The generative capabilities of CQML synthesized the metadata comprising all the information on the components, and the QoS needs for the dataflows on the connections. The payload and the number of iterations for each experiment are set as attributes on CIAO components, so they can be configured when the components are initialized. This configuration allows components to send the requisite network traffic for each experiment.

- **RACE planning** – The QoS requirements captured in the metadata generated by the CQML modeling tools are used by the RACE planner to communicate with the Bandwidth Broker, which in turn determines the appropriate DSCP marking for that dataflow. The RACE planner updates this information into the deployment and configuration metadata it generates.

- **CIAO network QoS policy and DAnCE frameworks** – The DAnCE framework then deploys the components according to the metadata generated by RACE. CIAO's policy framework uses the QoS configuration information to mark the packet headers of outgoing communication flows with the DSCP markings captured in the metadata.

We used the `TestNetQoPE` software to generate traffic among several pairs with different QoS (*e.g.*, class, bandwidth amount, transport) at the same time. In the corporate environment case study, for example, when the fire sensor detects a fire and sends the location of the fire to the monitoring subsystem, a video conferencing could be running simultaneously. The network could therefore experience both HP and MM traffic at the same time. These traffic flows also differ in the amount of bandwidth they require and the transport protocol they use.

The overall goals of the experiments were to evaluate the effectiveness of NetQoPE's (1) traffic classes, *i.e.* HP, HR, MM, and BE, (2) admission control mechanisms that ensure there is enough capacity for the flows that have been admitted for ensuring QoS, and (3) underlying network element mechanisms that police a flow for compliance, *i.e.*, not to exceed its allocated bandwidth amount.

4.2 Experiment Results and Analysis

We now describe the experiments performed using the ISISlab configuration described in Section 4.1. The results and the analysis of these results are also given for each experiment.

4.2.1 Experiment 1: Effectiveness of the Traffic Classes Supported

Rationale. The four traffic classes described in Section 4.1 support a judicious mix of importance, latency, jitter and reliability QoS dimensions. Although DiffServ allows 128 different DSCP markings or classes, generally the more the classes, the more the resource consumption and the slower the forwarding behavior. Moreover, there can only be at most eight classes in Layer-2 networks that use CoS. We therefore chose as few classes as needed and still satisfy the needs of many DRE domains, as exemplified by our corporate computing environment case study, as shown below.

This experiment was run with two variants: (1) applications communicate using the TCP transport protocol and (2) applications communicate using the UDP transport protocol. All experiments used high-resolution timer probes to measure the average roundtrip latency or one-way latency.

Methodology for the TCP experiment. `TestNetQoPE` was configured to make synchronous invocations with a payload of 200,000 bytes for 1,000 iterations. The experiments were repeated when the network QoS requested was configured to be one of the HP, HR, MM, and BE traffic classes. In each case, 20 Mbps of network bandwidth was requested for the flow supporting the invocations in that class to the Bandwidth Broker. To evaluate application performance in the presence of background network loads, several other applications were run, as described in Table 1:

The background network traffic generated for our experiments might or might not exist in production enterprise DRE systems, *i.e.*, they represent a worst-case scenario. We wanted to evaluate the performance of the NetQoPE framework,

Application Type	Background Traffi c in Mbps			
	BE	HP	HR	MM
Best Effort Application	85 to 100			
High Priority Application	30 to 40		28 to 33	28 to 33
High Reliability Application	30 to 40	12 to 20	14 to 15	30 to 31
Multimedia Application	30 to 40	12 to 20	14 to 15	30 to 31

Table 1. Background Traffic in the TCP Experiment

however, while operating in such worst-case scenarios, and hence generated such background loads.

