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

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

Abstract— **Distributed real-time and embedded (DRE) systems are composed of applications with diverse CPU utilization requirements. These applications participate in many end-to-end application flows with diverse network bandwidth requirements. Prior research has advanced the state-of-the-art on (1) bin-packing algorithms, such as first fit decreasing (FFD), to allocate per-application CPU resources, and (2) network-layer quality-of-service (QoS) mechanisms, such as differentiated services (DiffServ), to manage per-flow network resources. Relatively little work, however, has focused on how applications can be deployed and configured to leverage such advances for addressing their end-to-end 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 describe how our NetQoPE model-driven component middleware framework shields applications from the complexities of lower-level CPU and network QoS mechanisms to simplify (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 well NetQoPE provides QoS assurance for applications in DRE systems. Our results demonstrate that NetQoPE provides flexible QoS configuration and provisioning capabilities by leveraging CPU and network QoS mechanisms without modifying application source code.

I. Introduction

Emerging trends. 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 [2], inventory tracking systems [3], avionics mission computing systems [4], and intelligence, surveillance and reconnaissance systems [5]. These systems consist of applications that participate in different end-to-end application flows and operate in dynamic environments with varying levels of CPU, network connectivity, and bandwidth availability. CPU and network resources in such target environments must be configured so that DRE applications can have their requirements satisfied end-to-end.

Network quality of service (QoS) mechanisms, such as integrated services (IntServ) [6] and differentiated services (DiffServ) [7], support a range of network service levels for applications in DRE systems. To leverage the services of these QoS mechanisms, however, applications have conventionally used relatively low-level APIs provided by the switching elements.

Moreover, to ensure end-to-end QoS, applications must be deployed in appropriate end hosts so that required CPU and network resources for the application flow between those hosts are provided. To configure required CPU resources for applications, prior work has focused on resource allocation algorithms [8], [9] that satisfy timing requirements of applications in a DRE system. After the required resources are provisioned and the networking elements are configured correctly, applications invoke remote operations by adding a service level-specific identifier (*e.g.*, DiffServ codepoint (DSCP)) to the IP packets. DiffServ-enabled network routers parse the IP packets and provide the appropriate service level-specific packet forwarding behavior.

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. An earlier version of this paper appeared at the Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'2008) [1]

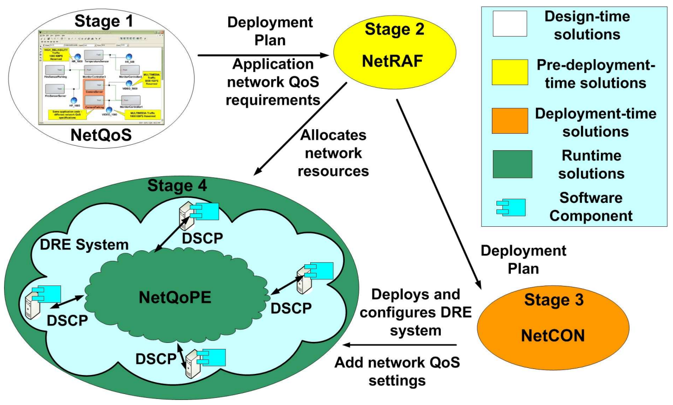
Addressing limitations of current approaches. Although the network and CPU QoS mechanisms described above are powerful, it is tedious and error-prone to develop applications that interact directly with low-level QoS mechanism APIs written imperatively in third-generation languages, such as C++ or Java. For example, applications must make multiple invocations on network QoS mechanisms to accomplish key network QoS activities, such as QoS mapping, admission control, and packet marking. To address part of this problem, middleware-based network QoS provisioning solutions [10], [11], [12], [13] have been developed that allow applications to specify their coordinates (source and destination IP and port addresses) and per-flow network QoS requirements via higher-level frameworks. The middleware frameworks—rather than the applications—are thus responsible for converting high-level specifications of QoS intent into low-level network QoS mechanism APIs.

Although middleware frameworks alleviate many accidental complexities of low-level network QoS mechanism APIs, they can still be hard to evolve and extend. In particular, application source code changes may be necessary whenever changes occur to the deployment contexts (*e.g.*, source and destination nodes of applications), per-flow requirements, IP packet identifiers, or middleware APIs. Moreover, applications must explicitly determine the optimal source and destination nodes before they can obtain network performance assurances via the underlying network QoS mechanisms.

