A Model-driven QoS Provisioning Engine for Cyber Physical Systems

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

Abstract

Developing cyber physical systems is hard since it requires a coordinated, physics-aware allocation of CPU and network resources to satisfy their end-to-end quality-of-service (QoS) requirements. This paper provides two contributions to address these challenges. First, we present model-driven middleware called NetQoPE that shields application developers from the complexities of programming the lower-level CPU and network QoS mechanisms by simplifying (1) the specification of per-application CPU and per-flow network QoS requirements subject to the physical constraints and dynamics, (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 CPS applications. 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.

I. Introduction

Emerging trends and limitations. Cyber physical systems (CPS), such as smart buildings, high confidence medical devices and systems, and traffic control and safety systems consist of applications that participate in multiple end-to-end application flows, operate in resource-constrained environments, and have varying quality-of-service (QoS) requirements driven by the dynamics of the physical environment in which they operate. For example, smart buildings can host different types of applications with diverse (1) CPU QoS requirements (*e.g.*, personal desktop applications versus fire sensor data analyzers), and (2) network QoS requirements (*e.g.*, transport of e-mails versus transport of security-related information). In such systems, there is a need to allocate CPU and network resources to contending applications subject to the constraints on resources imposed by the physical phenomena (*e.g.*, a fire may partition a set of resources requiring rerouting of network flows).

The QoS provisioning problem is 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 [1], [2]. Overprovisioning is often not a viable option in cost- and resource-constrained environments where CPS applications deployed, *e.g.* in emerging markets that cannot afford the expense of overprovisioning. CPS application developers must therefore seek effective resource management mechanisms that can efficiently provision CPU and network resources, and address the following two limitations in current research:

Limitation 1: Need for physics-aware integrated allocation of multiple resources. Prior work has focused predominantly on allocating and scheduling CPU [3], [4] or network resources [5], [6] in isolation. While single resource QoS mechanisms have been studied extensively, little work has focused on coordinated mechanisms that allocate multiple resources, particularly for CPS applications where the coordinated resource management must be aware of the physical dynamics. In the absence of such mechanisms, CPS applications systems may not meet their QoS goals. For example, an application CPU allocation algorithm [7], [3], could dictate multiple placement choices for application(s), but not all placement choices may provide the network *and* CPU QoS because physical limitations may not permit certain allocations (*e.g.*, the placement of a fire sensor impacts its wireless network connectivity to nearby access points). Coordinated mechanisms are therefore needed to allocate CPU and network resources in an integrated manner.

Limitation 2: Need for a non-invasive application-level resource management framework. Even if an integrated, physics-aware multi-resource management framework existed for CPS applications, developers would still incur accidental complexities in using the low-level APIs of the framework. Moreover, application source code changes may be needed 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, or IP packet identifiers.

Middleware frameworks that perform CPU [8], [9], [10], [11], [12] or network [13], [2], [14], [15] QoS provisioning often shield application developers from these accidental complexities.

Despite these benefits, CPS applications can still be hard to evolve and extend when the APIs change and middleware evolve. Addressing these limitations requires higher-level integrated CPU and network QoS provisioning technologies that decouple application source code from the variabilities (*e.g.*, different source and destination node deployments, different QoS requirement specifications) associated with their QoS requirements. 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 → Model-driven deployment and configuration middleware for CPS applications. To simplify the development of CPS applications, we developed a multistage, model-driven deployment and configuration framework called *Network QoS Provisioning Engine* (NetQoPE) that integrates CPU and network QoS provisioning via declarative domain-specific modeling languages (DSML) [16]. NetQoPE leverages the strengths of middleware while simultaneously shielding developers from specific middleware APIs. This design allows system engineers and software developers to perform *reusable* deployment-time analysis (such as schedulability analysis [17]) 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 the QoS requirements of CPS applications will be satisfied.

