

Using Composition of QoS Components to Provide Dynamic, End-to-End QoS in Distributed Embedded Applications – A Middleware Approach

Praveen Sharma, Joseph Loyall, Richard Schantz,
Jianming Ye, Prakash Manghwani, Matthew Gillen
BBN Technologies
{psharma, jloyall, schantz, jye, pmanghwa, mgillen}@bbn.com

George T. Heineman
Worcester Polytechnic Institute
heineman@cs.wpi.edu

Abstract. Maintaining end-to-end *Quality of Service* (QoS) in distributed applications operating within dynamically changing environments is challenging. We have been developing a middleware QoS management approach based upon composing *QoS Components*. In this paper, we illustrate and evaluate the approach using a real world medium-scale example we've built. We discuss the benefits of our approach and issues that arise while composing QoS components.

Keywords: Quality of Service, Component-Based Software Engineering, Middleware

1. Introduction

Distributed Real-Time Embedded (DRE) systems are increasingly at the core of a wide range of domains, including telecommunications, medicine and disaster response, military, and e-commerce. They are network-centric with real-time constraints and use Internet protocols and principles to communicate. In addition to requiring the stringent *quality of service (QoS)* of traditional *closed* embedded systems, DRE systems are subject to larger *end-to-end* QoS requirements (i.e., managing resources for all participants and shaping application data attributes throughout an application's life cycle) and *dynamic conditions* associated with being widely distributed across an often volatile network environment.

We have been developing a middleware approach to provide dynamic and end-to-end QoS management in DRE systems using encapsulations of QoS management code segments,

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contracts F33615-00-C-1694 and F33615-03-C-3317.

called *Qoskets* [1]. In prior work, we demonstrated how instantiations of Qoskets helped provide dynamic QoS management in DRE object applications [2]. With the emergence of component middleware, we have developed *component* instantiations of Qoskets, *qosket-components* (QCs) [3][4], for the CORBA Component Model (CCM) [5], an avionics domain-specific component model (PriSm) [6], and the Java Bean-based component model (Cougaar) [7].

The primary benefit of QCs is in demonstrating the feasibility to integrate end-to-end and dynamic QoS management into DRE systems by composing specialized QoS management mechanisms using middleware. Encapsulating QoS behaviors as components conceivably allows any QoS mechanism to be used by the middleware layer, as long as the QoS developer can provide suitable interfaces to control it. Furthermore, since QoS tradeoffs are inevitable in the dynamic, distributed environments we are targeting, our modular approach makes it feasible to identify and expose the configuration and runtime parameters through which dynamic tradeoffs can be made.

In this paper, we illustrate and evaluate progress on how these individual, off-the-shelf QCs can be composed to provide dynamic, end-to-end QoS, and discuss some of the tradeoffs, issues, and benefits of our approach, focusing on the CCM implementation of QCs. Section 2 briefly introduces a real-world example to illustrate the challenges of providing QoS management in DRE systems and motivates our approach for composing individual QoS behaviors to create aggregate, end-to-end behavior. Section 3 reviews the Qosket framework. Section 4 summarizes the roles that QCs fill and describes different categories and instances of QCs that we have built. Section 5 introduces important composition patterns and discusses benefits, issues, and challenges related to QC composition. Section 6 draws some conclusions based on current work and discusses future directions.

2. Motivating Example

We illustrate the need for and application of composing individual QoS behaviors using a real-world live-flight and live-fire DRE system we demonstrated at the White Sands Missile Range (WSMR) in April 2005 [8][9]. The full system included multiple subsystems – coded in different languages using a variety of component models – composed through middleware and Web Services interfaces. Many details omitted here for lack of space can be found in [9].

