Model-driven QoS Provisioning for Distributed Real-time and Embedded Systems

Submitted to the Special issue on Model-driven Embedded System Design

JAIGANESH BALASUBRAMANIAN, SUMANT TAMBE, ANIRUDDHA GOKHALE, DOUGLAS C. SCHMIDT
Department of EECS, Vanderbilt University, Nashville, USA
and
BALAKRISHNAN DASARATHY, SHRIRANG GADGIL
Telcordia Technologies, Piscataway, NJ, USA

Distributed real-time and embedded (DRE) systems consist of performance-sensitive applications that are deployed in resource-constrained environments and have diverse CPU and network quality-of-service (QoS) requirements. Coordinated allocation of CPU and network resources are required so that multiple DRE applications can effectively share the available resources and have their QoS requirements satisfied end-to-end. Although CPU QoS mechanisms, such as bin-packing algorithms, and network QoS mechanisms, such as differentiated services (DiffServ), can manage a single resource in isolation, relatively little work has been done on QoS-aware mechanisms for managing multiple heterogeneous resources in a coordinated, integrated, and non-invasive manner to support end-to-end application QoS requirements.

This paper provides two contributions to the study of middleware that supports QoS-aware deployment and configuration of applications in DRE systems. First, we present a model-driven component middleware framework called NetQoPE and describe how it shields applications from the complexities of lower-level CPU and network QoS mechanisms by simplifying (1) the specification of per-application CPU and per-flow network QoS requirements, (2) resource allocation and validation decisions (such as admission control), and (3) the enforcement of per-flow network QoS at runtime. Second, we empirically evaluate how NetQoPE provides QoS assurance for applications in DRE systems. Our results demonstrate that NetQoPE provides flexible and non-invasive QoS configuration and provisioning capabilities by leveraging CPU and network QoS mechanisms without modifying application source code.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; *Component Middleware*; *QoS Management*; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and Embedded Systems; D.2.2 [**Software Engineering**]: De-

This work is supported in part or whole by DARPA Adaptive and Reflective Middleware Systems Program Contract NBCH-C-03-0132. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of the Interior, National Business Center, Acquisition Services Division, Southwest Branch or DARPA.

Authors's addresses: Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt, Department of EECS, Vanderbilt University, 2015, Terrace Place, Nashville, TN 37203; e-mail: `jai,sutambe,gokhale,schmidt@dre.vanderbilt.edu`; Balakrishnan Dasarathy, and Shrirang Gadgil, Telcordia Technologies, Piscataway, NJ 08854; e-mail: `das,sgadgil@research.telcordia.com`.

An earlier version of this paper appeared at the Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'2008) [Balasubramanian et al. 2008].

sign Tools and Techniques—*Model-driven software engineering*; *multi-resource allocation*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific Architectures*; *Reusable Libraries*

1. INTRODUCTION

Emerging trends and motivation. Component middleware, such as CORBA Component Model (CCM), J2EE, and .NET, is increasingly being used to develop and deploy next-generation distributed real-time and embedded (DRE) systems, such as shipboard computing environments [Schmidt et al. 2001], inventory tracking systems [Nechypurenko et al. 2004], avionics mission computing systems [Sharp and Roll 2003], intelligence, surveillance and reconnaissance systems [Sharma et al. 2004], and smart buildings [Snoonian 2003].

Such systems consist of applications that participate in multiple end-to-end application flows and operate in resource-constrained environments that are highly dynamic with varying levels of CPU, and network bandwidth availabilities. For example, smart buildings can host different types of (1) embedded devices (*e.g.*, fire/temperature sensors and voice-over-IP phones), (2) applications with diverse CPU QoS requirements (*e.g.*, personal desktop applications versus fire sensor data analyzers), and (3) applications flows with diverse network QoS requirements (*e.g.*, transport of e-mails versus transport of security-related information). In such dynamic and heterogeneous environments, DRE systems need to provide services with diverse performance and quality-of-service (QoS) requirements, such as satisfactory average response times and throughput. For example, applications of a smart building environment need to share multiple heterogeneous resources while providing services with diverse end-to-end QoS requirements.

Problem description. With the advent of low-cost high-speed processors, and large network bandwidth availabilities, it seems conceptually simple to overprovision network bandwidth and CPU resources to ensure sufficient QoS for applications in dynamic DRE systems. In practice, however, the QoS provisioning problem is more complex due to the need to differentiate applications and application flows at the processors and the underlying network elements, respectively so that mission-critical applications receive better performance than non-critical applications [Nahrstedt 1999; Schantz et al. 2006]. Moreover, overprovisioning is not a viable option in cost- and resource-constrained environments in which DRE systems are often deployed, *e.g.* in emerging markets that cannot afford the expense of overprovisioning.

What is needed, therefore, are effective resource management mechanisms that can efficiently provision CPU and network resources so that DRE applications can share scarce resources in resource-constrained dynamic environments, yet still have their QoS requirements satisfied end-to-end. In particular, these mechanisms must address the following two limitations in current research:

Limitation 1: Need for integrated allocation of multiple resources. Prior research has advanced the state-of-the-art on network and CPU resource reservation and scheduling mechanisms through various architectures, algorithms, and protocols. To configure required CPU resources for applications, prior work has focused on resource allocation algorithms [de Niz and Rajkumar 2006; Gopalakrishnan and Caccamo 2006] that satisfy timing requirements of applications in a DRE system. After allocating applications to a processor, scheduling algorithms, such as rate-monotonic scheduling algorithm [Lehoczky et al. 1989], could be used to differentiate application operations and provide diverse CPU performance assurances for applications. Similarly, to configure the required network resources for application flows, prior research on network quality of service (QoS) mechanisms, such as integrated services (IntServ) [L. Zhang and S. Berson and S. Herzog and S. Jamin 1997] and differentiated services (DiffServ) [Blake et al. 1998], manage the available network bandwidth and support a range of network service levels for applications in DRE systems.

While QoS mechanisms for a single resource in isolation (*e.g.*, CPU or network) has been studied extensively, relatively little work has focused on deployment and configuration of applications with integrated allocation of multiple heterogeneous resources so that applications can have their QoS requirements satisfied end-to-end. In the absence of coordinated mechanisms that allocate multiple resources, applications in DRE systems may not meet their QoS goals. For example, application developers may erroneously believe that determining the source and destination nodes for the communicating entities of their applications using CPU QoS mechanisms, such as bin-packing algorithms [de Niz and Rajkumar 2006], can also assure network QoS. An application CPU allocation algorithm [Urgaonkar et al. 2007; Stewart and Shen 2005], however, could dictate multiple placement choices for application(s), but not all of those placement choices could provide the network QoS that is required simultaneously with CPU QoS. Mechanisms are therefore needed to allocate CPU and network resources in an integrated and coordinated manner.

Limitation 2: Need for an application non-invasive resource management framework. Although QoS mechanisms (*e.g.*, DiffServ) provide schemes for reserving resources (*e.g.*, network bandwidth), applications are ultimately responsible for determining and allocating the required resources. Conventionally, applications interact directly with provided low-level APIs written imperatively in third-generation languages, such as C++ or Java, to leverage the services of the QoS mechanisms. For example, applications must make multiple invocations on network QoS mechanisms (*e.g.*, DiffServ Bandwidth Broker [Foster et al. 2004]) to accomplish key network QoS activities, such as QoS mapping, admission control, and packet marking.

To address this problem, middleware-based network QoS provisioning solutions [Wang et al. 2000; Schantz et al. 1999; Schantz et al. 2003; Miguel 2002; El-Gendy et al. 2004] as well as middleware-based CPU QoS provisioning solutions [Eide et al. 2004; Nahrstedt et al. 2001; Urgaonkar and Shenoy 2004; Pyarali et al. 2003; Krishna et al. 2004] have been developed that allow applications to specify their coordinates (source and destination IP and port addresses), CPU utilization requirements, and per-flow network QoS requirements via higher-level frameworks. The middleware frameworks—rather than the applications—are thus responsible for converting high-level specifications of QoS intent into low-level network and CPU QoS mechanism APIs for allocating the required CPU and network resources.

Although middleware frameworks alleviate many accidental complexities of using low-level network QoS mechanism APIs, applications can still be hard to evolve and extend. In particular, application source code changes may be necessary whenever changes occur to the deployment contexts (*e.g.*, source and destination nodes of applications), per-flow network resource requirements, per-application CPU resource requirements, IP packet identifiers, or middleware APIs. To address the limitations with current approaches described above, therefore, what is needed are higher-level integrated CPU and network QoS provisioning technologies that can decouple application source code from the variabilities (*e.g.*, different source and destination node deployments, different QoS requirement specifications) associated with their QoS provisioning needs. This decoupling enhances application reuse across a wider range of deployment contexts (*e.g.*, different deployment instances each with different QoS requirements), thereby increasing deployment flexibility.

Solution approach → A model-driven deployment and configuration middleware framework, called *Network QoS Provisioning Engine* (NetQoPE), that integrates CPU and network QoS provisioning via declarative domain-specific modeling languages (DSML) [Balasubramanian et al. 2007]. To address the limitations with current solutions described above, NetQoPE provides end-to-end QoS assurances for applications by providing application non-invasive mechanisms to (1) employ well-known CPU QoS mechanisms such as bin-packing algorithms [de Niz and Rajkumar 2006] to find all the feasible CPU deployments for the applications to be deployed, (2) refine these CPU allocation decisions and ensure that the network traffic demands for the incoming and outgoing traffic are met by employing network QoS mechanisms such as DiffServ and its associated Bandwidth Broker [Dasarathy et al. 2007; Foster et al. 2004; Mahajan and Parashar 2003], and (3) enforce the required network QoS at runtime by configuring the underlying middleware.

To allocate CPU and network resources for applications—and to provision QoS needs of applications in an application-independent manner—NetQoPE raises the level of abstraction of DRE system design higher than using imperative third-generation programming languages. This design allows system engineers and software developers to perform deployment-time analysis (such as schedulability analysis [Gu et al. 2003]) of non-functional system properties (such as CPU and network QoS assurances for end-to-end application flows). The result is enhanced deployment-time assurance that application QoS requirements will be satisfied. NetQoPE provides multiple resource provisioning by leveraging existing CPU QoS provisioning mechanisms and complementing them with network QoS provisioning mechanisms in an application non-invasive manner.

To allocate CPU and network resources for applications in an integrated manner, NetQoPE provides a domain-specific modeling language [Karsai et al. 2003], called *Network QoS Specification Language* (NetQoS), that allows specification of required resources and QoS needs according to the the deployed contexts of the applications. On behalf of the applications, NetQoS initiates mapping of the captured QoS specifications to QoS-specific parameters and APIs [Dasarathy et al. 2007; Foster et al. 2004; de Niz and Rajkumar 2006] using a resource allocation middleware called *Network Resource Allocation Framework* (NetRAF).

NetRAF and NetQoS shield application developers from the complexity of interacting with complex QoS mechanism APIs (either directly or through middleware) to allocate CPU and network resources in an integrated manner. NetQoPE's *Network QoS Configurator* (NetCON) deploys applications in their allocated hosts, and auto-configures the underlying middleware to enforce network QoS at runtime (*e.g.*, by adding DiffServ Codepoints (DSCPs) to IP packets). This paper focuses on the design of the NetQoPE framework and demonstrates its efficiency using an empirical evaluation. Our results evaluate the flexibility and overhead of using NetQoPE to provide CPU and network QoS assurance to end-to-end application flows.

Paper organization. The remainder of the paper is organized as follows: Section 2 describes a case study that motivates common requirements associated with provisioning QoS for DRE systems; Section 3 explains how NetQoPE addresses those requirements via its multistage 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 QOS PROVISIONING CAPABILITIES

This section presents a case study of a representative DRE system taken from a smart office enterprise environment. We use this case study throughout the paper to motivate and evaluate NetQoPE's model-driven, middleware-guided CPU and network QoS provisioning capabilities.