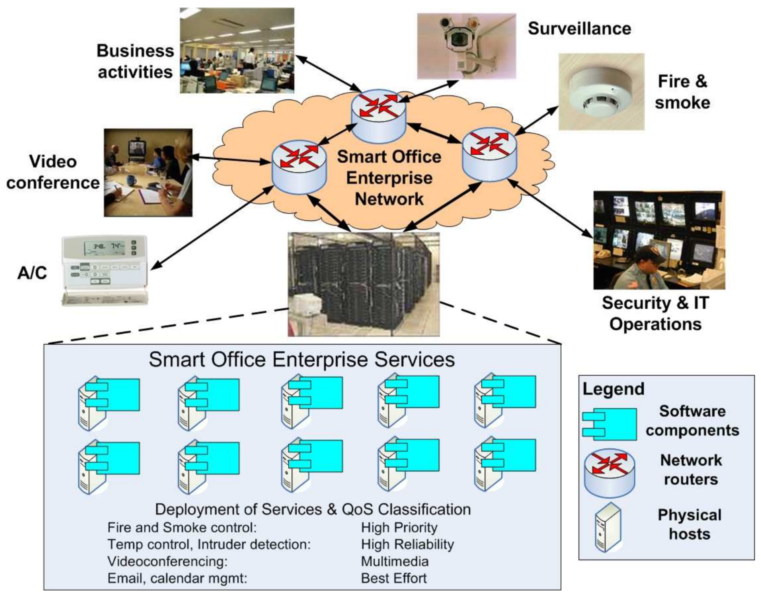
Paper organization. The remainder of the paper is organized as follows: Section II describes a case study that motivates common requirements associated with provisioning QoS for CPS applications; Section III explains how NetQoPE addresses those requirements via its multistage model-driven middleware framework; Section IV empirically evaluates the capabilities provided by NetQoPE in the context of a representative CPS application case study; 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

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

A. Smart Office Environment Case Study

Smart offices belong to a domain of systems called *Smart Buildings* [18] and showcase state-of-the-art computing and communication infrastructure in its offices and meeting rooms, as shown in Figure 1. Below we describe the cyber physical traits of the smart office environment, focusing on the development and deployment challenges CPS application developers face when ensuring the integration between the cyber and physical aspects of the system.



Fire and Smoke control:	High Priority
Temp control, Intruder detection:	High Reliability
Videoconferencing:	Multimedia
Email, calendar mgmt:	Best Effort

Fig. 1. **Network Configuration in a Smart Office Environment**

• *Fire and smoke management.* Detectors are placed in different rooms to send periodic sensory information to a fire and smoke management service. While designing and deploying this capability, developers must ensure the delivery of sensory data to the management service—and the outgoing traffic from this service—is *high priority*, *i.e.*, it should always obtain the desired CPU and network resources, even though the emergency mode operation (*e.g.*, in the event of a fire) of this service is infrequent. Moreover, sensory and actuation traffic must be reliable. The service should also adapt its policies of routing information to other resources when the current set of resources become unavailable, *e.g.*, due to fire or other adverse event.

• *Security surveillance.* This service uses a feed from cameras and audio sensors in different rooms and performs appropriate audio and video processing to sense physical movements and other intrusions. To notify the security control room, developers must ensure that the input feed from these sensors obtain high bandwidth for their multimedia traffic, while the outgoing alert notifications and activation of door controls are provided high priority. The image processing task must also be allocated its required CPU resources to perform intrusion detection.

• *Air conditioning and lighting control.* The air conditioning and lighting control service maintains appropriate ambient temperatures and lighting, respectively, in different parts of a building, including business offices, conference rooms and server rooms. It also turns off lights when rooms are not occupied to save energy. This service receives sensory data from thermostats and motion sensors, and controls the air conditioning vents and light switches. This service must be assured reliable transmission of information, though it does not necessarily require high priority.

- *Multimedia video and teleconferencing.* Offices often provide several multimedia-enabled conference rooms to conduct meetings simultaneously. These multimedia conferences require high bandwidth provisioning. 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 may also need to actuate a public address system informing people of a meeting. Since resources are finite, developers must make tradeoffs and assign this category of public address announcements to the best effort class of traffic, though that announcements about evacuations must be treated with high priority.
- *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.

B. Challenges in Provisioning and Managing QoS in the Smart Office

We now describe the challenges encountered when implementing the QoS provisioning and managing steps described above in the CPS applications that comprise our case study:

- **Challenge 1: Physics-aware QoS requirements specification.** Manually modifying 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 physical context in which they are deployed. For example, in our smart office case study, fire sensors have different importance levels (*e.g.*, fire sensors deployed in the parking lot have lower importance than those in the server room). The sensor to monitor flows thus have different network QoS requirements, even though the software controllers managing the fire sensor and the monitor are reusable units of functionality. It may be 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.

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. Section III-A describes how NetQoPE provides a solution to this challenge by providing a domain-specific modeling language (DSML) to support design-time application non-invasive specification of per-application network and CPU QoS requirements.