In the demonstration, command personnel at a command and control center (C2) manage several Unmanned Air Vehicles (UAVs) while engaging a time-critical target. These UAVs operate in a constrained network with limited CPU resources and are assigned different functional roles – surveillance, target tracking (TT) and battle damage indication (BDI). During surveillance, UAVs send imagery with sufficient resolution of the sur-

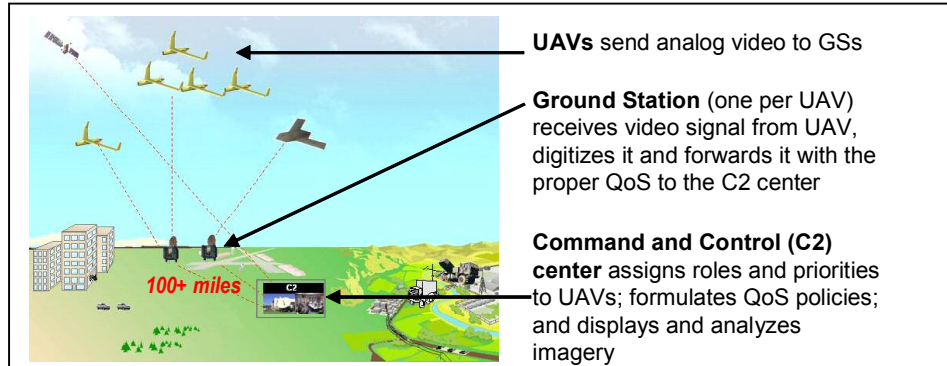
veiled area to help commanders determine items of interest. During TT, UAVs track targets and send images or streaming video at sufficient resolution and rate to enable a commander to follow unfolding situations. During BDI, UAVs send images with high resolution at regular intervals to enable human operators to analyze details. Each role has distinct QoS requirements for application data such as image rate, size, and resolution and resources are dynamically allocated based upon the roles. UAVs can change their roles as needed, e.g., when a commander identifies a potential threat. Each role is assigned a priority and has specific resource requirements; TT has the highest priority, followed by BDI, then surveillance.

As depicted in Figure 1a, we used six UAVs (2 real and 4 simulated) with sensors that sent imagery to the C2 center to be processed and displayed. Each UAV sensor transmitted analog video to its ground station (GS), where an imagery sender process digitized it and sent it (with the proper QoS) to the Displays in the C2 center, over 100 miles away connected via a shared fiber optic network. A Virtual LAN, using routers to control and distinguish traffic, provided 80MB/sec network capacity, shared by the six UAV-to-C2 image streams, C2 traffic, and other demonstration-specific traffic. This system illustrates several challenges:

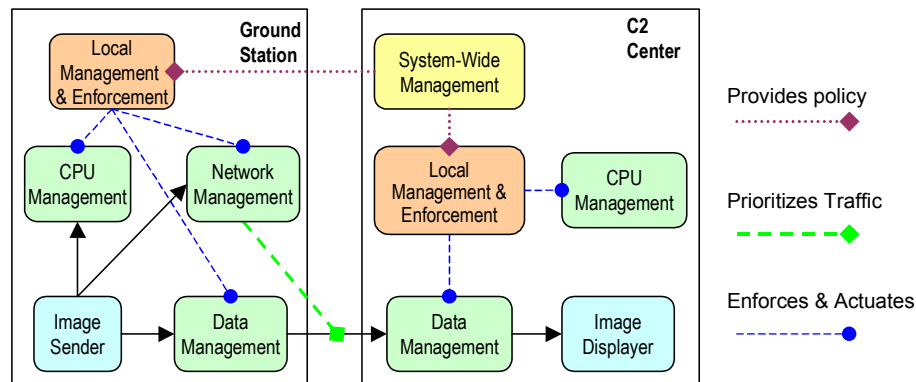
- Managing QoS *end-to-end* dictated by the *user*, provided by the application and delivered by the infrastructure within the specified time constraints.
- *Dynamic adaptation* of QoS based upon dynamically changing application requirements, operating conditions, and available resources.
- *Cross-layer* mapping of the application's QoS requirements (higher *mission-layer* concepts such as required fidelity) to lower *resource-layer* concepts such as resource availability and specific mechanisms to provision resources and control QoS.

Providing QoS in such a system is challenging, much less providing end-to-end and dynamic QoS. As illustrated in Figure 1b, the system needs QoS management software that provides:

- *System-wide management* of all participants, available resources, and QoS and application requirements of the system. It needs to allocate resources among the participants and needs to be located at the C2 center where the policies are formulated.
- *Local management and enforcement* of the policies based on the allocations made by actuating the adaptations. It needs to be associated with an asset, e.g., the ground station associated with a UAV.
- *Mechanisms* to control system resources and shape data streams to meet resource constraints. These need to be local to the resource or data source.



