NetQoPE: A Model-driven Network QoS Provisioning Engine for Distributed Real-time and Embedded Systems

Jaiganesh Balasubramanian[†], Sumant Tambe[†], Balakrishnan Dasarathy[‡],
Shrirang Gadgil[‡], Aniruddha Gokhale[†], and Douglas C. Schmidt[†]
[†]Department of EECS, Vanderbilt University, Nashville, TN, USA
[‡]Telcordia Technologies, Piscataway, NJ, USA

Abstract

This paper provides two contributions to the study of quality of service (QoS)-enabled middleware that supports the network QoS requirements of distributed real-time and embedded (DRE) systems. First, we describe the design and implementation of NetQoPE, which is a model-driven component middleware framework that shields applications from the details of network QoS mechanisms by (1) specifying per-flow network QoS requirements, (2) performing resource allocation and validation decisions (such as admission control), and (3) enforcing per-flow network QoS at runtime. Second, we empirically evaluate NetQoPE's capabilities on a representative DRE system that deploys reusable software code in a range of deployment contexts. Our results demonstrate that NetQoPE can provide network-level differentiated performance to each of those application flows without modifying their programming model or source code, thereby providing greater flexibility and extensibility in leveraging network-layer mechanisms.

1 Introduction

Emerging trends. Distributed real-time and embedded (DRE) systems, such as as shipboard computing systems, supervisory control and data acquisition (SCADA) systems, and enterprise security and hazard sensing subsystems, consist of multiple communication-intensive applications with multiple end-to-end application flows. These systems have network quality of service (QoS) requirements, such as low end-to-end roundtrip latency and jitter, that must be satisfied under varying levels of network connectivity and bandwidth availability. Network QoS mechanisms, such as integrated services (IntServ) [12] and differentiated services (DiffServ) [2], help provide diverse network service levels for applications in DRE systems.

For example, applications can use advanced network QoS mechanisms (*e.g.*, a DiffServ bandwidth broker [3]) to (1) request a network service level and (2) allocate and manage network resources for their remote invocations. Applications invoke remote operations by adding a service level-specific identifier (*e.g.*, DiffServ codepoint (DSCP)) to the IP packets. DiffServ-enabled network routers parse the IP packets and provide the appropriate service level-specific packet forwarding behavior.

Limitations with current approaches. Although advanced network QoS mechanisms are powerful, it is tedious and error-prone to develop applications that interact directly with low-level network QoS mechanism APIs written imperatively in third-generation languages, such as C++ or Java. To overcome this problem, middleware-based solutions [22, 18, 16, 4] have been developed that allow applications to specify their coordinates (source and destination IP and port addresses) and per-flow network QoS requirements via higher-level frameworks. The middleware frameworks—rather than the applications—are responsible for converting the higher-level QoS specifications into the lower-level network QoS mechanism APIs.

Although middleware frameworks alleviate many accidental complexities of low-level network QoS mechanism APIs, they can still be hard to evolve and extend. In particular, application source code changes may be needed whenever changes occur to the deployment contexts (source and destination nodes of the applications), per-flow requirements, IP packet identifiers, or the middleware APIs. What is needed, therefore, are middleware-guided network QoS provisioning solutions that (1) are not tied to a particular network QoS mechanism and (2) do not modify application source code to specify and enforce network QoS requirements. These solutions should ideally operate on well-defined system abstractions (*e.g.*, per-flow requirements and source/destination nodes) that are provided without programmatically modifying the application source code, thereby facilitating application reuse across a wide range of deployment and network QoS contexts.

Solution approach → A model-driven component middleware network QoS provisioning framework that uses declarative domain-specific techniques [1] to raise the level of abstraction of DRE system design higher than using imperative third-generation programming languages. A model-driven framework allows system engineers and software developers to perform deployment-time analysis (such as schedulability analysis [10]) of non-functional system properties (such as network QoS assurances for end-to-end application flows) and helps provide deployment-time assurance that application QoS requirements will be satisfied.

This paper describes the *Network QoS Provisioning Engine* (NetQoPE), which is a model-driven component middleware framework that deploys and configures applications in DRE systems and enforces their network QoS requirements using the four-stage design-, pre-deployment-, deployment- and runtime approach shown in Figure 1. The innovative

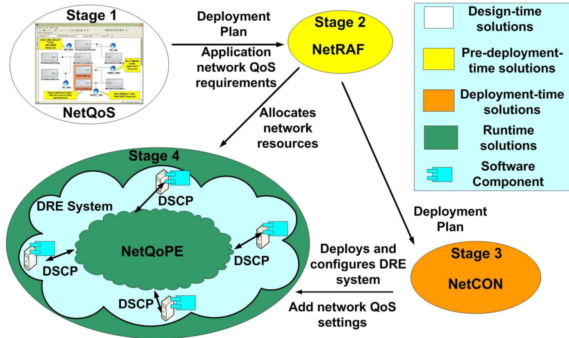


Figure 1: NetQoPE's Four-stage Architecture

elements of NetQoPE's four-stage architecture include the following:

- The **Network QoS Specification Language** (NetQoS), which is a domain-specific modeling language (DSML) that supports design-time specification of per-flow network QoS requirements, such as bandwidth and delay across a flow. By allowing application developers to focus on functionality—rather than the different deployment contexts (*e.g.*, different bandwidth and delay requirements) when they will be used—NetQoS simplifies the deployment of applications in contexts that require different network QoS requirements, *e.g.*, different bandwidth requirements.
- The **Network Resource Allocation Framework** (NetRAF), which is a middleware-based resource allocator framework that uses the network QoS requirements captured by *NetQoS* as input at pre-deployment time to help guide QoS provisioning requests on the underlying network QoS mechanism at deployment time. By providing application-transparent, per-flow resource allocation capabilities at pre-deployment-time, *NetRAF* minimizes runtime overhead and simplifies validation decisions, such as admission control.
- The **Network QoS Configurator** (NetCON), which is a middleware-based network QoS configurator that provides deployment-time configuration of component middleware containers, which at runtime add flow-specific identifiers (*e.g.*, DSCPs) to IP packets when applications invoke remote operations. By providing container-mediated and application-transparent capabilities to enforce runtime network QoS, NetCON allows DRE systems to leverage the QoS services of configured routers without modifying application source code. As shown in the Figure 1, the output of each stage in NetQoPE serves as input for the next stage, which helps automate the deployment and configuration of DRE applications with network QoS support.