- **Challenge 2: Application resource allocation.** Manual modifications to source code to reserve resources tightly couple application components with a network QoS mechanism API (*e.g.*, Telcordia's Bandwidth Broker [19]). This coupling complicates deploying the same application component with resources reserved using a different network QoS mechanism API (*e.g.*, GARA Bandwidth Broker [20]). Similarily, source code modifications are also required when the same application is deployed with different network QoS requirements (*e.g.*, requesting more bandwidth on its application flows). Allocating network resources may also depend on their IP addresses, which may be feasible only when CPU allocations are done, which may not be known at design-time.

Ideally, network resources should be allocated without modifying application source code and should handle complexities associated with specifying application source and destination nodes, which could vary depending on the deployment context. Section III-B describes how NetQoPE provides a solution to this challenge by providing a resource allocator framework that supports resource reservation for each application and all its application flows in a non-invasive and transparent manner.

- **Challenge 3: Application QoS configuration.** Application developers have historically written code that instructs the middleware to provide the appropriate runtime services, *e.g.*, DSCP markings in IP packets [2]. Since applications can be deployed in different contexts, modifying application code to instruct the middleware to add network QoS settings is tedious, error-prone, and non-scalable.

Application-transparent mechanisms are therefore needed to configure the middleware to add these network QoS settings depending on the application deployment context. Section III-C describes how NetQoPE provides a solution to this challenge by providing a network QoS configurator that provides deployment-time configuration of component middleware containers to automatically add flow-specific identifiers to support router layer QoS differentiations.

III. NetQoPE's Multistage Network QoS Provisioning Architecture

This section describes how NetQoPE addresses the challenges from Section II-B associated with allocating and providing network and CPU QoS in tandem to CPS applications. NetQoPE deploys and configures component middleware-based CPS applications and enforces their network and CPU QoS requirements using the multistage (*i.e.*, design-, pre-deployment-, deployment-, and run-time) architecture shown in Figure 2. NetQoPE's multistage architecture consists of the following elements in the workflow, which automates the task of QoS provisioning for CPS applications.

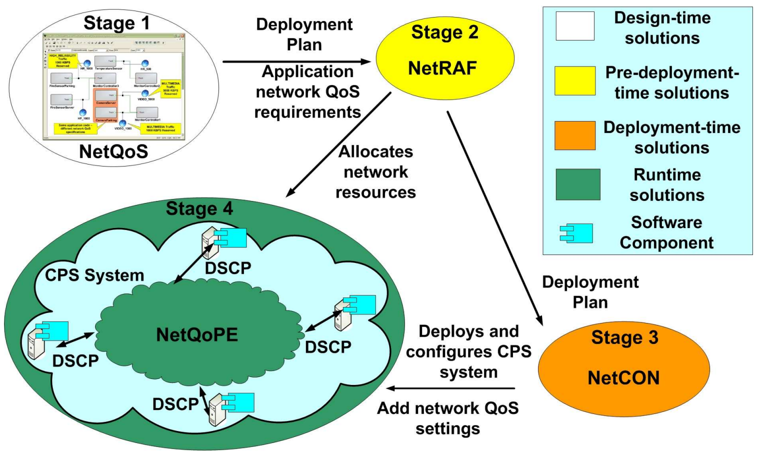


Fig. 2. **NetQoPE's Multistage Architecture**

• 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. NetQoPE uses NetQoS to resolve *Challenge 1* of Section II-B, as described in Section III-A.

• 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. NetQoPE uses NetRAF to resolve *Challenge 2* of Section II-B, as described in Section III-B.

• 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. NetQoPE uses NetCON to resolve *Challenge 3* of Section II-B, as described in Section III-C.

NetQoPE implementation technologies. We developed a prototype of the smart office environment case study using the Lightweight CORBA Component Model [21]. We also used a Bandwidth Broker [19] to allocate per-application-flow network resources using DiffServ network QoS mechanisms. In addition, we used the Generic Modeling Environment (GME) [22] to create domain-specific modeling languages (DSMLs) [23] that simplify the development and deployment of smart office environment applications.

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 CPS applications. Although the case study in this paper leverages LwCCM and DiffServ, NetQoPE can be used with other network QoS mechanisms (*e.g.*, IntServ) and component middleware technologies (*e.g.*, J2EE).