Analysis of results. Figure 7 shows the results of the experiments when the deployed LwCCM applications communicated using TCP. The results show that the average latency experienced by the application using the HP network QoS class in the presence of varied background network QoS traffic is much lower than the average latency experienced by the application using the BE network QoS class in the presence of background applications generating BE network QoS class traffic. This result indicates that NetQoPE can ensure an upper bound on latencies experienced by applications even in the presence of applications generating contending network traffic, thereby ensuring much better performance than in a best-effort network. In particular, the HP traffic is given preference over contending traffic from all other classes and the HP traffic limit is well within the link capacity.

The results also show that in the presence of a varied background network QoS traffic, the average latency experienced by the LwCCM application using the HR network QoS class is much lower than the average latency experienced by the application using the MM network QoS class. This result stems from the fact that different per-hop behavior mechanisms are configured for the different network QoS classes. The size of the queues at the routers for the HR network QoS class is higher than the size of the queues at the routers for the MM network QoS class. As a consequence, when network congestion occurs, the queues for the MM network QoS class are more likely to drop packets, which triggers the TCP protocol flow control to reduce its window size by half, thereby causing the application to experience more network delays while making the invocations. As the network delays add up, the average latencies experienced by the application increases, as shown by the MM network QoS class

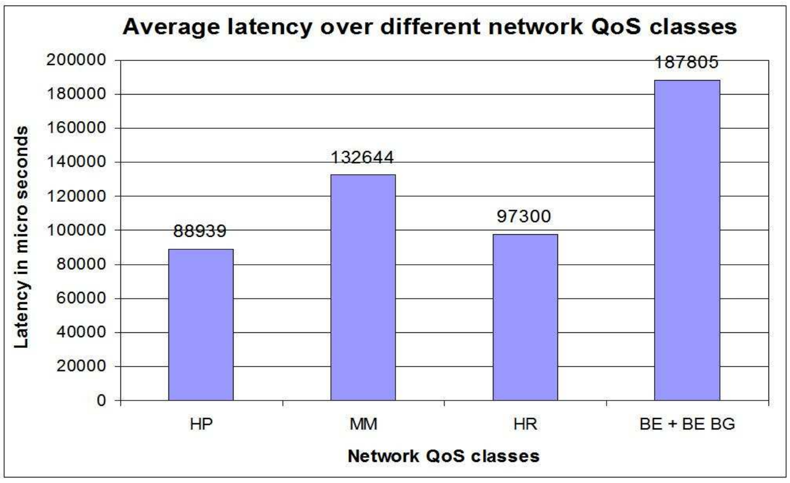


Figure 7. Average Latency under Different Network QoS Classes

in our experiments.

In general, the results of this experiment show that HR network QoS class is more suited for LwCCM applications communicating over TCP. Of course, the HP network QoS class gives the best performance, but not every application can use HP since its bandwidth capacity is relatively small and thus intended for low volume mission-critical traffic.

Methodology for the UDP experiment. `TestNetQoPE` was configured to make *oneway* invocations with a payload of 500 bytes for 100,000 iterations. We used high-resolution timer probes to measure the network delay for each invocation on the receiver side of the communication.

When the sender makes an invocation, it sends a timestamp along with its other invocation arguments to record the local time the invocation was made. When the receiver receives the invocation, it records the difference between its local time stamp and the time stamp that was sent in the invocation. An offset between the sender and the receiver clocks is accounted, and the adjusted difference is recorded as the network delay for the invocation received by the receiver.

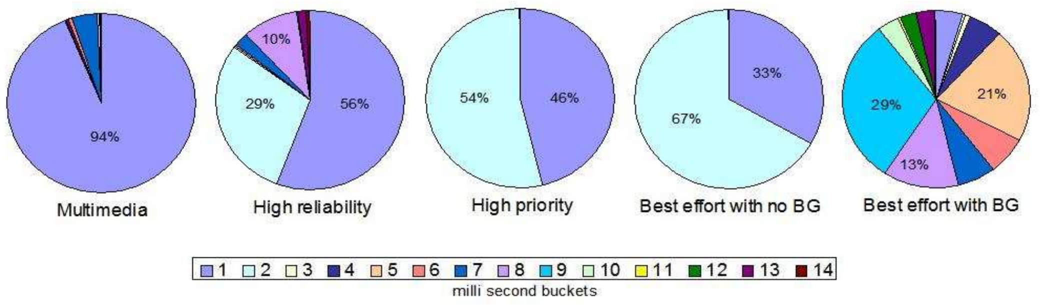
We also counted the number of invocations dispatched at the receiver to track the amount of UDP packet loss. The experiments were repeated using different network QoS classes in the presence of different background network traffic, as described in Table 2:

Application Type	Background Traffi c in Mbps			
	BE	HP	HR	MM
Best Effort Application	60 to 80			
High Priority Application	30 to 40		27 to 28	27 to 28
High Reliability Application	30 to 40	1 to 9		27 to 28
Multimedia Application	30 to 40	1 to 9	27 to 28	

Table 2. Background Traffic in the UDP Experiment

At the end of the experiments, at most 100,000 network delay values (in milliseconds) were recorded for each network QoS class, if there were no invocation losses. Those values were then arranged in increasing order, and every value was subtracted from the minimum value in the whole sample, *i.e.*, they were normalized with respect to the respective class minimum latency. The samples were divided into fourteen buckets based on their resultant values. For example, the 1 millisecond bucket contained only samples that are less than or equal to 1 millisecond in their resultant value and the 2 millisecond bucket contained only samples whose resultant values were less than or equal to 2 millisecond but greater than 1 millisecond and so on.

Analysis of results. Figure 8 shows the cardinality of the network delay groupings for different network QoS classes under different millisecond buckets. The results show that the LwCCM application configured to use HP network QoS class has network delays spread across just two buckets showing the effect of the priority queues at the routers configured for this class. The packets containing the invocations configured with HP class rarely had to wait at the queues, causing the application to have a low and predictable network delay. The results also show that in the case of an application configured to use MM network QoS class, about 94% of the invocations had their normalized network delays within 1 millisecond, which is



Total number of invocations received

Multimedia	**94901**
High reliability	**99998**
High priority	**99991**
Best effort with no BG	**100000**
Best effort with BG	**89924**

Figure 8. Network Delay Distribution Under Different Network QoS Classes

- Better than the network delay groupings recorded for application configured to use BE and HR network QoS class, where the values are spread over several buckets, indicating varied network delays, and

- Comparable to the network delay groupings recorded for application configured to use BE network QoS class in the presence of no background application network QoS traffic.

The latency performance of the application configured to use the MM class is better than the latency performance of the application configured to use the HR class because of the queue sizes configured for these classes. Since the queue size at the routers is smaller for the MM class than the queue size for the HR class, the UDP packets sent by the invocations do not experience as much queuing delays in the core routers as the packets belonging to the HR class.

The MM class traffic experienced more packet losses, however, which in turn means fewer invocations. This tradeoff of lower latency at the cost of higher packet loss may be acceptable—even ideal—in some situations. For example, in our corporate computing environment case study *oneway* invocations for sending out temperature readings may prefer to use this MM class. For such applications, losing a few readings over a period may be insignificant as long as the most recent updates consistently reach the destination with little delay.

4.2.2 Experiment 2: Evaluating NetQoPE Framework's Admission Control Capabilities

Rationale. Section 3.3 motivates the need for admission control capabilities in enterprise DRE systems to ensure QoS. This experiment demonstrates the need for such admission control.

Methodology. `TestNetQoPE` was configured to make synchronous invocations using the TCP transport protocol with a payload of 20,000 bytes for 10,000 iterations using the HR class. We repeated the experiments between a pair of blades with the following pairs of client-server communications: 1, 2, 3, 4, 5, 10, 15, and 20. The experiments had the same background application traffic as described in the UDP experiment in Section 4.2.1. We used the admission control capability of the Bandwidth Broker for 1, 2, 3, and 4 pairs of client-server communication.