Paper organization. The remainder of the paper is organized as follows: Section 2 describes a case study to motivate common requirements associated with provisioning network QoS for DRE systems; Section 3 explains how NetQoPE addresses those requirements via a model-driven component middleware framework; Section 4 empirically evaluates the capabilities provided by NetQoPE; 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

Figure 2 shows a representative DRE system in an office enterprise security and hazard sensing environment, which we use as a case study to demonstrate and evaluate NetQoPE's model-driven, middleware-guided network QoS provisioning capabilities. Enterprises often transport net-

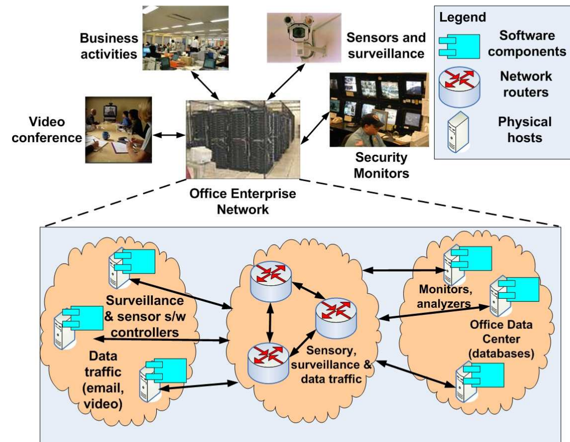


Figure 2: Network Configuration in an Enterprise Security and Hazard Sensing Environment

work 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 on office premises. Our case study makes the common assumption that safety/security traffic is more critical than other traffic, and thus focuses on model-driven, middleware-guided mechanisms to assure the specified QoS for this type of traffic in the presence of other traffic that shares the same network.

As shown in Figure 2, our case study uses software controllers to manage 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. These software controllers were developed using

Lightweight CCM (LwCCM) [14] and the traffic between these software controllers uses a bandwidth broker [3] to manage network resources via DiffServ network QoS mechanisms. Although this case study focuses on DiffServ and LwCCM, NetQoPE is designed for use with other network QoS mechanisms (*e.g.*, IntServ) and component middleware technologies (*e.g.*, J2EE).

Component-based applications in our case study obtain the services of the bandwidth broker via the following middleware-guided steps: (1) network QoS requirements are specified on each application flow, along with information on the source and destination IP and port addresses, (2) the bandwidth broker is invoked to reserve network resources along the network paths for each application flow, configure the corresponding network routers, and obtain per-flow DSCP values to help enforce network QoS, and (3) remote invocations are made with appropriate DSCP values added to the IP packets so that configured routers can provide per-flow differentiated performance. Section 3 describes the challenges we encountered when implementing these steps in the context of our case study and shows how NetQoPE's four-stage architecture shown in Figure 1 resolves these challenges.

3 NetQoPE's Multistage Network QoS Provisioning Architecture

As discussed in Section 1, conventional techniques for providing network QoS to applications incur several key limitations, including modifying application source code to (1) specify deployment context-specific network QoS requirements, and (2) integrate functionality from network QoS mechanisms at runtime. This section describes how NetQoPE addresses these limitations via its model-driven, middleware-guided network QoS provisioning architecture.

3.1 Challenge 1: Alleviating Complexities in QoS Requirements Specification

Context. For each application flow, DRE systems must specify a required level of service (*e.g.*, high priority vs. low priority), the source and destination IP and port addresses, and bandwidth and delay requirements, so that network resources are allocated and configured to provide the required QoS.

Problem. Network QoS requirements (such as the bandwidth and delay requirements mentioned above) can change depending on a deployed context. For example, in our case study from Section 2, multiple fire sensors are deployed at different importance levels and each sensor sends its sensory information to its corresponding monitors. A fire sensor deployed in the parking lot has a lower importance than those in the server room. The sensor-monitor flows thus have different network QoS requirements, even though the reusable software controllers managing the fire sensor and the monitor have the same functionality.

The use of conventional techniques, such as hard-coded API approaches [4], requires application source code modifications for each context. Writing this code manually to specify network QoS requirements is tedious, error-prone, and non-scalable. In particular, it is hard to envision at development time all the contexts in which the source code will be deployed.

Sidebar 1: Overview of Lightweight CORBA Component Model (LwCCM)

Application functionality in LwCCM is provided through *components* which collaborate with other components via *ports* to create component *assemblies*. Assemblies in LwCCM are described using XML descriptors (mainly the *deployment plan* descriptor) defined by the OMG D&C [15] specification. The *deployment plan* includes details about the components, their implementations, and their connections with other components. The *deployment plan* also has a placeholder *configProperty* that is associated with elements (*e.g.*, components, connections) to specify their properties (*e.g.*, priorities) and resource requirements. Components are hosted in *containers*, which provide the appropriate runtime operating environment (*e.g.*, transactions support) for components to invoke remote operations.

Solution approach → Model-driven visual network requirements specification. NetQoPE provides a DSML called the *Network QoS Specification Language* (NetQoS). Using NetQoS, DRE system developers (1) model component assemblies, (2) assign target node assignments for components, and (3) declaratively specify the following deployment context-specific network QoS requirements on the modeled application flows: (a) network QoS classes, such as HIGH PRIORITY (HP), HIGH RELIABILITY (HR), MULTIMEDIA (MM), and BEST EFFORT (BE), (b) bi-directional bandwidth and delay requirements, and (c) selection of transport protocol.