(a) The demonstration includes live UAVs sending imagery and receiving command and control signals with end-to-end QoS management.



(b) The demonstration includes a *system-wide manager* located at the C2 node providing QoS policy to *local managers*, which enforce QoS using resource control and data shaping *mechanisms*.

Figure 1: End-to-end QoS management in the live-flight and live-fire demonstration

To avoid the need for embedding QoS management code throughout the application, we separate QoS concerns from functional concerns by designing a QC encapsulating each individual management or mechanism behavior. We then compose these QCs to create end-to-end aggregate behavior. In the following section we briefly describe the QC Framework and discuss the types of QCs we developed for the demonstration.

3. The Qosket Encapsulation Framework

This section provides a brief overview of the Qosket framework. More detailed description is available in [1]. A Qosket is an encapsulation of QoS management code, supporting the separate development of QoS concerns from functional concerns and improved modularity and reuse. A Qosket includes code providing the following capabilities necessary for QoS management:

- (1) **Monitoring** the state of a particular QoS property in the system, often encompassing items such as available resources, resources used, and the satisfaction of a QoS policy.
- (2) **Decision Making** needed to control and deliver QoS, to mediate conflicting demands, to gracefully degrade and adapt as conditions change, and to meet applications' requirements within resource allocations.
- (3) **Actuation**, i.e., providing, enforcing, and controlling QoS through system, property, or resource manager interfaces.

QoS behavior in Qoskets is instantiated as code artifacts – objects, wrappers, classes, components and methods – that implement the QoS monitoring, decision making, and actuation throughout a distributed application. Previously, we have demonstrated QoS provisioning by instantiating Qoskets in distributed object applications [2]. In this paper, we concentrate on the instantiation of Qoskets as components, i.e., QCs.

4. Qosket Components and Categories of Qosket Components

4.1. Qosket Components

QCs are component instantiations of qoskets, consisting of qosket code wrapped inside standards-compliant components that can be assembled and deployed using existing tools and infrastructure [4]. QCs expose ports allowing them to be assembled between functional components and services, mechanisms, and system components (to intercept and adapt the interactions between the components). These QCs provide all the features of qoskets (allowing monitoring, decision making, and actuation) and all the features of components to provide lifecycle support for design, assembly, and deployment. Each QC encapsulates a single QoS behavior but can provide an aggregate, end-to-end behavior when composed with other QCs.

4.2. Categories of Qosket Components

As identified in Section 2, DRE systems require various types of QoS management software. To address this, we developed and classified QCs into three distinct categories based on the *scope* and *role* of the QC.

- **Managerial QCs** have a system-level view of the functional and system components of an application. They maintain specifications for QoS requirements and

domain specific application requirements that are translated into QoS policies. Relatively speaking, managerial QCs consist primarily of decision making, with only high-level monitoring, and little or no actuation. We developed a **System Resource Manager (SRM)** managerial QC that was hosted at the C2 center. The SRM QC dynamically allocates system resources based on the number of participants in the system, relative priorities and total available resources using weighted utility functions. It creates QoS policies that include the roles, priorities, and allowed quality ranges, and sends the policies and allocations to the enforcement QCs on each host.

- **Enforcement QCs** provide local management for a node (or group of nodes). They receive policies from managerial QCs, translate the policies into actions, and enforce the policies. They decide which resource and application controls (in the form of mechanism QCs) are best able to enforce the policies, in what combination, and at what granularity within the resource and time constraints. They combine decision making with actuation that turns on, configures, and accesses the control interfaces of mechanism QCs. We developed a **Local Resource Manager (LRM)** enforcement QC that manages resources and tradeoffs on each host. It configures specific mechanisms to control resources, shape data, and alter the application behavior to satisfy the resource allocations and QoS policies sent to it by the SRM QC.
- **Mechanism QCs** access system controls and interfaces to effect a particular QoS control or behavior. Their actuation can range from controlling a resource to shaping application data or altering algorithm parameters. These QCs monitor and act only on local information, and are directed by enforcement and managerial QCs. Mechanism QCs can include decision-making capabilities, e.g., to select the best compression algorithm to match the traffic format, but will – in general – have comparatively less decision-making capability than enforcement or managerial QCs. For the demonstration, we utilized the following mechanism QCs:
 - *Network Prioritization DiffServ QC* – prioritizes outgoing IP traffic.
 - *CPU Reservation Mechanism QC* – reserves a percent of the CPU.
 - *Encrypt and Decrypt Security QC* – encrypts and decrypts data.
 - *Pacing and Depacing QC* – minimizes network jitter.
 - *Application Adaptation or Data Shaping QCs*
 - *Compression QC* – compresses imagery.
 - *Cropping QC* – crops images by removing pixels from each side.
 - *Scaling QC* – scales images to either half or quarter size.