2.1 Overview of the Smart Office Case Study

As shown in Figure 1, smart office enterprises belong to a domain of systems called *Smart Buildings* [Snoonian 2003] and showcase state-of-the-art computing and communication infrastructure in its offices, and meeting rooms. Sensors and actuators pervade across a smart office enterprise, and control different functionality within the enterprise. For example, ventilation and air conditioning systems are controlled by sensors that monitor and send current room temperatures to an air conditioning service in the command and operations center using the communication infrastructures of the smart office enterprise. The air conditioning service analyzes the sensory data and automatically configures the actuators in response to control room temperatures.

In addition to the network traffic associated with the sensors, actuators, and other related embedded systems, the communication infrastructure of a smart office enterprise is also shared by the network traffic associated with the day-to-day enterprise operations of the employees (*e.g.*, e-mail, video conferencing). The different services provided by a smart office have different mission criticalities, and hence different and diverse QoS requirements. Below we describe some services in a smart office in the context of their capabilities and the QoS requirements they expect from the underlying platform:

—*Fire and smoke management.* Detectors in different rooms send periodic sensory information to a fire and smoke management service. In the event of a fire, this service activates the sprinkler system in the right places, activates the public address system announcing the right evacuation paths for occupants of the building, and notifies external entities, such as fire stations and hospitals of the incident

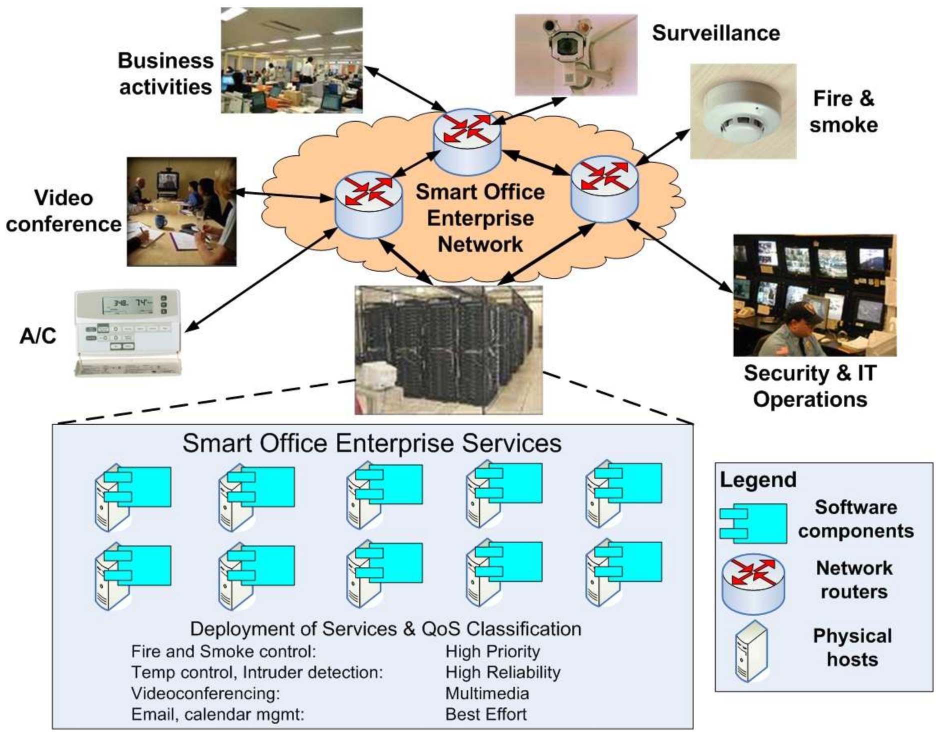

Business activities
Surveillance
Fire & smoke
Video conference
Smart Office Enterprise Network
A/C
Security & IT Operations
Smart Office Enterprise Services
Deployment of Services & QoS Classification
Fire and Smoke control: High Priority
Temp control, Intruder detection: High Reliability
Videoconferencing: Multimedia
Email, calendar mgmt: Best Effort
Legend
Software components
Network routers
Physical hosts

Fig. 1: Network Configuration in a Smart Office Enterprise

with the right details. Although the emergency mode operation of this service is infrequent, the delivery of sensory data to the service—and the outgoing traffic from this service—is *high priority*, and thus should always obtain the desired CPU and network resources. Moreover, sensory and actuation traffic must be reliable.

—*Security surveillance.* This service uses a feed from cameras and audio sensors fitted in different rooms and performs appropriate audio and image processing, and pattern matching operations to detect intruders. In the event of an intruder being detected, this service immediately notifies the security control room. It can also activate the mechanical control of the doors of the impacted rooms to lock automatically. Since this service receives a stream of video information from cameras, the input feed requires high bandwidth for the multimedia traffic, while the outgoing alert notifications and activation of door controls require high priority.

—*Air conditioning and lighting control.* The air conditioning and lighting control service maintains appropriate ambient temperatures and lighting, respectively, in different parts of the office including business offices, conference rooms and server rooms. It also turns off lights when rooms are not occupied thereby saving energy. This service receives sensory data from thermostats and motion sensors, and controls the air conditioning vents and light switches. This service requires reliable transmission of information but does not necessarily require high priority.

—*Video and teleconferencing.* Offices often provide several conference rooms with simultaneously conducted meetings. These meetings usually require video and teleconferencing facilities. It is therefore important to provision high bandwidth for these meetings. A moderator of each meeting submits a request for bandwidth to this service, which must be reliably transmitted to the service. The service in turn must provision the appropriate bandwidth for the multimedia traffic. This service can also actuate a public address system informing people of a meeting. These public address announcements belong to the best effort class of traffic.

—*Email and other web traffic.* Offices also involve a number of other kinds of traffic including email, calendar management, and web traffic. This service must manage these best effort class of traffic on behalf of the people.

As described above, assuring the correct operation of a smart office enterprise involves provisioning the diverse QoS needs of its different services that share the scarce resources across the different embedded and information systems of the smart office enterprise. The right set of CPU allocations need to be made for the services (*e.g.*, air conditioning service) to run on the shared cluster such that their QoS needs are met. Not all CPU allocations, however, always ensure the availability of network bandwidth, which these services require for processing sensory data from their respective sensors and controlling the actuators (*e.g.*, air conditioning controllers in the meeting rooms). For example, while the same CPU may be allocated to two or more multimedia tasks belonging to different services, their bandwidth requirements may make the CPU allocation decision inappropriate.

A smart office environment therefore needs algorithms and techniques that can (1) find all the feasible CPU deployments for the services, (2) refine these CPU allocation decisions by ensuring that the network traffic demands for the incoming and outgoing traffic are met, and (3) enforce the required network QoS at runtime.

2.2 Underlying Technologies

The individual tasks of each service in our case study were developed using (1) Lightweight CCM (LwCCM) to configure QoS aspects of the application separately from the functional aspects of the application [Wang et al. 2004] and (2) a Bandwidth Broker [Dasarathy et al. 2007] to manage network resources using DiffServ network QoS mechanisms, as described below.[1]

2.2.1 *Overview of Lightweight CORBA Component Model and CIAO.* The OMG Lightweight CCM (LwCCM) [Object Management Group 2003] specification is a subset of the complete CCM specification [Object Management Group 2008] designed to minimize code size for embedded environments and processing overhead for performance-sensitive applications. *Components* in LwCCM are the implementation entities that export a set of interfaces usable by conventional middleware clients as well as other components. Components can also express their intent to collaborate with other components by defining *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate

[1]Although the case study in this paper focuses on LwCCM and DiffServ, NetQoPE can be used with other network QoS mechanisms (*e.g.*, IntServ) and component middleware technologies (*e.g.*, J2EE).

a willingness to exchange typed messages with one or more components. *Homes* are factories that shield clients from the details of component creation strategies and subsequent queries to locate component instances. Figure 2 shows the layered

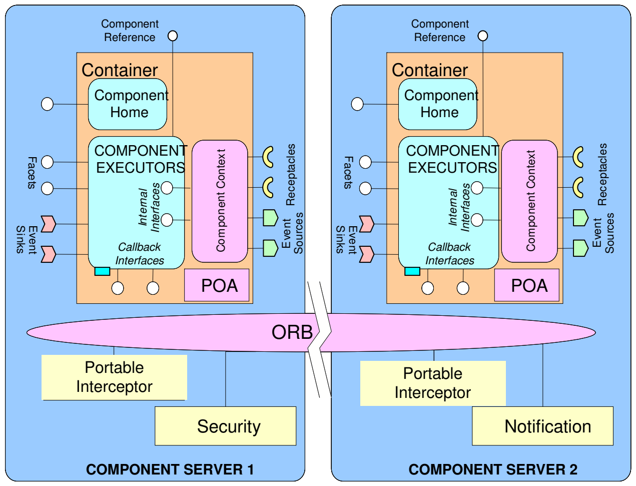


Fig. 2: Overview of LwCCM

architecture of LwCCM, which includes the following entities:

—LwCCM sits atop an *object request broker* (ORB) and provides *containers* that encapsulate and enhance the CORBA portable object adapter (POA) demultiplexing mechanisms. Containers support various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, to the components it manages.

—A *component server* plays the role of a process that manages the homes, containers, and components.

—Each container manages one type of component and is responsible for initializing instances of this component type and connecting them to other components and common middleware services.

—The *component implementation framework* (CIF) consists of patterns, languages and tools that simplify and automate the development of component implementations which are called as *executors*. Executors actually provide the component's business logic.

—To initialize an instance of a component type, a container creates a component home. The component home creates instances of servants and executors and combines them to export component implementations to the external world.

—Executors use servants to communicate with the underlying middleware and servants delegate business logic requests to executors. Client invocations made on the component are intercepted by the servants, which then delegate the invocations to the executors. Moreover, the containers can configure the underlying middleware to add more specialized services such as integrating event channel to allow components to communicate, add Portable Interceptors to intercept component requests, etc.

Assembly, which is an abstraction that groups components, in LwCCM is described using XML descriptors (mainly the *deployment plan* descriptor) defined by the OMG D&C [OMG 2006] specification. The *deployment plan* includes details about the components, their implementations, and their connections to 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* that provide the runtime operating environment (*e.g.*, load balancing, security, and event notification) for components to invoke remote operations.

The smart office enterprise software controllers we developed for this case study were build using CIAO, which is an open-source[2] implementation of the LwCCM and Real-time CORBA [Object Management Group 2002] specifications built atop The ACE ORB (TAO) [Schmidt et al. 1998]. CIAO's architecture is designed using patterns [Buschmann et al. 2007] for composing component-based middleware and reflective middleware techniques to enable mechanisms within the component-based middleware to support different QoS aspects, such as CPU scheduling priority [Wang et al. 2004].

2.2.2 *Overview of Telcordia's Bandwidth Broker.* Telcordia has developed a network management solution for QoS provisioning called the Bandwidth Broker - [Dasarathy et al. 2005], which leverages widely available mechanisms [Blake et al. 1998] that support Layer-3 DiffServ (Differentiated Services) and Layer-2 Class of Service (CoS) features in commercial routers and switches. DiffServ and CoS have two major QoS functionality/enforcement mechanisms:

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

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

These two features by themselves are insufficient to ensure end-to-end network QoS because the traffic presented to the network must be made to match the network capacity. What is also needed, therefore, is an adaptive admission control entity that ensures there are adequate network resources for a given traffic flow on any given link that the flow may traverse. The admission control entity should

[2]CIAO is available from `www.dre.vanderbilt.edu/CIAO`.

be aware of the path being traversed by each flow, track how much bandwidth is being committed on each link for each traffic class, and estimate whether the traffic demands of new flows can be accommodated. In Layer-3 networks, there is more than one equal-cost between a source and destination; so we employ Dijkstra's all-pair shortest path algorithms. In Layer-2 network, we discover the VLAN tree to find the path between any two hosts.

