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

Jaiganesh Balasubramanian[1], Sumant Tambe[1], Shrirang Gadgil[2], Frederick Porter[2], Balakrishnan Dasarathy[2], Aniruddha Gokhale[1], and Douglas C. Schmidt[1]

[1] Department of EECS, Vanderbilt University, Nashville, TN 37235, USA
[2] Telcordia Technologies, Piscataway, NJ 08854, USA

Abstract. Developers of distributed real-time and embedded (DRE) systems face challenges in provisioning network quality of service (QoS) properties due to the presence of application flows that require a range of network-level QoS properties, as well as the complexity of specifying per-flow network QoS requirements and mapping them to network QoS enforcement mechanisms. This paper describes a QoS-enabling component middleware framework called NetQoPE that addresses these challenges via a multi-stage approach that includes (1) a model-driven framework for declarative specification of per-flow network QoS requirements, (2) a network resource allocation and configuration framework that maps per-flow network QoS requirements to platform-specific network QoS provisioning mechanisms, (3) a deployment and configuration framework for deploying system components with the appropriate network QoS settings on the platforms, and (4) a runtime framework that enforces the network QoS properties for DRE application flows.

This paper evaluates the effectiveness of NetQoPE in the context of a representative DRE application in an IP network testbed that supports various types of network traffic. Our empirical results demonstrate that the capabilities provided by NetQoPE yield flexible, predictable, and efficient network QoS provisioning for DRE systems.

Keywords: model-driven architectures, middleware for quality-of-service, QoS-enabling component middleware, deployment and configuration tools

1 Introduction

Distributed real-time and embedded (DRE) systems often comprise multiple end-to-end application flows that may require various QoS properties affecting CPU, memory, and network resources. Examples of such DRE systems include shipboard computing systems, supervisory control and data acquisition (SCADA) systems, and corporate enterprises. Developers of DRE systems typically provision their network-level QoS properties via network QoS mechanisms, such as integrated services (IntServ) [1] and differentiated services (DiffServ) [2].

Prior work has concentrated on how applications can use network QoS mechanisms. For example, [3] focuses on encapsulating these mechanisms within operating systems and [4] focuses on modifying application business logic to work with QoS brokers. In these approaches, applications are often tightly coupled with the QoS APIs and the underlying network QoS mechanisms. As a result, these solutions require obtrusive application modifications when there are changes to OS QoS APIs, network QoS mechanisms, or per-flow network QoS requirements. Moreover, due to their complex APIs, network QoS mechanisms can be tedious and error-prone to program and manage, particularly as the number of application flows increases.

Solutions to network QoS provisioning are more effective when application business logic is decoupled from QoS provisioning techniques [5]. Achieving this objective requires identifying key elements in the QoS provisioning problem space that incur variability when mapping the problem space to the solution space, then designing solutions that address these sources of variability on behalf of applications. Experience has shown [6] that middleware is an appropriate level of abstraction at which to resolve these challenges, by providing services that integrate network QoS mechanisms without tightly coupling applications to low-level OS APIs and network QoS mechanisms.

This paper describes the design and evaluates the performance of the *Network QoS Provisioning Engine* (NetQoPE), which is a set of model-driven middleware frameworks designed to address variability in network QoS provisioning for DRE systems at multiple stages, including design-, deployment-, and run-time. NetQoPE's design-time capabilities include (1) model-driven declarative mechanisms that enable the specification of application network QoS requirements at the individual application communication flow level, *e.g.*, 10 Mbps forward and reverse bandwidth between peer components, and (2) a resource allocator and provisioner, which use commonly available network QoS mechanisms, such as DiffServ, to allocate and provision network resources.

NetQoPE's deployment-time capabilities include deploying and configuring the applications with the appropriate network QoS settings, such as DiffServ codepoints (DSCP). DSCPs are used to differentiate IP packets at the routers when applications make remote invocations. NetQoPE's run-time capabilities enforce the network QoS settings, such as IP packet markings with the right set of DSCPs, when applications communicate with each other. We evaluate the flexibility and effectiveness of NetQoPE in the context of a representative DRE system in an IP network that uses a Bandwidth Broker [7] to manage network resources via DiffServ QoS mechanisms.

The remainder of the paper is organized as follows: Section 2 uses a case study to motivate common requirements associated with provisioning network QoS for DRE systems; Section 3 describes how NetQoPE is designed to address key challenges associated with meeting these requirements; Section 4 provides experimental validation of NetQoPE in the context of the case study; Section 5 compares our work on NetQoPE with related research; and Section 6 presents concluding remarks and lessons learned.

2 Motivating NetQoPE's Network QoS Provisioning Capabilities: A Case Study

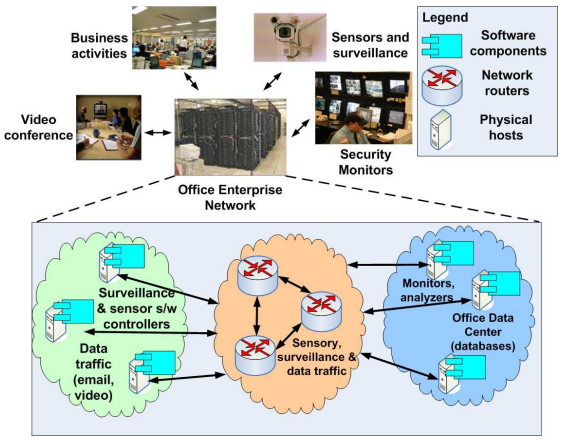

Fig. 1: A Corporate Environment and Its Network Configuration

Figure 1 shows a representative DRE system in a modern corporate enterprise, which we use as case study to demonstrate and validate NetQoPE's network QoS provisioning capabilities. These corporate enterprises often transport network traffic using an IP network over high-speed Ethernet. Network traffic in an enterprise can be grouped into several classes, including (1) e-mail, videoconferencing, and normal business traffic, and (2) sensory and imagery traffic of the safety/security hardware (such as fire/smoke sensors) installed in the premises. This paper makes the reasonable assumption that safety/security traffic is more critical than other traffic and focuses on middleware-guided mechanisms to assure the specified QoS for this mission-critical traffic in the presence of other traffic that shares the same network.

As shown in Figure 1, an office enterprise has many software controllers that manage the hardware devices, such as sensors and monitors. Each sensor/camera software controller filters the sensory/imagery information and relays them to the monitor software controllers that display the information. The traffic between these software controllers requires network QoS mechanisms—our case study uses DiffServ to provide network QoS to these applications.