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 completely 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 instance deployments 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) [14] to raise the level of abstraction of DRE system design higher than using imperative third-generation programming languages. NetQoPE allows system engineers and software developers to perform deployment-time analysis (such as schedulability analysis [15]) of non-functional system properties (such as network QoS assurances for end-to-end application flows). The result is enhanced deployment-time assurance that application QoS requirements will be satisfied.

NetQoPE deploys and configures component middleware-based applications in DRE systems and enforces their network and CPU QoS requirements using the four-stage (*i.e.*, design-, pre-deployment-, deployment-, and run-time) approach shown in Figure 1. The innovative elements of NetQoPE's



1: NetQoPE's Four-stage Architecture

four-stage architecture include the following:

- The **Network QoS Specification Language** (NetQoS), which is a DSML that supports design-time specification of per-application CPU resource requirements, as 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.
- The **Network Resource Allocation Framework** (NetRAF), which is a middleware-based resource allocator framework that uses the network QoS requirements captured by *NetQoS* as input at pre-deployment time to help guide QoS provisioning requests on the underlying network QoS mecha-

nism at deployment time. By providing application-transparent, per-fbw resource allocation capabilities at pre-deployment-time using a Bandwidth Broker [16], *NetRAF* provides network bandwidth guarantees for application fbws.

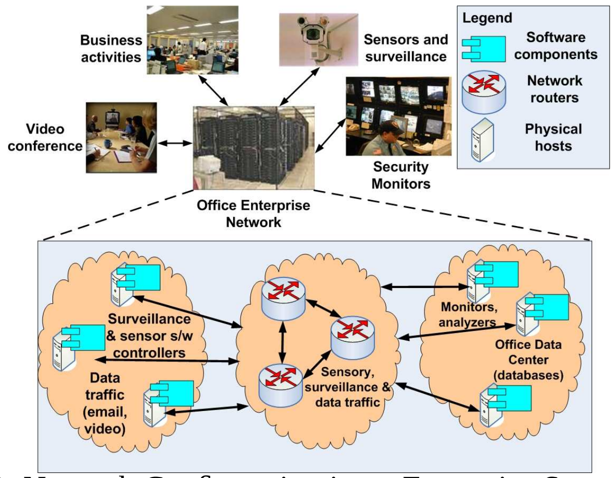
• The **Network QoS Configurator** (NetCON), which is a middleware-based network QoS configurator that provides deployment-time configuration of component middleware containers. NetCON adds fbw-specific identifiers (*e.g.*, DSCPs) to IP packets at runtime when applications invoke remote operations. By 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.

Figure 1 shows how the output of each stage in NetQoPE serves as input for the next stage, which helps automate the deployment and configuration of DRE applications with network QoS support.

Paper organization. The remainder of the paper is organized as follows: Section II describes a case study that motivates common requirements associated with provisioning network QoS for DRE systems; Section III explains how NetQoPE addresses those requirements via its model-driven component middleware framework; Section IV evaluates the capabilities provided by NetQoPE; Section V compares our work on NetQoPE with related research; and Section VI presents concluding remarks and lessons learned.

II. MOTIVATING NETQOPE'S QOS PROVISIONING CAPABILITIES

To demonstrate and evaluate NetQoPE's model-driven, middleware-guided network and CPU QoS provisioning capabilities this section presents a case study of a representative DRE system in an office enterprise security and hazard sensing environment, as which is shown in Figure 2. Enterprises often transport network traffic using an IP network over high-speed Ethernet. Network traffic in an enterprise can be grouped into several classes, including (1) e-mail, videoconferencing, and normal business traffic, and (2) sensory and imagery traffic of the safety/security hardware (such as fire/-smoke sensors) installed on office premises. Our case study assumes that safety/security traffic is more critical than other traffic, and thus focuses on


Business activities
Sensors and surveillance
Legend
Software components
Network routers
Physical hosts
Video conference
Security Monitors
Office Enterprise Network
Surveillance & sensor s/w controllers
Data traffic (email, video)
Sensory, surveillance & data traffic
Monitors, analyzers
Office Data Center (databases)

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

how NetQoPE's model-driven, middleware-guided mechanisms help ensure the specified QoS for this type of traffic in the presence of other traffic that shares the same network.

