Standards-based quality of service (QoS)-enabled component middleware is increasingly being used as a platform for developing distributed, real-time and embedded (DRE) systems that are network-centric, operate in environments where network resources are scarce, and applications need to have their network QoS requirements ensured even in the presence of network congestion. Traditional approaches to building network-based, QoS-sensitive applications, such as internet telephony and streaming multimedia applications, have relied on leveraging the low level APIs provided by QoS features of network elements and operating system architectures. Although QoS-enabled component middleware offers many desirable features, until recently it has lacked the ability to simplify and coordinate application-level services to leverage the advances in networks and endsystems end-to-end.

This paper presents two contributions to research on network QoS provisioning within QoS-enabled component middleware for DRE systems. First, we describe a declarative QoS provisioning solution called XYZ comprising modeling capabilities to specify application QoS requirements, resource management capabilities that seamlessly integrate with network elements, such as bandwidth brokers and routers, and a middleware-based policy framework to provision network QoS mechanisms for DRE applications. Second, we demonstrate and evaluate the effectiveness of XYZ in the context of a representative DRE system – a subset of an emergency response system focusing on an unmanned air vehicle (UAV) application. Our empirical results show that the capabilities provided by XYZ yields a predictable and efficient system for QoS sensitive applications even in the face of changing workloads and resource availabilities in the underlying network subsystem.

1 Introduction

Distributed real-time and embedded (DRE) systems form the core of many mission critical domains, such as shipboard computing environments [22], avionics mission computing [25], and intelligence, surveillance and reconnaissance missions [24]. Such systems must collaborate with multiple sensors, provide on-demand browsing and actuation capabilities for human operators, and provide wide range of capabilities including predictable performance, secure communications and fault tolerance.
**QoS-enabled component middleware**, such as CIAO [28], Qedo [17], and PHiSm [18], are increasingly being used to develop and deploy next-generation DRE systems. QoS-enabled component middleware leverage conventional component middleware (e.g., J2EE, .NET, CCM) capabilities that include: (1) standardized interfaces for application component interaction, (2) standards-based mechanisms with clear separation of concerns for the different lifecycle stages of applications including developing, installing, initializing, configuring and deploying application components, and (3) declarative as opposed to programmatic approaches to the lifecycle management activities, such as assembly, configuration and deployment.

Additionally, QoS-enabled component middleware platforms overcome shortcomings of conventional component middleware by explicitly separating QoS provisioning aspects of applications from their functionality [27], thereby yielding DRE systems that are less brittle and costly to develop, maintain, and extend [28]. For example, QoS-enabled component middleware [8, 9] provide mechanisms to enable applications to leverage CPU and memory resources of an end system thereby providing different processor priorities to applications.

An important trait of DRE systems of particular interest to us is the existence of multiple workflows through the interacting entities of the DRE system that need differentiated network-level QoS. The design and implementation of these systems must therefore be able to leverage advances in the network subsystem QoS and resource management mechanisms, such as integrated services (IntServ) [16], differentiated services (DiffServ) [2] and multi-protocol label switching (MPLS) [19] architectures.

Advances in QoS-enabled component middleware to date are, however, limited to only managing CPU and memory resources in the operating system subsystem. In particular, there are limited or no capabilities in current QoS-enabled component middleware to (1) isolate application communications flows from one another when they share the network subsystem resources and require differentiated QoS, and (2) control and configure many network subsystem resources using platform-specific QoS capabilities and mechanisms so that proper application QoS can be assured. These limitations force DRE system developers to seek alternate solutions including the use of out-of-band, low level network APIs to enable network QoS thereby adversely impacting the system flexibility, reusability and maintainability.

Overcoming these limitations requires QoS-enabled component middleware to provide additional capabilities for: (1) specifying, gathering and analyzing network QoS and bandwidth utilization requirements of varied communication flows applications in a DRE system, (2) leveraging the advances in the networking subsystem, such as sophisticated bandwidth brokering capabilities, to reserve network resources for communications within a DRE system based on their importance,
type, and other factors, (3) configuring and monitoring network devices and elements to provision mechanisms including queuing and policing to support network QoS, (4) providing traffic classification and marking e.g., DiffServ marking when applications communicate across wired and wireless networks, (5) providing resource reservation and admission control based on application identity, type and QoS needs, and (6) configuring the applications and their communications with platform-specific QoS settings (such as DiffServ markings for layer 3 network QoS settings) so that applications can seamlessly work with the lower-level network subsystem resources and elements to ensure application and their communication QoS.