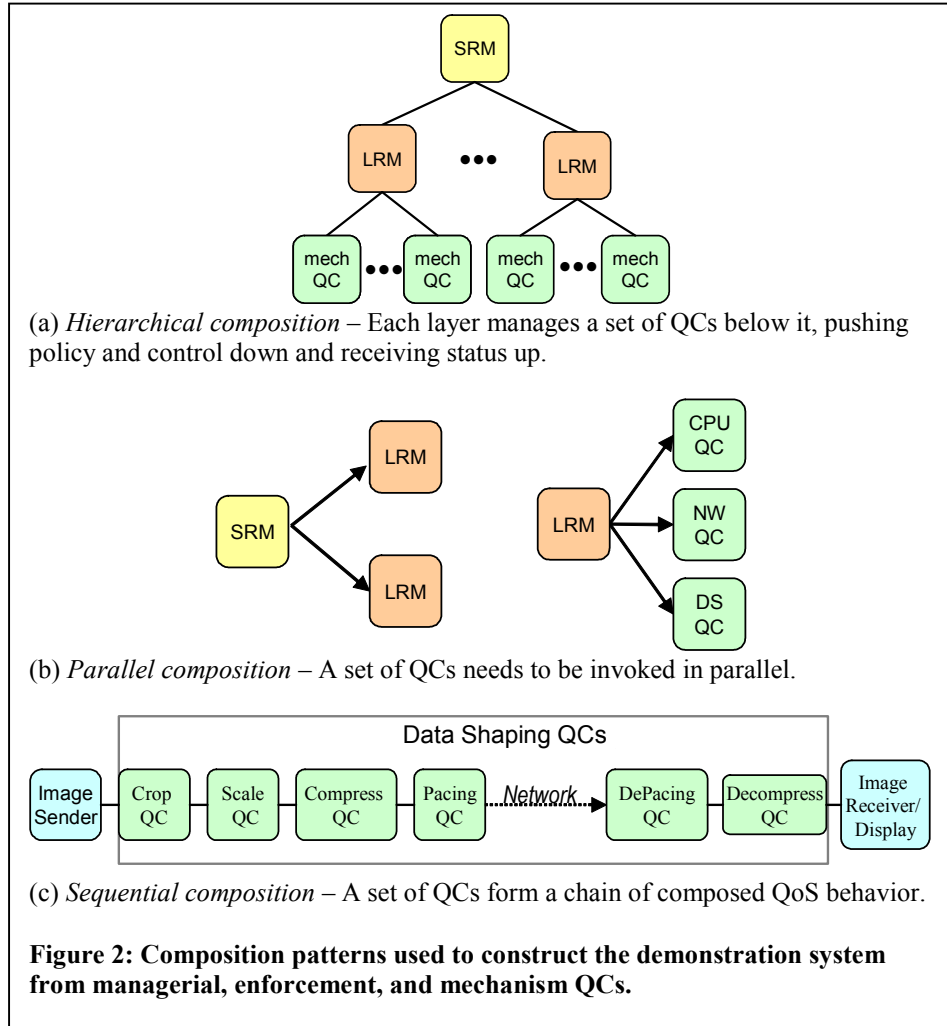
5. Composing QCs to Provide Dynamic, End-to-End QoS

We assembled the demonstration system by composing the QCs described in Section 4 with the application's functional components (e.g., the image sender, image receiver, and processing components) through the component and QC interfaces. During the demonstration, the SRM QC at the C2 center uses the number, priorities and roles of the demonstration participants to create a resource allocation and policy for each participant whenever there is a change in the mission, roles or available resources. The SRM QC pushes the allocation and policy to all LRM QCs, each of which then selects, configures, and activates the correct set of mechanism QCs to satisfy the policy within the allocated resources.