In each case, we allocated 6 Mbps for each pair of communication. The allocated capacity for the HR class on each link is 30 Mbps, so the admission control permitted all the requests, including the 4 pair case. For cases of 5, 10, 15, and 20 communicating pairs, we did not use the admission control capability. If the admission control capability were enabled, the Bandwidth Broker would not have admitted more than 5 pairs of communications or flows. Hence, we experimented 5, 10, 15, and 20 pairs of communication without this capability. The background application traffic in each of the network QoS classes was kept sufficiently high so that these HR flows did not get bandwidth from other traffic classes.

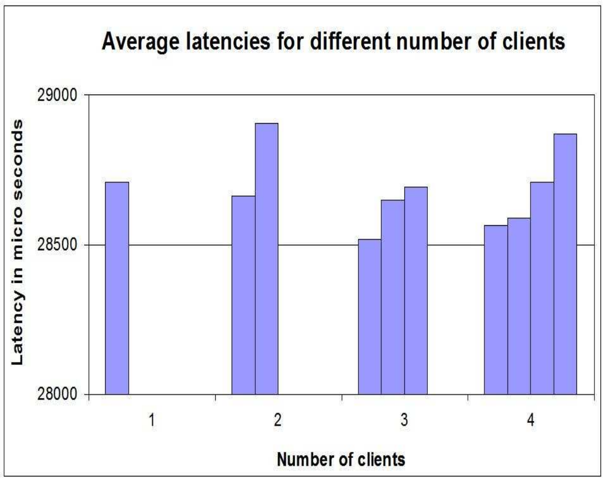


Figure 9. **Average Invocation Latencies with Admission Control**

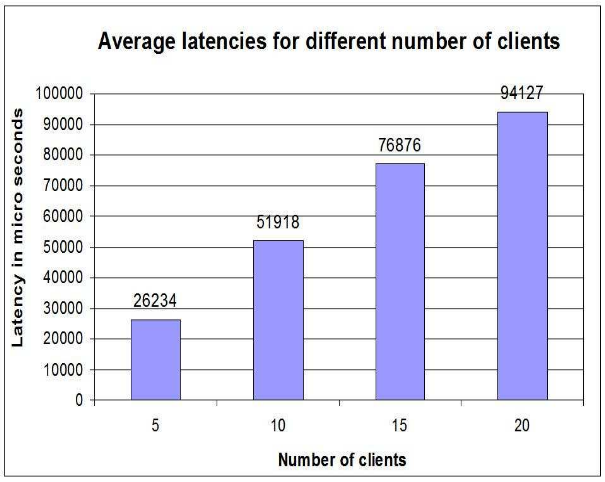


Figure 10. **Average Invocation Latencies Without Admission Control**

Analysis of results. Figure 9 and Figure 10 show the average roundtrip latencies experienced by different numbers of clients, with and without admission control of the Bandwidth Broker, respectively. When the experiments were run *with* admission control, all clients experienced the same average latency, as shown in Figure 9. This result occurred because all clients were allocated 6 Mbps of network bandwidth, and the applications were able to get that bandwidth.

When the experiment was run *without* admission control, all deployed applications shared the 30 Mbps of HR class bandwidth. When 5 communicating pairs were deployed, the available bandwidth was close to 6 Mbps for each of the application. The latency experienced by the clients was therefore almost equal to the latency experienced by the clients using the admission control capability.

As the number of client-server pairs increased to 10, 15, and 20, however, each communication pair bandwidth share started decreasing, causing the pairs to experience higher and higher latencies, as illustrated in Figure 10. These results demonstrate the need for QoS-enabled component middleware to incorporate admission control to ensure predictable performance for the application component communication and hence end-to-end application performance.