Our earlier work on a QoS provisioning framework has focused on developing the following capabilities: (1) developing a model-driven engineering (MDE) [20] tool suite called CoSMIC [6], which alleviates many accidental complexities associated with developing, deploying and configuring QoS-enabled component-based DRE systems, (2) developing a deployment and configuration engine (DAnCE) [5] that provides standards-based deployment and configuration mechanisms to deploy systems for component middleware platforms, such as our CIAO [29] lightweight CORBA Component Model (LwCCM) [13] implementation, (3) developing a dynamic resource allocation and control engine called RACE [23] to allocate and control resources for DRE applications, and (4) developing reflective middleware [30] techniques within CIAO to support runtime adaptive CPU and memory QoS management [31].

This paper describes novel enhancements we made to our QoS framework to address the limitations of network QoS control in standardized component middleware platforms. Our contributions include: (1) extending the CoSMIC tool suite to enable the modeling of network QoS requirements of DRE application workflows while seamlessly integrating it with existing capabilities for specifying the interfaces, communication and assembly details of the DRE system, (2) extending RACE to integrate sophisticated bandwidth brokers, which can analyze the network QoS requirements of DRE application workflows and provision individual network elements for traffic policing and marking, and (3) extending the CIAO and DAnCE frameworks to allow configuring the application components and their communications with platform-specific QoS settings determined by the bandwidth broker so that the applications and the network elements configured by the bandwidth broker can work collaboratively to ensure network QoS for DRE applications.

The remainder of the paper is organized as follows: Section 2 describes the challenges in provisioning network QoS using a representative DRE system case study as a guiding example; Section 3 describes the novel solutions we have developed in the context of our existing QoS provisioning framework; Section 4 provides experimental validation of our approach;
Section 5 compares our work with related research; and Section 6 provides concluding remarks, lessons learned and future work.

2 Challenges in Declarative Provisioning of Network QoS

This section describes the challenges faced in raising the level of abstraction of network QoS provisioning to the level of QoS-enabled component middleware, where the provisioning decisions can be made declaratively. To best motivate these challenges we first present a DRE system case study outlining the network-centric QoS requirements of a representative DRE application. We then illustrate the challenges that exist in the realm of QoS-enabled component middleware to enable DRE application developers to declaratively provision network QoS for the DRE applications.

2.1 DRE System Case Study

Our representative DRE system case study is drawn from an emergency response scenario comprising aerial reconnaissance by unmanned aerial vehicles (UAVs) coordinating activities with ground controllers. The UAVs operate under multiple modes including aerial imaging, survivor tracking, and damage assessment. Each of these modes is associated with a different set of QoS requirements. For example, a key QoS criteria involves the latency requirements in sending images from the flying UAVs to command and control centers under varying bandwidth availability. Similar QoS requirements manifest themselves in the traffic management, rescue missions, and fire fighting operations.

2.1.1 The UAV Software Architecture

In conjunction with colleagues at BBN Technologies and Washington University, we have developed a prototype of the UAV portion of the emergency response system [24] described above using the CCM and Real-time CORBA capabilities provided by CIAO [29]. This section describes (1) the structure/functionality of our UAV case study and (2) the key requirements that XYZ framework had to address. Naturally, XYZ’s capabilities can be applied to many DRE systems – we focus on the UAV case study in this paper to make our design discussions and performance experiments concrete.

Figure 1 illustrates the architecture for the application suite that motivates the specific directions of our work described in this paper. In the UAV example, each UAV is associated with providing a stream of images and notifications. Each UAV is associated with a ground controller, which receives data from the UAV and transmits them to a command and control center,
where the images and notifications are analyzed and appropriate actions are taken in response to the emergency and criticality of the situation depicted by the image or the notification. In this paper, we focus on the communications between the ground controllers, which serve as the proxies to the UAV they control, and the command and control center, and ensure network QoS across those communications.