In the context of our case study, NetQoS's network QoS classes correspond to the DiffServ levels of service provided by our Bandwidth Broker [3].[1] For example, the HP class represents the highest importance and lowest latency traffic (*e.g.*, fire sensing reporting in the server room). The HR class represents traffic with low drop rate (*e.g.*, surveillance data). NetQoS also supports the MM class for sending multimedia data and the BE class for sending traffic with no QoS requirements.

After a model has been created, NetQoS's model interpreter traverses the modeled application structure and generates a *deployment plan* (described in Sidebar 1).

[1] NetQoS's DSML capabilities can be extended to provide requirements specification conforming to a different network QoS mechanism, such as IntServ.

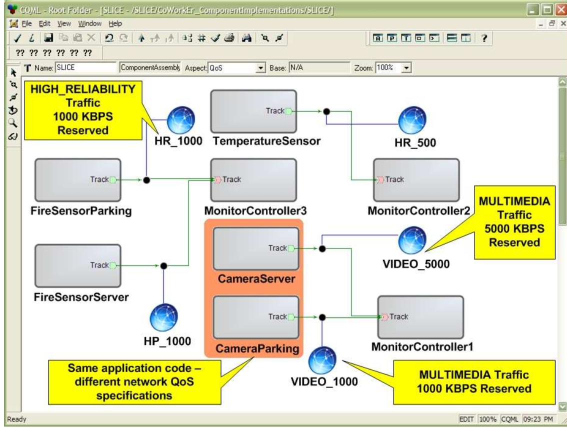


Figure 3: NetQoS Capabilities

NetQoS's model interpreter also traverses each modeled application flow and augments the *deployment plan config-Property* tags (also described in Sidebar 1) to express network QoS requirement annotations on the component connections. Section 3.2 describes how network resources are allocated based on requirements specified in the deployment plan descriptor.

Our case study has certain application flows (*e.g.*, a monitor requesting location coordinates from a fire sensor) where the client (monitor) controls the network priorities at which the requests and replies are sent. This capability enables real-time actions irrespective of network congestion. There are other examples (*e.g.*, a temperature sensor sends temperature sensory information to the monitors) where the server controls the network priorities at which the requests and replies are sent. This capability prevents misuse of DiffServ priority classes by clients, thereby avoiding unnecessary network congestion.

To support these two models, NetQoS can assign the following priority attributes to connections: (1) CLIENT_PROPAGATED network priority model, that allows the clients to dictate the bi-directional priorities, and (2) SERVER_DECLARED network priority model, that allows the server to dictate the bi-directional priorities. NetQoS's model interpreter updates the deployment plan with these priority models for each of the flows, and Section 3.3 explains how NetQoPE's runtime mechanisms honor these priority models when applications invoke remote operations.

Application to the case study. Figure 3 shows a NetQoS model highlighting many of its key capabilities. Multiple instances of the same reusable application components (*e.g.*, FireSensorParking and FireSensorServer components) can be annotated with different QoS attributes using an intuitive drag and drop technique. This method of specifying QoS requirements is thus much simpler than modifying application code for each deployment context, as demonstrated in Section 4.2.1. Moreover, the same QoS attribute (*e.g.*, HR_1000 in Figure 3) can potentially be reused across multiple connections. NetQoS thus increases the scalability of expressing requirements for large numbers of connections that are prevalent in large-scale DRE systems, such as our case study.

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

Context. DRE systems must allocate and configure network resources based on the QoS requirements specified on their application flows so that network QoS assurances can be provided at runtime.

Problem. In our case study, the temperature sensory information from the server room is more important than the information from a conference room. It is not desirable, however, to modify the temperature sensor software controller code to directly interact with a middleware API or network QoS mechanism API since certain deployment contexts (such as the deployment in a conference room) might not require network QoS assurances. Moreover, if application source code is modified to provide resource allocations, decisions on whether to allocate resources or not cannot be determined until the applications are deployed and operational. This approach forces DRE system deployers to stop application components and deploy them on different nodes if required resources cannot be allocated across the source and destination nodes.

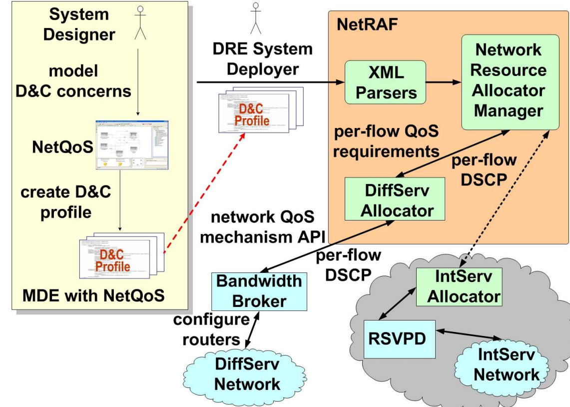


Figure 4: NetRAF's Network Resource Allocation Capabilities

Solution approach → Middleware-based Resource Allocator Framework. NetQoPE's *Network Resource Allocator Framework* (NetRAF) is a resource allocator engine that can provide network resource allocations for DRE systems using a variety of network QoS mechanisms, such as DiffServ and IntServ. As shown in Figure 4, the NetQoS DSML described in Section 3.1 captures the modeled per-flow network QoS requirements in the form of a *deployment plan* that is input to NetRAF.

The modeled deployment context could have many instances of the same reusable source code, such as the tem-

perature sensor software controller is instantiated two times, one for the server room, and one for the conference room. When using NetQoS, however, application developers only annotate the connection between the instance at the server room and the monitor software controller. Since NetRAF operates on the *deployment plan* that captures this modeling effort, network QoS mechanisms are used only for the connection on which QoS attributes are added. NetRAF thus improves conventional approaches [18] that modify application source code to work with network QoS mechanisms, which can become complex when source code is reused in a wide range of deployment contexts.