For applications to obtain the required network QoS using DiffServ, multiple steps are involved, as described below:

- **QoS requirements specification.** DiffServ provides different levels of service (*e.g.*, premium and assured) to meet the QoS demands of application traffic. For each pair of application communication endpoints, the service quality (*e.g.*, assured), and the bandwidth required (*e.g.*, 10 Mbps) must be specified without modifying the application code.
- **Network resource allocation and configuration.** A Bandwidth Broker [7] manages the network resources in our DiffServ domain using admission control capabilities to ensure that admitted flows have the requisite bandwidth and end-to-end delay, loss and jitter. Depending on QoS requirements for each pair of application communication end points, network-level resources must be allocated without the intervention of the application code, and a DSCP value should be assigned that is indicative of the required type of service, *e.g.*, assured vs. premium.

- **Runtime communication with network QoS settings.** When application communication is in progress, the pre-assigned DSCP value must be transmitted along with the IP packets that encapsulate the application-level messages. The routers use the DSCP markings in the IP packets to provide the appropriate QoS, *e.g.*, packets marked with premium DSCP values are given preference relative to packets marked with assured DSCP values.

Decoupling application code while meeting the three requirements above motivates our 3-stage NetQoPE architecture described in Section 3.

3 Network QoS Provisioning Capabilities in NetQoPE

This section describes the design-time, deployment-time, and run-time challenges that arose when developing a DiffServ-based prototype of the office enterprise case study outlined in Section 2. We then describe how NetQoPE helped resolve these challenges with its model-driven middleware frameworks.

3.1 NetQoPE's Multistage Network QoS Provisioning Architecture

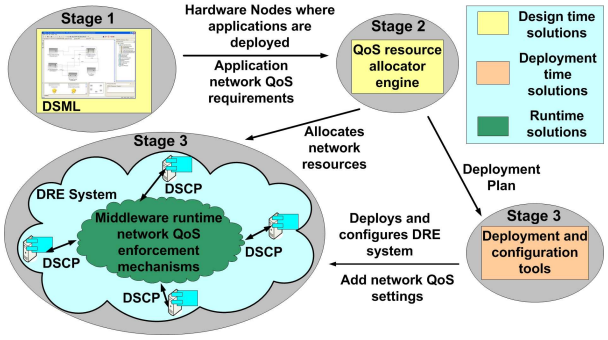

Fig. 2: NetQoPE's Three Stage Architecture

Figure 2 shows NetQoPE's three stage architecture and the techniques it uses for network QoS provisioning. Conventional techniques for adding network QoS to applications typically modify the OS [3] or applications [4]. Below we describe the challenges of applying such techniques and summarize how NetQoPE's model-driven middleware frameworks address these challenges.

3.1.1 Challenge 1: Alleviating Complexities in QoS Requirements Specification

Context. DRE applications must specify a level of service (*e.g.*, high priority versus high reliability) and their associated bandwidth requirements so that network elements provision resources properly. Ideally, multiple instances of the same reusable software components should be deployed and their QoS specifications should be customizable without requiring modifications to application code.

Problem. In our office enterprise case study, multiple fire sensors with different QoS requirements operate at different importance levels managed by reusable software controllers that deploy the same application code. For example, fire sensor deployed in a parking lot have lower importance than those in the server room. Since the fire sensors are managed by the same reusable software controllers, manually specifying the different QoS requirements needed for different important contexts is tedious, error-prone, and non-scalable.

Solution approach → Domain-specific modeling languages. Model-driven engineering (MDE) [8] captures network QoS requirements for data and control flows in a DRE system and helps raise the level of the abstraction of system design to a level higher than third-generation programming languages, such as C++ or Java. MDE also alleviates accidental complexities of using declarative mechanisms based on XML [9].

DRE system designers can use NetQoPE's *Component QoS Modeling Language* (CQML) to describe (1) the end-to-end application structure, including all source and destination pairs of applications, (2) the application network QoS requirements in terms of bandwidth, network transport mechanisms used, and traffic classes, and (3) the target environment, including the physical hosts on which the applications will be deployed. CQML traverses the application models and their QoS requirements and generates standards-based XML descriptors defined by the OMG Deployment and Configuration (D&C) specification [10]. Section 3.2.1 describes the design of CQML in more detail.

3.1.2 Challenge 2: Alleviating Complexities in Network Resource Allocation and Configuration

Context. DRE systems can allocate and configure network resources based on the QoS requirements specified by their applications.

Problem. In our office enterprise case study, a sensor monitoring a fire in a parking lot may have minimal QoS requirements, whereas the sensor monitoring a fire in a server room may have more stringent QoS requirements. As before, the reusable software controllers managing the fire sensors run the same application code. If this code were modified to use a Bandwidth Broker to allocate network resources, applications would incur unnecessary time/space overhead. For example, the software controller managing a fire sensor in a parking lot may need no network QoS, and thus need not allocate network resources. Ideally, applications should only use network QoS mechanisms when deemed necessary by the deployment context.

Solution approach → Resource allocator framework. NetQoPE's MDE tools in Stage 1 (see Section 3.1.1) capture the QoS requirements of different applications at a per-flow and per-deployment-context basis. This information is used in Stage 2 as input to NetQoPE's *QoS resource allocator engine* to provide resource allocation to different applications. NetQoPE's QoS resource allocation engine uses the Bandwidth Broker's network resource provisioners [7] to support resource allocation requests on a per-flow and per-deployment-context basis on behalf of applications.

After the network resources are allocated and network devices (*e.g.*, routers that support DiffServ) are configured, network resource provisioners provide network QoS settings (*e.g.*, DSCP markings) that applications use when interacting with the configured underlying network. NetQoPE encodes these DSCP markings on the deployment descriptors associated with the connections between applications. Applications are thus simplified by being shielded from (1) different network QoS mechanisms that are used to allocate network resources and (2) the

deployment contexts in which the resource allocations. Section 3.2.2 describes NetQoPE's QoS resource allocation engine in detail.