During an emergency response coordination, multiple UAVs work in different modes to capture images and observe events, and send them to the command and control center, which needs to take effective actions as a response to the kind of emergency dictated by the mode in which the UAV operates. For example, after a sudden and severe flooding, the command and control center needs to take faster actions in response to a UAV sending images showing survivors, rather than in response to a UAV sending images about property damage. So the workflow of interest to us, is the communication between the ground controllers, corresponding to a UAV and displays in the command and control centers. As shown in Figure 1, the ground controllers and the display components in the command and control center are software controller components implemented using CIAO, and they are deployed under different hardware nodes which are connected across a metropolitan-area network.

**Figure 1. Emergency Response System components**
2.1.2 Network QoS Requirements of UAV Scenario

The UAV application described in Section 2.1.1 presents a wide variety of characteristics that are representative of many DRE applications that operate under constrained resources, varying conditions and varying environments, including:

- **Varying data formats**, including MPEG and PPM, with different data sizes and compression characteristics, with which the ground controller components could send images or video or event notifications to the command and control center components. Since the UAVs can operate in different modes, the data they are scanning and sending could be very different depending on the mode they are operating. For example, when the UAVs are operating in the survivor tracking mode, the details about the survivors need to be sent in a quick and efficient fashion. This in turn means that the data size would be very less, but the data needs to travel very fast and without any delay. However, when the UAVs are operating in the aerial imaging mode, the speed with which the data travels is not important, but the reliability is important since details of the terrain which the UAVs are monitoring should not be missed at any cost.

- **Varying importances**, where, depending on the mission importance of the emergency response system, importance is given to images being tracked and sent by the UAVs to the control room of the command center. As explained in the example above, when the UAVs are operating after a sudden and severe flooding, some of the UAVs would be monitoring the survivors while some of the UAVs would be monitoring the damage made by the flooding. Simultaneously, all the images are being sent to the ground controllers which then subsequently send the images to the command and control center. However, the details about the survivors need to be sent faster and predictably to the command and control center so that effective actions can be taken to rescue the survivors. But the same network is being used to send both the information about the damages as well as about the survivors due to the scarcity of network resources available. This means that appropriate network QoS must be provided in the network transport between the ground controllers, and the command and control center so that importance is given to the data being sent even in the presence of network congestion and effective actions can be taken at the appropriate time instants.

2.2 Challenges Provisioning Network QoS in Component Middleware

To support the above described characteristics and requirements, and to provide reusable and extensible solutions, the UAV application suite requires a QoS provisioning middleware framework that addresses the following challenges:
Challenge 1: Specifying network QoS requirements for component interactions: Component-based applications communicate using ports that provide request-response or publisher-subscriber semantics. Irrespective of the communication semantics used, the interacting components require specific levels of QoS to be honored for the inter port communications. For example, for the correct behavior of the communicating application components, it may be necessary for the communication links between the ports to honor minimum bandwidth requirements. Since there may be many competing flows in a component-based system, it may be necessary to provide maximum bandwidth requirements so that the right capacity planning decisions can be made. Additional considerations should be given to the importance for different flows by prioritizing one flow over another.

In the UAV example, the streams involving the UAVs (i.e., their ground controllers) and the command center are organized according to their importance. During the system’s operational life time, these streams must be assured their QoS requirements even in the presence of excess workloads and unforeseen operating conditions. For example, in figure 1, the ground controller A could be sending images to the display Y and the ground controller B could be sending event notifications to the display X, and the network could be configured in a way that both the traffic flows through the same devices router P and router R. In order to provide network QoS, the networking elements need to be configured appropriately, so that flow QoS can be ensured.

Specifying and gathering network utilization details and priorities for the component flows allows the QoS-enabled component middleware to configure the underlying network elements such as routers on behalf of the applications, and also allows the applications to be configured with the network QoS settings to work efficiently with the configured networking elements. Since the DRE systems of interest to us are large scale, it is infeasible for application developers to programatically specify the QoS needs of components and their interactions on a per flow basis. A desired feature is a declarative and scalable means of specifying these QoS requirements.