NetRAF's *Network Resource Allocator Manager* accepts application QoS requests at pre-deployment-time and determines the network QoS mechanism (*e.g.*, DiffServ or IntServ) to use to serve the requests. As shown in Figure 4, NetRAF's Network Resource Allocator Manager works with QoS mechanism-specific allocators (*e.g.*, DiffServ Allocator), which shields it from interacting directly with the complex network QoS mechanism (*e.g.*, DiffServ Bandwidth Broker) APIs, thereby enhancing NetQoPE's flexibility and extensibility.

Multiple allocators (*e.g.*, IntServ Allocator and DiffServ Allocator) can be used by NetRAF's Network Resource Allocator Manager to serve the needs of small-scale deployments (where IntServ and DiffServ are both suitable) and large-scale deployments (where DiffServ often provides better scalability). For example, the shaded cloud connected to the Network Resource Allocator Manager in Figure 4 shows how NetRAF can be extended to work with other network QoS mechanisms, such as IntServ.

Application to the case study. Since our case study is based on DiffServ, NetRAF uses the *DiffServ Allocator* to allocate network resources. This allocator invokes the admission control capabilities of the Bandwidth Broker [3] by feeding it one application flow at a time. If all flows *cannot* be admitted, NetRAF allows developers an option to change the deployment context since applications have not yet been deployed. Example changes include changing component implementations to consume fewer resources or change the source and destination nodes. As demonstrated in Section 4.2.3, this capability helps NetRAF incur lower overhead than conventional approaches [22, 18] that perform validation decisions when applications are deployed and operated at runtime.

NetRAF's DiffServ Allocator instructs the Bandwidth Broker to reserve bi-directional resources in the specified classes. The Bandwidth Broker determines the bi-directional DSCPs and NetRAF encodes those values as connection attributes in the deployment plan. In addition, the Bandwidth Broker uses its *Flow Provisioner* [3] to configure the routers to provide appropriate per-hop behavior when they receive IP packets with the specified DSCP values. Section 3.3 describes how component containers are auto-configured to add these DSCPs when applications invoke remote operations.

3.3 Challenge 3: Alleviating Complexities in Network QoS Settings Configuration

Context. After network resources are allocated and network routers are configured, applications in DRE systems need to invoke remote operations using the chosen network QoS settings (*e.g.*, DSCP markings) so that the network layer can differentiate application traffic and provision appropriate QoS to each of the flow.

Problem. Application developers have historically written code that instructs the middleware to provide the appropriate runtime services, *e.g.*, DSCP markings in IP packets [16]. For example, fire sensors in our case study from Section 2 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 is 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 ideally 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.

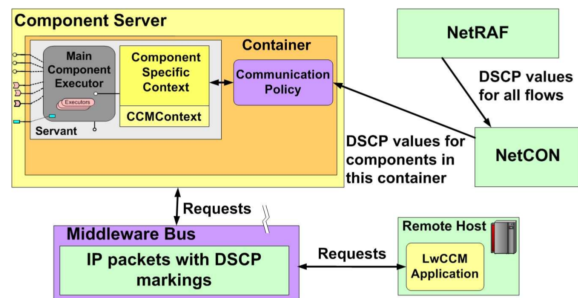


Figure 5: NetCON's Container Auto-configurations

Solution approach → Deployment and runtime component middleware mechanisms. Sidebar 1 describes how LwCCM containers provide a runtime environment for components. NetQoPE's *Network QoS Configurator* (NetCON) provides capabilities to auto-configure these containers to add DSCPs to IP packets when applications invoke remote operations. As shown in Figure 5, NetRAF performs network resource allocations, determines the bi-directional DSCP values to be used for each application flow, and encodes those DSCP values in the deployment plan.

During deployment, NetCON parses the deployment plan and its connection tags to determine (1) source and destination components, (2) the network priority model to be

used for their communication, (3) the bi-directional DSCP values, and (4) the target nodes on which the components are deployed. NetCON deploys the components on their respective containers, and creates the associated object references that can be used by clients in a remote invocation. When a component invokes a remote operation in LwCCM its container's context information provides it the object reference of the destination component. Other component middleware provide similar capabilities via containers, *e.g.*, EJB applications interact with containers to obtain the right runtime operating environment.

The NetCON container programming model can transparently add DSCPs and enforce the network priority models described in Section 3.1. To support SERVER_DECLARED network priority model, NetCON encodes a SERVER_DECLARED policy, and the associated request and reply DSCPs on the object reference of the server. When a client invokes a remote operation with this object reference, the client-side middleware checks the policy on the object reference, decodes the request DSCP and sends it on the request IP packets. In the server-side middleware, before sending the reply, the policy is checked again, and the reply DSCP is added on the IP packets.

To support CLIENT_PROPAGATED network priority model, NetCON configures the containers to apply a CLIENT_PROPAGATED policy at the point of binding an object reference with the client. In contrast to the SERVER_DECLARED policy, the CLIENT_PROPAGATED policy can be changed at runtime and different clients can access the servers with different network priorities. When the source component invokes a remote operation using the policy-applied object reference, NetCON adds the associated forward and reverse DSCP markings on the IP packets, thereby providing network QoS to the application flow. A container can therefore transparently add both forward and reverse DSCP values when components invoke remote operations using the container services.

Application to the case study. NetCON allows DRE system developers to focus on their application business logic, rather than wrestling with low-level mechanisms for provisioning network QoS. Moreover, NetCON provides these capabilities without having the applications to modify their application code, which simplifies development without incurring runtime overhead, as described in Section 4.2.2.

4 Evaluating NetQoPE

This section empirically evaluates the flexibility and overhead of using NetQoPE to provide network QoS assurance to end-to-end application flows. We first validate that NetQoPE's automated model-driven approach can provide differentiated network performance for a variety of applications in DRE systems, such as our case study. We then demonstrate that NetQoPE's network QoS provisioning capabilities significantly reduce application development effort incurred by conventional approaches.