Figure 3 illustrates the architecture (described in detail in [Dasarathy et al. 2005; Dasarathy et al. 2007; Gadgil et al. 2007]) of our network management solution for providing application QoS. The four components of the QoS management archi-

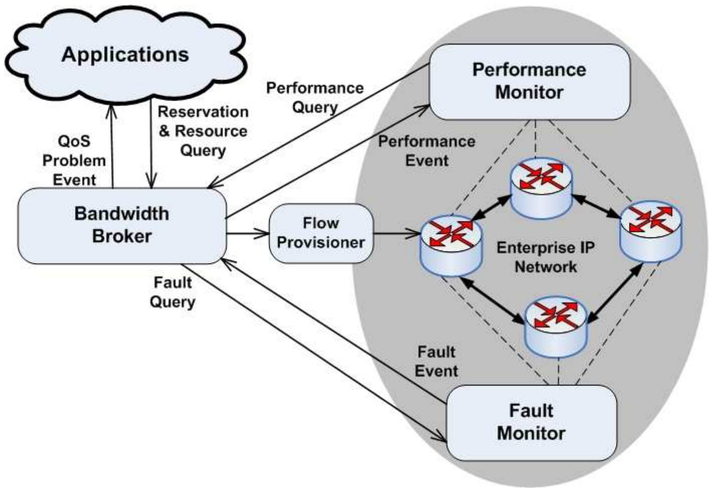


Fig. 3: Overview of Telcordia's Bandwidth Broker

tecture are (1) *Bandwidth Broker*, (2) *Flow Provisioner*, (3) *(Network) Performance Monitor*, and (4) *(Network) Fault Monitor*. Our network QoS components provide adaptive admission control that ensures there are adequate network resources to match the needs of admitted flows.

The Bandwidth Broker is responsible for admission control and assigning the appropriate traffic class to each flow. It tracks bandwidth allocations on all network links, rejecting new flow requests when bandwidth is not available. The Flow Provisioner enforces Bandwidth Broker admission control decisions by configuring ingress network elements to ensure that no admitted flow exceeds its allocated bandwidth. The Flow Provisioner translates technology-independent configuration directives generated by the Bandwidth Broker into vendor-specific router and switch commands to classify, mark, and police packets belonging to a flow. The Fault Monitor is the main feedback mechanism for adapting to network faults and the Performance Monitor provides information on the current performance information of flows and traffic classes. The Bandwidth Broker uses this information to adapt its admission control decisions.

The Bandwidth Broker admission decision for a flow is not based solely on requested capacity or bandwidth on each link traversed by the flow, but is also based on delay bounds requested for the flow. The delay bounds for new flows must be assured without damaging the delay bounds for previously admitted flows and without redoing the expensive job of readmitting every previously admitted flow. We have developed computational techniques to provide both deterministic and statistical delay-bound assurance [Dasarathy et al. 2007]. This assurance is based on relatively expensive computations of occupancy or utilization bounds for various classes of traffic, performed only at the time of network configuration/reconfiguration, and relatively inexpensive checking for a violation of these bounds at the time of admission of a new flow.

2.3 QoS Management in the Smart Office Enterprise.

Although state-of-the-art advances [de Niz and Rajkumar 2006; Foster et al. 2004; Dasarathy et al. 2007] have been made to reserve CPU and network resources, applications are still largely responsible for *determining* and *specifying* how much capacity of each resource must be reserved to meet the application's end-to-end QoS requirements. In particular, LwCCM-based applications in our smart office enterprise case study use bin-packing algorithms and Bandwidth Broker services via the following middleware-guided steps:

(1) CPU utilization requirements are specified for each application, and bin-packing algorithms are used to determine the physical hosts to deploy the applications
(2) Network QoS requirements are specified on each application flow, along with information on the source/destination IP/port addresses that were determined by bin-packing algorithms
(3) 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
(4) Remote operations are invoked 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 DRE systems, such as our case study, and shows how NetQoPE's multistage architecture shown in Figure 4 helps resolve these challenges.

3. NETQOPE'S MULTISTAGE NETWORK QOS PROVISIONING ARCHITECTURE

This section describes how NetQoPE addresses limitations with conventional techniques for CPU allocation and providing network QoS to applications in DRE systems. As discussed in Section 1, these limitations include requiring the modification of application source code to specify deployment context-specific network QoS requirements and integrate functionality from network QoS mechanisms at runtime. In contrast, NetQoPE deploys and configures component middleware-based applications in DRE systems and enforces their network and CPU QoS requirements using the multistage (*i.e.*, design-, pre-deployment-, deployment-, and run-time) architecture shown in Figure 4. NetQoPE's multistage architecture consists of the following elements:

• The **Network QoS Specification Language** (NetQoS), which is a DSML that supports design-time specification of per-application CPU resource requirements, as

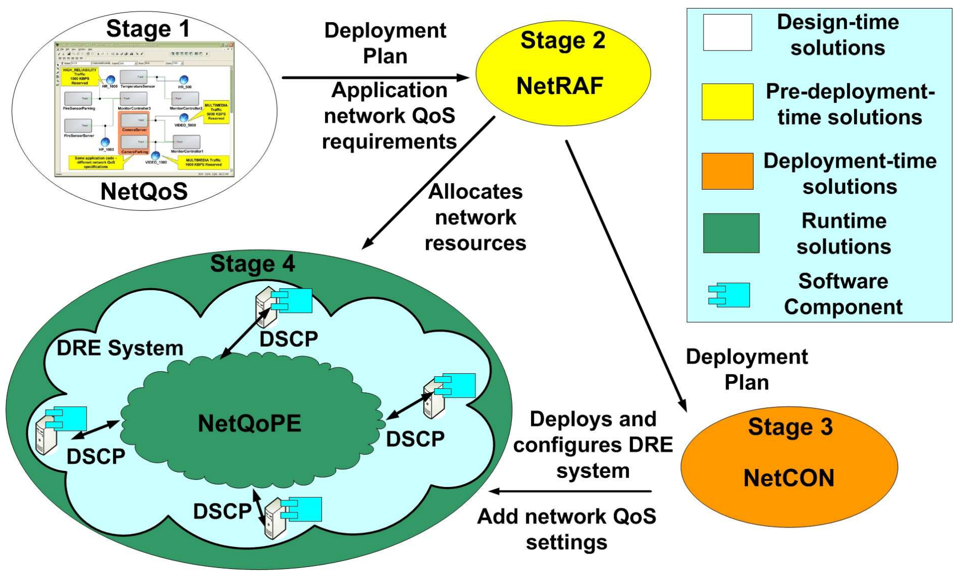


Fig. 4: NetQoPE's Multistage Architecture

well as 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 CPU, bandwidth, and delay requirements) where they will be used—NetQoS simplifies the deployment of applications in contexts that have different CPU and network QoS needs, *e.g.*, different bandwidth requirements. Section 3.1 describes NetQoS in more detail.

• 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 and CPU QoS mechanisms at deployment time. *NetRAF* provides an application-transparent and pre-deployment time per-application CPU resource allocation capabilities using bin-packing algorithms [de Niz and Rajkumar 2006] and per-flow resource allocation capabilities using a Bandwidth Broker [Dasarathy et al. 2007] that coordinates CPU utilization and network bandwidth assurances for applications and their end-to-end application flows. Section 3.2 describes NetRAF in more detail.

• The **Network QoS Configurator** (NetCON), which is a middleware-based network QoS configurator that provides deployment-time configuration of component middleware containers. NetCON adds flow-specific identifiers (*e.g.*, DSCPs) to IP packets at runtime 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. Section 3.3 describes NetCON in more detail.

Figure 4 shows how the output of each stage in NetQoPE's multistage architecture serves as input for the next stage, which helps automate the deployment and configuration of DRE applications with CPU and network QoS support in a non-invasive manner. The remainder of this section describes each element in the NetQoPE's multistage architecture and explains how they provide the functionality required to meet the end-to-end QoS requirements of applications in DRE systems.

3.1 NetQoS: Alleviating Complexities in CPU and Network QoS Requirements Specification

Context. As discussed in Section 1, integrated allocation and scheduling of multiple heterogeneous resources is required to allocate both CPU and network resources for applications and to provision per-application end-to-end QoS needs. To allocate CPU resources for applications using a bin-packing algorithm [de Niz and Rajkumar 2006; Chen et al. 2007], CPU utilization requirements need to be specified. Bin-packing algorithms determine the appropriate physical hosts for deployment and applications obtain their required CPU resources. To allocate network resources for each application flow in the system, application developers and deployers need to specify a required level of service (*e.g.*, high priority vs. low priority), the source and destination IP and port addresses, and ingress and egress bandwidth requirements. A network QoS mechanism (*e.g.*, a Bandwidth Broker [Foster et al. 2004]) uses this information to configure network resources between two endpoint nodes to provide the required QoS.

Problem. Although state-of-the-art advances [de Niz and Rajkumar 2006; Foster et al. 2004] have been made to reserve CPU and network resources, it is still the responsibility of the applications to *determine* and *specify* how much capacity of each resource is reserved to meet the application's end-to-end QoS requirements. Conventional techniques, such as hard-coded API approaches [de Miguel 2002], require application source code modifications for specifying resource requirements. Manual modifications to the application source code to specify both CPU and network QoS requirements is tedious, error-prone, and non-scalable.

In particular, applications could have different resource requirements depending on the context in which they are deployed. For example, in our smart office case study described in Section 2, depending on where they are deployed, fire sensors (which send their sensory data to monitors in the command and control center) will have different importance levels. For example, fire sensors deployed in the parking lot have a lower importance than those in the server room. The sensor to monitor flows thus have different network QoS requirements, even though the reusable software controllers (developed in LwCCM) managing the fire sensor and the monitor have the same functionality. It is hard to envision at development time all the contexts in which source code will be deployed; if such information is readily available, application source code can be modified to specify resource requirements for each of those contexts.

This problem is further complicated because of the need to provision both CPU and network resources in an integrated fashion. In particular, network bandwidth allocations depend on CPU allocations. It is important to know the source and destination addresses of an application flow before network resources could be reserved across the nodes for the particular application flow [Dasarathy et al. 2005; Foster et al. 2004]. The source and destination addresses of an application flow, however, are determined by the CPU allocation algorithms used to deploy the applications.

Multiple feasible CPU allocations are possible with a given set of component CPU requirements [de Niz and Rajkumar 2006]. For example, a component could be deployed on any node that has the capacity available to satisfy the component's CPU utilization requirement. Depending on the node chosen to deploy the component, the deployment of other components in the system can also change. Not all those placement choices, however, could provide the network QoS that is required simultaneously with CPU QoS. The need to know source and destination addresses of an application—coupled with the fact that multiple choices are possible for deploying applications—makes changing application source code to specify resource requirements inflexible and non-scalable.

Solution approach → Model-driven declarative CPU and network requirements specification. To resolve the above described problems, each application in NetQoPE specifies its resource requirements at application deployment-time using a domain specific modeling language (DSML) called the *Network QoS Specification Language* (NetQoS). Since NetQoS allows specifying resource requirements *right* before applications are deployed and configured in the target environment, its declarative mechanisms *decouple* this responsibility from application source code, and also *specialize* the process of specifying resource requirements for the particular deployment and usecase. Further, on behalf of the applications, NetQoS initiates the integrated CPU and network resource allocations for all the applications. It thus relieves application developers and deployers of the responsibilities to (1) specify multiple resource requirements, and (2) initiate resource reservations using CPU and network QoS mechanisms.