Section 3.2 describes a model-driven engineering (MDE) [20] approach we have developed that provides these declarative capabilities to capture QoS requirements. Our approach comprises extending our existing MDE capabilities that enable us to define the component workflows by providing the mechanisms to specify network QoS.

Challenge 2: Allocating resources to meet component QoS requirements: Satisfying the network QoS requirements of components involves, (1) identifying which hardware nodes the application components are to be deployed, (2) determining how much bandwidth is available in the network connections between the hardware nodes hosting the application compo-
ents, (3) committing network resources to be allocated for the connections so that resource requests (e.g., bandwidth) can be allocated accordingly, (4) configuring the networking elements like routers to honor the application QoS requirements when flows traverse through them, and (5) informing the applications with platform-specific QoS settings, so that the applications can be configured with them when they make remote calls, and the networking elements configured to work with those QoS settings can honor the QoS requirements. But current generation networking mechanisms require deployment target information including source and destination host, source and destination port addresses, for QoS configuration decisions to be made. In QoS-enabled component middleware, such deployment decisions are usually made using intelligent component placement algorithms [4, 11], based on details including component CPU resource profiles. Hence, network QoS provisioning mechanisms must work with such component placement algorithms inorder to configure the underlying network devices to provide network QoS for component middleware flows.

In the UAV example, there are multiple network streams that stream images and videos from the UAVs to the command center. Correspondingly, there are lot of network communications between the ground controllers and the command and control center. Since the ground controllers and the command and control center software components could be deployed in any of the available hardware nodes, the communications between those software components, can take many different network paths. For example, in figure 1, the UAVs could be sending an image stream and an event notification stream. The image stream ground controller could be deployed as ground controller A sending images to the display Y and traversing routers router P and router R. Alternatively, the image stream ground controller could be deployed as ground controller B sending images to the display Z and traversing routers router Q and router R.

Ensuring that the networking devices have the required resources to be allocated for a component communication, depends on identifying the source and destination hardware addresses and port addresses. Hence such provisioning decisions should be made in conjunction with intelligent component placement algorithms, so that network QoS can be ensured for component communications. Moreover, this also provides an opportunity to change the component placement decisions, if the network QoS cannot be provided if the components are deployed across a pair of hardware nodes.

Section 3.3 describes enhancements we made to the planning portions of our resource allocation and control engine (RACE) [23], which acts as a mediator between application components and low-level network elements, such as a bandwidth broker, to convey the application’s network QoS requirements to these elements, request resource reservations across the hardware nodes decided by the component placement algorithms, and configure networking elements and devices. The results
of these interactions are captured within the application’s deployment metadata, which describes how different application components will be deployed and configured within the computing infrastructure.

**Challenge 3: Configuring applications and their flows to meet network-level QoS requirements:** Once the network level elements, such as bandwidth brokers, reserve the desired network level resources, components are deployed on the hardware nodes of the infrastructure, and the routers are configured with popular layer-3 mechanisms like differentiated services (DiffServ) and layer-2 mechanisms like Class of Service (CoS), the applications need to be configured with appropriate QoS settings, to work efficiently with the networking elements and devices configured. But the philosophy of component-based systems is to allow the application developers to only focus on the business logic and not bother about complex details on how the middleware and underlying subsystems are configured to ensure the QoS requirements. In general, the applications developers should be alleviated from the responsibility of writing code to work with endsystem (like network) specific APIs, thereby reducing opportunities for code reuse.

In the UAV example, the UAVs can serve in different modes and send images or video or event notifications to the ground controllers. The number of types of ground controllers deployed depends on the number of modes in which the UAVs operate during an emergency response coordination. For example, if there are many UAVs producing notifications on the damage assessment of a severe flooding, then there could be multiple ground controllers deployed communicating with those UAVs. Depending on the importance of those notifications, those communications between the ground controllers and the command and control center could have different network QoS requirements. For example, in figure 1, the UAVs could be sending multiple event notification streams. The notification stream ground controller could be deployed as *ground controller A* sending notifications to the *display Y* and traversing routers *router P* and *router R*. Also, another notification stream ground controller could be deployed as *ground controller B* sending notifications to the *display Z* and traversing routers *router Q* and *router R*.