3.1.3 Challenge 3: Alleviating Complexities in Network QoS Settings Configuration

Context. DRE applications need to invoke remote operations using the chosen network QoS settings (*e.g.*, DSCP markings) so that the network layer can differentiate application traffic based on QoS,

Problem. Application developers historically have written code that instructs the middleware to provide the appropriate runtime services, *e.g.*, DSCP markings in IP packets. In our office enterprise case study, for example, fire sensors can be deployed in different QoS contexts that are managed by reusable software controllers. Modifying application code to instruct the middleware to add network QoS settings would be tedious, error-prone, and non-scalable because (1) the same application code could be used in different contexts requiring different network QoS settings and (2) application developers might not (and should not) know the different QoS contexts in which the applications are used during the development process. Application-transparent mechanisms are therefore needed to configure the middleware to add these network QoS settings depending on the deployment context in which applications are used.

Solution approach → Deployment and runtime middleware mechanisms. NetQoPE uses a deployment and configuration (D&C) engine [11] to map application components to the appropriate nodes in a target environment. This D&C engine is a middleware framework that processes deployment descriptors generated during Stage 1 and Stage 2 and deploys application components to their respective nodes. Since deployment descriptors capture the network QoS settings (*e.g.*, DSCP values) to use on a per-connection or a per-flow granularity, the D&C engine can configure the underlying middleware to use the appropriate settings when operations are invoked at runtime. As a result, middleware can be configured automatically on behalf of applications on a per-flow and per-context granularity, thereby simplifying business logic.

NetQoPE also uses runtime middleware mechanisms [6] to invoke remote operations. Its D&C engine provisions and configures the middleware to provide the required application-specific connection services and associated network QoS settings, *e.g.*, DSCP values. At runtime, these network QoS settings are automatically added to the IP packets when applications invoke remote operations. Section 3.2.3 describes NetQoPE's D&C and runtime capabilities in detail.

3.2 A Concrete Instantiation of the NetQoPE Architecture

Section 3.1 described the NetQoPE's model-driven middleware architecture for network QoS provisioning. To prototype the concepts described in Section 3.1—and showcase how middleware can simplify the provisioning of network QoS capabilities for DRE applications—we used the Lightweight CORBA Component Model (LwCCM), which provides QoS-enabling component middleware capabilities for DRE systems. We have extensive experience developing and applying LwCCM-based QoS-enabling middleware infrastructure, such as the CIAO

LwCCM middleware [6] and the DAnCE LwCCM D&C engine [11], and model-driven tools, such as CoSMIC [9], that simplified our NetQoPE prototype.

The remainder of this section describes a concrete realization of the NetQoPE architecture using LwCCM as the component middleware platform and DiffServ as the network-level QoS mechanism. NetQoPE uses a Bandwidth Broker [7] that reserves and allocates DiffServ-related network resources for DRE applications. Note that this concrete instantiation is just one way to realize the multi-stage NetQoPE architecture. In particular, the concepts described in Section 3.1 could also be used to provision network QoS for other QoS-enabling component middleware platforms and tools [8, 12, 13].

3.2.1 Network QoS Modeling Language The *Component QoS Modeling Language* (CQML) is a domain-specific modeling language (DSML) [14] NetQoPE uses to specify network QoS requirements of LwCCM-based DRE applications that use DiffServ. CQML annotates connections between components in DRE applications with the following attributes: (1) network QoS classes, such as HIGH PRIORITY (HP), HIGH RELIABILITY (HR), MULTIMEDIA (MM), and BEST EFFORT (BE), (2) bi-directional bandwidth requirements, and (3) transport protocol used. Figure 3 shows a CQML model for key flows in our case study from Section 2. If multiple flows require the same network QoS class, CQML enhances scalability by allowing models with different flows to share the network QoS class attribute, which significantly reduces the effort of modeling traffic flows.

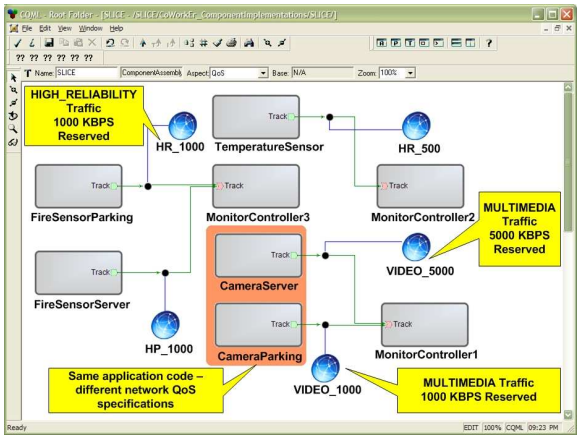

Fig. 3: NetQoPE's Network QoS Requirements Specification Capabilities

CQML's network QoS classes correspond to the DiffServ levels of service provided by a Bandwidth Broker [7]. For example, the HP class represents the highest importance and lowest latency traffic (*e.g.*, fire sense reporting in the server room). The HR class represents traffic with low drop rate (*e.g.*, surveillance data). CQML also supports the MM class for sending multimedia data and the BE class for sending traffic with no QoS requirements.

Clients make forward and reverse bandwidth reservations to assure network QoS when they invoke remote operations with server applications. In some scenarios, however, the clients cannot dictate the network QoS and must rely on the network QoS guarantees provided by server applications. For example, some applications in the office enterprise might have many clients, so the network could be overloaded if all clients reserve forward and reverse bandwidth. In these scenarios, clients invoke remote operations on the server applications, and the server applications reply according to priorities based on the available resources.

To support these types of scenarios, CQML allows DRE system designers to designate two endpoints of component-to-component communication via NetQoPE's network priority models, which are inspired by Real-time CORBA's request priority models. The two network priority models supported by NetQoPE include (1) CLIENT_PROPAGATED network priority model, where the client dictates the priorities at which the requests and replies are sent, and (2) SERVER_DECLARED network priority model, where the server dictates the priority at which the requests and replies are sent. When connections are made, CQML ensures that endpoints of the connection have semantically consistent priority models. In particular, CQML detects and prevents designating one end of a connection to have the CLIENT_PROPAGATED network priority model and the other end of the connection to have the SERVER_DECLARED network priority model.

Generation of deployment metadata. CQML is part of the CoSMIC modeling tool suite [9], which allows developers to specify the computing nodes for deploying application components. CQML generates standards-based deployment descriptors [10] that capture the application deployment details, as well as their network QoS requirements. As described in Section 3.2.2, these requirements are used by NetQoPE's network resource allocators to provision the needed network QoS.