There is an LRM QC on the UAV ground station side and one on the C2 side for each information stream. Each LRM is composed with a set of mechanism QCs that it configures and activates to satisfy the policy with the allocation provided by the SRM [10].

We composed the QCs using the following composition patterns, illustrated in Figure 2:

- *Hierarchical (Layered) Composition* (figure 2a) – In this pattern, managerial QCs are layered upon enforcement QCs, which in turn are layered upon mechanism QCs. In our demonstration, the SRM QC formed the top of the hierarchy, pushing policy down to the LRM QCs and receiving status back. The LRM QCs, in turn, controlled a set of mechanism QCs (resource control and data shaping QCs).
- *Parallel Composition* (figure 2b) – When QCs receive data simultaneously and perform their QoS behavior independently, they can be composed in parallel. Our demonstration system had two examples of parallel composition. First, the SRM QC must send its policy to all the LRM QCs associated with a data stream (e.g., at the GS source of the data and the C2 destination) in parallel, since they must coordinate QoS management. Second, the LRM QC enforcing the QoS policy must simultaneously control the CPU resource mechanism QC, the Network mechanism QC, and the data shaping QCs to produce the proper aggregate behavior.
- *Sequential Composition* (figure 2c) – Often a set of QCs must be tightly integrated such that a set of QoS behaviors are performed sequentially, with the output of each QC becoming the input to the next QC. In our demonstration system, the data shaping QCs used this pattern, controlling the rate by which data is sent, then altering the data (by cropping, scaling, and/or compression), and finally pacing it (i.e., sending it in pieces over a period of time to control jitter).



5.1. Factors affecting QC Composition

We continue to explore the formalisms, tools, and analyses that can guide effective composition. A complete treatment of all the issues is beyond the limits of this paper. However we briefly discuss the following issues illustrated by our demonstration system:

- *Composition Order* – The composition order of some QCs are important to the effectiveness or feasibility of end-to-end QoS management. For example, we

compose cropping before compression because the cropping QC can only process non-compressed image formats. As another example, a QC adjusting the rate of imagery should be composed before other data shaping QCs, otherwise one might waste time and resources performing compression or cropping on an image that is not sent.

- *Paired QCs* – Some QCs have to combine their behavior with corresponding “undo” behaviors. For example, in our demonstration each compression QC on the source side requires a corresponding decompression QC on the receiver side. The “undo” QCs usually need to be composed in reverse order of the composition of their paired QCs.
- *Explicit and Implicit Dependencies* – There are dependencies between QCs that can be useful to guide the composition or that can restrict the circumstances in which they can be usefully composed. Explicit dependencies can be reflected in QC interfaces, such as the policy interface provided by the LRM and used by the SRM, or type matching, such as the cropping QC working only with specific, uncompressed data types. Other implicit dependencies can be due to semantic or algorithmic factors, and can be more difficult to detect and manage. For example, some encryption and compression algorithms might not compose well, since encrypting might restrict the ability to compress data very much and compressing data might produce a result that cannot be properly encrypted. We are still investigating methods to incorporate this type of information so that automated tools can verify appropriate composition.
- *Placement of QCs* – The host boundaries on which QCs are deployed can play a crucial role in the effectiveness of the aggregate QoS management. Placing data shaping QCs closest to the data source makes the most sense *unless* the data is used by multiple consumers demanding different qualities. Some QCs are only effective when separated by host boundaries. For example, compression can only reduce network traffic if the compression and decompression QCs are placed on different hosts. In addition, because our demonstration was based on a military scenario, there was a defined central authority in the C2 center and, therefore, an obvious place to put the SRM managerial QC. A peer-to-peer or ad hoc system, however, might need a different number and placement of managerial QCs. We are still experimenting with understanding and how to express the relationships between placement of QCs and outcome of the end-to-end compositions.