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 dual CPU blades in ISISlab, where 8 blades hosted linux router software. The remaining 8 blades

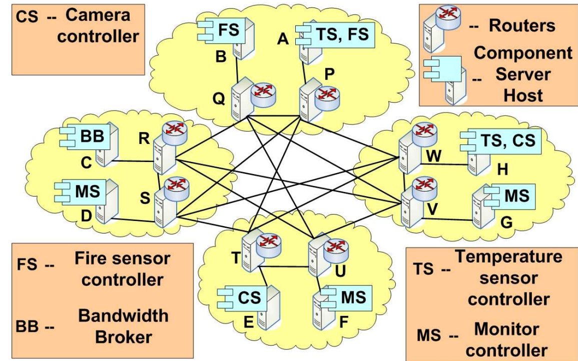


Figure 6: Experimental Setup

hosted software controllers (*e.g.*, a fire sensor controller) developed using the CIAO middleware, which is an open-source LwCCM implementation developed on top of TAO real-time CORBA Object Request Broker (ORB). Our evaluations used DiffServ QoS and the associated Bandwidth Broker [3] 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 via virtual 100 Mbps links.

In our evaluation scenario, a number of sensory and imagery software controllers sent their monitored information to monitor controllers so that appropriate control actions could be performed by enterprise supervisors monitoring abnormal events. For example, Figure 6 shows several *fire sensor controller* components deployed on blades A and B. These components sent their monitored information to *monitor controller* components deployed on blades D and F. communication between these software controllers used one of the traffic classes defined in Section 3.1 with the following capacities on all links: HP = 20 Mbps, HR = 30 Mbps, and MM = 30 Mbps. The BE class used the remaining available bandwidth in the network.

To emulate the network traffic behavior of the software controllers developed using NetQoPE, we developed the `TestNetQoPE` performance test. This test creates a session for component-to-component communication with configurable bandwidth consumption. High resolution timer

probes we used to measure roundtrip latency accurately for each invocation made by a 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 NetQoPE's QoS Customization Capabilities

Rationale. NetQoPE's model-driven approach provides the flexibility of developing application source code once and reusing it multiple times in different deployment contexts. It can also address the QoS needs of a wide variety of applications by supporting multiple DiffServ classes and network priority models. This experiment empirically evaluates the benefits of these capabilities.

Methodology. We identified four flows from Figure 6 and modeled them using NetQoS as follows: (1) a fire sensor controller component on blade A uses the high reliability (HR) class and sends potential fire alarms in the parking lot to monitor controller component on blade D, (2) a fire sensor controller component on blade B uses the high priority (HP) class and sends potential fire alarms in the server room to monitor controller component on blade F, (3) a camera controller component on blade E uses the multimedia (MM) class and sends imagery information of the break room to the monitor controller component on blade G, and (4) a temperature sensor controller component on blade A uses the best effort (BE) class and sends temperature readings to the monitor controller component on blade F. CLIENT_PROPAGATED network policy was used for all flows, *except* for the the temperature sensor and monitor controller component flow, which used the SERVER_DECLARED network policy.

We performed two variants of this experiment. The first variant used TCP as the transport protocol and 20 Mbps of forward and reverse bandwidth was requested for each type of QoS traffic. 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. The second variant used UDP as the transport protocol and `TestNetQoPE` was configured to make *oneway* invocations with a payload of 500 bytes for 100,000 iterations. We used high-resolution timer probes to measure the network delay for each invocation on the receiver side of the communication.

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

In both the experiments, 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"). NetRAF allocated the network resources for each flow and determined the DSCP values to use. After deploying the applications, NetCON configured the containers to use the appropriate network priority models to add DSCP values to IP packets when applications invoke remote operations.

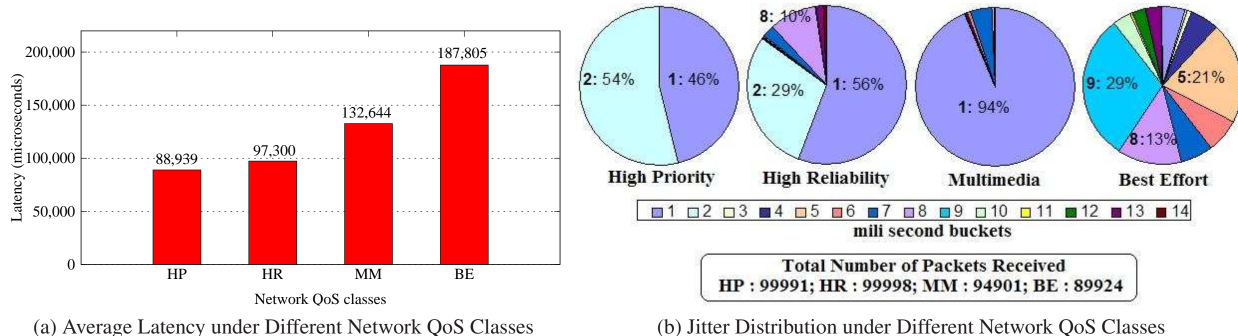
Traffic Type	Background Traffic in Mbps			
	BE	HP	HR	MM
BE (TS - MS)	85 to 100			
HP (FS - MS)	30 to 40		28 to 33	28 to 33
HR (FS - MS)	30 to 40	12 to 20	14 to 15	30 to 31
MM (CS - MS)	30 to 40	12 to 20	14 to 15	30 to 31

Table 1: Application Background Traffic

Analysis of results. Figure 7a shows the results of experiments when the deployed applications were configured with different network QoS classes and were sending TCP traffic. This figure shows that irrespective of the heavy background traffic, the average latency experienced by the fire sensor controller component using the HP network QoS class is lower than the average latency experienced by all other components. In contrast, the traffic from the BE class does not get differentiated from the competing background traffic and incurs a high latency (*i.e.*, throughput is very low). Moreover, the latency increases while using the HR and MM classes when compared to the HP class.