A. NetQoS: Supporting Physics-aware CPU and Network QoS Requirements Specification

To resolve *Challenge 1* of Section II-B, NetQoPE enables CPS application developers to specify their resource requirements at application deployment-time using a DSML called the *Network QoS Specification Language* (NetQoS). NetQoS is built using the Generic Modeling Environment (GME) [22] and works in concert with the *Platform Independent Component Modeling Language* (PICML) [23]. NetQoS provides applications with an application-independent, declarative (as opposed to application-intrusive [14], middleware-dependent [8], and OS-dependent [24]) mechanism to specify multi-resource requirements simultaneously that can account for the physical context in which the system is deployed.

NetQoS also allows specifying resource requirements as applications are deployed and configured in the target environment. Its declarative mechanisms (1) decouple this responsibility from application source code, and (2) specialize the process of specifying resource requirements for the particular deployment and usecase. Below we describe the steps in using NetQoS' capabilities.

1. Declarative specification of resource requirements. CPS applications 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.[1] NetQoS's network QoS classes correspond to the DiffServ levels supported by an underlying network-level resource allocator, such as the Bandwidth Broker [19] we used in our case study.[2] 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). Figure 3 show how NetQoS was used to model the QoS requirements of our case study.

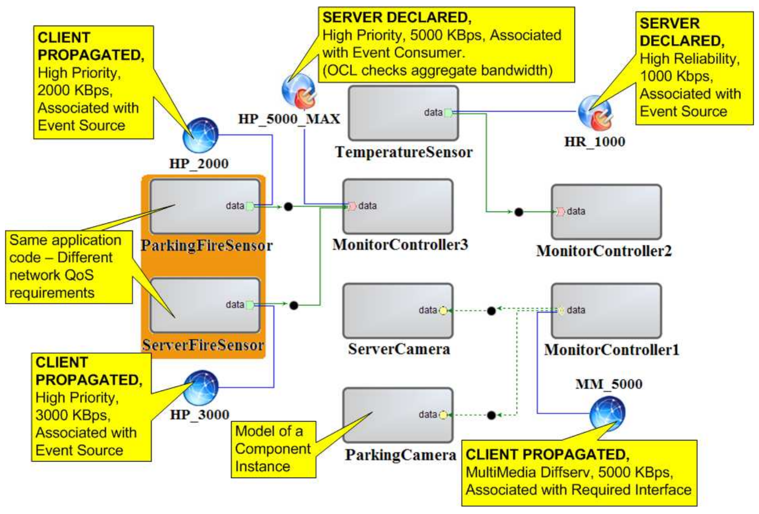

CLIENT PROPAGATED, High Priority, 2000 KBps, Associated with Event Source
SERVER DECLARED, High Priority, 5000 KBps, Associated with Event Consumer. (OCL checks aggregate bandwidth)
SERVER DECLARED, High Reliability, 1000 Kbps, Associated with Event Source
HP_5000_MAX
TemperatureSensor
HR_1000
HP_2000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

Fig. 3. **Applying NetQoS Capabilities to the Case Study**

2. Flexible enforcement of network QoS. In certain application flows in the smart office case study, (*e.g.*, a monitor requesting location coordinates from a fire sensor) clients control the network priorities at which requests/replies are sent. In other application flows (*e.g.*, a temperature sensor sending temperature sensory information to monitors), the 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.

To support both models of communication (*i.e.*, whether clients or servers control network QoS for a flow), NetQoS supports annotating each bi-directional flow using either: (1) the CLIENT_PROPAGATED network priority model, which allows clients to request real-time network QoS assurance even in the presence of network congestion, or (2) the SERVER_DECLARED network priority model, which allows servers to dictate the service that they wish to provide to

[1]Middleware such as the Lightweight CORBA Component Model allow components to communicate using *ports* that provide application-level communication endpoints. NetQoS provides capabilities to annotate communication ports with the network QoS requirement specification capabilities.

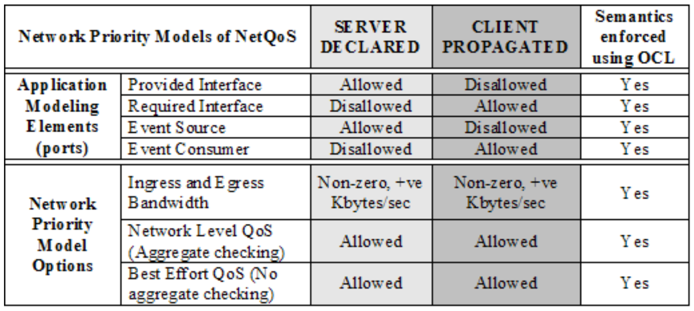
[2]NetQoS's DSML capabilities can also be extended to provide requirements specification conforming to other network QoS mechanisms, such as IntServ.

the clients to prevent clients from wasting network resources on non-critical communication.