Benefits of the approach. Multiple instances of the same applications could be deployed in different/same deployment contexts with different QoS requirements. CQML captures the context-specific per-flow QoS requirements by virtue of modeling a deployment context that includes all applications and their communications. Application developers can thus focus on writing their business logic without being tightly coupled to the context in which it will be used. Specifying those QoS requirements manually or programmatically is tedious and error-prone. CQML auto-generates XML descriptors that capture inter-component communications and QoS requirements, thereby enabling per-flow resource allocations that are "correct-by-construction." Section 4.2.2 describes an experiment that evaluates CQML empirically.

3.2.2 Network Resource Reservation and Device Configuration NetQoPE's network resource allocator middleware is integrated with a Bandwidth Broker [7] that allocates and manages network resources in DiffServ networks. Figure 4 illustrates the architecture of NetQoPE's network resource allocator.

NetQoPE's network resource allocation uses a two-phase process. In the first phase, *i.e.*, before deploying the applications, the network resource allocator parses the deployment descriptors generated by CQML (Section 3.2.1) to acquire the DRE application flows being deployed. NetQoPE next invokes the Bandwidth Broker admission control capabilities by feeding it one application flow at a time. If all flows can be admitted, NetQoPE then proceeds with the configuration of network resources for those flows. Conversely, if not all flows can be admitted, NetQoPE enables application developers to change the deployment (*e.g.*, change component implementation to consume less resources) since applications have not yet been deployed.

In the second phase, NetQoPE instructs the Bandwidth Broker to reserve all resources in the specified classes, and provide the DSCPs used by application

operation invocations. The Bandwidth Broker uses its *Flow Provisioner* to configure the routers to provide the appropriate per-hop behavior when they receive IP packets with the specified DSCP values. The DSCPs used by communicating components are encoded as connection attributes in deployment descriptors. Section 3.2.3 discusses how NetQoPE adds these DSCPs to communicating components without modifying application code.

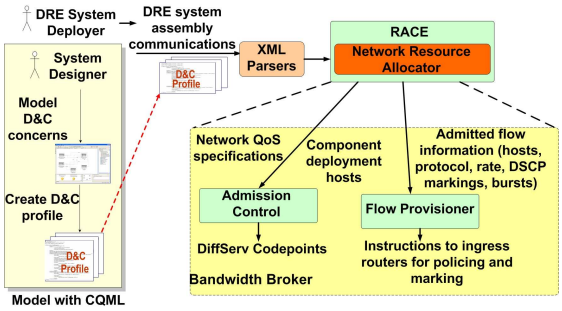

Fig. 4: NetQoPE's Network Resource Allocation and Configuration Capabilities

Benefits of the approach. Rather than modifying applications or the operating systems to integrate network QoS mechanisms, NetQoPE's middleware-based network resource allocators allows application developers to focus on writing business logic, rather than wrestling with complex network QoS APIs. NetQoPE's capability to determine if sufficient resources can be provisioned before deploying applications also provides an opportunity to change the deployment (*e.g.*, node, implementation, etc.), thereby allowing DRE application designers to plan for a degraded QoS prior to runtime. Section 4.2.3 describes an experiment that evaluates NetQoPE's network resource allocator empirically.

3.2.3 Component Middleware Mechanisms to Configure Application QoS Settings NetQoPE leverages the DAnCE [11] deployment and configuration (D&C) engine to create a *network configurator* for configuring network QoS policies of DRE applications developed using the CIAO [6] component middleware. DAnCE automates the configuration of semantically consistent QoS policies for application and service components. For example, NetQoPE's network configurator parses the deployment descriptors and annotated DSCPs on the connections to add the following QoS policies automatically:

- **The** CLIENT_PROPAGATED **network policy**, which allows clients to define the forward and reverse DSCPs used for communication. This policy corresponds to the CLIENT_PROPAGATED network priority model defined in the CQML model described in Section 3.2.1.
- **The** SERVER_DECLARED **network policy**, which allows servers to define the forward and reverse DSCPs used for communication. This policy corresponds to the SERVER_DECLARED network priority model defined in the CQML model.

Honoring of policies. As mentioned above, NetQoPE supports both the SERVER_DECLARED and CLIENT_PROPAGATED network priority policies. To support the SERVER_DECLARED network policy, NetQoPE encodes the request and reply DSCPs in a CCM interoperable object reference (IOR). When a client makes a call on the object using the IOR, NetQoPE uses the underlying CIAO middle-

ware to decode the request DSCP from the IOR and add it to the IP packets. Likewise, CIAO's portable object adapter (POA) in the server checks the policy on the IOR and adds the reply DSCP to the IP packets. NetQoPE's network configurator is part of the DAnCE D&C engine that configures the POA as part of creating the component containers and IORs. Its network configurator can therefore automatically add the designated policies to the IOR and configure the POA to reply with the right set of DSCPs for each object it hosts.

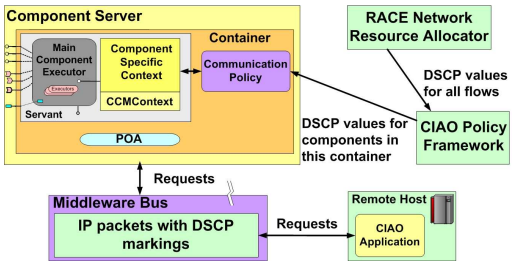

Fig. 5: NetQoPE's Network QoS Settings Configuration Capabilities

Likewise, to support the CLIENT_PROPAGATED network policy, NetQoPE uses the container programming model shown in Figure 5, where component A obtains the IOR of a remote component B using the container. To allow component A to dictate the request and reply priorities of the remote communication, NetQoPE applies the client propagated network policy on the object reference of component B provided to component A. The policy specifies the request and reply DSCPs to be used. When component A makes a call on component B via the IOR, NetQoPE uses CIAO to decode the request DSCP from the policy associated with the IOR and add it to the IP packets. Moreover, it also decodes the reply DSCP from the policy, and sends it along with the request as part of the *service context*. Before the server sends the reply, the POA checks the request *service context* and adds the reply DSCP on the reply IP packets. To allow the container to add the request and reply DSCPs to the IOR at deployment time, NetQoPE's network configurator configures the containers with the DSCP values for all the components hosted by the container.

Benefits of the approach. Using the NetQoPE D&C engine to provision network QoS settings allows DRE application developers to focus on their business logic, rather than wrestling with low-level mechanisms for provisioning network QoS. Moreover, the NetQoPE container programming model provides an effective means to apply the correct policies when components communicate. For example, NetQoPE's D&C engine can automatically configure containers to provide semantically consistent priority model support (*e.g.*, SERVER_DECLARED or CLIENT_PROPAGATED) for components they hosts. This capability allows reuse of components in multiple contexts, *e.g.*, the middleware and MDE tools declaratively specify the priority rather than having clients imperatively program the priority into application code.

