NetQoPE: Middleware-based Network QoS Provisioning Engine for 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 network-centric, distributed, real-time and embedded (DRE) systems where applications require ensured network QoS even when network resources are scarce or congestion is present. Traditional approaches to building network-based, QoS-sensitive applications, such as Internet telephony and streaming multimedia applications, have relied on leveraging 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.

Our work models and provisions network QoS in QoS-enabled component middleware for DRE systems, extending our previous work that predominantly used host resources to provision end-system QoS. By integrating a Bandwidth Broker (BB) that provides network QoS into our DRE middleware, we enable end-to-end QoS assurance. We describe a declarative QoS provisioning solution called NetQoPE. Its modeling capabilities allow a user to specify application QoS requirements in terms of network and computational needs. The model, in turn, leads to deployment decisions that take into account network resources (in addition to host resources). Finally, our DRE middleware provisions network and middleware elements to enable and enforce QoS decisions.

We demonstrate and evaluate the effectiveness of NetQoPE in the context of a representative DRE system – a modern enterprise environment 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 underlying network subsystem.

1 Introduction

Distributed real-time and embedded (DRE) platforms are at the core of many mission critical systems such as shipboard computing, avionics, intelligence, surveillance and reconnaissance. 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.

QoS-enabled component middleware, such as CIAO [40], and Qedo [32], are increasingly used to develop and deploy next-generation DRE systems. QoS-enabled component middleware leverages conventional component middleware (e.g.,

*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

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 programmatic approaches to the lifecycle management activities, such as assembly, configuration and deployment. QoS-enabled component middleware platforms overcome shortcomings of conventional component middleware by explicitly separating QoS provisioning aspects of applications from their functionality, thereby yielding DRE systems that are less brittle and costly to develop, maintain, and extend [17, 40].

Advances in QoS-enabled component middleware to date are, however, limited to managing CPU and memory resources in the operating system. Although there exist standard mechanisms to support network QoS including integrated services (IntServ) [23] and differentiated services (DiffServ) [3], there are limited or no capabilities in current 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, specifically 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 get their required treatment.

To overcome these limitations, our QoS-enabled component middleware includes mechanisms to: (1) declaratively specify network QoS requirements (in terms of priority, reliability, bandwidth required, latency and jitter) of different communication flows in a DRE system, (2) assure network resources for communications within a DRE system based on the specified QoS, especially for critical communications, (3) automatically mark application packets for classification (with DSCPs) at the entrance to the network, and (4) police for usage compliance at the ingress points of the network so that QoS is assured for all admitted traffic.

Earlier work of the authors at Vanderbilt University on a declarative QoS provisioning framework has focused on developing the following capabilities: (1) a model-driven engineering (MDE) [34] tool suite called CoSMIC [16], which alleviates many accidental complexities associated with developing, deploying and configuring QoS-enabled component-based DRE systems, (2) a deployment and configuration engine (DAnCE) [11] that provides standards-based deployment and configuration mechanisms to deploy systems for component middleware platforms, such as our CIAO [41] lightweight CORBA Component Model (LwCCM) [28] implementation, (3) a dynamic resource allocation and control engine called RACE [37] to allocate and control resources for DRE applications, and (4) reflective middleware techniques within CIAO to support runtime adaptive CPU and memory QoS management [42].

This paper describes how we addressed the aforementioned limitations of QoS-enabled component middleware to provide network QoS management and control by extending our standardized QoS-enabled provisioning framework. Our main contributions are as follows:

- We extend the CoSMIC tool suite to enable modeling of network QoS requirements of DRE applications. The QoS modeling is seamlessly integrated with existing capabilities for specifying the interfaces, communication and assembly details of the DRE system.

- We extend RACE to integrate with the Bandwidth Broker (BB) [7,8] developed by the Telcordia team. Bandwidth Broker assures network resources for flows by using an admission control process that tracks network capacity against bandwidth commitments. Its Flow Provisioner provisions ingress routers to police traffic at the level of a flow.

- We extend the CIAO and DAnCE frameworks to allow configuring application communications with network QoS markings determined by the Bandwidth Broker so that the applications and network elements have the same view on the importance of a communication flow.

Although much of this network QoS technology is developed and validated for the DARPA ARMS program for shipboard computing, we will illustrate the technology using an easily understood commercial application: a modern office/enterprise network environment described in Section 2.1. The remainder of the paper is organized as follows: Section 2 uses a representative DRE system case study as a guide to describe the challenges in provisioning network QoS for DRE systems; Section 3 describes the novel solutions we have developed in the context of our existing QoS provisioning framework; Section 4 provides experimental validation of our approach; Section 5 compares our work with related research; and Section 6 provides 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 DRE system case study, (2) the key network traffic and QoS requirements in the DRE system case study, and (3) the challenges in provisioning the network QoS requirements for such a DRE system using QoS-enabled component middleware.

2.1 DRE System Case Study

Figure 1 illustrates a representative DRE system case study, that involves a modern office/enterprise. The enterprise consists of data centers hosting the computing facilities, desktops throughout the enterprise accessing data and compute cycles in the computing facilities, videoconferencing facilities throughout the enterprise, laboratories hosting expensive and sensitive instruments, and 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 office premises for detecting intruders and unauthorized accesses. All these sensors are connected to the office network where the data is streamed across potentially multiple but a small 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 office network. These monitoring systems are assumed to 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 also implemented as CIAO [40] components. There is essentially 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 such as Fast Ethernet that is IP-aware, for instance as supported by CISCO 6500 switches.

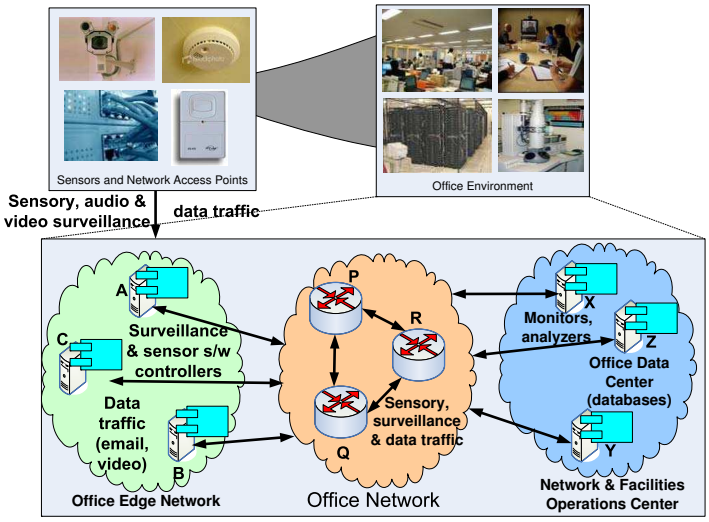


Figure 1. An Office Environment and Network Setup

2.1.1 Network Traffic and QoS Requirements in Office Scenario

The traffic across the office network is diverse and has specific QoS requirements. For example, the traffic between sensors that monitor factors like smoke or fire send periodic events of small or moderate sized data and require low and predictable latencies. On the other hand, video surveillance cameras send a constant stream of images back to the monitoring system. These data streams consume more bandwidth and while latencies 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 DRE applications using UDP require appropriate latency, jitter and bandwidth guarantees though they may tolerate some losses. These requirements demonstrate the need for differentiated QoS over different transport protocol for different applications.