NetQoS initiates the allocation of CPU and network resources on behalf of applications by triggering the next stage of the workflow. Section III-C describes how NetQoPE uses component middleware frameworks at runtime to *realize* the design intent captured by NetQoS and *enforce* network QoS for applications.

3. 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 lifecycle stages (such as deployment and runtime) when they are 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 4 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

Fig. 4. **Network QoS Models Supported by NetQoS**

4. Preparation for allocating CPU and network resources. After a model has been created and checked for type violations using NetQoS's built-in constraints, network resources must be allocated using a network QoS mechanism [19], [20]. As described in Section II-B, 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 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 (see Section III-B), 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 in which the components participate.

Application to the case study. Figure 3 shows a NetQoS model that highlights many 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 drag-and-drop.

Our case study 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 3) can be reused across multiple connections, which increases the scalability of expressing requirements for a number of connections prevalent in large-scale CPS applications, such as our smart office environment case study. Section IV-B and Section IV-C empirically evaluate these capabilities.

B. NetRAF: Alleviating Complexities in Network Resource Allocation and Configuration

NetQoPE's *Network Resource Allocator Framework* (NetRAF) is a resource allocator engine that allocates network resources for CPS applications using DiffServ network QoS mechanisms, which resolves *Challenge 2* described in Section II-B.. NetRAF allocates network resources for application flows on behalf of the applications (recall how NetQoS invokes NetRAF on behalf of the applications as part of their workflow) 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 [19], [20]), NetRAF uses 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 underlying network QoS mechanism, such as DiffServ, IntServ, and RSVP.

NetRAF provides a clean separation of functionality between resource reservation (provided by NetRAF) and QoS enforcement (done by underlying network elements), as described in the following steps:

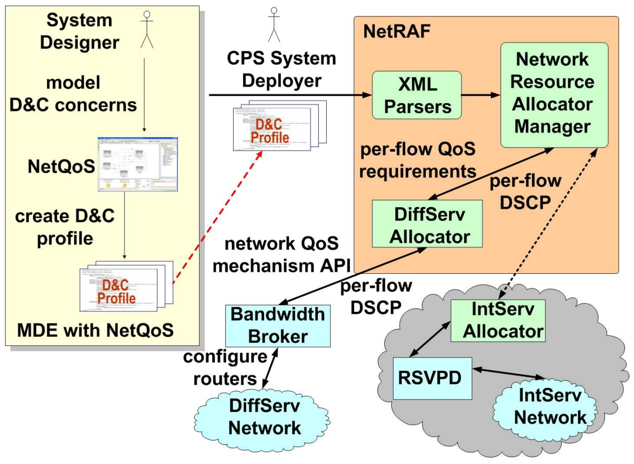


Fig. 5. **NetRAF's Network Resource Allocation Capabilities**

1. Network resource allocations. Figure 5 shows how 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 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 (see Section IV-B).

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 infeasible resource allocation errors. Although NetQoS checks the well-formedness of network requirement specifications at application level, it cannot identify every situation that may lead to scenarios with infeasible resource allocations, since these depend on the dynamics of the physical environment.

To handle such scenarios, 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 invokes the API of the QoS mechanism-specific allocator, providing it one flow at a time without actually reserving network resources, and (2) it 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 its *DiffServ Allocator* to allocate network resources, which in turn invokes the Bandwidth Broker's admission control capabilities [19] by feeding it one application flow at a time. 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. This paper assumes the underlying network QoS mechanism (*e.g.*, the Bandwidth Broker) is responsible for configuring the routers to provide the per-hop behavior [19].

C. NetCON: Alleviating Complexities in Network QoS Settings Configuration

NetQoPE's *Network QoS Configurator* (NetCON) resolves *Challenge 3* described in Section II-B by enabling the auto-configuration of component middleware containers, which provide a hosting environment for application component functionality. Through NetCON auto-configuration, containers can add DSCPs to IP packets when applications invoke remote operations. The current version of NetCON is developed for the LwCCM component middleware and is shown in Figure 6.

During deployment, NetCON parses the deployment plan (which now includes both the

CPU allocations and network DSCP tags for the connections) 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.