Declarative specification of resource requirements. NetQoS, which is built using Generic Modeling Environment (GME) [Ákos Lédeczi et al. 2001] and the Platform Independent Component Modeling Language (PICML) [Balasubramanian et al. 2005], provides applications with an application-independent, and declarative (as opposed to application-intrusive [de Miguel 2002], middleware-dependent [Eide et al. 2004], and OS-dependent [Mehra et al. 2000]) mechanism to specify multi-resource requirements.

DRE system developers can use NetQoS to (1) model application elements, such as interfaces, components, connections, and component assemblies, (2) specify CPU utilization of components, and (3) specify the network QoS classes, such as HIGH PRIORITY (HP), HIGH RELIABILITY (HR), MULTIMEDIA (MM), and BEST EFFORT (BE), bi-directional bandwidth requirements on the modeled application elements[3]. NetQoS's network QoS classes correspond to the DiffServ levels of service provided by our Bandwidth Broker [Dasarathy et al. 2005].[4] For example, the HP class represents the highest importance and lowest latency traffic (*e.g.*, fire detection reporting in the server room) whereas the HR class represents traffic with low drop rate (*e.g.*, surveillance data).

Flexible enforcement of network QoS. In certain application flows in the smart

[3]In LwCCM, components communicate using *ports* (described in Section 2.2) that provide application-level communication endpoints. NetQoS provides capabilities to annotate communication ports with the network QoS requirement specification capabilities.

[4]NetQoS's DSML capabilities can also be extended to provide requirements specification conforming to other network QoS mechanisms, such as IntServ.

office case study, (*e.g.*, a monitor requesting location coordinates from a fire sensor in our case study) clients control the network priorities at which requests/replies are sent. In other application flows (*e.g.*, a temperature sensor sends temperature sensory information to monitors), servers control the reception and processing of client requests. If such *design intents* are not captured, applications could potentially misuse network resources at runtime, and also affect the performance of other applications that share the network. For example, multiple applications (irrespective of their mission criticality and importance) could potentially invoke remote operations by using HIGH PRIORITY (HP) network service class in DiffServ, thereby affecting QoS of applications that are *really* mission critical.

To support both models of communication (*i.e.*, whether clients vs. servers control network QoS for a flow), NetQoS supports annotating each bi-directional flow using either:

—The CLIENT_PROPAGATED network priority model, which allows clients to request real-time network QoS assurance even in the presence of network congestion or

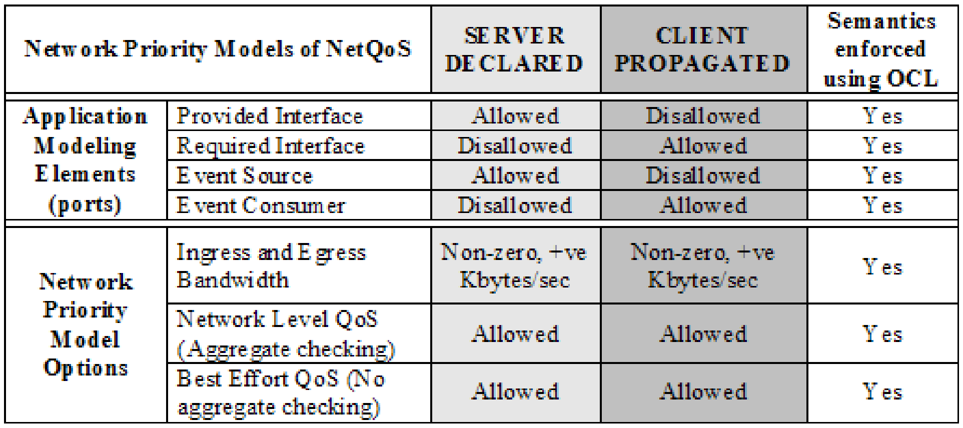
—The SERVER_DECLARED network priority model, which allows servers to dictate the service that they wish to provide to the clients to prevent clients from wasting network resources on non-critical communication.

On behalf of applications, NetQoS initiates the allocation of CPU and network resources, and in Section 3.3, we will describe how NetQoPE uses component middleware frameworks at runtime to *realize* the design intent captured by NetQoS and *enforce* network QoS for applications.

Early detection of QoS specification errors. Defining network and CPU QoS specifications in source code or through NetQoS is a human-intensive process. Errors in these specifications may remain undetected until later stages of development, such as deployment and runtime, when they are much more costly to identify and fix. To identify common errors in network QoS requirement specification early in the development phase, NetQoS uses built-in constraints specified via the OMG Object Constraint Language (OCL) that check the application model annotated with network and CPU priority models.

For example, NetQoS detects and flags specification network resource specification errors, such as negative or zero bandwidth. It also enforces the semantics of network priority models via syntactic constraints in its DSML. For example, the CLIENT_PROPAGATED model can be associated with ports in the client role only (*e.g.*, required interfaces), whereas the SERVER_DECLARED model can be associated with ports in the server role only (*e.g.*, provided interfaces). Figure 5 shows other examples of network priority models supports by NetQoS.

A server using the SERVER_DECLARED network priority model can also dictate that the total ingress bandwidth from all communicating clients cannot exceed a designated network bandwidth (*e.g.*, 30 Mbps). NetQoS checks the aggregation of egress bandwidth requested using all clients that communicate with the server and raise an error if the total exceeds the preferred total bandwidth. Without this capability, applications could fail at runtime where clients invoke remote operations on servers after reserving more network bandwidth than the server's reply will use, which wastes available network bandwidth that could be used by other application flows. NetQoS provides this capability so application deployers can provision the



Network Priority Models of NetQoS		SERVER DECLARED	CLIENT PROPAGATED	Semantics enforced using OCL
Application Modeling Elements (ports)	Provided Interface	Allowed	Disallowed	Yes
	Required Interface	Disallowed	Allowed	Yes
	Event Source	Allowed	Disallowed	Yes
	Event Consumer	Disallowed	Allowed	Yes
Network Priority Model Options	Ingress and Egress Bandwidth	Non-zero, +ve Kbytes/sec	Non-zero, +ve Kbytes/sec	Yes
	Network Level QoS (Aggregate checking)	Allowed	Allowed	Yes
	Best Effort QoS (No aggregate checking)	Allowed	Allowed	Yes

Fig. 5: Network QoS Models Supported by NetQoS

underlying network QoS mechanisms efficiently and flexibly.

Preparation for allocating CPU and network resources. After a model has been created and checked for type violations using built-in constraints, the specified network resource requirements need to be captured so that a network QoS mechanism [Dasarathy et al. 2007; Foster et al. 2004] can be used to allocate network resources for the application flows. However, as described before, this process requires determination of source and destination IP addresses of the applications.

NetQoS allows the specification of CPU utilization requirements of each component and also the target environment where components are deployed. NetQoS's model interpreter traverses CPU requirements of each application component and generates a set of feasible deployment plans (described in Section 2.2) using CPU allocation algorithms, such as *first fit*, *best fit*, and *worst fit*, as well as *max* and *decreasing* variants of these algorithms. NetQoS can be used to choose the desired CPU allocation algorithm and to generate the appropriate deployment plans automatically, thereby shielding developers from tedious and error-prone manual component-to-node allocations.

To perform network resource allocations (described in Section 3.2), NetQoS's model interpreter captures the details about (1) the components, (2) their deployment locations (determined by the CPU allocation algorithms), and (3) the network QoS requirements for each application flow they are part of, using the *deployment plan configProperty* tags (see Section 2.2). Section 3.2 describes how a later stage in NetQoPE allocates network resources based on requirements specified in the deployment plan descriptor.

Application to the case study. Figure 6 shows a NetQoS model that highlights many of the capabilities described above. In this model, multiple instances of the same reusable application components (*e.g.*, FireSensorParking and FireSensorServer components) are annotated with different QoS attributes using an intuitive drag and drop technique. Specifying QoS requirements via NetQoS is much simpler than modifying application code for each deployment context, as shown in Section 4.2.

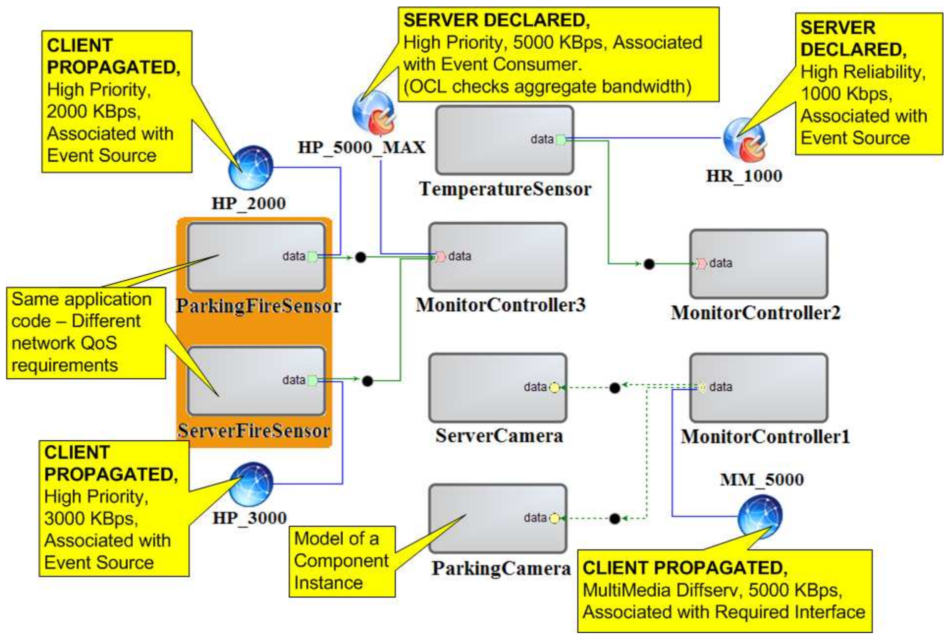

CLIENT PROPAGATED, High Priority, 2000 KBps, Associated with Event Source
SERVER DECLARED, High Priority, 5000 KBps, Associated with Event Consumer. (OCL checks aggregate bandwidth)
SERVER DECLARED, High Reliability, 1000 Kbps, Associated with Event Source
HP_2000
HP_5000_MAX
TemperatureSensor
HR_1000
data
ParkingFireSensor
MonitorController3
MonitorController2
Same application code – Different network QoS requirements
ServerFireSensor
ServerCamera
MonitorController1
CLIENT PROPAGATED, High Priority, 3000 KBps, Associated with Event Source
HP_3000
MM_5000
Model of a Component Instance
ParkingCamera
CLIENT PROPAGATED, MultiMedia Diffserv, 5000 KBps, Associated with Required Interface

Fig. 6: Applying NetQoS Capabilities to the Case Study

Our case study also has scores of application flows with different client- and server-dictated network QoS specifications, which are modeled using CLIENT_PROPAGATED and SERVER_DECLARED network priority models, respectively. The well-formedness of these specifications are checked using NetQoS's built-in constraints. In addition, the same QoS attribute (*e.g.*, HR_1000 in Figure 6) can be reused across multiple connections, which increases the scalability of expressing requirements for the number of connections prevalent in large-scale DRE systems, such as our smart office enterprise environment case study.

Finally, NetQoS's support to plug-in different bin-packing algorithms to determine CPU allocations decouples the applications from the responsibility of manually specifying all possible allocations to allocate network resources. This feature of NetQoS coupled with its declarative mechanisms to specify resource requirements, completely shields applications (and hence modifications to its source code) from the complexities of QoS specification and allocation. Section 4.2 validates these capabilities provided by NetQoS.