The sensor traffic also has different importance levels. For example, a fire or smoke alarm will have higher priority over sensors that track air conditioning temperature control within the office environment. Thus, certain traffic flows within the office network need to take preference over others.

The office scenario described here presents a wide variety of characteristics that are representative of many DRE applications, specifically from the network QoS requirements perspective. The DRE applications demand different degrees of reliability, timeliness/latency, jitter and importance/priority.[1] Even within a process, these characteristics typically vary for

[1] Although higher priority translates to lower latency, to guarantee a specific latency, the network utilization has to be kept very low, i.e., priority and

different flows, as illustrated here for the office case study.

2.2 Challenges Provisioning Network QoS in QoS-enabled Component Middleware

As shown in Figure 1, a modern office environment could have many different data flows between the same or different sets of components, and those communications could have different network QoS requirements. To support this scenario, the modern office 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 to be honored for the inter-component communications e.g., bandwidth amount, desired latency, and priority. For example, different flows within the office 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 the dataflows involved in the system. Section 3.2 describes how NetQoPE framework 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 to be deployed onto, (2) determining how much bandwidth is available in every network connection between the hardware nodes hosting the application components, (3) committing network resources to be allocated 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 framework 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, and the edge routers are configured for appropriate policing and mark down behaviors for the flows, the application communications need to be made with appropriate QoS settings so that the 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 ORB middleware. Hence middleware must provide mechanisms for interacting with applications, to configure them with appropriate QoS settings, so that such QoS settings can be carried onto to the underlying network subsystem, which can then provision network QoS for those applications. Section 3.4 describes how NetQoPE framework supports this requirement.

latency are non-overlapping dimensions of network QoS.

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 DRE systems. First, we provide an overview of our existing QoS provisioning framework. Next we describe 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 R&D, which we leverage.

3.1.1 Model-driven Component Middleware

The DOC group at Vanderbilt University has developed a framework for the developmental and operational lifecycle management of distributed, real-time and embedded (DRE) systems. Our solution illustrated in Figure 2 is an open source tool chain[2] that uses model-driven engineering (MDE) [34] tools to capture application structural and behavioral characteristics converting it into metadata that are used by resource planners to determine resource allocations and component deployments. This information is then used by a deployment and configuration tool to host the application components in a QoS-enabled component middleware framework. We briefly describe individual artifacts of our existing framework below. Subsequent sections describe how we have enhanced each element of this tool chain to realize the network QoS provisioning goals.

CoSMIC Model-Driven Engineering Tool-suite. To simplify the development of component-based applications, we have developed the *Component Synthesis with Model Integrated Computing* (CoSMIC), which is an open-source set of model-driven engineering (MDE) tools that support the deployment, configuration, and validation of component-based DRE systems. A key capability supported by CoSMIC is the definition and implementation of *domain-specific modeling languages* (DSMLs) [14], which use concrete and abstract syntax to define the concepts, relationships, and constraints used to express domain entities [19].

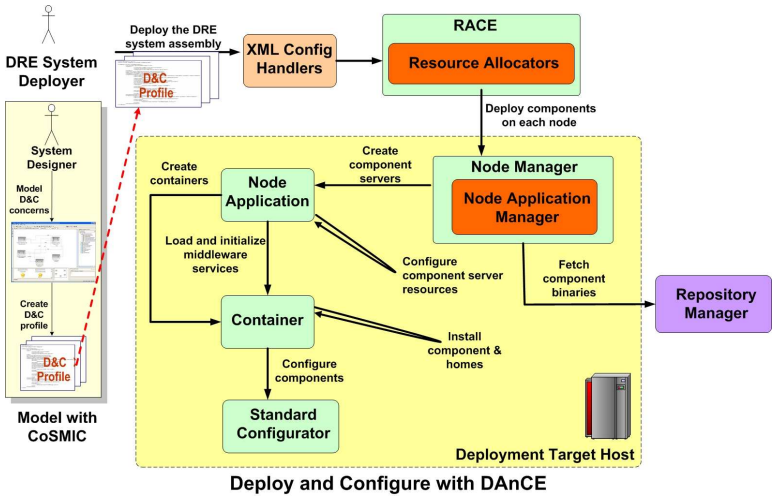


Figure 2. Model-driven Component Middleware Framework

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

A key CoSMIC DSML called *Platform Independent Component Modeling Language* (PICML) [2] enables graphical manipulation of modeling elements and performs various types of generative actions, such as synthesizing XML-based deployment plan descriptors defined in the OMG Deployment and Configuration (D&C) specification [30]. CoSMIC has been developed using a DSML metaprogramming environment called Generic Modeling Environment (GME) [24].

Resource and Control Engine (RACE). The Resource and Control Engine (RACE) [37] is an adaptive resource management framework that provides (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) *configurator* 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.

Deployment and Configuration Engine (DAnCE). DRE system application component assemblies developed using CIAO are deployed and configured via DAnCE [11], which implements the OMG *Deployment and Configuration* (D&C) specification [30]. DAnCE manages the mapping of DRE application components onto nodes in the target environment. The information about the component assemblies and the target environment in which the components will be deployed are captured in the form of standard XML assembly descriptors and deployment plans generated by CoSMIC tools and enhanced by RACE. DAnCE's runtime framework parses these descriptors to extract connection and deployment information, deploys the assemblies onto the CIAO component middleware platform described below, and establishes the connections between component ports.

Component Integrated ACE ORB (CIAO). CIAO is an open-source implementation of the OMG Lightweight CORBA Component Model (LwCCM) [28] and Real-time CORBA [29] specifications built atop TAO [35], which is our real-time CORBA ORB. CIAO's architecture is designed based on (1) patterns for composing component-based middleware [39] and (2) reflective middleware techniques to enable mechanisms within the component-based middleware to support different QoS aspects [40].