Figure 7b shows the (1) cardinality of the network delay groupings for different network QoS classes under different millisecond buckets and (2) losses incurred by each network QoS class. These results show that the jitter values experienced by the application using the BE class are spread across all the buckets (*i.e.*, are highly unpredictable). When combined with packet or invocation losses, this property is undesirable in DRE systems. In contrast, predictability and loss-ratio improves when using the HP class as evidenced by the spread of network delays across just two buckets. The application's jitter is almost constant and is not affected by heavy background traffic.

The results in Figure 7b also show that application using the MM class experiences predictable latency than applications using BE and HR class. Approximately 94% of the MM class invocations had their normalized delays within 1



(a) Average Latency under Different Network QoS Classes

(b) Jitter Distribution under Different Network QoS Classes

Figure 7: Performance of NetQoPE

millisecond. This result occurs because the queue size at the routers is smaller for the MM class than the queue size for the HR class, so UDP packets sent by the invocations do not experience as much queuing delay in the core routers as packets belonging to the HR class. The HR class provides better loss-ratio, however.

These results demonstrate that NetQoPE's automated model-driven mechanisms (1) support the needs of a wide variety of applications by simplifying the modeling of QoS requirements via various DiffServ network QoS classes and (2) provide those modeled applications with differentiated network performance validating the automated network resource allocation and configuration process. By using NetQoPE, applications can leverage the functionalities of network QoS mechanisms with minimal effort (as described in Section 4.2.3).

The results also demonstrated the following QoS customization possibilities for a set of application communications (*e.g.*, fire sensor and monitor controller component): (1) *different network QoS performance*, *e.g.,* HP communication between blades A and D, and HR communication between blades B and F, (2) *different transport protocols for communication*, *e.g.*, TCP and UDP, and (3) *different network access models*, *e.g.*, monitor controller components were accessed using the CLIENT_PROPAGATED network priority model and the SERVER_DECLARED network priority model.

Taken together, these results demonstrate that NetQoPE's "write once, deploy multiple times for different QoS" capabilities increase deployment flexibility and extensibility for environments where many reusable software components are deployed. To provide this flexibility, NetQoS generates XML-based deployment descriptors that capture context-specific QoS requirements of applications. For our 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 our 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, NetQoS auto generates $\sim$1,300 lines of XML code, which would otherwise need to 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 XML Code

4.2.2 Evaluating the Overhead of NetQoPE for Normal Operations

Rationale. NetQoPE provides network QoS to applications by using the four-stage architecture shown in Figure 1. This experiment evaluates the overhead of using NetQoPE to enforce network QoS.

Methodology. As described in Section 3.1, DRE system developers can use NetQoPE at design time to specify network QoS requirements on the application flows. Based on the specified network QoS requirements, NetRAF interacts with the Bandwidth Broker at pre-deployment time to allocate per-flow network resources. By providing design- and pre-deployment-time capabilities, NetQoS and NetRAF thus incur no runtime overhead. In contrast, NetCON provides deployment-time configuration of component middleware containers by adding DSCP markings to IP packets when applications invoke remote operations, as described in Section 3.3. There is thus the potential for runtime overhead when containers apply one of the network policy models to provide the the source application with an object reference to the destination application.

To measure the runtime overhead incurred by NetCON, we ran an experiment to determine the runtime over-

head of the container when it performs extra work to apply the policies to add DSCPs to IP packets. This experiment had the following variants: (1) the client container not configured by NetCON (no network QoS required), (2) the client container configured by NetCON to apply the CLIENT_PROPAGATED network policy, and (3) the client container configured by NetCON to apply the SERVER_DECLARED network policy. All experiment variants had no background network load.

In our experiment, the network priority models were configured with DSCP values of 0 for both the forward and reverse direction flows, as there was no network congestion and QoS support was not needed. `TestNetQoPE` was configured to make 200,000 invocations that generated a load of 6 Mbps, and average roundtrip latency was calculated for each experiment variant. The routers were not configured to perform DiffServ processing (provide routing behavior based on the DSCP markings), and hence no edge router processing overhead was incurred. We configured the experiment to pinpoint only the overhead of the container and not of any other entity in the path of the remote communication invoked by the clients.

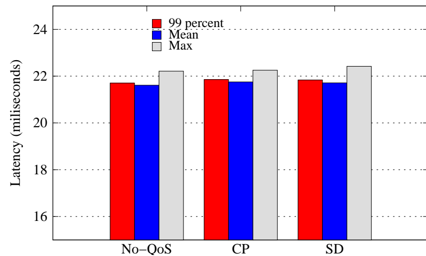


Figure 8: Overhead of NetQoPE's Policy Framework

Analysis of results. Figure 8 (CP refers to CLIENT_PROPAGATED and SD refers to SERVER_DECLARED network priority models) shows the different average roundtrip latencies experienced by clients in the three different variants of the experiment. To honor the network policy models, the NetQoPE middleware added the request and reply DSCPs to the IP packets. The latency results shown in Figure 8 are all similar, which shows that NetCON is efficient and adds negligible overhead to applications.

Since Network QoS was not needed for this experiment the 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 the appropriate DSCP values will be used in those application flows. The middleware overhead will remain the same, however, since the same middleware infrastructure is used, only with different DSCP values. This result thus shows that NetCON incurs minimal runtime overhead when enforcing network QoS support for applications.

4.2.3 Evaluating NetQoPE's Model-driven QoS Provisioning Capabilities

Rationale. As discussed in Section 3, a key design goal of NetQoPE is to provide network QoS to applications in an extensible manner. This experiment evaluates NetQoPE's application-transparent network QoS provisioning capabilities.