4.2.3 Experiment 3: Evaluating NetQoPE Framework's Traffic Policing Capabilities

Rationale. A key basis of NetQoPE's network QoS assurance is that an admitted flow will not routinely exceed its allocated bandwidth. If a flow were to consistently violate its usage of the network resource, one or more flows that share a link with the flow may not be able to get its/their share of the bandwidth. Policing for compliance at the granularity of a flow at the ingress router is therefore required, and modern routers and switches support this feature. Experiment 3 validates this policing feature and demonstrates its usefulness in the context of NetQoPE and our corporate computing environment.

Methodology. `TestNetQoPE` was configured to make synchronous invocations using the TCP transport protocol with a payload of 20,000 bytes for 10,000 iterations using the HR class. We repeated the experiments with the following bandwidth reservation requests for the flow carrying the invocations: 20 Mbps, 16 Mbps, 12 Mbps, 8 Mbps, and 4 Mbps. The experiments ran with the same background application network traffic described in the UDP experiment from Section 4.2.1.

Analysis of results. Figure 11 shows the average latency experienced by the clients for the different bandwidth requests. The results show that the average latency increases linearly as the requested/reserved bandwidth amount decreases (the ingress router for the flow was provisioned to police at the requested bandwidth rate). Since the experiments were run to generate the same network load regardless of the bandwidth reserved, the lower the reserved amount, the sooner (or more certain) that the policing functionality will start drop packets of the flow.

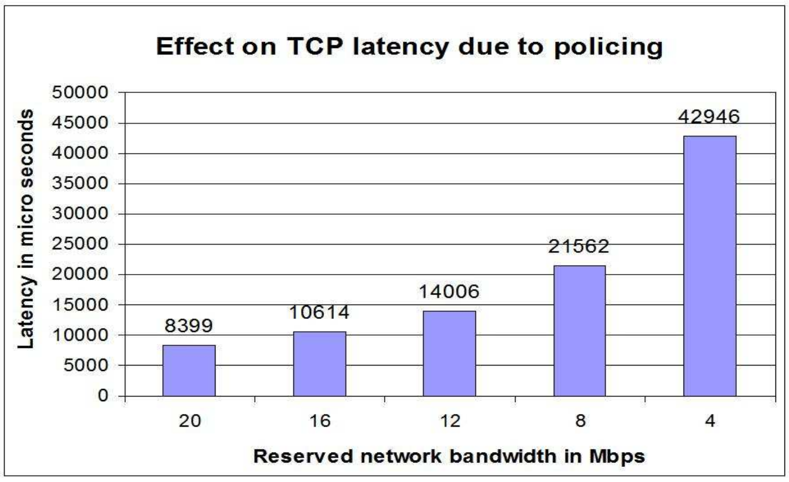


Figure 11. Effect on Invocation Latency due to Policing

When packets loss is detected, the TCP protocol at the sending end halves the transmission window size, thereby causing application to experience more network delay than before (since the window size was halved) while making the invocations, *i.e.*, lower the bandwidth size, the higher the delay. The policing feature, however, is independent of the protocol used, although each protocol may adjust differently when a packet loss is detected, *i.e.*, when the policing function is triggered.

These experiments clearly show that applications configured with a specific network bandwidth reservation do not exceed this limit, due to the Bandwidth Broker's policing feature. This feature combined with NetQoPE's admission control capability ensures end-to-end QoS for all the admitted flows.

5 Related Work

The general problem of QoS implementation, management, and enforcement has been the focus of many research projects in recent years. There has been relatively little work, however, on the specific problems addressed in this paper, namely, the integration of (1) model-driven engineering to specify network QoS for communication flows in DRE systems, (2) subsequent automated allocation and provisioning of network resources during deployment, and (3) automated component middleware runtime support to effect appropriate QoS for applications in a transparent and portable manner. This section compares our R&D activities on NetQoPE with several areas of related work.