Our case study uses software controllers to manage hardware devices, such as sensors and monitors, shown in Figure 2. Each sensor/camera software controller filters the sensory/imagery information and relays them to the monitor software controllers that display the information. Modern office enterprises deploy many sensors and monitors, which may have different CPU utilization requirements. Moreover, communication between sensors may have various network QoS requirements, such as different network bandwidth requirements, *e.g.*, 20 Mbps vs 5 Mbps.

The software controllers in our case study were developed using Lightweight CCM (LwCCM) [17], which is described in Sidebar 1. Moreover, the traffic between these software controllers uses a Bandwidth Broker [16] to manage network resources using DiffServ network QoS mechanisms. Although the case study in this paper focuses on LwCCM and DiffServ, NetQoPE is designed for use with other network QoS mechanisms (*e.g.*, IntServ) and component middleware technologies (*e.g.*, J2EE).

Component-based applications in our case study use Bandwidth Broker services via the following middleware-guided steps: (1) network QoS requirements are specified on each application fbw, along with information on the source/destination IP/port addresses that were determined by bin packing algorithms, (2) the Bandwidth Broker is invoked to reserve network resources along the network paths for each application fbw, configure the corresponding

Sidebar 1: Overview of Lightweight CORBA Component Model

Our case study is based on Lightweight CCM (LwCCM). Application functionality in LwCCM is provided through *components* that collaborate with other components via *ports* to create component *assemblies*. There are four types of CCM ports: *facet*, *receptacle*, *event source*, and *event sink*. CCM components provide services to other components by either *synchronous* communication in the form of operations on facet/-receptable ports or *asynchronous* communication in the form of eventtypes exchanged between event source/sink ports.

Assemblies in LwCCM are described using XML descriptors (mainly the *deployment plan* descriptor) defined by the OMG D&C [18] specification. The *deployment plan* includes details about the components, their implementations, and their connections with other components. The *deployment plan* also has a placeholder *configProperty* that is associated with elements (*e.g.*, components, connections) to specify their properties (*e.g.*, priorities) and resource requirements. Components are hosted in *containers* that provide the runtime operating environment (*e.g.*, load balancing, security, and event notification) for components to invoke remote operations.

network routers, and obtain per-flow DSCP values to help enforce network QoS, and (3) remote operations are invoked with appropriate DSCP values added to the IP packets so that configured routers can provide per-flow differentiated performance. Section III describes the challenges we encountered when implementing these steps in the context of our case study and shows how NetQoPE's four-stage architecture in Figure 1 helps resolve these challenges.

III. NetQoPE's Multistage Network QoS Provisioning Architecture

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

A. Challenge 1: Alleviating Complexities in CPU and Network QoS Requirements Specification

Context.: Every component's CPU utilization requirements guide the selection of the physical node to which it is deployed. After identifying the nodes on which to deploy the components, each application flow in a DRE system can 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. NetQoPE uses this information to configure network resources between two endpoint nodes to provide the required QoS.

Problem. Multiple feasible CPU allocations are possible with a given set of component CPU requirements. 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. Manually enumerating all these possible CPU allocation permutations is hard.

Network QoS requirements can affect CPU allocations. For example, a deployment plan for a component with multiple CPU allocations (*e.g.*, component A could be deployed on either node X or node Y) must ensure the deployment also satisfies network QoS requirements. Source and destination IP address are therefore needed to specify network QoS requirements properly.

Network QoS requirements can also change depending on the deployed context. For example, in our case study from Section II, multiple fire sensors are deployed at different importance levels and each sensor sends its sensory information to its corresponding monitors. 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 managing the fire sensor and the monitor have the same functionality.

Conventional techniques, such as hard-coded API approaches [13], require application source code modifications for each context. Writing this code manually to specify both CPU and network QoS requirements is tedious, error-prone, and non-scalable. In particular, it is hard to envision at development time all the contexts in which source code will be deployed.

Solution approach → Model-driven declarative CPU and network requirements specifi cation. NetQoPE provides a DSML called the *Network QoS Specification Language* (NetQoS), which is built using Generic Modeling Environment (GME) [19] and the Platform Independent Component Modeling Language (PICML) [20]. 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. NetQoS's network QoS classes correspond to the DiffServ levels of service provided by our Bandwidth Broker [21].[1] 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).