3.2 NetRAF: Alleviating Complexities in Network Resource Allocation and Configuration

Context. After deciding where to deploy components on source and destination nodes, DRE systems must communicate with a network QoS mechanism API (*e.g.*, Bandwidth Broker for DiffServ networks) to allocate and configure network resources based on the network QoS requirements specified on the application flows.
Problem. It is often undesirable to tightly couple application components (*e.g.*, the temperature sensor software controller code in our case study) with a network

QoS mechanism API. This coupling complicates deploying the same application component in a different context (*e.g.*, the temperate sensor software controllers for sensing the temperature at the server room *and* the conference room) with different network QoS requirements. Manually programming application components to handle all possible combinations of network resources is tedious and error-prone.

Moreover, network QoS mechanism APIs that allocate network resources require IP addresses for hosts where the resources are allocated. Components that require network QoS must therefore know the physical node placement of the components with which they communicate. This component deployment information may be unknown at development time since deployments are often not finalized until CPU allocation algorithms decide them. Maintaining such deployment information at the source code level or querying it at runtime is unnecessarily complex. Ideally, network resources should be allocated without modifying application source code and should handle difficulties associated with specifying application source and destination nodes, which could vary depending on the deployment context.

Solution approach → Middleware-based Resource Allocator Framework. NetQoPE's *Network Resource Allocator Framework* (NetRAF) is a resource allocator engine that allocates network resources for DRE systems using DiffServ network QoS mechanisms. NetRAF does not sit between the applications and the underlying operating system kernel; rather, it complements the functionality of the operating system kernel, and applications are oblivious to NetRAF.

NetRAF allocates network resources for application flows on behalf of the applications and shields applications from interacting with complex network QoS mechanism APIs. To ensure compatibility with different implementations of network QoS mechanisms (*e.g.*, multiple DiffServ Bandwidth Broker implementations [Dasarathy et al. 2007; Foster et al. 2004]), NetRAF works with XML descriptors that capture CPU and network resource requirement specifications (which were specified using NetQoS in the previous stage) in *QoS-independent* manner. These specifications are then mapped to *QoS-specific* parameters depending on the chosen network QoS mechanism. The task of enforcing those QoS specifications are then left to the network QoS mechanism implementation (with configuration support required from applications, which we discuss in detail in the context of NetQoPE in Section 3.3). This provides a clean separation of functionality between resource reservation (provided by NetRAF) and QoS enforcement (done by underlying network elements).

Network resource allocations. As shown in Figure 7, input to NetRAF is the set of feasible deployment plans generated by the NetQoS model interpreter, which also embeds per-flow network QoS requirements. The modeled deployment context could have many instances of the same reusable source code, *e.g.*, the temperature sensor software controller could be instantiated two times: one for the server room and one for the conference room. When using NetQoS, however, application developers annotate only 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 [Schantz et al. 1999] that modify application source code to work with network QoS mechanisms, which become complex when source code is reused in a

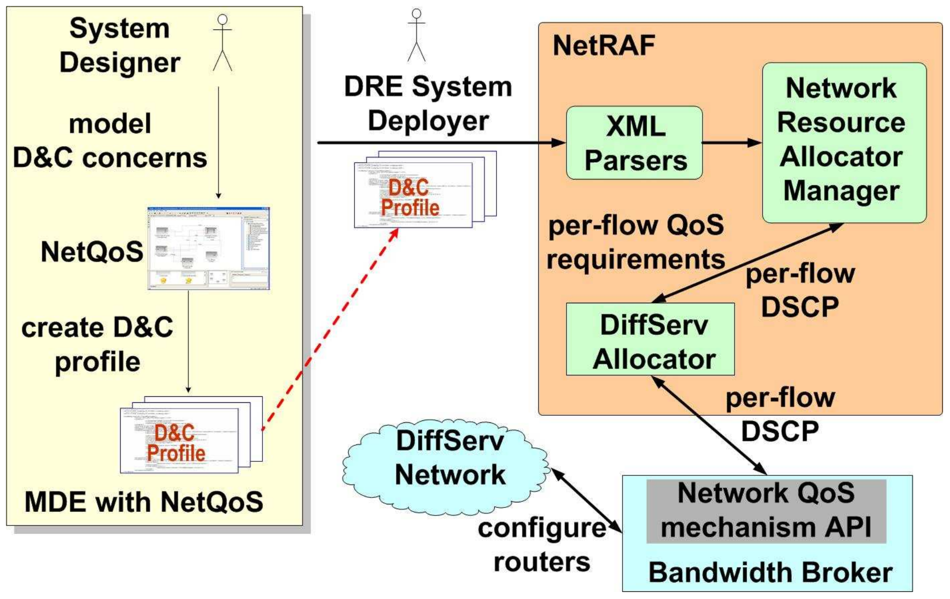


Fig. 7: NetRAF's Network Resource Allocation Capabilities

wide range of deployment contexts.

NetRAF's *Network Resource Allocator Manager* accepts application QoS requests at pre-deployment-time. It processes these requests in conjunction with a *Diff-Serv Allocator*, using deployment specific information (*e.g.*, source and destination nodes) of components and per-flow network QoS requirements embedded in the deployment plan created by NetQoS. This capability shields applications from interacting directly with complex APIs of network QoS mechanisms thereby enhancing the flexibility NetQoPE for a range of deployment contexts. Moreover, since NetRAF provides the capability to request network resource allocations on behalf of components, developers need not write source code to request network resource allocations for all applications flows, which simplifies the creation and evolution of application logic, as validated in Section 4.2.

Integrated CPU and network QoS provisioning. While interacting with network QoS mechanism specific allocators (*e.g.*, a Bandwidth Broker), NetRAF's Network Resource Allocator Manager may need to handle exceptional conditions, such as failures in resource allocation. Failures during allocation may occur due to insufficient network resources between the source and destination nodes hosting the components. Although NetQoS checks the well-formedness of network requirement specifications at application level, it cannot identify every situation that may lead to failures during actual resource allocation.

To handle failure scenarios gracefully, NetRAF provides hints to regenerate CPU allocations for components using the CPU allocation algorithm selected by application developers using NetQoS. For example, if network resource allocations fails for a pair of components deployed in a particular source and destination node, NetRAF requests revised CPU allocations by adding a constraint to not deploy the components in the same source and destination nodes. After the revised CPU allocations are computed, NetRAF will (re)attempt to allocate network resources for the components.

NetRAF automates the network resource allocation process by iterating over the set of deployment plans until a deployment plan is found that satisfies both types of requirements (*i.e.*, both the CPU and network resource requirements) thereby simplifying system deployment via the following two-phase protocol:

(1) It first invokes the API of the QoS mechanism-specific allocator, providing it one flow at a time without actually reserving network resources.
(2) It then commits the network resources if and only if the first phase is completely successful and resources for all the flows can be successfully reserved.

This protocol prevents the delay that would otherwise be incurred if resources allocated for a subset of flows must be released due to failures occurring at a later allocation stage. If no deployment plan yields a successful resource allocation, the network QoS requirements of component flows must be reduced using NetQoS.

Application to the case study. Since our case study is based on DiffServ, NetRAF uses the *DiffServ Allocator* to allocate network resources, which in turn invokes the Bandwidth Broker's admission control capabilities [Dasarathy et al. 2005] by feeding it one application flow at a time. If all flows *cannot* be admitted, NetRAF provides developers with an option to modify the deployment context since applications have not yet been deployed. Example modifications include changing component implementations to consume fewer resources or change the source/-destination nodes. As shown in Section 4.2, this capability helps NetRAF incur lower overhead than conventional approaches [Wang et al. 2000; Schantz et al. 1999] 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 network QoS classes, as described in Section 3.1. 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* [Dasarathy et al. 2007] 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 NetCON: 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 the network can differentiate application traffic and provision appropriate QoS to each 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 [Schantz et al. 2003]. 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 application deployment context.

Solution approach → Deployment and runtime component middleware mechanisms. Section 2.2 describes how LwCCM containers provide a runtime environment for components.[5] NetQoPE's *Network QoS Configurator* (NetCON) can auto-configure these containers by adding DSCPs to IP packets when applications invoke remote operations. NetRAF performs network resource allocations, determines the bi-directional DSCP values to use for each application flow and encodes those DSCP values in the deployment plan, as shown in Figure 8.

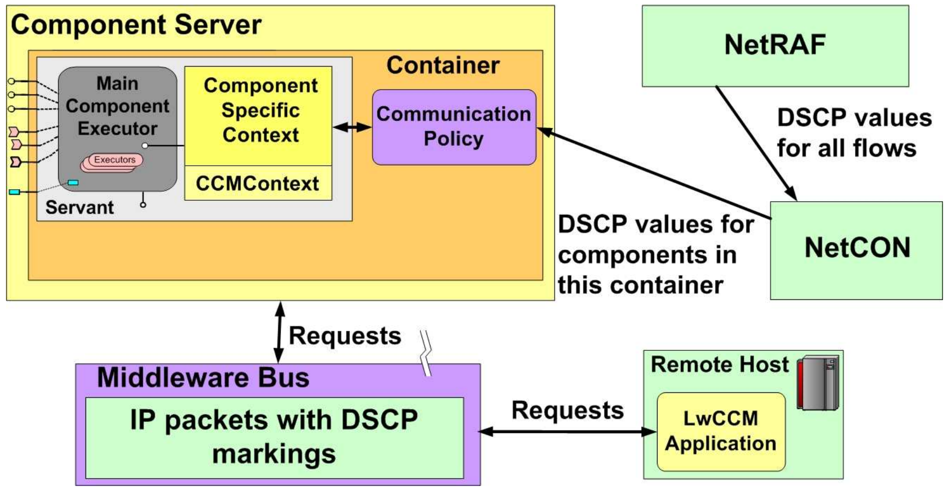


Fig. 8: NetCON's Container Auto-configurations

During deployment, NetCON parses the deployment plan and its connection tags to determine (1) source and destination components, (2) the network priority model to use for their communication, (3) the bi-directional DSCP values (obtained via NetRAF), 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 for use by clients in a remote invocation. When a component invokes a remote operation in LwCCM, its container's context information provides the object reference of the destination component.

NetCON's container programming model can transparently add DSCPs and enforce the network priority models described in Section 3.1. To support the SERVER_DECLARED network priority model, NetCON encodes a SERVER_DECLARED policy and the associated request/reply DSCPs on the server's object reference. When

[5]Other component middleware provide similar capabilities via containers, *e.g.*, EJB applications interact with containers to obtain the right runtime operating environment.

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 includes it in the request IP packets. Before sending the reply, the server-side middleware checks the policy again and the reply DSCP is added to the associated IP packets.

To support the 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 allows the clients to control the network priorities with which their requests and replies traverse the underlying network 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 NetQoPE-enabled 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. In our case study shown in Figure 6, the FireSensor software controller component is deployed in two different instances to control the operation of the fire sensors in the parking lot and the server room. There is a single MonitorController software component (MonitorController3 in Figure 6) that communicates with the deployed FireSensor components. Due to differences in importance of the FireSensor components deployed, however, the MonitorController software component uses CLIENT_PROPAGATED network priority model to communicate with the FireSensor components with different network QoS requirements.

After software components are modeled using NetQoS—and the required network resources are allocated using NetRAF—NetCON configures the *container* hosting the MonitorController3 component with the CLIENT_PROPAGATED policy, which corresponds to the CLIENT_PROPAGATED network priority model defined on the component by NetQoS. This capability is provided automatically by containers to ensure that, at runtime appropriate DSCP values are added to both forward and reverse communication paths when the MonitorController3 component communicates with either the FireSensorParking or FireSensorServer component. Communication between the MonitorController3 and the FireSensorParking or FireSensorServer components thus receives the required network QoS since NetRAF configures the routers between the MonitorController3 and FireSensorParking components with the source IP address, destination IP address, and DSCP tuple.

NetCON therefore allows application developers in DRE systems to focus on their application component logic (*e.g.*, the MonitorController component in the case study), rather than wrestling with low-level mechanisms for provisioning network QoS. Moreover, NetCON provides these capabilities without modifying application code, thereby simplifying development and minimizing runtime overhead, as validated in Section 4.3.