Network QoS management in middleware. Earlier work [26, 34] on integrating network QoS with middleware focused on the integration of priority and reservation-based OS and network QoS management using IntServ with standards-based middleware, such as Real-time CORBA, to provide end-to-end QoS for DRE systems. In IntServ every router on the path of a requested flow decides whether or not to admit the flow with a given QoS requirement. Each router in the network keeps the status of all flows that it has admitted as well as the remaining available (uncommitted) bandwidth on its links. Some well-known drawbacks with IntServ are that (1) it requires per-flow state at each router, which can be an issue from a scalability perspective, (2) its admission decisions are based on local information rather than on an adaptive, network-wide policy, and (3) it is applicable only to Layer-3 IP networks. In contrast, NetQoPE does not have these drawbacks, as described in Section 3.3.

Telcordia has successfully applied Bandwidth Broker technologies in other settings [3, 17]. Neither of these two prior endeavors, however, deal with layer-2 QoS. Moreover, this earlier work did not focus on integration with network resource management using a model-driven QoS-enabled component middleware framework, as we have done by integrating and enhancing PICML, CIAO, DAnCE, and RACE to create NetQoPE.

QoS management in component middleware. Other research has focused on adding various types of QoS capabilities to component middleware. For example, [14] illustrates mechanisms added to J2EE containers to allow application isolations by allowing uniform and predictable access to the CPU resources, thereby providing CPU QoS to applications. Likewise, [5] shows the use of aspect-oriented techniques to plug-in different non-functional behaviors into containers, so that QoS aspects can be statically linked to applications. In addition, [8] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive QoS support. NetQoPE differs from this related work by providing (1) network QoS to component middleware applications, (2) scalable techniques to specify QoS requirements, (3) automated deployment time mechanisms to work with network QoS mechanisms to do QoS negotiations, and (4) runtime capabilities to easily integrate network QoS settings into application communication flows.

QoS management in multimedia applications. Research has produced QoS architectures and models for real-time multimedia applications, that demand predictable QoS from both endsystem and network resources. For example, AQUA (Adaptive QUality of service Architecture) [15] is a resource management architecture, at the endsystem level, in which the applications and OS cooperate to dynamically adapt to variation is resource availability and requirements. Likewise, [22] describes a QoS broker that is used to do QoS negotiation for providing end-to-end QoS management for multimedia applications. In addition, [11, 13, 21, 29] have considered the co-reservation of multiple resource types for providing QoS to real-time and distributed multimedia and high-end applications. Our work on NetQoPE framework differs by providing (1) network QoS provisioning and enforcing techniques that are not specific to specific resource types like multimedia resources, (2) a model-based advance reservation specification tool at per-flow level, and (3) runtime capabilities to easily integrate and deploy network QoS settings into application communication flows.

QoS-aware deployment of applications. Our work on NetQoPE is also related to other recently proposed approaches for QoS-aware deployment of applications in heterogeneous and dynamically changing distributed environments. For example, GARA [12] and Darwin [4] focus on identifying and reserving appropriate network resources to satisfy application requirements. Likewise, Petstore [20] describes how the service usage patterns of J2EE-based web applications can be analyzed to decide on their deployments across wide-area environments, so that certain client requirements on faster access times can be satisfied. In addition, [32] focuses on improving J2EE performance by collocating applications that communicate heavily with one another. NetQoPE differs from this other work by utilizing popular network architectures, such as DiffServ, to provide network QoS assurance for applications, rather than trying to collocate them or understand their service usage patterns. NetQoPE can also support the deployment of applications by considering multiple resources, including network and CPU by adding planning algorithms in RACE, which can then sequentially or combinatorially generate deployment decisions for component-based DRE applications.

6 Concluding Remarks

This paper describes the design and implementation of NetQoPE, which is a model-driven middleware framework that handles network QoS management and enforcement for component-based DRE systems. This work extends our prior work [26] by integrating Telcordia's Bandwidth Broker [6, 7] with Vanderbilt University DOC Group's QoS-enabled component middleware framework [33]. The Bandwidth Broker is a network QoS management system that leverages DiffServ/CoS features that are common in today's routers and switches. Our experiments systematically validated NetQoPE's end-to-end QoS assurance capabilities in standards-based DRE systems.