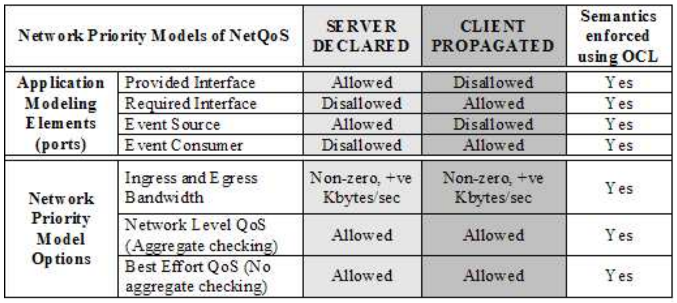
In LwCCM, components communicate using *ports* (described in Sidebar 1) that provide application-level communication endpoints. NetQoS provides capabilities to annotate communication ports with the network QoS requirement capabilities described above. In certain application fbws (*e.g.*, a monitor requesting location coordinates from a fire sensor in our case study) clients control the network priorities at which requests/replies are sent. In other application fbws (*e.g.*, a temperature sensor sends temperature sensory information to monitors), servers control the reception and processing of client requests. To support both models of communication (*i.e.*, whether clients vs. servers control network QoS for a fbw), NetQoS supports annotating each bi-directional fbw 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.

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

Defining network 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 priority models.

For example, NetQoS detects and fhgs 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 3 shows other examples of network priority models supports by NetQoS.



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

3: Network QoS Models Supported by NetQoS

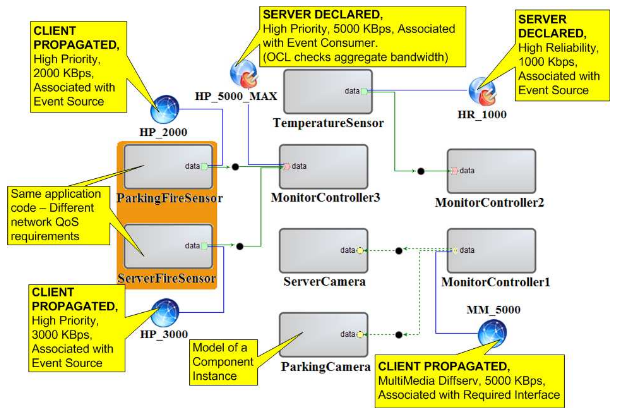
A server using the SERVER_DECLARED network priority model can also dictate that the total ingress bandwidth from all communicating clients cannot exceed a designated network bandwidth (*e.g.*, 30 Mbps). NetQoS checks the aggregation of egress bandwidth requested using all clients that communicate with 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 fbws. NetQoS provides this capability so application deployers can provision the underlying network

QoS mechanisms efficiently and flexibly.

After a model has been created and checked for type violations using built-in constraints, network resources must be allocated, which requires identifying component source and destination nodes. 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 Sidebar 1) 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 placements.

To perform network resource allocations (described in Section III-B), NetQoS's model interpreter captures the details about the components, their deployment locations (determined by the CPU allocation algorithms), and the network QoS requirements for each application flow they are part of the *deployment plan configProperty* tags (see Sidebar 1). Section III-B 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 4 shows a NetQoS model that highlights many of the capabilities described above. In this model, multiple


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_5000_MAX
HP_2000
HR_1000
TemperatureSensor
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

4: Applying NetQoS Capabilities to the Case Study

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 IV-C.

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 4) 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 enterprise office environment case study.