Section 4.2.2 describes an experiment that evaluates how reusable software components deployed in multiple contexts with different priority models can provide different end-to-end network QoS to applications that invoke remote operations. Section 4.2.1 describes an experiment that evaluates the overhead of adding support for CLIENT_PROPAGATED and SERVER_DECLARED network pri-

ority models within the runtime middleware applications use for making remote communications.

4 Empirical Validation of NetQoPE

This section empirically evaluates the network QoS provisioning capabilities provided by the NetQoPE model-driven frameworks 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 consists of (1) 56 dual-CPU blades running 2.8 Gz XEONs with 1 GB memory, 40 GB disks, and 4 NICs per blade, and (2) 6 Cisco 3750G switches with 24 10/100/1000 MPS ports per switch. As shown in Figure 6, our experiments were conducted on 16 of those dual CPU blades, with 8 of them hosting linux router software, and the rest hosting software controllers (*e.g.*, a fire sensor controller) developed using our NetQoPE middleware.

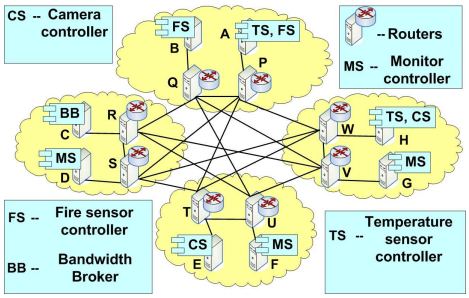

Fig. 6: Experimental Setup

Our evaluations used DiffServ QoS and the associated Bandwidth Broker [7] software was hosted on blade *C*. All blades ran Fedora Core 4 Linux distribution configured using the real-time scheduling class. The blades were connected over a 1 Gbps LAN with virtual 100 Mbps links. The NetQoPE middleware frameworks were implemented atop CIAO and DAnCE version 0.5 and CoSMIC version 0.4.8.

We evaluated NetQoPE in a scenario where a number of sensory and imagery software controllers send their monitored information to monitor controllers so that appropriate control actions could be taken by enterprise supervisors monitoring abnormal events. For example, as shown in Figure 6, several *fire sensor controller* components are deployed in blades A and B. These components send their monitored information to *monitor controller* components deployed in blades D and F. The communications between these software controllers could use one of the traffic classes defined in Section 3.2.1 with the following capacities on all links: HP = 20 Mbps, HR = 30 Mbps, and MM = 30 Mbps for the experiments. The BE class used the remaining available bandwidth in the network.

To emulate the network traffic behavior of the software components created using NetQoPE, we developed a "`TestNetQoPE`" performance test, which is distributed with CIAO. For a component A to communicate with component B, this test creates a session with a configurable number of invocations and sleep time. All experiments use high resolution timer probes to accurately measure the roundtrip latency for each invocation made by the client.

4.2 Experimental Results and Analysis

Below we describe the experiments performed using the ISISlab configuration described in Section 4.1 and analyze the results.

4.2.1 Evaluating the Overhead of NetQoPE for Normal Operations

Rationale. NetQoPE provides network QoS to applications by using the three stage architecture shown in Figure 2. Since all the DRE applications require network QoS, it is worthwhile to evaluate the overhead of NetQoPE when it is used to design, deploy, and configure applications with *no* QoS requirements.

Methodology. The NetQoPE CQML, network resource allocator, and network configurator described in Section 3 are used at design- and deployment-time, and hence incur no additional run-time overhead. DRE application designers can use CQML, which is part of the CoSMIC tool suite [9], to model the DRE application structure and generate deployment descriptors used by the DAnCE D&C engine to deploy and configure component-based applications. NetQoPE's network resource allocators and network configurator thus are invoked only if network QoS attributes are modeled using CQML. As a result, NetQoPE incurs no design-time and deployment-time overhead if network QoS is not needed.

NetQoPE enhances CIAO to support the CLIENT_PROPAGATED and SERVER_DECLARED network priority models. To measure runtime overhead, we ran the experiment with the following variants: (1) a CIAO client calling a CIAO server with no network QoS, (2) a CIAO client configured with the CLIENT_PROPAGATED network policy calling a CIAO server, and (3) a CIAO client calling a CIAO server configured with the SERVER_DECLARED network policy.

`TestNetQoPE` was configured to make 200,000 invocations that generated a load of 6 Mbps, and average roundtrip latency was calculated. For all experiment variants, there was no background network load. The CLIENT_PROPAGATED and SERVER_DECLARED network priority models were configured with DSCP values of 0, which indicates no QoS preferences were given for IP packets. The routers were not configured to do DiffServ processing (provide routing behavior based on the DSCP markings), and hence no edge router processing overhead was incurred.

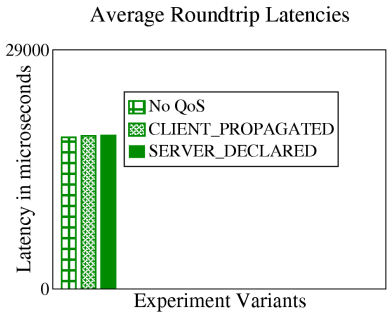

Fig. 7: Overhead of NetQoPE's Policy Framework

Analysis of results. Figure 7 shows the different average roundtrip latencies experienced by clients in the three different variants of the experiment, which are all are similar. To honor the CLIENT_PROPAGATED and SERVER_DECLARED network policy models, the NetQoPE middleware added the request and reply DSCPs to the IP packets. As shown by these results, these enhancements are quite efficient and add little overhead to DRE applications that do not require any network QoS.

Network QoS was not needed in this experiment, and therefore network resources were not allocated and a DSCP value of 0 was used. If a different variant of the experiment is run with background network loads, and network QoS is required for some of the application flows, network resources will be allocated, and

appropriate DSCP values will be used in those application flows. However, the middleware overhead will remain the same, as the same middleware infrastructure is used, but different DSCP values are added. Applications with network resource reservations will perform with smaller latencies than the applications without network resource reservations, and network QoS can be added with no runtime middleware overhead. Moreover, the priority models provide considerable flexibility in deploying reusable components in different deployment contexts without modifying the software. The following Section 4.2.2 evaluates these capabilities in detail.