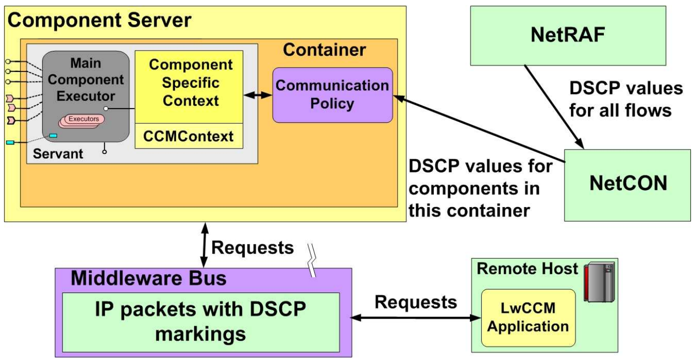


Fig. 6. **NetCON's Container Auto-configurations**

NetCON's container programming model can transparently add DSCPs and enforce the network priority models (see Figure 3). 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 allows 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 3, 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 uses CLIENT_PROPAGATED network priority model to communicate with the FireSensor components with different network QoS requirements.

After the first two stages of NetQoPE, 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 appropriate DSCP values are added at runtime 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. Section IV-B and Section IV-C empirically evaluate these capabilities.

IV. Empirical Evaluation of NetQoPE

This section empirically evaluates NetQoPE's capabilities 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 CPS applications, such as our case study in Section II.

A. Evaluation Scenario

Hardware and software testbed. Our empirical evaluation of NetQoPE was conducted on 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. 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. Figure 7 depicts these details.

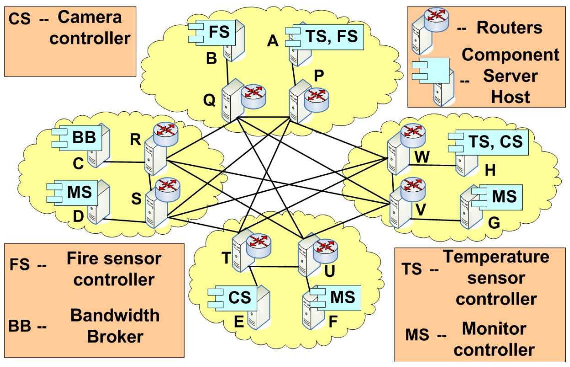


Fig. 7. **Experimental Setup**

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 [11]. Our evaluations used DiffServ QoS and the associated Bandwidth Broker [19] 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 this scenario six sensory and imagery software controllers sent their monitored information to three 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. 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 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 CPU and network behavior of the software controllers when different 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 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

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

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.

B. *Evaluating NetQoPE's Model-driven QoS Provisioning Capabilities*

Rationale. 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.

To evaluate the effort saved using NetQoPE, we developed a taxonomy of technologies that provide CPU and network QoS assurances to end-to-end CPS application flows. This taxonomy is used to compare NetQoPE's methodology of provisioning integrated network and CPU QoS for these flows with conventional approaches, including (1) object-oriented [15], [13], [2], (2) aspect-oriented [25], and (3) component middleware-based [14], [26] 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 [15], [13], (2) contract definition languages [2], (3) runtime aspects [25], or (4) specialized component middleware container interfaces [14] 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.

In such approaches, 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 III-A.

• **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 [3] 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 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 [2] or programming model [14] 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.

Based on this taxonomy, 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 CPS application structure, specify per-application CPU resource and per-flow network resource requirements, and allocate needed CPU and network resources.

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:[4]

• **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 CPS application structure of all 4 end-to-end application flows using NetQoS (1 modeling step), (2) annotate QoS specifications on each application and each end-to-end application flow (1 modeling step), (3) deploy all components (1 deployment step – this step also involved allocation

[4]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.

of both CPU and network resources for applications using NetRAF's two step allocation process described in Section III-B), 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).

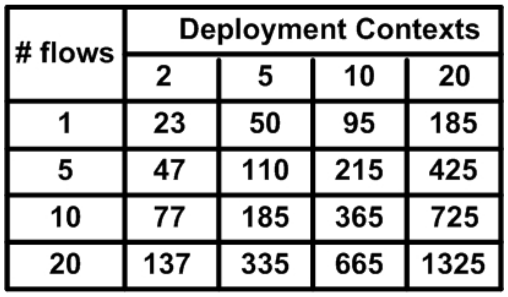
Approaches	# Steps in exp. variants		
	First	Second	Third
NetQoPE	4	3	5
Conventional	10	10	13

Fig. 8. **Effort Comparison Across Different Approaches**

Figure 8 summarizes the step-by-step analysis described above. These results show 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 [13] and/or network QoS mechanism APIs [5].