Methodology. We first define a taxonomy for evaluating technologies that provide network QoS assurances to end-to-end DRE application flows. Conventional approaches can be classified as being (1) object-oriented [8, 18, 22, 16], (2) aspect-oriented [7], and (3) component middleware-based [4, 19]. Below we describe how each approach provide the following functionalities needed to leverage network QoS mechanism capabilities:

- **Requirements Specification.** In conventional approaches applications use (1) middleware-based APIs [8, 22], (2) contract definition languages [18, 16], (3) run-time aspects [7], or (4) specialized component middleware container interfaces [4] to specify network QoS requirements. Whenever the deployment context and the associated QoS requirements change, however, application source code must also change, thereby limiting reusability. In contrast, as described in Section 3.1, NetQoS provides domain-specific, declarative techniques that alleviate the need to programmatically specify QoS requirements and increase reusability across different deployment contexts.
- **Network Resource Allocation.** Conventional approaches require the deployment of applications before their per-flow network resource requirements can be provisioned by network QoS mechanisms. If those applications cannot have their required resources allocated they must be stopped, their source code must be modified to specify new resource requirements, and the resource reservation process needs to start again. This approach is tedious since it involves deploying and re-deploying applications (potentially in different nodes) multiple times. In contrast, NetRAF handles deployment changes through NetQoS models, as described in Section 3.2. This process occurs during pre-deployment before applications have been deployed, which reduces the efforts needed to change deployment topology or application QoS requirements.
- **Network QoS Enforcement.** Conventional approaches modify application source code [16] or programming model [4] to instruct the middleware to enforce runtime QoS for their remote invocations. Applications must therefore be designed to handle two different usecases—to enforce QoS and when no QoS is required—thereby limiting application reusability. In contrast, as described in Section 3.3, NetCON uses a container programming model that transparently enforces runtime QoS for applications without

changing their source code or programming model.

Using the conventional approaches and the NetQoPE approach, we now compare the manual effort required to provide network QoS to the four end-to-end application flows described in Section 4.2.1. We decompose the manual effort across the following general steps: (1) *implementation*, which involves software engineers writing code, (2) *deployment*, which involves the system deployers to map (or stop) application components to their target nodes, and (3) *modeling tool use*, which involves the application developers to use NetQoPE to model a DRE application structure and specify per-flow QoS requirements. In the context of our evaluation, a complete QoS provisioning lifecycle consists specifying requirements, allocating resources, deploying applications, and stopping applications when they are finished.

To compare the manual efforts, we devised a realistic scenario for the four end-to-end application flows described in Section 4.2.1. In this scenario, three sets of experiments are conducted with the following different deployment variants:

• In the first variant, all the four end-to-end application flows are configured with the QoS requirements as specified in Section 4.2.1.

• In the second variant, to demonstrate the effect of changes in QoS requirements on manual efforts we modify bandwidth requirements from 20 Mbps to 12 Mbps for each of the four end-to-end flows.

• In the third variant, we demonstrate the effect of changes in QoS requirements and resource (re)reservations taken together on manual efforts. We modify bandwidth requirements of all the flows from 12 Mbps to 16 Mbps. We also change temperature sensor controller component to use the high reliability (HR) class instead of the best effort BE class as described in Section 4.2.1. We also increased the background HR class traffic across the blades, so that the resource reservation request for the flow between temperature sensor and monitor controller components fails. In response, deployment contexts (*e.g.*, bandwidth requirements, source and destination nodes) were changed and resource re-reservation was performed.

For the first deployment, the effort required using conventional approaches is the following 10 steps: (1) modify source code of each of the eight components to specify their QoS requirements (8 implementation steps), (2) deploying all the components (1 deployment step), and (3) shutdown all the components (1 deployment step). The effort required using NetQoPE involves the following 4 steps: (1) model the DRE application structure of all the 4 end-to-end application flows using NetQoS (1 modeling step), (2) annotate QoS specifications on each of the end-to-end application flow (1 modeling step), (3) deploying all the components (1 deployment step), and (4) shutdown all the components (1 deployment step).

For the second deployment, the effort required using a conventional approach is also 10 steps because this approach require source code modifications as the deployment contexts changed (in this case, the bandwidth requirements changed across four different deployment contexts). In contrast, the effort required using NetQoPE is 3 steps and is described as follows: (1) annotate QoS specifications on each of the end-to-end application flow (1 modeling step), (3) deploying all the components (1 deployment step), and (4) shutdown all the components (1 deployment step). For the second deployment, application developers reused the NetQoS application structure model that was created for the initial deployment and this helps reduce required efforts by a step.

For the third deployment, the effort required using a conventional approach is the following 13 steps: (1) modify source code of each of the eight components to specify their QoS requirements (8 implementation steps), (2) deploying all the components (1 deployment step), (3) shutdown the temperature sensor component (1 deployment step – resource allocation failed for the component), (4) modify source code of temperature sensor component back to use BE network QoS class (deployment context change) (1 implementation steps), (5) redeploy the temperature sensor component (1 deployment step), and (6) shutdown all the components (1 deployment step).

In contrast, the effort required using NetQoPE for the third deployment is the following 4 steps: (1) annotate QoS specifications on each of the end-to-end application flow (1 modeling step), (2) re-annotate QoS requirements for the temperature sensor component flow (1 deployment step – NetRAF's pre-deployment-time allocation capabilities determined the resource allocation failure and prompted NetQoPE application developer to change the QoS requirements) (3) deploying all the components (1 deployment step), and (4) shutdown all the components (1 deployment step).

Approaches	# Steps in Experiment Variants		
	First	Second	Third
NetQoPE	4	3	4
Conventional	10	10	13

Table 3: Comparison of Manual Efforts Incurred in Conventional and NetQoPE Approaches

As shown in Table 3, the results from this exercise show that conventional approaches incur roughly an order of magnitude more effort than NetQoPE to provide network QoS assurance for end-to-end application flows. Closer examination shows that in conventional approaches, application developers spend substantially more effort designing and implementing software that can work across different

deployment contexts. Moreover, this process must be repeated as and when the deployment contexts and the associated QoS requirements change. Moreover, implementations are complex since the requirements are specified using external APIs, such as middleware-based APIs [22] or network QoS mechanism APIs [12].