The following is a summary of our lessons learned from this project:

- Standard Lightweight CORBA Component Middleware (LwCCM) interfaces can be extended with relative ease to develop a scalable and flexible middleware infrastructure that can support marking IP header with the right DiffServ codepoint determined by the Bandwidth Broker in an OS-independent, portable, and application-transparent manner, *i.e.*, application developers are shielded from working with low-level network interfaces. The resulting QoS-enabled component middleware facilitates QoS assurance to software deployed to multiple deployment targets and network environments. This effectively improves both software reuse and QoS.

- NetQoPE makes network provisioning decisions in coordination with component placement algorithms that consider host resources, specifically CPU. This coordination is a significant step in systems oriented QoS R&D, since it handles two key resources, but it falls short of true allocation integration. Integrated allocation problems that take into account both CPU and network resources are integer programming problems that are less tractable than bin-packing problem formulations for the CPU resource alone. In our RACE-based integration, we judiciously combined tractable placement solutions that consider only the CPU resource with the network resource availability checking by the Bandwidth Broker. Our solution is by no means optimal, but it is practical.

An area of focus for our future work is on continued QoS guarantees in the case of network faults. NetQoPE assumes that the properties of network links where the component-based DRE system is deployed remain fixed during the system lifetime. In reality, of course, network links do fail. To cope with this, NetQoPE must be enhanced with capabilities to model, monitor and adapt to network failures. The Bandwidth Broker incorporates mechanisms to adapt dynamically to network faults. When a fault is detected one or more admitted flows take new paths to circumvent the link or router that has failed. This, in turn, could overload some of the links, causing QoS contracts of some previously admitted flows to be missed. In such cases, the Bandwidth Broker bases its re-admission decision on the priority of the flows. NetQoPE does not currently model classifying applications or classifying communication flows based on priorities. Classifying applications based on priorities will allow NetQoPE to give preferential treatment to applications during admission as well as re-admission or reconfiguration, so that higher priority applications receive resource allocation in the face of reduced network capacity.

Dropping flows even of low priority may not always be preferable. Our future work with NetQoPE will include developing selection algorithms [35] that can automatically choose the most suitable component implementation to operate in the case of network resource scarcity. A selection algorithm, for instance, may substitute a less communication-intensive algorithm when it detects a network fault. We will implement these selection algorithms using NetQoPE's integrated QoS capabilities and validate them in the context of a DRE system, such as the office of the future environment presented in this paper.

References

[1] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A platform-independent component modeling language for distributed real-time and embedded systems. *Elsevier Journal of Computer and System Sciences*, 2006. To appear.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *Internet Society, Network Working Group RFC 2475*, pages 1–36, Dec. 1998.

[3] R. Chadha, Y.-H. Cheng, T. Cheng, S. Gadgil, A. Hafi d, K. Kim, G. Levin, N. Natarajan, K. Parmeswaran, A. Poylisher, and J. Unger. Pecan: Policy-enabled confi guration across networks. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 52, Washington, DC, USA, 2003. IEEE Computer Society.

[4] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable resource management for value-added network services. *IEEE Network*, 15(1):22–35., 2001.

[5] D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel. Integration of Non-Functional Properties in Containers. *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[6] B. Dasarathy, S. Gadgil, R. Vaidhyanathan, K. Parmeswaran, B. Coan, M. Conarty, and V. Bhanot. Network QoS Assurance in a Multi-Layer Adaptive Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework. In *Proceedings of the IEEE Real-time Technology and Applications Symposium (RTAS)*, San Francisco, CA, Mar. 2005. IEEE.

[7] B. Dasarathy, S. Gadgil, R. Vaidyanathan, A. Neidhardt, B. Coan, K. Parameswaran, A. McIntosh, and F. Porter. Adaptive network qos in layer-3/layer-2 networks for mission-critical applications as a middleware service. *Journal of Systems and Software: special issue on Dynamic Resource Management in Distributed Real-time Systems*, 2006.