3.1.2 Network QoS Provisioning Tools

The Telcordia authors have developed a network management solution for QoS provisioning called the Bandwidth Broker (BB) [7]. The BB leverages widely available vendor mechanisms that support layer-3 DiffServ (Differentiated Services) and layer-2 CoS (Class of Service) features in commercial routers and switches. These two features by themselves are insufficient to guarantee end-to-end network QoS because the traffic presented to the network must be made to match the network capacity. What is also needed 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 accomodated.

The BB provides these capabilities. In Layer-3 network, it is the shortest path that is used for communication between any two hosts. We employ Djkstra's allpair shortest path algorithms. We employ Djkstra's all pair shortest path algorithms to discover all the paths between all pairs of hosts. In Layer-2 network, we discover the VLAN tree to find the path between any two hosts.

The BB provides the following mechanisms:

- **Flow Admission Functions –** to reserve, commit, modify, and delete flows.
- **Queries –** for bandwidth availability in different classes among pairs of pools and subnets.
- **Bandwidth Allocation Policy Changes –** to support mission-mode changes.
- **Feedback/Monitoring Services –** for 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 it is also based on delay bounds requested for the flow. The delay bounds for new flows need to be guaranteed without damaging the delay guarantees 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 guarantees [8].

3.2 Providing Mechanisms for Specifying Network QoS Requirements

Context: Section 2.2 described the need for gathering the network QoS requirements of all the dataflows in a DRE system to make appropriate network resource reservations for those flows. DRE systems of interest to us are moderate to large scale, and number of flows to be specified is in the order of thousands.

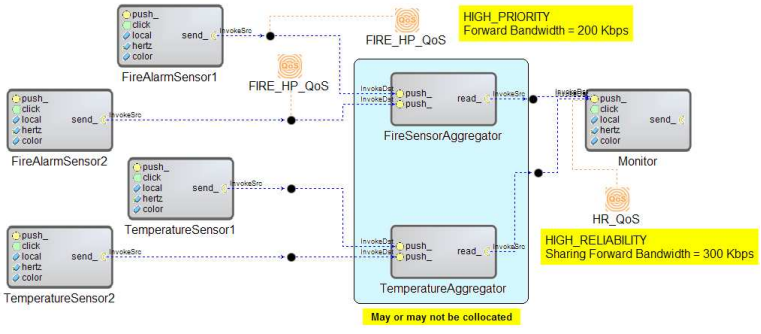


Figure 3. Network QoS Modeling in CQML

Problem: It is infeasible for application developers to programmatically or manually specify the QoS needs of components and their interactions on a per flow basis. Moreover, the QoS specifications are often made on a trial and error basis, i.e., the

specification-implementation (simulation)-testing cycle has to be repeated several times. A desired feature therefore is an easy to use, faster, declarative and scalable means of specifying these QoS requirements.

Solution Approach: Domain-specific Modeling and Generative Programming. Model-Driven Engineering (MDE) is a promising approach to specify and gather 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 as well as eliminates the need for low level and potentially out of band programming. At the heart of our MDE approach to provision network level QoS support for DRE systems is a platform-specific DSML for LwCCM, called the Component QoS Modeling Language (CQML).

Modeling network QoS requirements: CQML allows a modeler to annotate the elements modeled in CQML 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 large amount of traffic which can tolerate certain amount of packet loss, but require predictable delays (e.g video conferencing, news tickers). In addition there is a default BEST EFFORT (BE) class which requires no QoS treatment (e.g., email).

- **Generation of deployment metadata.** The network QoS requirements have to be specified along with other component deployment-specific details (e.g. CPU requirements, 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 the system deployers from the 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 modern office environment scenario, there could be multiple fire sensors deployed, and each of the sensors need to send the information with the same urgency, and hence the same network QoS. In order to facilitate the system modeler to associate the same network QoS across multiple connections, we follow the principle of ”write once, deploy everywhere”. We allow the modeler to define the QoS (in this case, the property **FIRE_HP_QOS**) once and refer it across multiple connections as shown in Figure 3. In the future, any

number of fire sensors could be added, and the QoS can be rapidly added, thereby increasing the scalability of the modeling process.

- **Facilitating ease of deployment.** In Figure 3, the **Monitor** component could dictate the bandwidth reservations for all the different aggregator components (e.g **FireSensorAggregator**) deployed in the modern office 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 need to be made for sending data from the aggregator components to the **Monitor** components. But at modeling time, the hosts hosting the aggregator components are not known, and hence the system modeler cannot specify the bandwidth reservations for each and every connection between the aggregator components and the **Monitor** component. CQML facilitates the ease of modeling such deployments by defining the *bandwidth pipe* to be shared between the different aggregator components and the **Monitor** component. The information about the *bandwidth pipe* is captured in the deployment metadata using the 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.

3.3 Providing Deployment-time Network Resource Reservation and Device Configuration

Context: 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 to satisfy the QoS requirements of component communications. This 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 [10, 22], based on details including component CPU resource profiles. Bandwidth Broker (BB) [7] provides with mechanisms to configure the network resources including switches and routers, to provision network QoS to applications. Hence network QoS provisioners like Bandwidth Broker must work with component placement algorithms to provide a well-integrated network allocation and placement algorithm that provides an opportunity to retry a component placement decision if the network QoS cannot be met with a particular placement decision. At deployment time, DRE applications can therefore be provided with required resources at the host and network layer, so that application end-to-end QoS can be assured.

Solution Approach: Network resource allocation planners. Section 3.1.1 described how RACE [37] uses standard component middleware mechanisms to allocate CPU resources to applications [22] and control DRE system performance once applications are deployed and running. Since RACE provides mechanisms to plug in a series of resource alloca-

tion algorithms, we extended RACE to add a network resource allocator that utilizes the Bandwidth Broker (BB) [7]. This provides our QoS-enabled component middleware framework 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, (3) communicate with the QoS-enabled component middleware on behalf of the component with desired QoS settings, specifically DSCP 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.

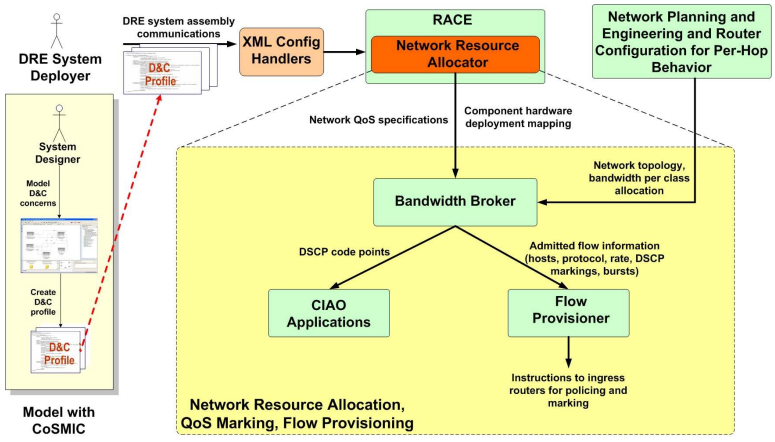


Figure 4. RACE Network Planning and Configuration

Figure 4 illustrates now BB is integrated as a RACE *network resource allocator*. The BB leverages widely available vendor mechanisms that support layer-3 DiffServ (Differentiated Services) and layer-2 CoS (Class of Service) 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, typically based on the 5-tuple (source IP address and port, destination IP address and port, and protocol) or any subset of this 5-tuple information. The classified traffic is marked with a DSCP codepoint 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, based on the DSCP marking, traffic is placed into different classes and provided differentiated, but consistent per-class treatment. This includes 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 Weighed Random Early Detection (WRED).