These ground controllers A and B are software components, that serves the same business logic of sending notifications, irrespective of the QoS requirement with which the notifications need to be sent. So in an ideal condition, the application developer would like to write code once, but deploy it in multiple targets (depending on the number of UAVs and their modes in deployment) with different QoS requirements. Hence the UAV scenario, requires underlying QoS-enabled middleware capabilities to automatically add QoS settings to be used to work with the configured network devices and elements in order to provide different QoS guarantees to the same software written and deployed under multiple deployment targets.
Section 3.4 describes a network QoS policy framework we have developed within our CIAO component middleware framework to transparently add platform specific QoS settings when inter-component remote calls are made, thereby honoring the network-level QoS management decisions made at deployment time. These decisions can be made at varied levels in which the applications can operate, including at the port level, at the thread level and at the object level, depending on how the remote calls are made.

3 Declarative Network QoS Provisioning Capabilities in Component Middleware

This section describes the novel enhancements we made to our QoS provisioning framework for component-based DRE systems. First, we provide an overview of our existing QoS provisioning framework. Next we describe our enhancements to this framework to enable network-level QoS provisioning.

3.1 Overview of Model-driven Component Middleware

The DOC group at Vanderbilt University has developed a framework for the developmental and operational lifecycle management of distributed, real-time and embedded (DRE) systems. Our solution illustrated in Figure 2 is a tool chain that uses model-driven engineering (MDE) [20] tools to capture application structural and behavioral characteristics converting it into metadata that are used by resource planners to determine resource allocations and component deployments. This information is then used by a deployment and configuration tool to host the application components in a QoS-enabled component middleware framework. We briefly describe individual artifacts of our existing framework below. Subsequent sections describe how we have enhanced each element of this tool chain to realize the network QoS provisioning goals.

CoSMIC Model-Driven Engineering Toolsuite. To simplify the development of component-based applications, we have developed the Component Synthesis of Model Integrated Computing (CoSMIC), which is an open-source\(^1\) set of model-driven engineering (MDE) tools that support the deployment, configuration, and validation of component-based DRE systems. A key capability supported by CoSMIC is the definition and implementation of domain-specific modeling languages (DSMLs) [7], which use concrete and abstract syntax to define the concepts, relationships, and constraints used to express domain entities [10].

\(^1\)CoSMIC is available for download from www.dre.vanderbilt.edu/cosmic.
A key CoSMIC DSML called *Platform Independent Component Modeling Language* (PICML) [1] enables graphical manipulation of modeling elements and performs various types of generative actions, such as synthesizing XML-based deployment plan descriptors defined in the OMG Deployment and Configuration (D&C) specification [15]. CoSMIC has been developed using a DSML development environment called Generic Modeling Environment (GME) [12].

**Resource and Control Engine (RACE).** The Resource and Control Engine (RACE) [23] is an adaptive resource management framework that provides (1) *resource monitor* components that track utilization of various system resources, such as CPU, memory, and network bandwidth, (2) *QoS monitor* components that track application QoS, such as end-to-end delay, (3) *resource allocator* components that allocate resource to components based on their resource requirements and current availability of system resources, (4) *configurator* components that configure QoS parameters of application components, (5) *controller* components that compute end-to-end adaptation decisions to ensure that QoS requirements of applications are met, and (6) *effector* components that perform controller-recommended adaptations.

**Deployment and Configuration Engine (DAnCE).** DRE system application component assemblies developed using CIAO are deployed and configured via DAnCE [5], which implements the OMG *Deployment and Configuration* (D&C) specifi-
cation [15]. DAnCE manages the mapping of DRE application components onto nodes in the target environment. The information about the component assemblies and the target environment in which the components will be deployed are captured in the form of standard XML assembly descriptors and deployment plans generated by CoSMIC tools and enhanced by RACE. DAnCE’s runtime framework parses these descriptors to extract connection and deployment information, and then automatically deploys the assemblies onto the CIAO component middleware platform described below establishing the connections between component ports.

**Component Integrated ACE ORB (CIAO).** CIAO is an open-source implementation of the OMG Lightweight CORBA Component Model (LwCCM) [13] and Real-time CORBA [14] specifications built atop TAO [21], which is our real-time CORBA ORB. CIAO’s architecture is designed based on (1) patterns for composing component-based middleware [26] and (2) reflective middleware techniques to enable mechanisms within the component-based middleware to support different QoS aspects [28].