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 CPS 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 CPS applications such as our case study, however, the same communication patterns between components could occur in many deployment contexts.



# flows	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

Fig. 9. **Generated lines of XML code**

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 Figure 9, 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 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.

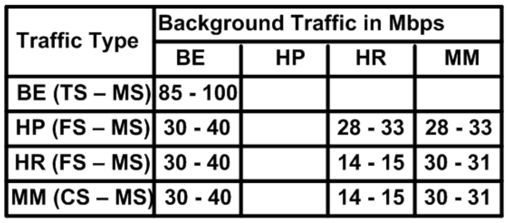
C. Evaluating NetQoPE's QoS Customization Capabilities

Rationale. This experiment empirically evaluates the benefits of the the flexibility and decoupling resulting from NetQoPE's multi stage architecture, and whether the CPS applications indeed obtain their required QoS.

Methodology. From Figure 7, the four flows that were described in Section IV-B were modeled with the same set of network and CPU QoS requirements using NetQoS. 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 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.



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

Fig. 10. **Background Traffic**

To evaluate application performance in the presence of background network loads, several other applications were run in both experiments, as described in Figure 10 (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 determined the DSCP values which were then enforced in each outgoing packet through the container auto-configuration effected by NetCON.

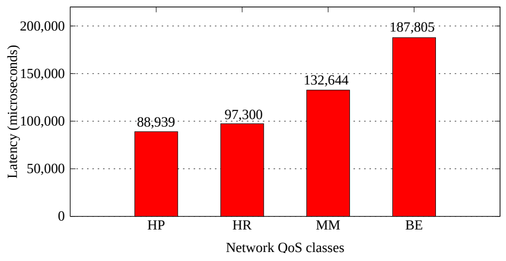


Fig. 11. **Average Latency under Different Network QoS Classes**

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. 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 CPS applications. 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.

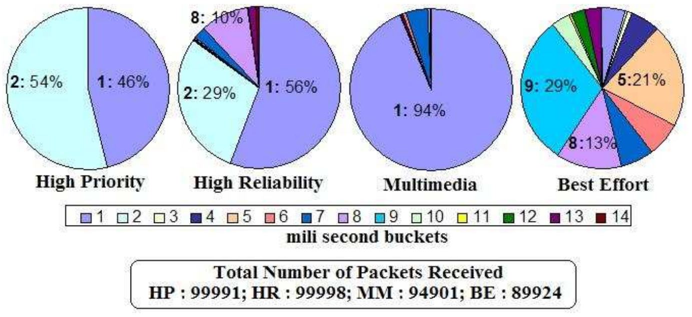


Fig. 12. **Jitter Distribution under Different Network QoS Classes**

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.

These results demonstrate that NetQoPE can provide significant flexibility and customizability, while ensuring that applications obtain their required QoS.

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 [13], [2], [15] 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, [27] describes J2EE container resource management mechanisms that provide CPU availability assurances to applications. Likewise, 2K [28] provides QoS to applications from varied domains using a component-based runtime middleware. In addition, [14] extends EJB containers to integrate QoS features by providing negotiation interfaces which the application developers need to implement to receive desired QoS support. 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 [21] 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.

Model-based design tools. Prior work has been done on model-based design tools. PICML [23] enables developers of CPS applications to define component interfaces, their implementations, and assemblies, facilitating deployment of LwCCM-based applications. VEST [29] and AIRES [17] analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provides automated allocation of components to processors. SysWeaver [30] 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 CPS 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 described the design and evaluation of NetQoPE, which is a model-driven middleware framework that manages CPU and network QoS for CPS applications. The lessons we learned developing NetQoPE and applying it to a representative CPS application 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 before deployment may be too limiting for certain types of CPS applications. In particular, because of the physical nature of the systems, faults might occur at runtime, and applications might not consume all their resource allotment at runtime. Similarily, applications in open systems might require dynamic provisioning of resources based on application demand. Our future work is therefore extending NetQoPE to overprovision resources for applications on the assumption that not all applications will use their allotment.

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 Bandwidth Broker is a product licensed by Telcordia.

References

[1] K. Nahrstedt, "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*. Washington, DC, USA: IEEE Computer Society, 1999, p. 35.