These two features by themselves are insufficient to guarantee end-to-end network QoS because the traffic presented to the network must be made to match the network capacity. What is also needed 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 accomodated. In Layer-3 network, it is the shortest path that is used for communication between any two hosts. We employ Djkstra's allpair shortest path algorithms. In Layer-2 network, we discover the VLAN tree to find the path between any two hosts. In MPLS networks,

we can force a path for a particular flow with the use of a MPLS label for that path, and of course, the routers in the path have to be configured for the next hop correctly using the MPLS label.

The BB is responsible for assigning the appropriate traffic class to each flow, and, using the Flow Provisioner, provisioning 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, independently of CQML. Much of this information is input to the Bandwidth Broker as reference data. The network topology itself is discovered by the Bandwidth Broker. CQML currently does not address this pre-deployment configuration issues.

Currently, the Bandwidth Broker provides these functions to support deployment decisions:

- **Flow Admission Functions:** Reserve, commit, modify, and delete flows.
- **Queries:** Bandwidth availability in different classes among pairs of pools and subnets.
- **Bandwidth Allocation Policy Changes:** In support of mission-mode changes.
- **Feedback/Monitoring Services:** 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 it is also based on delay bounds requested for the flow. The delay bounds for new flows need to be guaranteed without damaging the delay guarantees 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 guarantees [8]. Delay guarantees raise the level of abstraction of the Bandwidth Broker to application or higher-level resource management components and enable these components to provide better end-to-end mission guarantees. The basic framework we have developed is capable of dealing with any number of priority classes and, within a priority class, any number of weighted fair queuing subclasses. These guarantees are 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.

Bandwidth Broker Integration into RACE Figure 4 illustrates the architecture of the BB and shows how it is integrated as a RACE *network resource allocator*. As noted earlier, the BB leverages widely available vendor mechanisms that support

layer-3 DiffServ (Differentiated Services) and layer-2 CoS (Class of Service) features in commercial routers and switches. RACE's CPU resource allocators need to determine the hardware nodes in 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. Currently, it is a sequential process. The CPU resource allocator makes the hardware node decisions. The Bandwidth Broker is then invoked for the network QoS viability and reservation.

The BB in turn is responsible for assigning the appropriate traffic class to each flow, and, using the Flow Provisioner module, provisioning complex parameters for policing and marking, such that a contracted flow obtains 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.

The BB returns a DSCP marking for each flow, which the QoS-enabled component middleware needs to use 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 marking returned by BB as part of the component connection description so that DAnCE can configure the applications with those QoS settings when remote communications are made.

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. In order to enhance application components' reusability, it is important to separate QoS provisioning support from components' application logic. A policy framework will allow component implementations to be installed and used under different contexts by configuring component instances' QoS settings transparently using the underlying middleware at runtime.

As discussed in Section 3.2 and Section 3.3, the modeling tools allow the developers to specify the bandwidth requirements for various components, including only forward or reverse bandwidth or both. The RACE network allocator makes the network resource allocation decisions as QoS settings (DSCP markings) for various component connections. The components need to be configured with these QoS settings at deployment time at various deployment locations of a DRE system.

Problem: QoS-enabled component middleware, such as CIAO and DAnCE, can be extended to support configuring QoS-related policies of component instances via XML deployment descriptors. As shown in our RT-CCM work [40], components in a real-time application can be configured to run with different execution priorities by simply modifying the extended XML deployment descriptors. Such CCM extensions depend on applying various RT-CORBA [29] policies. Unfortunately, there is currently no standard CORBA policy for configuring the ORB protocols to set DSCP markings while either invoking a request or replying to a client. Furthermore, the existing CIAO RT-CCM extensions only support configuration of server-side policies.

Solution Approach: CIAO Network QoS Policy Framework. The address these limitations, we first need to extend CORBA policies [36] to support network QoS configurations, and to extend CIAO's RT extensions to support these new network policies. Similar to RT-CORBA's priority models, the actual network QoS can be configured on either the client-side or server-side depending on the usage scenarios. For example, some of the components may dictate how they answer queries, and hence they define the bandwidth requirements at which they receive and service requests, and clients using those services need to obtain that utilization information, before making requests. We have extended TAO, our CORBA implementation, 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. 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.

CIAO's policy configuration extension to standard deployment descriptors [30] has also been enhanced to parse and set these extended network policies that RACE resource allocator generates.

Furthermore, to accomodate different usage scenarios, CIAO need 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 override [36]. In order for a client to override a server's network policy with its own forward and reverse bandwidth requirements, client's own network policies need to be applied on the object reference pointing to the server. In component middleware, object references are shielded from components using the container programming model, and are provided to components using the **Context** objects, which are initialized with possible values at deployment time, depending on the connections a component could have. As depicted in Figure 5, to support configuring the client-side policies, we enhanced the the container programming model to (1) keep track of the policies associated with each component hosted within the container, (2) automatically override the object references to reference the appropriate policies, when a particular component wants to make a communication, and (3) return the overridden object references to the components using the **Context** object.

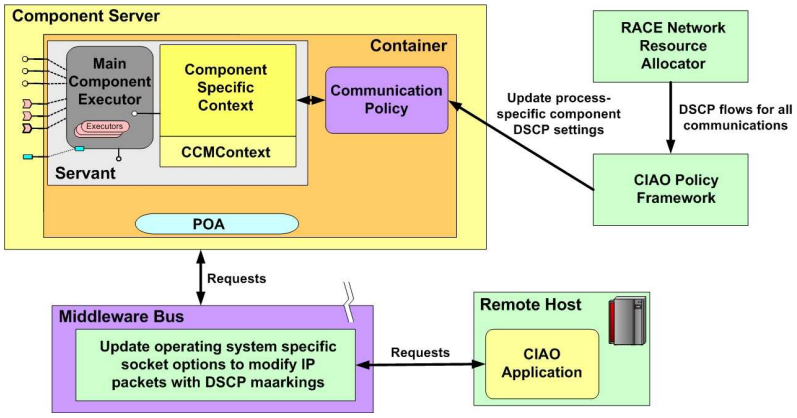


Figure 5. CIAO Network QoS Policy Framework

4 Empirical Validation of NetQoPE

This section validates the network QoS provisioning capabilities provided by the declarative mechanisms described in Section 3.