4. EMPIRICAL EVALUATION OF NETQOPE

This section empirically evaluates the flexibility and overhead of using NetQoPE to provide CPU and network QoS assurance to end-to-end application flows. We first demonstrate how NetQoPE's model-driven QoS provisioning capabilities can significantly reduce application development effort compared with conventional approaches. We then 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 in Section 2.

4.1 Evaluation Scenario

Hardware and software testbed. Our empirical evaluation of NetQoPE was conducted at ISISlab (`www.dre.vanderbilt.edu/ISISlab`), which consists of (1) 56 dual-CPU blades running 2.8 GHz 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 9, our experiments were conducted on 15 of dual CPU blades in ISISlab, where (1) 7 blades (A, B, D, E, F, G, and H) hosted our smart office enterprise case study software components (*e.g.*, a fire sensor software controller) and (2) 8 other blades (P, Q, R, S, T, U, V, and W) hosted Linux router software.

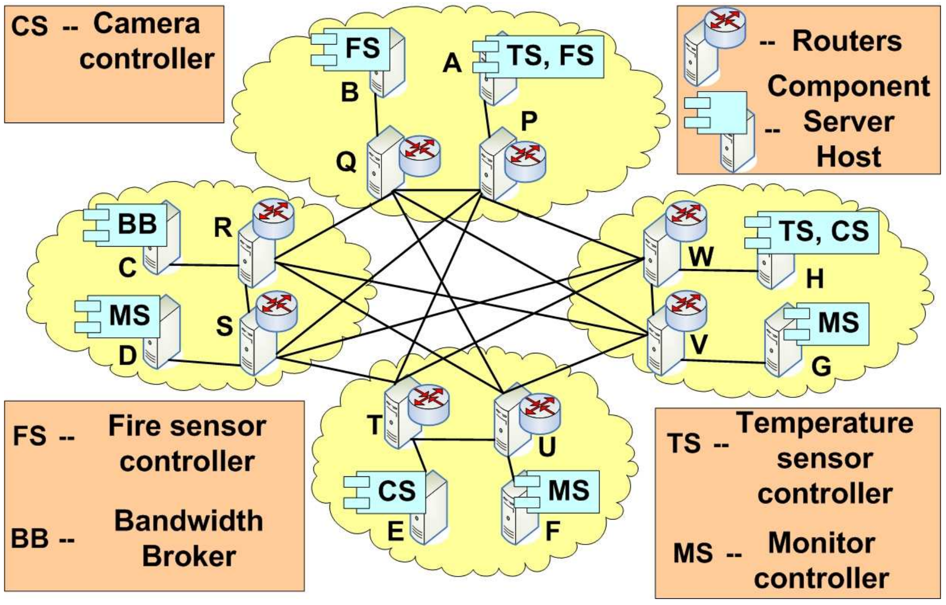


Fig. 9: Experimental Setup

The software controller components were developed using the CIAO middleware, which is an open-source LwCCM implementation developed atop the TAO real-time CORBA object request broker [Schmidt et al. 2002]. Our evaluations used DiffServ QoS and the associated Bandwidth Broker [Dasarathy et al. 2007] 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.

Evaluation scenario. 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 9 shows two *fire sensor controller* components deployed on hosts A and B. These components sent their monitored information to *monitor controller* components deployed on hosts D and F. Each of these software controller components have their own CPU resource requirements and the physical node allocations for those components were determined by the CPU allocation algorithms employed by NetQoS. Further, communication between these software controllers used one of the traffic classes (*e.g.*, HIGH PRIORITY (HP)) 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 CPU and network behavior of the software controllers when different QoS requirements are provisioned, we created the `TestNetQoPE` performance benchmark suite.[6] We used `TestNetQoPE` to evaluate the flexibility, overhead, and performance of using NetQoPE to provide CPU and network QoS assurance to end-to-end application flows. In particular, we used `TestNetQoPE` to specify and measure diverse CPU and network QoS requirements of the different software components that were deployed via NetQoPE, such as the application flow between the *fire sensor controller* component on host A and the *monitor controller* component on host D. These tests create a session for component-to-component communication with configurable bandwidth consumption (components also consume a configurable percentage of CPU resource on their hosted processors). High-resolution timer probes were used to measure roundtrip latency accurately for each client invocation.

We now describe the experiments performed using the ISISlab configuration described in Section 4.1 and analyze the results.

4.2 Evaluating NetQoPE's Model-driven QoS Provisioning Capabilities

Rationale. As discussed in Section 3, NetQoPE provides extensible provisioning of application CPU and network QoS mechanisms. This experiment evaluates the effort application developers spend using NetQoPE to (re)deploy applications and provision QoS and compares this effort against the effort needed to provision QoS for applications via conventional approaches.

Methodology. We first identified four flows from Figure 9 whose network QoS requirements are described as follows:

—A fire sensor controller component on host A uses the high reliability (HR) class to send potential fire alarms in the parking lot to the monitor controller component on host D.

—A fire sensor controller component on host B uses the high priority (HP) class to send potential fire alarms in the server room to the monitor controller component on host F.

—A camera controller component on host E uses the multimedia (MM) class and sends imagery information from the break room to the monitor controller com-

[6]`TestNetQoPE` can be downloaded as part of the CIAO open-source middleware available at (`www.dre.vanderbilt.edu/CIAO`).

ponent on host G.

—A temperature sensor controller component on host A uses the best effort (BE) class and sends temperature readings to the monitor controller component on host F.

The clients dictated the network priority for requests and replies in all flows *except* for the temperature sensor and monitor controller component flow, where the server dictated the priority. TCP was used as the transport protocol and 20 Mbps of forward and reverse bandwidth was requested for each type of network QoS traffic.

We also define a taxonomy for evaluating technologies that provide CPU and network QoS assurances to end-to-end DRE application flows. This taxonomy is used to compare NetQoPE's methodology of provisioning network QoS for these flows with conventional approaches, including (1) object-oriented [El-Gendy et al. 2004; Schantz et al. 1999; Wang et al. 2000; Schantz et al. 2003], (2) aspect-oriented [Duzan et al. 2004], and (3) component middleware-based [de Miguel 2002; Sharma et al. 2004] approaches. Below we describe how each approach provides the following functionality needed to leverage network QoS mechanism capabilities:

• **QoS Requirements specification**. In conventional approaches applications use (1) middleware-based APIs [El-Gendy et al. 2004; Wang et al. 2000], (2) contract definition languages [Schantz et al. 1999; Schantz et al. 2003], (3) runtime aspects [Duzan et al. 2004], or (4) specialized component middleware container interfaces [de Miguel 2002] to specify QoS requirements. These approaches do not, however, provide capabilities to specify both CPU and network requirements and assume that physical node placement for all components are decided (*i.e.*, applications are already deployed in appropriate hosts) before the network resource allocations are requested using the appropriate APIs. This assumption allows those applications to specify the source and destination IP addresses of the applications when requesting network resources for an end-to-end application flow.

Moreover, application source code must change whenever the deployment context (*e.g.*, different physical node allocations, component deployment for a different usecase) and the associated QoS requirements (*e.g.*, CPU or network resource requirements) change, which limits reusability. In contrast, NetQoS provides domain-specific, declarative techniques that increase reusability across different deployment contexts and alleviate the need to specify QoS requirements programmatically, as described in Section 3.1.

• **Resource allocation**. Conventional approaches require application deployment before their per-flow network resource requirements can be provisioned by network QoS mechanisms. Recall that appropriate hosts for each application is determined by intelligent CPU allocation algorithms [de Niz and Rajkumar 2006] before their per-flow network resource requirements can be provisioned by network QoS mechanisms. If the required network resources cannot be allocated for these applications after a CPU allocation decision is made, however, the following steps occur:

(1) The applications must be stopped

(2) Their source code must be modified to specify new resource requirements (*e.g.*, either source and destination nodes of the components can be changed, forcing

application re-deployments as well or for the same pair of source and destination nodes the network resource requirements could be changed) and

(3) The resource reservation process must be restarted.

This approach is tedious since applications may be deployed and re-deployed multiple times, potentially on different nodes. In contrast, NetRAF handles deployment changes via NetQoS models (see Section 3.2) at pre-deployment, *i.e.*, *before* applications have been deployed, thereby reducing the effort needed to change deployment topology or application QoS requirements.

• **Network QoS enforcement**. Conventional approaches modify application source code [Schantz et al. 2003] or programming model [de Miguel 2002] 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, NetCON uses a container programming model that transparently enforces runtime QoS for applications without changing their source code or programming model, as described in Section 3.3.

We now compare the effort required to provision end-to-end QoS to the 4 end-to-end application flows described above using conventional manual approaches vs. the NetQoPE model-driven approach. We decompose this effort across the following general steps: (1) *implementation*, where software developers write code to specify resource requirements and allocate needed resources, (2) *deployment*, where system deployers map (or stop) application components on their target nodes, and (3) *modeling tool use*, where application developers use NetQoPE to model a DRE application structure, specify per-application CPU resource and per-flow network resource requirements, and allocate needed CPU and network resources. In our evaluation, a complete QoS provisioning lifecycle consists of specifying resource requirements, allocating both CPU and network resources, deploying applications, and stopping applications when they are finished.

To compare NetQoPE with other conventional efforts, we devised a realistic scenario for the 4 end-to-end application flows described above. In this scenario, three sets of experiments were conducted with the following deployment variants[7]:

• **Baseline deployment**. This variant configured all 4 end-to-end application flows with the CPU and network QoS requirements as described above. The manual effort required using conventional approaches for the baseline deployment involved 10 steps: (1) modify source code for each of the 8 components to specify their QoS requirements (8 implementation steps – note that CPU allocation algorithms were used to determine the appropriate physical node allocations for the applications before network resources were requested for each application flow), (2) deploy all components (1 deployment step), and (3) shutdown all components (1 deployment step).

In contrast, the effort required using NetQoPE involved the following 4 steps: (1) model the DRE application structure of all 4 end-to-end application flows using NetQoS (1 modeling step), (2) annotate QoS specifications on each application

[7]In each of the experiment variants, we kept the same per-application CPU resource requirements, but varied the network resource requirements for the application flows.

and each end-to-end application flow (1 modeling step), (3) deploy all components (1 deployment step – this step also involved allocation of both CPU and network resources for applications using NetRAF's two step allocation process described in Section 3.2), and (4) shutdown all components (1 deployment step).

• **QoS modification deployment**. This variant demonstrated the effect of changes in QoS requirements on manual efforts by modifying the bandwidth requirements from 20 Mbps to 12 Mbps for each end-to-end flow. As with the baseline variant above, the effort required using a conventional approach for the second deployment was 10 steps since source code modifications were needed as the deployment contexts changed (in this case the bandwidth requirements changed across 4 different deployment contexts – however, the CPU resource requirements did not change, and hence the application physical node allocations did not change as well).

In contrast, the effort required using NetQoPE involved 3 steps: (1) annotate QoS specifications on each end-to-end application flow (1 modeling step), (2) deploy all components (1 deployment step), and (3) shutdown all components (1 deployment step). Application developers also reused NetQoS' application structure model created for the initial deployment, which helped reduce the required efforts by a step.

• **Resource (re)reservation deployment**. This variant demonstrated the effect of changes in QoS requirements and resource (re)reservations taken together on manual efforts. We modified bandwidth requirements of all flows from 12 Mbps to 16 Mbps. We also changed the temperature sensor controller component to use the high reliability (HR) class instead of the best effort BE class. Finally, we increased the background HR class traffic across the hosts 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.