[8] M. A. de Miguel. Integration of QoS Facilities into Component Container Architectures. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, 2002.

[9] D. de Niz and R. Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems*, 2005.

[10] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale. DAnCE: A QoS-enabled Component Deployment and Confi guration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment*, Grenoble, France, Nov. 2005.

[11] I. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, Sept. 2004.

[12] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service (IWQOS'99)*, London, UK, May 1999.

[13] G. Hoo, W. Johnston, I. Foster, and A. Roy. QoS as middleware: Bandwidth broker system design. Technical report, LBNL, 1999.

[14] M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner. Extending a j2ee server with dynamic and fexible resource management. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2004), Toronto, Canada*, pages 439–458, 2004.

[15] R. F. K. Lakshman, Raj Yavatkar. Integrated CPU and Network-I/O QoS Management in an Endsystem. In *Proceedings of the IFIP Fifth International Workshop on Quality of Service (IWQoS '97)*, 1997.

[16] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[17] B. Kim and I. Sebuktekin. An integrated ip qos architecture, performance. *Proceedings of the IEEE MILCOM Conference (MILCOM 2002)*, Oct. 2002.

[18] L. Zhang and S. Berson and S. Herzog and S. Jamin. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specifi cation. *Network Working Group RFC 2205*, pages 1–112, Sept. 1997.

[19] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specifi c Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[20] D. Llambiri, A. Totok, and V. Karamcheti. Effi ciently distributing component-based applications across wide-area environments. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 412, Washington, DC, USA, 2003. IEEE Computer Society.

[21] A. Mehra, A. Indiresan, and K. G. Shin. Structuring Communication Software for Quality-of-Service Guarantees. *IEEE Transactions on Software Engineering*, 23(10):616–634, Oct. 1997.

[22] K. Nahrstedt and J. Smith. The QoS Broker. *IEEE Multimedia Magazine*, pages 53–67, Spring 1995.

[23] Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, Nov. 2002.

[24] Object Management Group. *Real-time CORBA Specifi cation*, OMG Document formal/05-01-04 edition, Aug. 2002.

[25] Object Management Group. *Deployment and Confi guration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.

[26] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali. Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. In *Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms*, Rio de Janeiro, Brazil, June 2003. IFIP/ACM/USENIX.

[27] T. Ritter, M. Born, T. Unterschütz, and T. Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the 36^{th} Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, Honolulu, HW, Jan. 2003. HICSS.

[28] W. Roll. Towards Model-Based and CCM-Based Applications for Real-time Systems. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*. IEEE/IFIP, May 2003.

[29] V. Sander, W. A. Adamson, I. Foster, and A. Roy. End-to-end provision of policy information for network qos. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 115, Washington, DC, USA, 2001. IEEE Computer Society.

[30] D. C. Schmidt and S. Vinoski. An Overview of the CORBA Messaging Quality of Service Framework. *C++ Report*, 12(3), Mar. 2000.

[31] N. Shankaran, J. Balasubramanian, D. C. Schmidt, G. Biswas, P. Lardieri, E. Mulholland, and T. Damiano. A Framework for (Re)Deploying Components in Distributed Real-time and Embedded Systems. In *Poster paper in the Dependable and Adaptive Distributed Systems Track of the 21st ACM Symposium on Applied Computing*, Dijon, France, Apr. 2005.

[32] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proc. 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005), Boston, MA*, 2005.

[33] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Confi guring Real-time Aspects in Component Middleware. In *Lecture Notes in Computer Science: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, Oct. 2004. Springer-Verlag.

[34] P. Wang, Y. Yemini, D. Florissi, and J. Zinky. A Distributed Resource Controller for QoS Applications. In *Proceedings of the Network Operations and Management Symposium (NOMS 2000)*. IEEE/IFIP, Apr. 2000.

[35] D. M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, 42(1), 2003.