4.1 Experimental Testbed and Setup

The empirical validation of the NetQoPE framework was conducted at the Vanderbilt's ISISLab (www.dre.vanderbilt.edu/ISISlab), which is an open testbed for experimentation on distributed, real-time and embedded systems. It consists 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.

All of our experiments were conducted on 16 of those dual CPU blades, with 8 of them running as routers, and 8 of them running as the hosts hosting the office environment scenario applications developed using our QoS-enabled component middleware. All the machines were running Fedora Core 4 Linux distribution, which supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. All PCs were connected over a 100 Mbps LAN and the standard linux router software was used to configure the policing behavior. RACE is bundled as part of the CIAO distribution, and CIAO version 0.5 was used as the component middleware infrastructure. CoSMIC version 0.4.8 was used to model the network QoS requirements of the office environment scenario applications. The benchmarks ran in the POSIX real-time thread scheduling class [20] to enhance the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

Our experiment set up is described in Figure 6. *A, B, C, D, E, F, G,* and *H* are the hosts hosting the software components developed using our QoS-enabled component middleware. Software components include application software components, specifically software controllers controlling the sensors, the software controllers controlling the monitoring system, video servers and clients and surveillance data generators. RACE and its integrated network resource Allocator (which interfaces with Bandwidth Broker), developed using CIAO, is hosted on *C*. The Bandwidth Broker runs on *C* as well. *P, Q, R, S, T, U, V*, and *W* are the hosts hosting the router software, which is configured by the Flow Provisioner

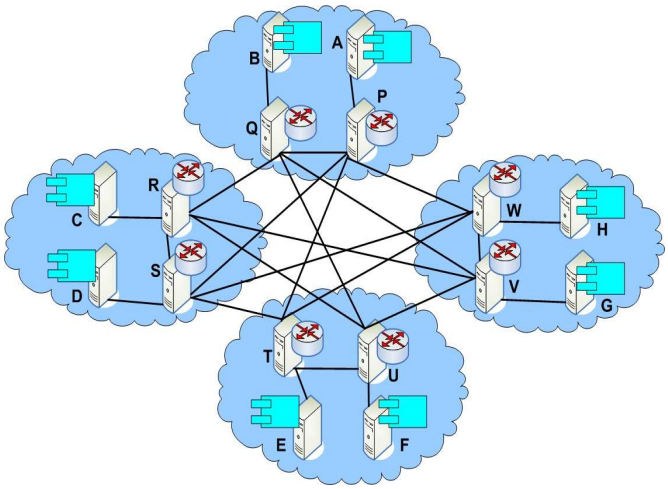


Figure 6. Experimental Setup

also hosted on *C*.

Our office environment case study comprises of different classes of traffic, as alluded to earlier 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) need to 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 the surveillance cameras, which 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., none to very little packet loss. The TCP protocol is used for this class. The latency requirement for this class is not that high as that of the HP class.

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

There is one other class, **BEST_EFFORT (BE)**. This traffic class comprises any other network traffic for which no QoS requirements are required. We configured our network topology described in Section 4.1 with the following class capacities on all the links: **HP** = 20 Mbps, **HR** = 30 Mbps, and **MM** = 30 Mbps for all the experiments. Moreover, the scheduling discipline used among these classes is strict priority; **HP** has the highest priority and **MM** has the lowest priority. The **BE** class could use the remaining available bandwidth in the network.

Since all the network traffic is going to be sent by software components developed using CORBA and Lightweight CCM, the core benchmarking software was based on the single-threaded version of the "`TestNetQoPE`" performance test distributed with CIAO[3]. This performance test consists of two components, say A and B, and creates a session for A to communicate with B. The session is to permit 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 way, we could experiment for different types of traffic by sending different types of payload, thereby varying the bandwidth used for the communications.

We recapture briefly how the various tools in the tool chain are used to run the experiments:

- **Modeling the QoS requirements** – We used the modeling capabilities described in Section 3.2 to model the QoS requirements of the applications for which we were interested in observing the delivered QoS. The generative capabilities of the modeling tools 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 the CIAO component, so that they can be configured when the components are initialized. This allows the components to send the targeted network traffic for each experiment.

[3] "`TAO/CIAO/performance-tests/NetQoPE`" in the CIAO release contains the source code for this benchmarking suite

- **RACE planning** – The QoS requirements captured in the metadata generated by the modeling tools is 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 is responsible for deploying the components according to the metadata. The CIAO policy framework uses the QoS configuration information to mark the packet headers of outgoing communication flows with the DSCP markings captured in the metadata.

As to be expected, 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 office environment case study, for example, when the fire sensor senses a fire, and sends the location of the fire to the monitoring subsystem, a video conferencing could be happening. Therefore, the network could 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.

The overall goals of the experiments were:

- Evaluating the effectiveness of the NetQoPE framework's traffic classes supported.

- Evaluating the effectiveness of the NetQoPE framework's admission control mechanisms that ensures there is enough capacity for the flows that have been admitted and guaranteed QoS.

- Evaluating the effectiveness of the NetQoPE framework's underlying network element mechanisms to police a flow for compliance, i.e., not to exceed its allocated bandwidth amount.

4.2 Experiments and Results

We now describe the experiments we performed using the lab setup described in Section 4.1. Results and rationale for the results obtained are also given for each experiment

4.2.1 Experiment 1: Effectiveness of the Traffic Classes Supported

Rationale. As we alluded to earlier, **HIGH_PRIORITY (HP)**, **HIGH_RELIABILITY (HR)**, and **MULTIMEDIA (MM)** are the three classes being supported. In addition, there is a default class; the **BEST_EFFORT (BE)** class. The **HP** class is the highest priority class for very critical traffic. **HR** is for DRE applications that cannot tolerate packet loss where as the **MM** class is for high volume DRE applications that can tolerate some loss, but require predictable delays. The **HP, HR, MM** and **BE** are supported by a Weighted Fair Queuing discipline with most scheduling slots for the **MM** traffic and the least scheduling slots for the **BE** class. The **HR** class has also a very large queue to minimize packet drops in our configuration.

These four traffic classes currently supported judiciously mix importance, latency, jitter and reliability QoS dimensions. In other words, there is a dimension reductionality. Although DiffServ allows 128 different DCSP markings or classes,

generally the more the classes, the more the resource consumption and the slower the forwarding behavior . Moreover, in Layer-2 networks (using CoS), there can only be at most eight classes. So, we chose as few classes as needed but still satisfying the needs of our domain, the office environment, as we will demonstrate now.