4.2.2 Evaluating NetQoPE's QoS Customization Capabilities

Rationale. NetQoPE provides network QoS capabilities to applications without modifying the applications, thereby enhancing flexibility by allowing the deployment of DRE applications in different contexts with different QoS requirements. This experiment empirically validates the adequacy and efficacy of network QoS classes, which is essential to support network QoS in diverse deployment contexts, *i.e.*, without changes in application code.

Methodology. We deployed multiple instances of the same pair of applications, and show the following types of application-transparent QoS customizations: (1) different network QoS classes, (2) same network QoS class, but different bandwidth requirements, and (3) different priority access models. We identified four different flows from Figure 6 as described below:

- Two instances of fire sensor controller components deployed in blades A and B send fire alarm information to the monitor controller components deployed in blades D and F. The fire sensor controller component on blade A monitors fire in the parking lot, whereas fire sensor controller component on blade B monitors the fire in server room. Even though the same set of application components are deployed, the traffic between blades A and D has a higher priority than traffic between blades B and F due to the importance of sensors involved.
- A *camera controller* component deployed in blade E sends imagery information to the monitor controller component in blade G, and requires multimedia QoS. Finally, a *temperature sensor controller* component on blade A sends temperature readings to the monitor controller component on blade F, and does not require any QoS.

We used CQML to model the four different flows in the experiment and their associated network QoS requirements. CQML then synthesized deployment descriptors that capture the DRE system structure and their deployed blades.

In the first experiment, the flows between the fire sensor and monitor controller components were configured to use the high priority (HP) and high reliability (HR) traffic classes. The flow between the camera and monitor controllers was configured to use the multimedia (MM) class, and the flow between the temperature sensor and monitor controller components was configured to use the best effort (BE) class. For each type of QoS traffic, 20 Mbps of forward and reverse bandwidth was requested, TCP transport protocol was used, and NetQoPE's network resource allocator determined the DSCP values to use.

Traffic Type	Background Traffic in Mbps			
	BE	HP	HR	MM
BE (TS - MS)	85 to 100			
HP (FS - MS)	30 to 40		28 to 33	28 to 33
HR (FS - MS)	30 to 40	12 to 20	14 to 15	30 to 31
MM (CS - MS)	30 to 40	12 to 20	14 to 15	30 to 31

Table 1: Application Background Traffic

To add DSCP values to the IP packets, the CLIENT_PROPAGATED network policy was used for all component communication, *except* for the temperature sensor and monitor controller component communication, which used the SERVER_DECLARED network policy. For each application flow, `TestNetQoPE` was configured to generate a load of 20 Mbps, and the average roundtrip latency over 200,000 iterations was calculated. To evaluate application performance in the presence of background network loads, several other applications were run, as described in Table 1, where TS stands for "temperature sensor controller," MS stands for "monitor controller", FS stands for "fire sensor controller," and CS stands for "camera controller."

In the second experiment, all four flows were created with fire sensor and monitor controller components and configured to use the HR class. For each application flow, `TestNetQoPE` was configured to generate a load of 20 Mbps, the average roundtrip latency over 200,000 iterations were calculated, and TCP transport protocol was used. Four application flows were made with the following forward and reverse bandwidth reservations in Mbps: 4, 8, 12, and 16. No background load was generated and the CLIENT_PROPAGATED network policy was used.

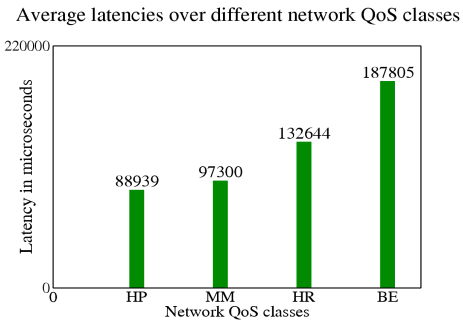

Fig. 8: Average Latency under Different Network QoS Classes

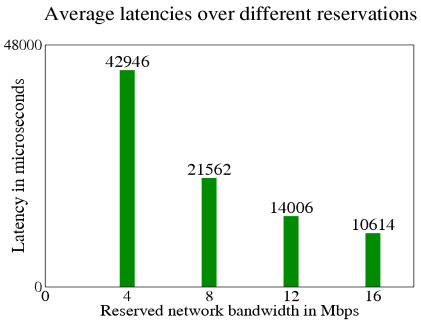

Fig. 9: Average Latency under Different Reservations

Analysis of results. Figure 8 shows the results of experiments when the deployed applications were configured with different network QoS classes. The results show that the average latency experienced by the fire sensor controller component using the HP network QoS class is lower than the average latency experienced by the fire sensor controller component using the HR network QoS class. Moreover, the latency increases while using the MM and BE classes. This result shows that the different network QoS classes supported are adequate to provide varied network QoS performance to DRE applications deployed in different contexts.

Figure 9 shows the results of experiments when the deployed applications were configured with different bandwidth reservations. The figure shows that

the latency increases as the reservation decreases. NetQoPE's network resource allocator used the traffic policing features provided by the *Flow Provisioner* [7] to configure the network devices to allow no more flows than the reserved bandwidth. 20 Mbps are sent and reservations use no more than 16 Mbps. When packet 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 reserved bandwidth size, the higher the delay. These results indicate that NetQoPE can efficiently integrate the DiffServ network QoS mechanisms to provide different levels of network QoS services to applications.

The Figure 8 results also indicate that the same set of application communications (fire sensor and monitor controller components) could get different network QoS depending on their deployed contexts (*e.g.,* HP communication between A and D, and HR communication between blades B and F). These results demonstrate that the declarative capabilities of CQML allow modeling and deploying the same applications (without any source modifications) with different QoS requirements. CQML's "write once, deploy multiple times for different QoS" capabilities greatly increase deployment flexibility for environments like the office enterprise where many reusable software components are deployed.

To provide this flexibility, CQML generated XML-based deployment descriptors that captured context-specific QoS requirements of applications so that context-specific network QoS can be provided. For the experiment, communication between fire sensor and monitor controllers was deployed in multiple deployment contexts, *i.e.*, HR and HP QoS requirements. In DRE systems like the office enterprise case study, however, the same communication patterns between components could occur in many deployment contexts.

For example, the same communication patterns could use any of the four network QoS classes (HP, HR, MM, and BE). The communication patterns that use the same network QoS class (*e.g.*, HP) could make different forward and reverse bandwidth reservations (*e.g.*, 4, 8, 10 Mbps). In such scenarios, as shown in Table 2, CQML auto-generates ∼1,300 lines of XML code, which would otherwise be handcrafted by application developers.