The effort required using a conventional approach for the third deployment involved 13 steps: (1) modify source code for each of the 8 components to specify their QoS requirements (8 implementation steps), (2) deploy all components (1 deployment step), (3) shutdown the temperature sensor component (1 deployment step – note that the 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 step), (5) redeploy the temperature sensor component (1 deployment step – note that the CPU allocation algorithms were rerun to change physical node allocations), and (6) shutdown all components (1 deployment step).

In contrast, the effort required using NetQoPE for the third deployment involved 4 steps: (1) annotate QoS specifications on each end-to-end application flow (1 modeling step), (2) begin deployment of all components, though NetRAF's pre-deployment-time allocation capabilities determined the resource allocation failure and prompted the NetQoPE application developer to change the QoS requirements (1 pre-deployment step), (3) re-annotate QoS requirements for the temperature sensor component flow (1 modeling step) (4) deploy all components (1 deployment step), and (5) shutdown all components (1 deployment step).

Table I summarizes the step-by-step analysis described above. These results show

Approaches	# Steps in Experiment Variants		
	First	Second	Third
NetQoPE	4	3	5
Conventional	10	10	13

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

that conventional approaches incurred roughly an order of magnitude more effort than NetQoPE to provide CPU and network QoS assurance for end-to-end application flows. Closer examination shows that in conventional approaches, application developers spend substantially more effort developing software that can work across different deployment contexts. Moreover, this process must be repeated when deployment contexts and their associated QoS requirements change. In addition, conventional implementations are complex since the requirements are specified directly using middleware [Wang et al. 2000] and/or network QoS mechanism APIs [L. Zhang and S. Berson and S. Herzog and S. Jamin 1997].

Application (re)deployments are also required whenever reservation requests fail. In this experiment only 1 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 often have scores of flows—conventional approaches require significantly more effort.

In contrast, NetQoPE's ability to "write once, deploy multiple times for different QoS requirements" increases deployment flexibility and extensibility in environments that deploy many reusable software components. 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.*, with bandwidth reservations of 20 Mbps, 12 Mbps, and 16 Mbps. 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 could make different forward and reverse bandwidth reservations (*e.g.*, 4, 8, or 10 Mbps). As shown in Table II, NetQoS auto-generates over 1,300 lines of XML code for these scenarios, which would otherwise be handcrafted by application developers. These results demonstrate that NetQoPE's model-driven

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 II: Generated Lines of XML Code

CPU and network QoS provisioning capabilities significantly reduce application development effort compared with conventional approaches. Moreover, NetQoPE also provides increased flexibility when deploying and provisioning multiple application end-to-end flows in multiple deployment and diverse QoS contexts.

4.3 Evaluating the Overhead of NetQoPE for Normal Operations

Rationale. NetQoPE provides network QoS to applications via the multi-stage architecture shown in Figure 4. This experiment evaluates the runtime performance overhead of using NetQoPE to enforce network QoS.

Methodology. DRE system developers can use NetQoPE at design time to specify network QoS requirements on the application flows, as described in Section 3.1. Based on the specified network QoS requirements, NetRAF interacts with the Bandwidth Broker to allocate per-flow network resources at pre-deployment time. By providing design- and pre-deployment-time capabilities, NetQoS and NetRAF thus incur no runtime overhead. In contrast, NetCON configures component middleware containers at post-deployment-time by adding DSCP markings to IP packets when applications invoke remote operations (see Section 3.3). NetCON may therefore incur runtime overhead, *e.g.*, when containers apply a network policy models to provide the source application with an object reference to the destination application.

To measure NetCON's overhead, we conducted an experiment to determine the runtime overhead of the container when it performs extra work to apply the policies that add DSCPs to IP packets. This experiment had the following variants: (1) the client container was not configured by NetCON (no network QoS required), (2) the client container was configured by NetCON to apply the CLIENT_PROPAGATED network policy, and (3) the client container was configured by NetCON to apply the SERVER_DECLARED network policy. This experiment had no background network load to isolate the effects of each variant.

Our experiment had no network congestion, so QoS support was thus not needed[8]. The network priority models were therefore configured with DSCP values of 0 for both the forward and reverse direction flows. `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), so no edge router processing overhead was incurred. We configured the experiment to pinpoint only the overhead of the container no other entities in the path of client remote communications.

Analysis of results. Figure 10 shows the average roundtrip latencies experienced by clients in the three experiment variants (in this figure CP is the CLIENT_PROPAGATED network priority model and SD is the SERVER_DECLARED model). To honor the network policy models, the NetQoPE middleware added the request/reply DSCPs to the IP packets. The latency results shown in Figure 10 are all similar, which shows that NetCON is efficient and adds negligible overhead to applications. If another variant of the experiment was run with background network loads, network resources will be allocated and the appropriate DSCP values used for those application flows. The NetCON runtime overhead will remain the same, however, since

[8]Our experimentation goal was to measure the runtime overhead of using NetQoPE middleware to enforce network QoS. So we wanted to remove other effects in the experiment such as network congestion.

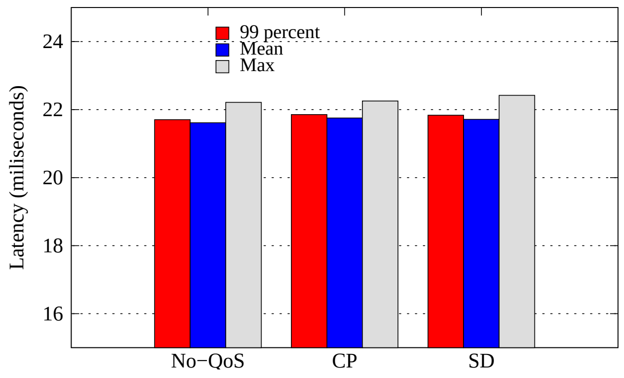


Fig. 10: Overhead of NetQoPE's Policy Framework

the same middleware infrastructure is used, only with different DSCP values.

4.4 Evaluating NetQoPE's QoS Customization Capabilities

Rationale. NetQoPE's model-driven approach enhances flexibility by enabling the reuse of application source code 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 evaluates the benefits of these capabilities empirically.

Methodology. We identified four flows from Figure 9 and modeled them using NetQoS as follows:

—A fire sensor controller component on blade A uses the high reliability (HR) class to send potential fire alarms in the parking lot to the monitor controller component on blade D.

—A fire sensor controller component on blade B uses the high priority (HP) class to send potential fire alarms in the server room to the monitor controller component on blade F.

—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.

—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.

The CLIENT_PROPAGATED network policy was used for all flows, except for the temperature sensor and monitor controller component flow, which used the SERVER_DECLARED network policy.

We executed two variants of this experiment. The first variant used TCP as the transport protocol and requested 20 Mbps of forward and reverse bandwidth for each type of QoS traffic. `TestNetQoPE` configured each application flow 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

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 III: Application Background Traffic

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 we recorded 100,000 network delay values (in milliseconds) for each network QoS class. Those network delay values were then sorted 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 resulting values. For example, the 1 ms bucket contained only samples that are <= to 1 ms in their resultant value, the 2 ms bucket contained only samples whose resultant values were <= 2 ms but > 1 ms, etc.

To evaluate application performance in the presence of background network loads, several other applications were run in both experiments, as described in Table III (in this table 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 which DSCP values to use[9]. After deploying the applications, NetCON configured the containers to use the appropriate network priority models to add DSCP values to IP packets when applications invoked remote operations.

Analysis of results. Figure 11 shows the results of experiments when the deployed applications were configured with different network QoS classes and sent TCP traffic.

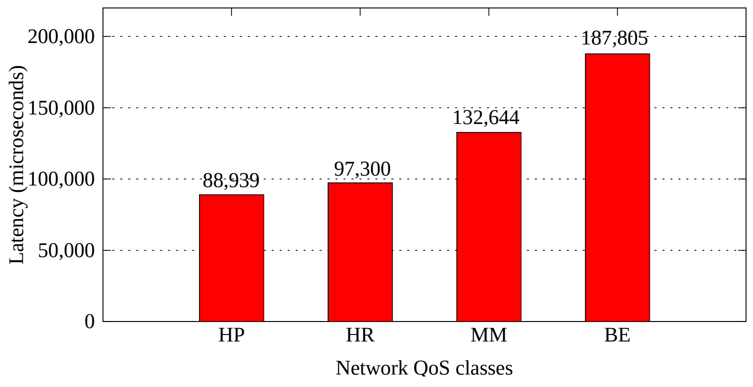


Fig. 11: Average Latency under Different Network QoS Classes

[9]To allocate network resources for each flow, NetRAF used the physical node allocations determined by the CPU allocation algorithms employed by NetQoS.

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 is not differentiated from the competing background traffic and thus 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 12 shows the (1) cardinality of the network delay groupings for different network QoS classes under different ms 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, the 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 12 also show that the application using the MM class experienced more predictable latency than applications using BE and HR class. Approximately 94% of the MM class invocations had their normalized delays within 1 ms. 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, because the queue sizes at the routers are large enough to hold more packets when the network is congested.

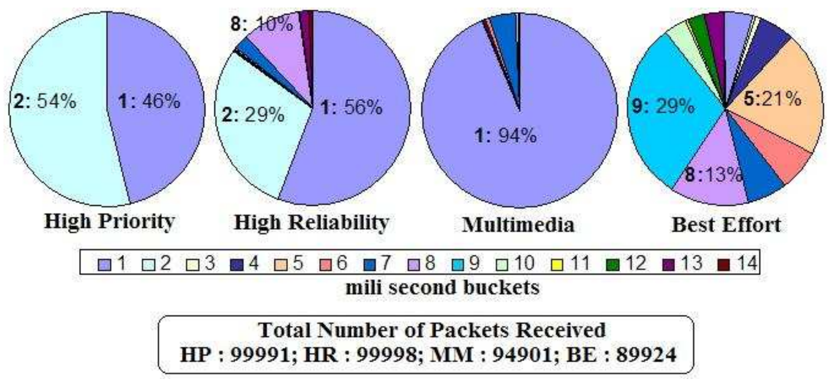


Fig. 12: Jitter Distribution under Different Network QoS Classes

These results demonstrate that NetQoPE's automated model-driven middleware-guided 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, therefore, applications can leverage the capabilities of network QoS mechanisms with minimal effort, as described in Section 4.2.

These results also demonstrate the following QoS customization possibilities for a set of application communications (*e.g.*, fire sensor and monitor controller component):

—*Different network QoS performance*, *e.g.,* HP communication between blades A and D, and HR communication between blades B and F.

—*Different transport protocols for communication*, *e.g.*, TCP and UDP.

—*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.

These results show how NetQoPE's ability to "write once, deploy multiple times for different QoS requirements" increased 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.

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 [Wang et al. 2000; Schantz et al. 1999; Schantz et al. 2003; El-Gendy et al. 2004] 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 this related work 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, [Jordan et al. 2004] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [Wichadakul et al. 2001] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [de Miguel 2002] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. Synergy [Repantis et al. 2006] 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.

Our previous work [Wang et al. 2004] has focused on mechanisms that add real-time QoS aspects into a component middleware, so that component middleware applications can enforce CPU QoS at runtime in a non-invasive manner. NetQoPE builds on that work but solves the following orthogonal but important problems - how to decide what all applications need to operate in a particular processor such that both their CPU and network resources can be provisioned, and how to enforce network QoS for such applications at runtime. Combined with our previous work, NetQoPE can thus manage and enforce both CPU and network QoS for applications. The work reported in this paper, however, focuses on how to combine CPU allocation algorithms with network QoS mechanisms and provision and enforce network QoS for applications.