Further, application (re)deployments are required whenever reservation requests fail. In this experiment, only one flow required re-reservation and that incurred additional effort of 3 steps. If there are large number of flows —and enterprise DRE systems like our case study tend to have dozens or hundreds of flows—the level of effort required is significantly more than for conventional approaches.

5 Related Work

This section compares our R&D activities on NetQoPE with related work on middleware-based QoS management and model-based design tools.

Network QoS management in middleware. Prior work on integrating network QoS mechanisms with middleware [22, 18, 16, 8] focused on providing middleware APIs to shield applications from directly interacting with complex network QoS mechanism APIs. Middleware frameworks transparently converted the specified application QoS requirements into lower-level network QoS mechanism APIs and provided network QoS assurances. These approaches, however, modified applications to dictate QoS behavior for the various flows. NetQoPE differs from these approaches by providing application-transparent and automated solutions to leverage network QoS mechanisms, thereby significantly reducing manual design and development effort to obtain network QoS.

QoS management in middleware. Prior research has focused on adding various types of QoS capabilities to middleware. For example, [11] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [24] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [4] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. Synergy [17] describes a distributed stream processing middleware that provides QoS to data streams in real time by efficient reuse of data streams and processing components. These approaches are restricted to CPU QoS assurances or application-level adaptations to resource-constrained scenarios. NetQoPE differs by providing network QoS assurances in a application-agnostic fashion.

Deployment-time resource allocation. Prior work has focused on deploying applications at appropriate nodes so that their QoS requirements can be met. For example, prior work [13, 21] has studied and analyzed application communication and access patterns to determine collocated placements of heavily communicating components. Other research [6, 9] has focused on intelligent component placement algorithms that maps components to nodes while satisfying their CPU requirements. NetQoPE differs from these approaches by leveraging network QoS mechanisms to allocate network resources at pre-deployment-time and enforcing network QoS at runtime.

Model-based design tools. Prior work has been done on model-based design tools. PICML [1] enables DRE system developers to define component interfaces, their implementations, and assemblies, facilitating deployment of LwCCM-based applications. VEST [20] and AIRES [10] analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provides automated allocation of components to processors. SysWeaver [5] supports design-time timing behavior verification of real-time systems and automatic code generation and weaving for multiple target platforms. In contrast, NetQoPE provides model-driven capabilities to specify network QoS requirements on DRE system application flows, and subsequently allocate network resources automatically using network QoS mechanisms. NetQoPE thus helps assure that application network QoS requirements are met at deployment-time, rather than design-time or runtime.

6 Concluding Remarks

This paper describes the design and evaluation of NetQoPE, which is a model-driven component middleware framework that manages network QoS for applications in DRE systems. The following is a summary of the lessons we learned developing NetQoPE and applying it to a representative DRE system case study:

- 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 and flexibility 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 framework helps to guide resource allocation strategies *before* application deployment, thereby simplifying validation and adaptation decisions.
- NetQoPE's model-driven deployment and configuration tools help transparently configure the underlying component middleware on behalf of applications to add context-specific network QoS settings. These settings can be enforced by NetQoPE's runtime middleware framework 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 open DRE systems [23] might not consume their resource allotment at runtime, which may underutilize system resources. We are therefore extending NetQoPE to overprovision resources for applications on the assumption that not all applications will use their allotment. If runtime resource contentions occur, we are also developing dynamic resource management strategies that can provide predictable network performance for mission-critical applications.

NetQoPE's model-driven middleware platforms and tools are available in open-source format from `www.dre.vanderbilt.edu/cosmic`, and along with the CIAO component middleware available at `www.dre.vanderbilt.edu`.

References

[1] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. *Journal of Computer Systems Science*, 73(2):171–185, 2007.

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

[3] 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 *RTAS 2005*, San Francisco, CA, Mar. 2005. IEEE.

[4] 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.

[5] D. de Niz, G. Bhatia, and R. Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *Proc. of RTAS'06*, pages 231–242, Washington, DC, USA, August 2006.

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

[7] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In *Proc. of AOSD '04*, pages 66–73, New York, NY, USA, 2004.

[8] M. A. El-Gendy, A. Bose, S.-T. Park, and K. G. Shin. Paving the First Mile for QoS-dependent Applications and Appliances. In *Proc. of IWQOS'04*, Montreal, Canada, June 2004.

[9] S. Gopalakrishnan and M. Caccamo. Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems. In *RTAS '06*, pages 199–207, Washington, DC, USA, 2006.

[10] Z. Gu, S. Kodase, S. Wang, and K. G. Shin. A Model-Based Approach to System-Level Dependency and Real-time Analysis of Embedded Software. In *RTAS'03*, pages 78–85, Washington, DC, May 2003. IEEE.

[11] M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner. Extending a J2EE Server with Dynamic and Flexible Resource Management. In *Proc. of Middleware'04, Toronto, Canada*, 2004.

[12] 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.

[13] D. Llambiri, A. Totok, and V. Karamcheti. Efficiently Distributing Component-Based Applications Across Wide-Area Environments. In *Proc. of ICDCS'03*, 2003.

[14] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[15] OMG. *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 edition, Apr. 2006.

[16] 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 Middleware'03*, Rio de Janeiro, Brazil, June 2003. IFIP/ACM/USENIX.

[17] T. Repantis, X. Gu, and V. Kalogeraki. Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In *Proc. of Middleware 2006*.

[18] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, and J. Loyall. An Object-level Gateway Supporting Integrated-Property Quality of Service. *ISORC*, 00:223, 1999.

[19] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, and G. Duzan. Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004.

[20] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. Vest: An aspect-based composition tool for real-time systems. In *Proc. of RTAS'03*, page 58, Washington, DC, USA, 2003.

[21] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proc. of NSDI'05, Boston, MA*, pages 71–84, May 2005.

[22] 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.

[23] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 18(7):996–1009, 2007.

[24] D. Wichadakul, K. Nahrstedt, X. Gu, and D. Xu. 2K: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework. In *Proc. of Middleware'01*, 2001.