B. Challenge 2: Alleviating Complexities in Network Resource Allocation and Configuration

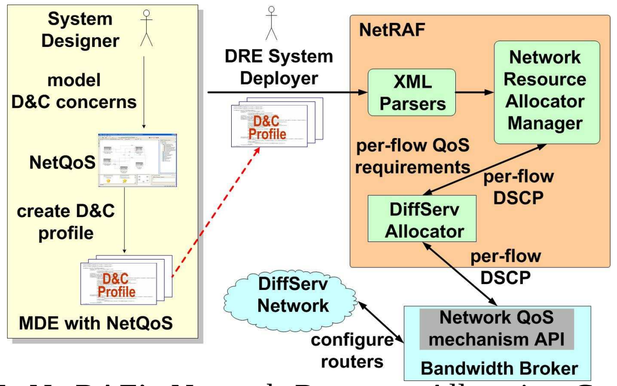
Context. After deciding where to deploy components on source and destination nodes, DRE systems must communicate with a network QoS mechanism API (*e.g.*, Bandwidth Broker for DiffServ networks) to allocate and configure network resources based on the network QoS requirements specified on the application 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 program 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 placement of the components with which they communicate. This component deployment information may unknown at development time since deployments are often not finalized until CPU placement 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. As shown in Figure 5, input to NetRAF is the set of feasible deployment plans generated by NetQoS model interpreter, which also embeds per-flow network QoS requirements.



5: NetRAF's Network Resource Allocation Capabilities

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 [11] that modify application source code to work with network QoS mechanisms, which become complex when source code is reused in a wide range of deployment contexts.

NetRAF's *Network Resource Allocator Manager* accepts application QoS requests at pre-deployment-time. It processes these requests in conjunction with a *DiffServ 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 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 shown in Section IV-C.

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 gracefullly, NetRAF provides hints to regenerate CPU placements 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 placements by adding a constraint to not deploy the components in the same source and destination nodes. After the revised CPU placements 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 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 fbws 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 fbws 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 [21] by feeding it one application fbw at a time. If all fbws *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 IV-C, this capability helps NetRAF incur lower overhead than conventional approaches [10], [11] that perform validation decisions when applications are deployed and operated at runtime.

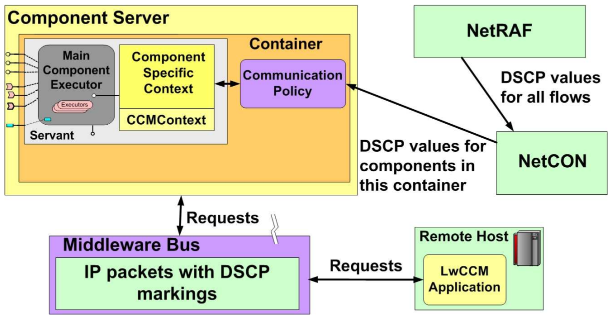
NetRAF's DiffServ Allocator instructs the Bandwidth Broker to reserve bi-directional resources in the specified network QoS classes, as described in Section III-A. The Bandwidth Broker determines the bi-directional DSCPs and NetRAF encodes those values as connection attributes in the deployment plan. In addition, the Bandwidth Broker uses its *Flow Provisioner* [16] to configure the routers to provide appropriate per-hop behavior when they receive IP packets with the specified DSCP values. Section III-C describes how component containers are auto-configured to add these DSCPs when applications invoke remote operations.

C. Challenge 3: Alleviating Complexities in Network QoS Settings Configuration

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

Problem. Application developers have historically written code that instructs the middleware to provide the appropriate runtime services, *e.g.*, DSCP markings in IP packets [12]. For example, fire sensors in our case study from Section II 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 $\rightarrow$ Deployment and runtime component middleware mechanisms. Sidebar 1 describes how LwCCM containers provide a runtime environment for components.[2] 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 fbw and encodes those DSCP values in the deployment plan, as shown in Figure 6.



6: 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, 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

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

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 III-A. 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 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 can be changed at runtime and different clients can access the servers with different network priorities. When the source component invokes a remote operation using the policy-applied object reference, NetCON adds the associated forward and reverse DSCP markings on the IP packets, thereby providing network QoS to the application flow. A 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 4, 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 4) that communicates with the deployed FireSensor components. Due to differences in importance of the FireSensor components deployed, however, the MonitorController software component the 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 the appropriate DSCP values are added to both forward and reverse communication paths when the MonitorController3 component communicates with either the FireSensorParking or FireSensorServer component at runtime. 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.