This experiment was run with two variants: one when the applications were communicating using the TCP transport protocol, and another when the applications were communicating using the UDP tranport protocol. In all the experiments, We 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 CORBA invocations with a payload of 200000 bytes for 1000 iterations. The experiments were repeated when the network QoS requested was configured to be one of: **HP**, **HR**, **MM**, and **BE**. In each of the cases corresponding to these classes, 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, few other applications were run:

Background Traffic	BE (Mbps)	HP (Mbps)	HR (Mbps)	MM (Mbps)
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

Please note that the background network traffic generated for our experiments are not normally prevalent in everyday DRE systems, and they are more of a worst case scenario. But we wanted to evaluate the performance of the NetQoPE framework 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 CORBA applications communicated using the TCP transport protocol. The results clearly show that, the average latency experienced by the CORBA application using the **HP** network QoS class in presence of varied background network QoS traffic, is much lower than the average latency experienced by the CORBA application using the **BE** network QoS class in the presence of background applications generating **BE** network QoS class traffic. This shows that the NetQoPE framework can ensure an upper bound on latencies experienced by CORBA applications even in the presence of applications generating contending network traffc, thereby ensuring a performance much better than the performance in a non-QoS network. This should not be surprising, as the **HP** traffic gets priority treatment over contending traffic from all other classes and as the **HP** traffic limit is well within the link capacity.

The results also shows that the average latency experienced by CORBA application using the **HR** network QoS class, in the presence of a varied background network QoS traffic is much lesser than the average latency experienced by CORBA application using the **MM** network QoS class, in the presence of a varied background network QoS traffic. This is because of the different per-hop behavior mechanisms 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. This means that, in the event of network congestion as generated in our experiments, the queues for the **MM** network QoS class are more likely to drop packets. This causes the TCP protocol flow control to reduce its window size by half, thereby causing application to experience more network delays while making the CORBA invocations. As more network delays add up, the average latencies experienced by the CORBA application increase, as illustrated for the **MM** network QoS class in our experiments.

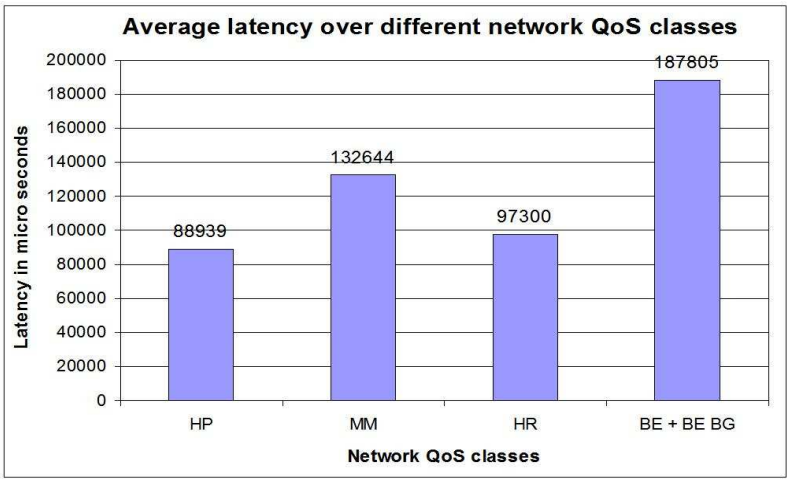


Figure 7. Average Latency under Different Network QoS Classes

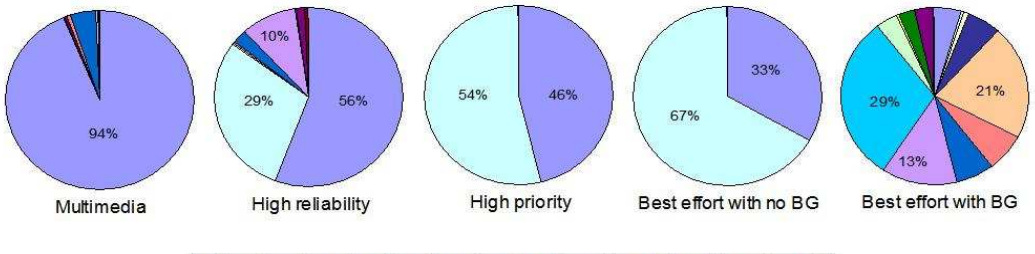
This clearly shows that **HR** network QoS class is more suited for CORBA applications communicating using the TCP transport protocol. Of course, the **HP** network QoS class gives the best performance, but not every application can use HP, as its bandwidth capacity is relatively small and as it is it is for extremely critical, but low volume traffic. In such cases, the applications should prefer using the **HR** network QoS class.

Methodology for the UDP experiment. "`TestNetQoPE`" was configured to make *oneway* CORBA invocations with a payload of 500 bytes for 100000 iterations. We used high resolution timer probes to measure the network delay for each CORBA invocation on the receiver side of the communication. When the sender makes an invocation, along with its invocation arguments, it also sends a timestamp, which is 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 kept count of the number of invocations received by the receiver, to keep track of the losses since the tranport protocol used is UDP. The experiments were repeated using different network QoS classes, in the presence of different background network traffic, as described below:

Background Traffic	BE (Mbps)	HP (Mbps)	HR (Mbps)	MM (Mbps)
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, for each network QoS class, at most 100000 network delay values (in milliseconds) were recorded, if there are no invocation losses. Then those values are arranged in the 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. 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. Figure 8 deals mainly with the cardinality of each bucket.



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

Analysis of results. Figure 8 shows the network delay groupings for different network QoS classes under different millisecond buckets. The results clearly show that in the case of CORBA application configured to use **MM** network QoS class, about 94 percent of the invocations had their normalized network delays within 1 millisecond. This performance is:

- 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.
- 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 in comparison to the queue size for the **HR** class, the UDP packets sent as part of the CORBA invocations do not experience as much queuing delays in the core routers as the packets belonging to the **HR** class. However, the **MM** class traffic experienced more packet losses, which in turn means fewer invocations. This behavior of less latency at the cost of packet drops may be acceptable even ideal in some situations. For example, in the modern office environment *oneway* CORBA calls for sending temperature updates or even company stock updates tickers may prefer to use

this **MM** class. For such applications, losing a few readings over a period does not matter as long as the most recent updates reaches the destination in time.

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

Rationale. In Section 3.3, we argued for the admission control capability for DRE systems over and beyond the two DiffServ features to guarantee QoS: (1) per-flow classification, marking and policing at the ingress, and (2) per-class differentiated treatment in packet forwarding at each of the routers. In this experiment, we demonstrate the need for admission control unequivocally.