Number of communications	Deployment contexts			
	2	5	10	20
1	23	50	95	185
5	47	110	215	425
10	77	185	365	725
20	137	335	665	1325

Table 2: Generated Lines of Code

As shown in the Figure 8 results, different average latencies are experienced by the fire sensor controller and the temperature sensor controller components when they communicated with the monitor controller component. The fire sensor controller component used the CLIENT_PROPAGATED network policy to dictate the priority of the requests and replies to ensure lower average latency and hence better network QoS. However, when the temperature sensor controller component communicated with the monitor controller component, the latter (server) dictated the priority of the requests and replies using the

SERVER_DECLARED network priority policy. This result shows that the same component (*i.e.*, the monitor controller component) can be accessed using different priority models to dictate different network QoS in the system, thereby enhancing flexibility.

4.2.3 Evaluating NetQoPE's Resource Allocation Capabilities

Rationale. NetQoPE's network resource allocator works in two phases (see Section 3.2.2), where the first phase determines if sufficient network resources are available for all network flows. This experiment empirically validates the admission control capabilities of NetQoPE that enables DRE application designers to change deployment options to assure the required QoS.

Methodology. A fire sensor controller component deployed on blade A sends fire alarm information to the monitor controller component on blade D. At design-time, DRE application designers do not know how many pairs of the fire sensor and monitor controller components can be used to share the network link between blades A and D. We therefore deployed the following number of pairs of fire sensor and monitor controller components between A and D: 1, 2, 3, 4, 5, 10, and 20. The goal is to check if NetQoPE's admission control capabilities can help designers determine the right number of pairs of fire sensor and monitor controller components between A and D, so that all the pairs of communications can obtain the required network QoS.

The allocated link capacity for the HR class between blades A and D is 30 Mbps. We allocated 6 Mbps of forward and reverse bandwidth for each pair of fire sensor and monitor controller component communication. NetQoPE's admission control capabilities were used to deploy up to 4 pairs of fire sensor and monitor controller components. Admission control capabilities were not used when more than 4 pairs of fire sensor and monitor controller components were deployed since deploying 5 pairs will utilize 30 Mbps, which would saturate the link.

The background traffic was kept sufficiently high during this experiment so that the HR communications between blades A and D do not use all the bandwidth from other classes. NetQoPE's network resource allocator provided the DSCP values used for each communication. These allocated values were configured as policies (the CLIENT_PROPAGATED model was used and clients dictated the DSCPs to be used in both the directions) by NetQoPE's policy framework, and the designated DSCP value was added to the IP packets by the CIAO middleware when application components communicated.

Analysis of results. Figure 10 and Figure 11 show the average roundtrip latencies experienced by different pairs of fire sensor and monitor controller components. When admission control is enabled for up to 4 pairs of components, the total link bandwidth used is 24 Mbps (each fire sensor and monitor controller component pair requested for 6Mbps bandwidth). Since the requested bandwidth (24 Mbps) is less than the total available bandwidth (30 Mbps), all 4 pairs of communications obtained the required resources, and thus achieved their required network QoS. Figure 10 shows that all 4 pairs of communicating components obtained similar average latencies.

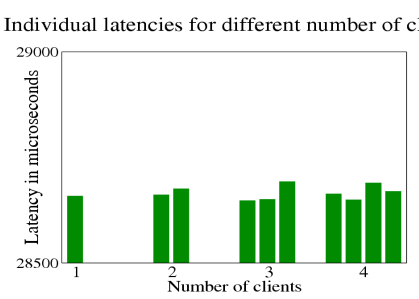

Fig. 10: Average Invocation Latencies with Admission Control

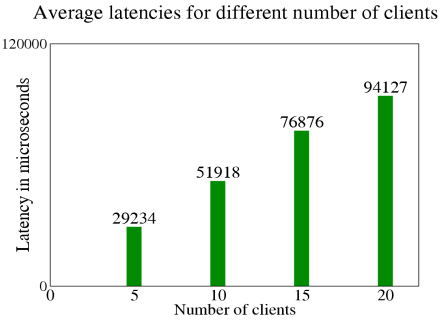

Fig. 11: Average Invocation Latencies without Admission Control

When the experiment was run without admission control, the bandwidth requirements of the different pairs of fire sensor and monitor controller communications far exceeded the total bandwidth (30 Mbps). When 5 pairs of these components were deployed, even though admission control was not used, the required bandwidth (30 Mbps) equaled the total available bandwidth (30 Mbps). Hence in Figure 11 the average latency for 5 pairs of fire sensor and monitor controller component communications is nearly equivalent to the average latency for 4 pairs of these communications shown in figure 10. When more than 5 pairs of fire sensor and monitor controller components are deployed, the required bandwidth exceeds the available bandwidth, causing the applications to compete for the bandwidth. Hence the average latencies for these pairs increase as shown in Figure 11, causing no pairs to meet their required network QoS.

These results demonstrate that NetQoPE's network resource allocator not only shields applications from interacting with network QoS mechanisms, but also provides design-time suggestions on how many pairs of application components can be deployed between two blades. As shown in the experiment, if NetQoPE's admission control capabilities were used throughout, not more than 4 pairs of fire sensor and monitor controller components would have been deployed between blades A and D. For the other pairs, DRE system designers are given the option to change the deployment (*e.g.,* change blades), thereby assuring network QoS. As shown in Figure 11, when more than 5 pairs of fire sensor and monitor controller components were deployed without NetQoPE's admission control capabilities, no pairs obtained their required network QoS.

5 Related Work

This section compares our R&D activities on NetQoPE with related work on middleware-based QoS implementation, management, and enforcement.

QoS management in middleware. Other research has focused on adding various types of QoS capabilities to middleware. For example, [5] illustrates mechanisms added to J2EE containers to allow application isolation by allowing uniform and predictable access to the CPU resources, thereby providing CPU availability assurances to applications. Likewise, 2K [15] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [16] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to

receive desired QoS support. NetQoPE differs from related work on QoS management in middleware by (1) using model-driven tools to specify and verify network QoS capabilities at design time, (2) automating deployment and runtime mechanisms to negotiate and adapt QoS, and (3) enforcing network QoS at runtime in a manner transparent to the application.