### 3.2 Modeling Network QoS Requirements

**Context:** Challenge 1 in Section 2.2 describes the need for a scalable and declarative mechanism for specifying network QoS requirements including prioritizing flows along the inter port communication paths between components.

**Solution Approach: Domain-specific Modeling and Generative Programming.** Model-Driven Engineering (MDE) is a promising approach to address these challenges by raising the abstraction of system design to a level higher than third-generation programming languages. Modeling network QoS requirements and synthesizing the metadata from the model alleviates many deployment time concerns as well as eliminates the need for low level and potentially out of band programming. At the heart of our MDE approach to provision network level QoS support for DRE systems is a platform-specific DSML for LwCCM. The DSML is called the Component QoS Modeling Language (CQML), which is a platform-specific mapping of PICML explained in Section 3.1.

1. **Modeling network QoS requirements:** CQML annotates the elements modeled in PICML with platform-specific details and QoS requirements. The artifacts which can be annotated are component instances, component implementations,

---

2CIAO is available from [www.dre.vanderbilt.edu/CIAO](http://www.dre.vanderbilt.edu/CIAO).
connections between component ports, component assemblies, among others. Network QoS modeling is part of a of the larger QoS modeling capabilities of CQML. CQML allowes a deployer to model the network level QoS requirements in the QoS view of the DRE system as shown in Figure 3.

**Figure 3. Network QoS Modeling in CQML**

The QoS view exposes the basic structure of the DRE system in terms of the component instances in an assembly, component ports and the connections between them. In addition it captures network QoS requirements in the model in the following ways:

- **Annotating component connections.** Connections in a LwCCM-based application can be of two types: facet-receptacle and publisher-consumer. QoS attributes such as `HIGH_PRIORITY`, `HIGH_RELIABILITY`, `HIGH_THROUGHPUT`, `VIDEO` and `VOICE` can be assigned to the inter port connections. Moreover, bandwidth requirements (both minimum and maximum) for the connection can be captured so that appropriate bandwidth reservation can be made during deployment of the system.

- **Annotating component ports.** CQML provides a short hand for annotating the connections with network QoS attributes in a succinct way. If a component follows the *server declared* model of network QoS provisioning (see Section 3.3 for more details), then it can be succinctly represented in the model by simply specifying the appropriate ports of the component with desired QoS and the maximum bandwidth needs.

- **Sharing of the bandwidth pipe.** CQML allows modeling of a shared bandwidth pipe among multiple component connections. A set of $n$ client components connected to a remote provider component can share a pipe of bandwidth of say, X Mbps. If all of the $n$ client components are collocated then the aggregate bandwidth usage between the pair of hosts is limited to X Mbps. If the client components are distributed across multiple hosts then one strategy of sharing that is employed is to divide the bandwidth into $n$ equal parts.

**(2) Generation of annotated deployment metadata** Capturing the network QoS requirements is only half the story. These requirements have to be transformed into deployment descriptors so that the runtime environment can be configured if sufficient resources are available. A model interpreter, which is developed as a plugin in GME, traverses the annotated model of a DRE system in CQML and synthesizes annotated deployment descriptors. The advantages of the *visual formalism* of CQML are
immediately observed in the form of automated metadata generation. Some more non-trivial benefits of using MDE are summarised below.

- **Constraint checking.** The interpreter can also validate the network QoS requirements against the network resources provided that a model of the target domain in which the system is to be deployed is also captured in CQML.

- **Aspect weaving.** The deployment aspect of the DRE system is taken into account while interpreting the network QoS requirements. In particular, when sharing of a bandwidth pipe is intended, the pipe of bandwidth is split into multiple equal parts depending upon whether the components are collocated or distributed.

### 3.3 Deployment-time network resource reservation and device configuration

**Context:** Challenge 2 in Section 2.2 describes the need for network QoS provisioning mechanisms to work with component placement algorithms in order to configure the underlying network devices to provide network QoS for component middleware flows.