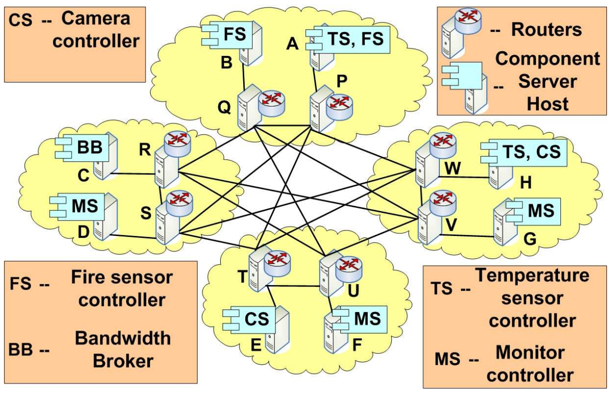
NetCON therefore allows DRE system developers to focus on their application component logic (*e.g.*, the MonitorController component in the case study), rather than wrestling with low-level mechanisms for provisioning network QoS. Moreover, NetCON provides these capabilities without modifying application code, thereby simplifying development and minimizing runtime overhead, as shown in Section IV-C.1.

IV. EVALUATING 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 validate that NetQoPE's automated model-driven approach can provide differentiated network performance for a variety of applications in DRE systems, such as our case study. We then demonstrate how NetQoPE's network QoS provisioning capabilities can significantly reduce application development effort compared with conventional approaches.

A. Evaluation Scenario

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 Gz XEONs with 1 GB memory, 40 GB disks, and 4 NICs per blade, and (2) 6 Cisco 3750G switches with 24 10/100/1000 MPS ports per switch. As shown in Figure 7, 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 modern 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.



7: 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 [22]. Our evaluations used DiffServ QoS and the associated Bandwidth Broker [21] software was hosted on blade *C*. All blades ran Fedora Core 4 Linux distribution configured using the real-time scheduling class. The blades were connected over a 1 Gbps LAN via virtual 100 Mbps links.

In our evaluation scenario, a number of sensory and imagery software controllers sent their monitored information to monitor controllers so that appropriate control actions could be performed by enterprise supervisors monitoring abnormal events. For example, Figure 7 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. Communication between these software controllers used one of the traffic classes (*e.g.*, HIGH PRIORITY (HP)) defined in Section III-A with the following capacities on all links: HP = 20 Mbps, HR = 30 Mbps, and MM = 30 Mbps. The BE class used the remaining available bandwidth in the network.

To emulate the network behavior of the software controllers when different network QoS requirements are provisioned, we created the `TestNetQoPE` performance benchmark suite.[3] We used `TestNetQoPE` to evaluate the flexibility, overhead, and performance of using NetQoPE to provide 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. High-resolution timer probes were used to measure roundtrip latency accurately for each client invocation.

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

B. Experimental Results and Analysis

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

C. Evaluating NetQoPE's Model-driven QoS Provisioning Capabilities

Rationale. As discussed in Section III, NetQoPE is designed to provision application CPU and network QoS mechanisms in an extensible manner. 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 7 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 component 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 network QoS assurances to end-to-end DRE application flows to compare NetQoPE's methodology of provisioning network QoS for these flows with conventional approaches, including (1) object-oriented [23], [11], [10], [12], (2) aspect-oriented [24], and (3) component middleware-based [13], [5] 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 [23], [10], (2) contract definition languages [11], [12], (3) runtime aspects [24], or (4) specialized component middleware container interfaces [13] to specify network 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 before the network resource allocations are requested.

Moreover, application source code must change whenever the deployment context (*e.g.,* different physical node placements, component deployment for a different usecase) and the associated QoS requirements (*e.g.,* CPU or network resource requirements) changes, 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 III-A.

• **Network resource allocation.** Conventional approaches require application deployment before their per-flow network resource requirements can be provisioned by network QoS mechanisms. If the required resources cannot be allocated for these applications, however, the following steps occur:

1) They 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 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 III-B) 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 [12] or programming model [13] 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 III-C.

We now compare the effort required to provision network 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, (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 and specify per-flow QoS requirements. In our evaluation, a complete QoS provisioning lifecycle consists of specifying requirements, allocating resources, deploying applications, and stopping applications when they are finished.

To compare NetQoPE with manual 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:

• **Baseline deployment.** This variant configured all 4 end-to-end application flows with the network QoS requirements as described above. The manual effort required using conventional approaches for the first deployment involved 10 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), 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 end-to-end application flow (1 modeling step), (3) deploy all components (1 deployment step), 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 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). 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's 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 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 – 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), 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 that conventional

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

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

approaches incurred roughly an order of magnitude more effort than NetQoPE to provide network QoS assurance for end-to-end application flows. Closer examination shows that in conventional approaches, application developers spend substantially more effort 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 [10] and/or network QoS mechanism APIs [6].