Network QoS management in middleware. Earlier work [17, 18] on integrating network QoS with middleware focused on 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. This work, however, modified applications to dictate QoS behavior for the various flows. NetQoPE enhances prior work on network QoS management in middleware by transparently marking packet QoS and configuring network elements for desired QoS in the middleware.

QoS-aware composition of applications. Research has produced QoS architectures for multimedia and stream processing applications that require predictable QoS from endsystems and network elements. For example, Synergy [19] describes a distributed stream processing middleware that provides QoS to data streams in real time by efficient reuse of data streams and processing components. Likewise, [20, 21] focus on appropriate component placement in stream processing applications to exploit efficient use of network resources and maximize query performance. NetQoPE differs from related work on QoS-aware composition of applications via its (1) model-driven tools that reserve QoS at a per-flow level and (2) middleware-based runtime capabilities that integrate and deploy network QoS settings automatically with application communication flows.

QoS-aware deployment of applications. NetQoPE is also related to work on QoS-aware deployment of applications in heterogeneous and dynamically changing distributed environments. For example, GARA [22] focuses on identifying and reserving appropriate network resources to satisfy application requirements. Likewise, Petstore [23] describes how the service usage patterns of J2EE-based web applications can be analyzed to determine how to deploy them in wide-area environments so that access time can be minimized. In addition, [24] focuses on improving J2EE performance by collocating applications that communicate frequently. NetQoPE differs from related work on QoS-aware deployment of applications by providing standard network QoS assurance, rather than trying to collocate components or discern application service usage patterns.

6 Concluding Remarks

This paper describes the design and evaluation of NetQoPE, which is a set of model-driven middleware frameworks that manage network QoS for component-based DRE systems. The following is a summary of our lessons learned:

• NetQoPE's domain-specific modeling languages help capture per-deployment network QoS requirements of applications so that network resources can be allocated appropriately. Application business logic consequently need not be modified to specify deployment-specific QoS requirements, thereby increasing software reuse across a range of deployment contexts.

• Programming network QoS mechanisms directly in application code requires that applications are deployed and running before they can determine if

the required network resources are available to meet QoS needs. Providing these capabilities via NetQoPE's model-driven middleware frameworks helps guide resource allocation strategies *before* application deployment, thereby simplifying validation and adaptation decisions.

• NetQoPE's deployment and configuration tools help transparently configure the underlying middleware on behalf of applications to add context-specific network QoS settings. These settings can be enforced by NetQoPE's runtime middleware mechanisms without modifying the middleware programming model used by applications. Applications consequently need not change the way they communicate at runtime since network QoS settings can be added transparently.

• NetQoPE's strategy of allocating network resources to applications before they are deployed may be too limiting for certain types of DRE systems. In particular, applications in so-called "open" DRE systems might not consume the allocated resources at runtime, which may underutilize system resources. We are extending NetQoPE to deploy more applications even though resources might not be available by providing runtime adaptations that change the service levels of low-priority applications to make resources available for mission-critical applications.

NetQoPE's model-driven middleware platforms and tools are available in open-source format from `www.dre.vanderbilt.edu`.

References

1. L. Zhang and S. Berson and S. Herzog and S. Jamin: Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification. Network Working Group RFC 2205 (September 1997) 1–112
2. Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., Weiss, W.: An Architecture for Differentiated Services. Internet Society, Network Working Group RFC 2475 (December 1998) 1–36
3. Mehra, A., Verma, D.C., Tewari, R.: Policy-based diffserv on internet servers: The AIX approach (on the wire). IEEE Internet Computing **4**(5) (2000) 75–80
4. El-Gendy, M.A., Bose, A., Park, S.T., Shin, K.G.: Paving the first mile for QoS-dependent applications and appliances. In: Proceedings of the International Workshop on Quality of Service (IWQOS'2004), Montreal, Canada (June 2004)
5. Jordan, M., Czajkowski, G., Kouklinski, K., Skinner, G.: Extending a J2EE Server with Dynamic and Flexible Resource Management. In: Proceedings of the International Middleware Conference (Middleware 2004), Toronto, Canada. (2004)
6. Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04). Volume 3291., Agia Napa, Cyprus, Springer-Verlag (October 2004) 1520–1537
7. Dasarathy, B., Gadgil, S., Vaidyanathan, R., Neidhardt, A., Coan, B., Parameswaran, K., McIntosh, A., Porter, F.: 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. Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A model-based approach to system-level dependency and real-time analysis of embedded software. In: Proc. of IEEE Real-time and Embedded Tech. and Applications Symp. (RTAS'03). (2003)

9. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. Elsevier Journal of Computer and System Sciences (2006) 171–185
10. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document mars/03-05-08 edn. (July 2003)
11. Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In: Proc. of the 3rd Working Conf. on Component Deployment (CD 2005), Grenoble, France (2005)
12. Ritter, T., Born, M., Unterschütz, T., Weis, T.: A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In: Proc. of the 36^{th} Hawaii International Conference on System Sciences (HICSS'03), Honolulu, HW (2003)
13. Akkerman, A., Totok, A., Karamcheti, V.: Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In: 3rd International Working Conference on Component Deployment (CD 2005). (2005)
14. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-Integrated Development of Embedded Software. Proceedings of the IEEE **91**(1) (January 2003) 145–164
15. Wichadakul, D., Nahrstedt, K., Gu, X., Xu, D.: 2K: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework. In: Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). (2001)
16. de Miguel, M.A.: 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)
17. 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: Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2003), Rio de Janeiro, Brazil (2003)
18. Florissi, P.G.S., Yemini, Y., Florissi, D.: Qosockets: a new extension to the sockets api for end-to-end application qos management. Comput. Networks **35**(1) (2001) 57–76
19. Repantis, T., Gu, X., Kalogeraki, V.: Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In: Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2006)
20. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proc. of the 22nd International Conference on Data Engineering (ICDE'06). (2006)
21. Seshadri, S., Kumar, V., Cooper, B.F.: Optimizing multiple queries in distributed data stream systems. In: Proc. of the 22nd International Conference on Data Engineering Workshops (ICDEW'06). (2006)
22. Foster, I., Fidler, M., Roy, A., Sander, V., Winkler, L.: End-to-end Quality of Service for High-end Applications. Computer Communications **27**(14) (September 2004) 1375–1388
23. Llambiri, D., Totok, A., Karamcheti, V.: Efficiently Distributing Component-Based Applications Across Wide-Area Environments. In: Proc. of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003). (2003)
24. Stewart, C., Shen, K.: 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)