Methodology. "`TestNetQoPE`" was configured to make synchronous CORBA invocations using the TCP transport protocol with a payload of 20000 bytes for 10000 iterations using the **HR** class. We repeated the experiments with the following pairs of client-server communications between a pair of hosts: 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 of these cases, 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 5, 10, 15 and 20 communicating pairs, we did not use the admission control capability. If the admission control capability were to be turned on, 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.

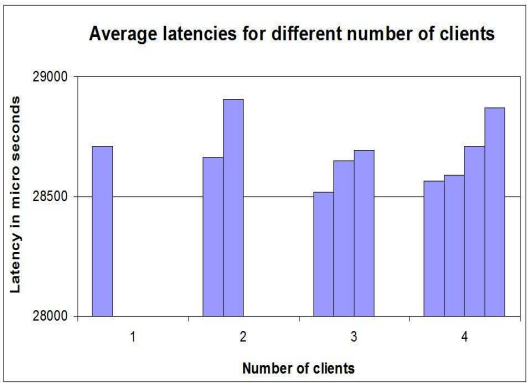


Figure 9. Average Invocation Latencies with Admission Control

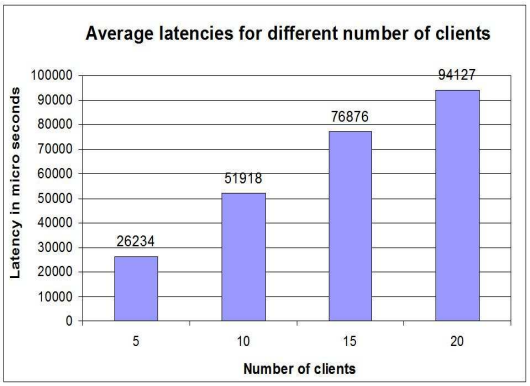


Figure 10. Average Invocation Latencies without Admission Control

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 the clients experienced the same average latency, as Figure 10 illustrates. This is because all the 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 the deployed applications were to share 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, and hence the latency experienced by the clients was almost equal to the latency experienced by the clients using the admission control capability. But as the number of client-server pairs increased to 10, 15 and 20, each communication pair bandwidth share started decreasing, causing the pairs to experience higher and higher latencies, as illustrated in Figure 10.

This demonstrates the need for DRE system middleware to incorporate admission control so as 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 our network QoS guarantees 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. So, policing for compliance at the granularity of a flow at the ingress router is required, and modern routers and switches support this feature. In this experiment, we validate this policing feature and demonstrate its usefulness.

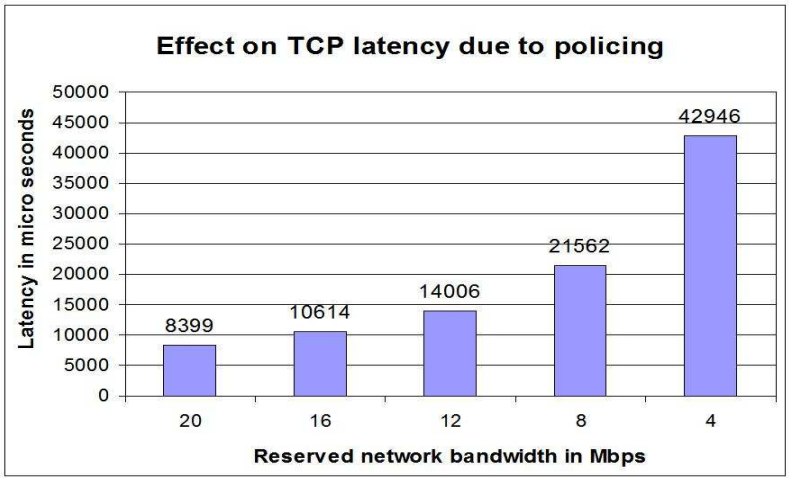


Figure 11. Effect on Invocation Latency due to Policing

Methodology. "`TestNetQoPE`" was configured to make synchronous CORBA invocations using the TCP transport protocol with a payload of 20000 bytes for 10000 iterations using the **HR** class. We repeated the experiments with the following bandwidth reservation requests for the flow carrying the CORBA invocations: 20 Mbps, 16 Mbps, 12 Mbps, 8 Mbps, and 4 Mbps. The experiments ran with the same background application network traffic as 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 linearly increases 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 kicks in dropping packets of the flow. 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 (the window size was halved) while making the CORBA invocations , i.e., lower the bandwidth size is, the higher the delay. It is to be stressed that the policing feature is independent of the protocol used, although each protocol may adjust differently when a packet loss is detected, i.e., when the policing function kicks in.

These experiments clearly show that applications configured with a specific network bandwidth reservation does not exceed this limit, thanks to the policing feature. This feature combined with the framework's admission control capability ensures QoS for all the admitted flows.

5 Related Work

The general problem of QoS implementation, management and enforcement is receiving increased attention . However, there has been little work on the specific problems addressed in this paper, namely, the use of a model-driven approach to specify network QoS for communication flows in DRE systems, subsequent automated allocation and provisioning of network resources during deployment and automated runtime support to effect appropriate QoS by component middleware transparent to applications in a portable manner.

Our work makes use of a Bandwidth Broker to reserve network resources and a Flow Provisioner to configure ingress network elements to achieve desired QoS for wired networks in both layer 3 and layer 2 networks. Such networks predominate enterprises, as exemplified by the office environment case study. The Bandwidth Broker leverages widely available DiffServ and CoS features in commercial routers and switches, respectively. The two main technologies for providing differentiated treatment of traffic are DiffServ/CoS and IntServ. [31] focusses on the integration of priority and reservation-based OS and network QoS management using IntServ with standards based DOC middleware like RT-CORBA to provide end-to-end QoS for DRE systems. In IntServ, however, 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 drawbacks with IntServ are that (1) it requires per-flow state at each router, which can be an issue from a scalability perspective; (2) it makes its admission decisions based on local information rather than some adaptive, network-wide policy; and (3) it is applicable only to layer-3 IP networks. Our network QoS does not have any of these drawbacks. In addition to [31], there is one other work that approaches network QoS through middleware. [43] is about a middleware broker that automatically maps application QoS requirements to underlying network provisioning mechanisms.

Telcordia has successfully applied Bandwidth Broker technologies in other settings [4, 21]. None of these endeavors, however, deals with layer-2 QoS, let alone unified management of QoS across multi-layers. None of these works, moreover, has focused on integration with network resource management using a component middleware framework, as we have done here with RACE, PICML and DAnCE.