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 [Llambiri et al. 2003; Stewart and Shen 2005] has studied and analyzed application communication and access patterns to determine collocated placements of heavily communicating components. Other research [de Niz and Rajkumar 2006; Gopalakrishnan and Caccamo 2006] has focused on intelligent component allocation 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 [Balasubramanian et al. 2007] enables DRE system developers to define component interfaces, their implementations, and assemblies, facilitating deployment of LwCCM-based applications. VEST [Stankovic et al. 2003] and AIRES [Gu et al. 2003] analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provides automated allocation of components to processors. SysWeaver [de Niz et al. 2006] 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 CPU and network QoS for applications in distributed real-time and embedded (DRE) systems. The lessons we learned developing NetQoPE and applying it to a representative DRE system case study thus far include:

—NetQoPE's domain-specific modeling languages (*e.g.*, NetQoS) help capture per-deployment QoS requirements of applications so that CPU and 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, as shown in

Section 3.1.

—Programming network QoS mechanisms directly in application code requires the deployment and execution of applications before they can determine if the required network resources are available to meet QoS needs. Conversely, providing these capabilities via NetQoPE's model-driven, middleware framework helps guide resource allocation strategies *before* application deployment, thereby simplifying validation and adaptation decisions, as shown in Section 3.2.

—NetQoPE's model-driven deployment and configuration tools help configure the underlying component middleware transparently 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 programming model used by applications. Applications therefore need not change how they communicate at runtime since network QoS settings can be added transparently, as shown in Section 3.3.

—NetQoPE's strategy of allocating network resources to applications before deployment may be too limiting for certain types of DRE systems. In particular, applications in open DRE systems [Wang et al. 2007] might not consume all their resource allotment at runtime, in which case NetQoPE may underutilize system resources. Our future work is 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 integrating dynamic resource management algorithms [Lardieri et al. 2007] with NetQoPE to provide predictable network performance for applications in open DRE systems.

Most of NetQoPE's model-driven middleware platforms and tools described in this paper and used in the experiments are available in open-source format from `www.dre.vanderbilt.edu/cosmic` and in the CIAO component middleware available at `www.dre.vanderbilt.edu`. The one exception is the Bandwidth Broker, which is a product licensed by Telcordia.

REFERENCES

Ákos Lédeczi, Árpád Bakay, Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. 2001. Composing domain-specific design environments. *Computer 34,* 11, 44–51.

Balasubramanian, J., Tambe, S., Dasarathy, B., Gadgil, S., Porter, F., Gokhale, A., and Schmidt, D. C. 2008. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS' 08: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 113–122.

Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., and Schmidt, D. C. 2005. A platform-independent component modeling language for distributed real-time and embedded systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 190–199.

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

Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. 1998. An Architecture for Differentiated Services.

Buschmann, F., Henney, K., and Schmidt, D. C. 2007. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4*. Wiley and Sons, New York.

CHEN, J.-J., YANG, C.-Y., KUO, T.-W., AND TSENG, S.-Y. 2007. Real-time task replication for fault tolerance in identical multiprocessor systems. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 249–258.

DASARATHY, B., GADGIL, S., VAIDYANATHAN, R., NEIDHARDT, A., COAN, B., PARMESWARAN, K., MCINTOSH, A., AND PORTER, F. 2007. Adaptive network qos in layer-3/layer-2 networks as a middleware service for mission-critical applications. *J. Syst. Softw. 80,* 7, 972–983.

DASARATHY, B., GADGIL, S., VAIDYANATHAN, R., PARMESWARAN, K., COAN, B., CONARTY, M., AND BHANOT, V. 2005. Network qos assurance in a multi-layer adaptive resource management scheme for mission-critical applications using the corba middleware framework. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 246–255.

DE MIGUEL, M. A. 2002. Integration of qos facilities into component container architectures. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Computer Society, Washington, DC, USA, 394.

DE NIZ, D., BHATIA, G., AND RAJKUMAR, R. 2006. Model-based development of embedded systems: The sysweaver approach. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 231–242.

DE NIZ, D. AND RAJKUMAR, R. 2006. Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems 2,* 3, 196–208.

DUZAN, G., LOYALL, J., SCHANTZ, R., SHAPIRO, R., AND ZINKY, J. 2004. Building Adaptive Distributed Applications with Middleware and Aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 66–73.

EIDE, E., STACK, T., REGEHR, J., AND LEPREAU, J. 2004. Dynamic cpu management for real-time, middleware-based systems. In *RTAS '04: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 286.

EL-GENDY, M., BOSE, A., PARK, S.-T., AND SHIN, K. 2004. Paving the first mile for qos-dependent applications and appliances. In *IWQoS '04: Proceedings of the 12th International Workshop on Quality of Service*. IEEE Computer Society, Washington, DC, USA, 245–254.

FOSTER, I., FIDLER, M., ROY, A., SANDER, V., AND WINKLER, L. 2004. End-to-end Quality of Service for High-end Applications. *Computer Communications 27,* 14 (Sept.), 1375–1388.

GADGIL, S., DASARATHY, B., PORTER, F., PARMESWARAN, K., AND VAIDYANATHAN, R. 2007. Fast recovery and qos assurance in the presence of network faults for mission-critical applications in hostile environments. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, Washington, DC, USA, 283–292.

GOPALAKRISHNAN, S. AND CACCAMO, M. 2006. Task partitioning with replication upon heterogeneous multiprocessor systems. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 199–207.

GU, Z., KODASE, S., WANG, S., AND SHIN, K. G. 2003. A model-based approach to system-level dependency and real-time analysis of embedded software. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 78.

JORDAN, M., CZAJKOWSKI, G., KOUKLINSKI, K., AND SKINNER, G. 2004. Extending a j2ee^{TM}server with dynamic and flexible resource management. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 439–458.

KARSAI, G., SZTIPANOVITS, J., LEDECZI, A., AND BAPTY, T. 2003. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE 91,* 1 (Jan.), 145–164.

KRISHNA, A. S., SCHMIDT, D. C., AND KLEFSTAD, R. 2004. Enhancing real-time corba via real-time java features. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. IEEE Computer Society, Washington, DC, USA, 66–73.

L. ZHANG AND S. BERSON AND S. HERZOG AND S. JAMIN. 1997. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification.

LARDIERI, P., BALASUBRAMANIAN, J., SCHMIDT, D. C., THAKER, G., GOKHALE, A., AND DAMIANO, T. 2007. A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems. *Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems 80,* 7 (July), 984–996.

LEHOCZKY, J., SHA, L., AND DING, Y. 1989. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS '89: Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, USA, 166–171.

LLAMBIRI, D., TOTOK, A., AND KARAMCHETI, V. 2003. Efficiently distributing component-based applications across wide-area environments. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 412.

MAHAJAN, M. AND PARASHAR, M. 2003. Managing qos for multimedia applications in the differentiated services environment. *J. Netw. Syst. Manage. 11,* 4, 469–498.

MEHRA, A., VERMA, D. C., AND TEWARI, R. 2000. Policy-based diffserv on internet servers: The AIX approach (on the wire). *IEEE Internet Computing 4,* 5, 75–80.

MIGUEL, M. A. D. 2002. Qos-aware component frameworks. In *IWQOS '02: Proceedings of the Tenth International Workshop on Quality of Service*. IEEE Computer Society, Washington, DC, USA, 51.

NAHRSTEDT, K. 1999. To overprovision or to share via qos-aware resource management? In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, Washington, DC, USA, 35.

NAHRSTEDT, K., XU, D., WICHADAKUL, D., AND LI, B. 2001. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine 39,* 11 (Nov.), 140–148.

NECHYPURENKO, A., LU, T., DENG, G., TURKAY, E., SCHMIDT, D. C., AND GOKHALE, A. S. 2004. Concern-based composition and reuse of distributed systems. In *ICSR*. Vol. 3107. Springer, Madrid, Spain, 167–184.

Object Management Group 2002. *Real-time CORBA Specification*, OMG Document formal/05-01-04 ed. Object Management Group.

Object Management Group 2003. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed. Object Management Group.

Object Management Group 2008. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 ed. Object Management Group.

OMG 2006. *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 ed. OMG.

PYARALI, I., SCHMIDT, D., AND CYTRON, R. 2003. Techniques for enhancing real-time corba quality of service. *Proceedings of the IEEE 91,* 7 (July), 1070–1085.

REPANTIS, T., GU, X., AND KALOGERAKI, V. 2006. Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In *Middleware '06: Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 322–341.

SCHANTZ, R., ZINKY, J., KARR, D., BAKKEN, D., MEGQUIER, J., AND LOYALL, J. 1999. An Object-level Gateway Supporting Integrated-Property Quality of Service. *ISORC 00*, 223.

SCHANTZ, R. E., LOYALL, J. P., RODRIGUES, C., AND SCHMIDT, D. C. 2006. Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper. 36,* 11-12, 1189–1208.

SCHANTZ, R. E., LOYALL, J. P., RODRIGUES, C., SCHMIDT, D. C., KRISHNAMURTHY, Y., AND PYARALI, I. 2003. Flexible and adaptive qos control for distributed real-time and embedded middleware. In *Middleware*. Lecture Notes in Computer Science, vol. 2672. Springer, Rio de Janeiro, Brazil, 374–393.

SCHMIDT, D. C., LEVINE, D. L., AND MUNGEE, S. 1998. The Design and Performance of Real-time Object Request Brokers. *Computer Communications 21,* 4 (Apr.), 294–324.

SCHMIDT, D. C., NATARAJAN, B., GOKHALE, A., WANG, N., AND GILL, C. 2002. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online 3,* 2 (Feb.).

SCHMIDT, D. C., SCHANTZ, R., MASTERS, M., CROSS, J., SHARP, D., AND DIPALMA, L. 2001. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. In *CrossTalk - The Journal of Defense Software Engineering*. Software Technology Support Center, Hill AFB, Utah, USA, 10–16.

SHARMA, P. K., LOYALL, J. P., HEINEMAN, G. T., SCHANTZ, R. E., SHAPIRO, R., AND DUZAN, G. 2004. Component-based dynamic qos adaptations in distributed real-time and embedded systems. In *CoopIS/DOA/ODBASE (2)*. Springer, Agia Napa, Cyprus, 1208–1224.

SHARP, D. C. AND ROLL, W. C. 2003. Model-Based Integration of Reusable Component-Based Avionics System. Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003.

SNOONIAN, D. 2003. Smart Buildings. *IEEE Spectrum 40,* 8, 18–23.

STANKOVIC, J. A., ZHU, R., POORNALINGAM, R., LU, C., YU, Z., HUMPHREY, M., AND ELLIS, B. 2003. VEST: An Aspect-Based Composition Tool for Real-Time Systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 58.

STEWART, C. AND SHEN, K. 2005. Performance modeling and system management for multi-component online services. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 71–84.

URGAONKAR, B., ROSENBERG, A., AND SHENOY, P. 2007. Application Placement on a Cluster of Servers. *International Journal of Foundations of Computer Science 18,* 5, 1023–1042.

URGAONKAR, B. AND SHENOY, P. 2004. Sharc: Managing cpu and network bandwidth in shared clusters. *IEEE Trans. Parallel Distrib. Syst. 15,* 1, 2–17.

WANG, N., GILL, C., SCHMIDT, D. C., AND SUBRAMONIAN, V. 2004. Configuring Real-time Aspects in Component Middleware. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*. Vol. 3291. Springer-Verlag, Agia Napa, Cyprus, 1520–1537.

WANG, P., YEMINI, Y., FLORISSI, D., AND ZINKY, J. 2000. A distributed resource controller for qos applications. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*. IEEE Computer Society, Los Alamitos, CA, USA, 143–156.

WANG, X., JIA, D., LU, C., AND KOUTSOUKOS, X. 2007. DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems. *Parallel and Distributed Systems, IEEE Transactions on 18,* 7, 996–1009.

WICHADAKUL, D., NAHRSTEDT, K., GU, X., AND XU, D. 2001. 2k: An integrated approach of qos compilation and reconfigurable, component-based run-time middleware for the unified qos management framework. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, London, UK, 373–394.

Received September 2008;