[2] R. E. Schantz, J. P. Loyall, C. Rodrigues, and D. C. Schmidt, “Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with auto-adaptive and reconfigurable systems,” *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1189–1208, 2006.

[3] 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.

[4] S. Gopalakrishnan and M. Caccamo, “Task partitioning with replication upon heterogeneous multiprocessor systems,” in *RTAS ’06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 199–207.

[5] L. Zhang and S. Berson and S. Herzog and S. Jamin, “Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification,” pp. 1–112, Sept. 1997.

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

[7] B. Urgaonkar, A. Rosenberg, and P. Shenoy, “Application Placement on a Cluster of Servers,” *International Journal of Foundations of Computer Science*, vol. 18, no. 5, pp. 1023–1042, 2007.

[8] E. Eide, T. Stack, J. Regehr, and J. Lepreau, “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*. Washington, DC, USA: IEEE Computer Society, 2004, p. 286.

[9] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li, “QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments,” *IEEE Communications Magazine*, vol. 39, no. 11, pp. 140–148, Nov. 2001.

[10] B. Urgaonkar and P. Shenoy, “Sharc: Managing cpu and network bandwidth in shared clusters,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 1, pp. 2–17, 2004.

[11] I. Pyarali, D. Schmidt, and R. Cytron, “Techniques for enhancing real-time corba quality of service,” *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1070–1085, July 2003.

[12] A. S. Krishna, D. C. Schmidt, and R. Klefstad, “Enhancing real-time corba via real-time java features,” in *ICDCS ’04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS’04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 66–73.

[13] P. Wang, Y. Yemini, D. Florissi, and J. Zinky, “A distributed resource controller for qos applications,” in *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 143–156.

[14] M. A. de Miguel, “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*. Washington, DC, USA: IEEE Computer Society, 2002, p. 394.

[15] M. El-Gendy, A. Bose, S.-T. Park, and K. Shin, “Paving the first mile for qos-dependent applications and appliances,” in *IWQoS ’04: Proceedings of the 12th International Workshop on Quality of Service*. Washington, DC, USA: IEEE Computer Society, June 2004, pp. 245–254.

[16] M. Mernik, J. Heering, and A. M. Sloane, “When and How to Develop Domain-specific Languages,” *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[17] 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: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Toronto, Canada: IEEE Computer Society, 2003, pp. 78–85.

[18] D. Snoonian, “Smart Buildings,” *IEEE Spectrum*, vol. 40, no. 8, pp. 18–23, 2003.

[19] B. Dasarathy, S. Gadgil, R. Vaidyanathan, A. Neidhardt, B. Coan, K. Parmeswaran, A. McIntosh, and F. Porter, “Adaptive network qos in layer-3/layer-2 networks as a middleware service for mission-critical applications,” *J. Syst. Softw.*, vol. 80, no. 7, pp. 972–983, 2007.

[20] I. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler, “End-to-end Quality of Service for High-end Applications,” *Computer Communications*, vol. 27, no. 14, pp. 1375–1388, Sept. 2004.

[21] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, “Configuring Real-time Aspects in Component Middleware,” in *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, vol. 3291. Agia Napa, Cyprus: Springer-Verlag, Oct. 2004, pp. 1520–1537.

[22] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, “Model-Integrated Development of Embedded Software,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

[23] 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.

[24] A. Mehra, D. C. Verma, and R. Tewari, “Policy-based diffserv on internet servers: The AIX approach (on the wire),” *IEEE Internet Computing*, vol. 4, no. 5, pp. 75–80, 2000. [Online]. Available: citeseer.ist.psu.edu/mehra00policybased.html

[25] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky, “Building Adaptive Distributed Applications with Middleware and Aspects,” in *AOSD ’04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2004, pp. 66–73.

[26] P. K. Sharma, J. P. Loyall, G. T. Heineman, R. E. Schantz, R. Shapiro, and G. Duzan, “Component-based dynamic qos adaptations in distributed real-time and embedded systems,” in *CoopIS/DOA/ODBASE (2)*. Agia Napa, Cyprus: Springer, 2004, pp. 1208–1224.

[27] M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner, “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*. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 439–458.

[28] 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 *Middleware ’01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. London, UK: Springer-Verlag, 2001, pp. 373–394.

[29] 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 *RTAS ’03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Toronto, Canada: IEEE Computer Society, 2003, pp. 58–69.

[30] D. de Niz, G. Bhatia, and R. Rajkumar, “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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–242.