**Solution Approach: Network resource allocation planners** Section 3.1 described how RACE [23] uses standard component middleware mechanisms to allocate CPU resources to applications [11] and control DRE system performance once applications are deployed and running. Since RACE provides mechanisms to plug-in a series of resource allocation algorithms, we extended RACE to add a network resource allocator that utilizes a Bandwidth Broker (BB) [3] to provide our QoS-enabled component middleware framework with network QoS provisioning capabilities that (1) allocate network resources for component communication flows, (2) configure network devices with appropriate platform-specific QoS settings so that they can provide QoS requirements when component communication flows traverse through them, and (3) provision applications with desired QoS settings based on requirements captured in models so that the applications can work with the configured network devices and derive end-to-end network QoS.

Figure 4 illustrates the architecture of the BB and shows how it is integrated as a RACE *network resource allocator*. The BB leverages widely available vendor mechanisms that support layer-3 DiffServ (Differentiated Services) and layer-2 CoS (Class of Service) features in commercial routers and switches. In this context, the BB also provides mechanisms to configure the network devices to provision the requested QoS. The following are the key functionalities of the network resource allocator:
• **Admission control**, where the BB provides capabilities to reserve, commit, modify, and delete flows in support of allocation and scheduling for use by the UAV assembly components developed using CIAO.

• **Usage queries**, where the BB provides capabilities to query information about bandwidth availability in different classes of traffic.

• **Allocation policies**, where the BB provides capabilities to adapt to existing and future bandwidth reservations in support of any emergencies in the UAV emergency response deployments.

• **Device configurations**, where the BB provides capabilities to provision and configure routers and switches to manage network QoS for different UAV application flows traversing through those devices.

• **Application QoS configuration**, where the resource allocator captures the platform-specific QoS settings (like the DiffServ QoS settings) to be configured on the applications, so that the applications can work with the configured network devices in a seamless fashion.

The model-driven solution described in Section 3.2 provides design-time capabilities for component-based applications to express their network resource utilization requirements, network resource usage intentions and network QoS requirements. At
deployment time, when component resource allocations are made, RACE’s CPU resource allocators need to determine the hardware nodes in which the components are deployed, and then check if the determined hardware node allocation can also satisfy the network utilization requirements of the components deployed in those hardware nodes.

In order to assure network QoS guarantees to applications, the BB leverages layer-2 and layer-3 mechanisms in the following manner:

- Policing and marking, where the BB uses Access Control Lists (ACLs) available on COTS network elements to classify traffic into flows that can easily be policed. The classification usually depends on the TCP/IP five tuple: source address, destination address, protocol, source port and destination port. The source address and destination address are made available to the BB by the RACE resource allocator, but the port numbers are not known until the components are deployed. So the BB returns a DSCP marking for each flow, which the QoS-enabled component middleware needs to send whenever such a flow is initiated by the source component. RACE’s network resource allocator also updates the component deployment metadata to capture these DSCP markings as part of the component connection descriptions so that DAnCE can configure the applications with those QoS settings when remote communications are made.

- Scheduling and buffer management, where the BB uses popular vendor mechanisms, such as simple round-robin mechanisms to more sophisticated mechanisms like the weighted fair queuing. The BB provides capabilities to configure those queues to honor network QoS.

- Transport of QoS markings, where the BB uses the DSCP markings available in the IP header to make scheduling and buffer management decisions. When the traffic traverses layer-2 network segments, the DSCP markings are translated to corresponding layer-2 CoS values. Atmost 8 distinct classes of service can be identified on the layer-2 segments. The translation between the DSCP markings to layer-2 classes of services are done by the BB and the layer-2 network segments are configured to provide per-hop behaviors based on those translations. So the applications need to send data with the appropriate DSCP markings so that the configured routers and switches can then capture all those packets that are marked and translate them into appropriate traffic classes thereby providing network QoS for the marked flows. As described in Section 3.4, our enhancements to CIAO middleware make it feasible to mark outgoing application packets with these DSCP markings.
3.4 Middleware Policy Framework

4 Experimental Validation of XYZ

4.1 Experimental Setup

4.2 Experiment 1:

4.3 Experiment 2:

4.4 Experiment 3:

5 Related Work

6 Concluding Remarks

References