Research has been done in adding QoS capabilities to component middleware. [17] 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. [6] illustrate 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. [9] extends an EJB container to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive QoS support. Our work on NetQoPE framework differs in 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) run-time capabilities to easily integrate network QoS settings into application communication flows.

Research has been done to provide QoS architectures and models for real-time multimedia applications, that demand predictable QoS from both endsystem and network resources. AQUA (Adaptive QUality of service Architecture) [18] 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. [27] describes a QoS broker that is used to do QoS negotiation for providing end-to-end QoS management for multimedia applications. [12, 15, 26, 33] have considered the co-reservation of mutiple resource types for providing QoS to real-time and distributed multimedia and high-end applications. Our work on NetQoPE framework differs in (1) providing network QoS provisioning and enforcing techniques that are not specific to specific resource types like multimedia resources, (2) providing a model-based advance reservation specification tool at per-flow level, so that it scales well with DiffServ services, and (3) run-time capabilities to easily integrate and deploy network QoS settings into application communication flows.

The research objectives of our work is closely related to other recently proposed approaches for QoS-aware deployment of applications in heterogeneous and dynamically changing distributed environments. GARA [13] and Darwin [5] focuses on identifying and reserving appropriate network resources to satisfy application requirements. Petstore [25] 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. [38] focuses on deploying J2EE applications, that colloborate a lot with other applications, collocated, so that the network delay can be minimized and application performance can be greatly improved. Our work on NetQoPE framework differs in utilizing popular network architectures like DiffServ to provide network QoS guarantees for applications, rather than trying to collocate them, and also in providing a possibility of deploying applications by considering multiple resources including network and CPU by adding planning algorithms in RACE, which can then sequentially or combinatorially work on providing deployment decisions for DRE applications.

6 Concluding Remarks

This paper describes the design and implementation of NetQoPE, a model-driven middleware framework that handles network QoS management and enforcement for component-based DRE systems. This work is an extension of our previous work that only considered host or CPU resources. The work integrates a Bandwidth Broker into a QoS-enabled middleware framework to ensure offered load matches with the capacity throughout the network and thereby providing network QoS guarantees. The Bandwidth Broker leverages DiffServ/CoS features that are commonly available in today's routers and switches. Two accomplishments/contributions, in our view, stand out:

- Standard Lightweight CORBA Component Middleware (CCM) interfaces can be extended with relative ease to develop a scalable and flexible middleware infrastructure that can support marking IP header with the right DSCP codepoint determined by the Bandwidth Broker, the network QoS management subsystem, in an operating system independent portable and in application-transparent manner, i.e., application developers cam be shielded from working with low level network interfaces. More importantly, the resultant QoS-enabled middleware facilitates different QoS guarantees to the same software written and deployed under multiple deployment targets and network environments. This effectively increases software code reuse.

- NetQoPE makes network provisioning decisions in coordination with component placement algorithms that consider host resources, specifically CPU. This capability we have demonstrated is a major practical step in quality of service research to take into account two main resources simultaneously. Allocation problems that take into account both CPU and network resources are l integer programming problems that are more intractable than bin-packing problem formulations for the CPU resource.

The Bandwidth Broker technology [7, 8] incorporates mechanisms to dynamically adapt to network faults. When there is a fault, in essence, 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 guarantees of some previously admitted flows to be missed. The Bandwidth Broker is capable of rediscovering paths as soon as the fault is detected and making re-admission decisions. The readmission decision can be policy-driven. Policies such as "drop as few flows as possible", "drop only low priority flows", etc can be supported in a plug-and-play manner.

However, an application may not be prepared to work with the degraded quality of service for the demoted flows regardless of the readmission policy chosen. One strategy is application-level adaptation that essentially involves substituting a different implementation for the same component during runtime [1] for one or more components making up the application. This is because two different implementations for the same component may require different network QoS. Our future work will focus on developing selection algorithms [44] that can automatically choose the most suitable component implementation to operate in the case of network resource scarcity. We will implement these selection algorithms and validate them in the context of a DRE application such as the office of the future environment discussed in this paper.

References

[1] J. Balasubramanian, B. Natarajan, D. C. Schmidt, A. Gokhale, G. Deng, and J. Parsons. Middleware Support for Dynamic Component Updating. In *International Symposium on Distributed Objects and Applications (DOA 2005)*, Agia Napa, Cyprus, Oct. 2005.

[2] 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. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 190–199, San Francisco, CA, Mar. 2005. IEEE.

[3] 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.

[4] R. Chadha, Y.-H. Cheng, T. Cheng, S. Gadgil, A. Hafid, K. Kim, G. Levin, N. Natarajan, K. Parmeswaran, A. Poylisher, and J. Unger. Pecan: Policy-enabled configuration 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-Specific Modeling. In *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*. CRC Press, Dec. 2006.

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

[16] Institute for Software Integrated Systems. Component Synthesis using Model Integrated Computing (CoSMIC). www.dre.vanderbilt.edu/cosmic, Vanderbilt University, Nashville, TN.

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

[18] 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.

[19] 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.

[20] Khanna, S., *et al.* Realtime Scheduling in SunOS 5.0. In *Proceedings of the USENIX Winter Conference*, pages 375–390. USENIX Association, 1992.

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

[22] J. Kinnebrew, N. Shankaran, G. Biswas, and D. Schmidt. A Decision-Theoretic Planner with Dynamic Component Reconguration for Distributed Real-Time Applications. In *Poster paper at the Twenty-First National Conference on Artificial Intelligence*, Boston, MA, July 2006.

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

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

[25] D. Llambiri, A. Totok, and V. Karamcheti. Efficiently 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.

[26] 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.
[27] K. Nahrstedt and J. Smith. The QoS Broker. *IEEE Multimedia Magazine*, pages 53–67, Spring 1995.
[28] Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, Nov. 2002.
[29] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/05-01-04 edition, Aug. 2002.
[30] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.
[31] 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.
[32] 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.
[33] 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.
[34] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):41–47, 2006.
[35] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.
[36] D. C. Schmidt and S. Vinoski. An Overview of the CORBA Messaging Quality of Service Framework. *C++ Report*, 12(3), Mar. 2000.
[37] 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.
[38] 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.
[39] M. Volter, A. Schmid, and E. Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*. Wiley Series in Software Design Patterns, West Sussex, England, 2002.
[40] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Configuring 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.
[41] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill. QoS-enabled Middleware. In Q. Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2004.
[42] N. Wang, D. C. Schmidt, K. Parameswaran, and M. Kircher. Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *24th Computer Software and Applications Conference*, Taipei, Taiwan, Oct. 2000. IEEE.
[43] 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.
[44] D. M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, 42(1), 2003.