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

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

II: Generated Lines of XML Code

results demonstrate that NetQoPE's model-driven 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 network QoS contexts.

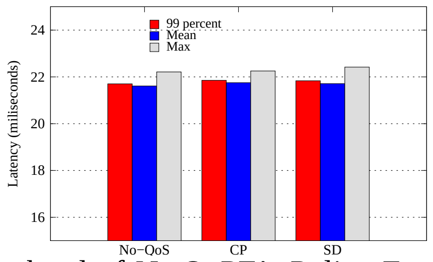
1) Evaluating the Overhead of NetQoPE for Normal Operations: **Rationale.** NetQoPE provides network QoS to applications via the four-stage architecture shown in Figure 1. This experiment evaluates the runtime performance overhead overhead of using NetQoPE to enforce network QoS.

Methodology. DRE system developers can use NetQoPE at design time to specify network QoS requirements on the application flows, as described in Section III-A. 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 III-C). 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. 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 8 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 8 are all similar, which shows that NetCON is efficient and adds negligible overhead to applications. If another variant



8: Overhead of NetQoPE's Policy Framework

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

2) 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 7 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 performed 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 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 ms) 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.

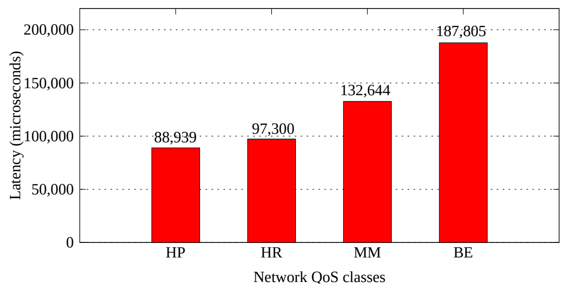
To evaluate application performance in the presence of background network loads, several other applications were run in both experiments, as described in Table III (in this table TS stands for "temperature sensor controller," MS stands for "monitor controller", FS stands for "fire sensor controller," and CS stands for "camera controller"). NetRAF

Traffi c Type	Background Traffi c 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

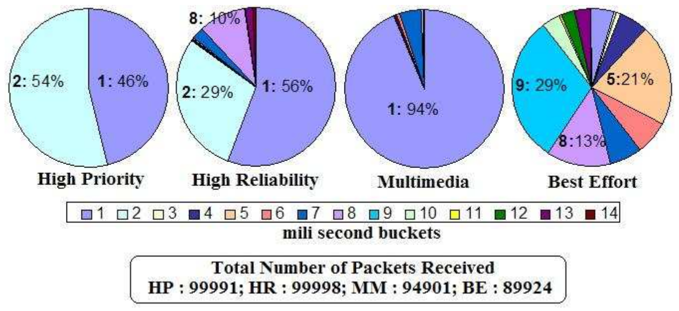
III: Application Background Traffic

allocated the network resources for each flow and determined which DSCP values to use. After deploying the applications, NetCON configured the containers to use the appropriate network priority models to add DSCP values to IP packets when applications invoked remote operations.

Analysis of results. Figure 9a shows the results of experiments when the deployed applications were configured with different network QoS classes and sent TCP traffic. This figure shows that irrespective of the heavy background traffic, the average latency experienced by the fire sensor controller component using the HP network QoS class is lower than the



(a) Average Latency under Different Network QoS Classes



(b) Jitter Distribution under Different Network QoS Classes

9: Performance of NetQoPE

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 9b 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 9b 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 queueing 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.

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 capabiltiies of network QoS mechanisms with minimal effort, as described in Section IV-C.

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.

V. 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 [10], [11], [12], [23] focused on providing middleware APIs to shield applications from directly interacting with complex network QoS mechanism APIs. Middleware frameworks transparently converted the specified application QoS requirements into lower-level network QoS mechanism APIs and provided network QoS assurances. These approaches, however, modified applications to dictate QoS behavior for the various flows. NetQoPE differs from these approaches by providing application-transparent and automated solutions to leverage network QoS mechanisms, thereby significantly reducing manual design and development effort to obtain network QoS.

QoS management in middleware. Prior research has focused on adding various types of QoS capabilities to middleware. For example, [25] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [26] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [13] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. Synergy [27] describes a distributed stream processing middleware that provides QoS to data streams in real time by efficient reuse of data streams and processing components. These approaches are restricted to CPU QoS assurances or application-level adaptations to resource-constrained scenarios. NetQoPE differs by providing network QoS assurances in a application-agnostic fashion.

Deployment-time resource allocation. Prior work has focused on deploying applications at appropriate nodes so that their QoS requirements can be met. For example, prior work [28], [29] has studied and analyzed application communication and access patterns to determine collocated placements of heavily communicating components. Other research [8], [9] has focused on intelligent component placement algorithms that maps components to nodes while satisfying their CPU requirements. NetQoPE differs from these approaches by leveraging network QoS mechanisms to allocate network resources at pre-deployment-time and enforcing network QoS at runtime.

Model-based design tools. Prior work has been done on model-based design tools. PICML [14] enables DRE system developers to define component interfaces, their implementations, and assemblies, facilitating deployment of LwCCM-based applications. VEST [30] and AIRES [15] analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provides automated allocation of components to processors. SysWeaver [31] 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.

VI. 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 III-A.
- 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 III-B.

• 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 III-C.

• 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 [32] 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 [33] with NetQoPE to provide predictable network performance for applications in open DRE systems.

All of NetQoPE's model-driven middleware platforms and tools—except the Bandwidth Broker—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`.

REFERENCES

[1] J. Balasubramanian, S. Tambe, B. Dasarathy, S. Gadgil, F. Porter, A. Gokhale, and D. C. Schmidt, "NetQoPE: A Model-Driven Network QoS Provisioning Engine for Enterprise Distributed Real-time and Embedded Systems," in *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium*, St. Louis, MO, USA, Apr. 2008.

[2] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," *CrossTalk - The Journal of Defense Software Engineering*, Nov. 2001.

[3] A. Nechypurenko, D. C. Schmidt, T. Lu, G. Deng, A. Gokhale, and E. Turkay, "Concern-based Composition and Reuse of Distributed Systems," in *Proceedings of the 8th International Conference on Software Reuse*. Madrid, Spain: ACM/IEEE, July 2004.

[4] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*. Washington, DC: IEEE Computer Society, May 2003.

[5] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, and G. Duzan, "Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, Oct. 2004.

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

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

[8] D. de Niz and R. Rajkumar, "Partitioning Bin-Packing Algorithms for Distributed Real-time Systems," *International Journal of Embedded Systems*, vol. 2, no. 3, pp. 196–208, 2006.

[9] S. Gopalakrishnan and M. Caccamo, "Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems," in *RTAS '06*, San Jose, CA, USA, 2006, pp. 199–207.

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

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

[12] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware," in *Proc. of Middleware'03*. Rio de Janeiro, Brazil: IFIP/ACM/USENIX, June 2003.

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

[14] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," *Journal of Computer Systems Science*, vol. 73, no. 2, pp. 171–185, 2007.

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

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

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

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

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

[20] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems," in *RTAS '05: Proceedings of the 11th*

IEEE Real Time on Embedded Technology and Applications Symposium, Los Alamitos, CA, USA, 2005, pp. 190–199.

[21] B. Dasarathy, S. Gadgil, R. Vaidhyanathan, K. Parmeswaran, B. Coan, M. Conarty, and V. Bhanot, "Network QoS Assurance in a Multi-Layer Adaptive Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework," in *RTAS 2005*. San Francisco, CA: IEEE, Mar. 2005.

[22] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

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

[24] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky, "Building Adaptive Distributed Applications with Middleware and Aspects," in *Proc. of AOSD '04*, New York, NY, USA, 2004, pp. 66–73.

[25] M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner, "Extending a J2EE Server with Dynamic and Flexible Resource Management," in *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, 2004.

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

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

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

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

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

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

[32] X. Wang, D. Jia, C. Lu, and X. Koutsoukos, "DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 7, pp. 996–1009, 2007.

[33] P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano, "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*, vol. 80, no. 7, pp. 984–996, July 2007.