5.2. Benefits of QC Composition

There are a number of benefits associated with building systems using embedded QCs, rather than as a one-of-a-kind, custom crafted stove-piped system, including the following:

- *Reusability* – Many of the QCs that were used in this demonstration were reused or adapted from earlier contexts and are members of our QC library. Part of our ongoing work involves exploring the tradeoffs associated with decoupling a QC from functional interfaces (thereby increasing its reusability in different contexts) but increasing the work associated with composing it in a specific context.
- *Simplified development* – Using the embedded QC approach, providing QoS management in a DRE system becomes more of a configuration issue rather than a programming exercise. One can, therefore, assemble the components required for QoS management into an existing or developing component-based distributed application. In our demonstration system, this allowed us to rapidly prototype versions of the system with or without specific QoS behaviors and with specific combinations of QCs, simply by assembling the system using available CCM assembly tools
- *QoS management at different epochs* – Traditional embedded systems rely on static QoS provisioning, at design time or system configuration time. Our approach supports QoS provisioning at several different lifecycle epochs of an application as follows:
 - At configuration time, one can set the default values of QC attributes. For example, the attributes of an LRM QC can be set to define the default strategy for selecting and activating a QC.
 - At assembly time, one can compose QCs to provide a desired aggregate or end-to-end QoS.
 - At deployment time, the placement of QCs and their monitoring sources affects the provided QoS. Prior knowledge of the host and network load can facilitate the process of selecting suitable hosts.
 - At runtime, QCs facilitate the dynamic control of QoS and adaptation to changing conditions.

5.3. Challenges and Proposed Solutions

While we have had success in developing QCs and in composing them to create DRE systems, such as the demonstration system, there are still many issues left to investigate on the path to common practice and operational use. In this section, we discuss a few of the most important ones, which derive from our recent experience with putting these ideas into practice.

Data-specific QCs. There is a tradeoff to be made in developing a QC that is specific to a particular data format versus one that is not. A QC that is not specific to a particular data

format should be more widely reusable. However, this might not always be feasible. For example, an attempt to develop a format-neutral compression QC leads to the following pitfalls:

- *A useless QC* – Trying to remove code that understands specific data formats from a QC might result in an empty shell that contains little behavior and requires everything to be specified at assembly time or compensated for elsewhere.
- *A bloated QC* – Including a wide variety of algorithms that work with many different formats might create an unwieldy QC that is too heavyweight for any specific context.
- *An inefficient QC* – There are format neutral compression algorithms, such as gzip, that could be used. However, in many cases, data specific algorithms are more useful. JPEG compression, for example, is more useful for imagery because it compresses efficiently and comes with display software. Over time, emerging standards, de facto or de jure, are likely to help alleviate some of these issues, if for no other reason than to reduce the space of acceptable choices.

In much of our work we have made QCs as format-neutral as possible, even while continuing to work on additional solutions, such as QC interface templates. This can lead to the problem of QC *data incompatibility*, in which data emitted from one set of QCs may not be compatible as input to another set of QCs. Currently, we have no way to specify this or annotate the QCs to aid the assemblers. The assemblers need knowledge of the domain’s data types and functional components, and therefore must either work with domain experts or possess domain expertise. This problem does not propagate to application code because each QC that alters the output data is paired with a QC that undoes the alteration, as described in Section 5.1.

Need for hardware and system support. QCs that provide system-level controls and monitoring require support from the system infrastructure to work correctly. For instance, a DiffServ QC that provides network prioritization requires universal support for DiffServ capabilities at all intermediate routers. This becomes difficult over an uncontrolled network, such as the Internet. A solution to this problem is to use only more controlled subsets, to emphasize traffic shaping techniques instead, or to use a reactive approach that adjusts to the provided QoS, even in a “best effort” environment such as the Internet.

Maintaining QoS while integrating with other middleware services. It is tricky to provide end-to-end QoS dynamically when the QCs need to interoperate with other middleware services such as the Joint Battlespace Infosphere (JBI), a publish/subscribe oriented service [11] or the CORBA Notification Service [12]. While these other middleware services are individually compliant with the standards, there is no uniform protocol for communicating among them, or maintaining QoS while doing so. Our solution has been to provide as much QoS as possible, up to the boundaries of entering uncontrolled services and introducing QCs that react to the observed QoS in uncontrolled environments, while

at the same time promoting and helping to develop QoS management awareness and capabilities for these other middleware services.

6. Conclusion

Encapsulating QoS code as components promises to simplify the construction of applications with managed QoS, by providing similar lifecycle and composition benefits for QoS development that is currently available for functional development. Our Qosket approach takes this a step further by supporting components that combine monitoring, decision making, and actuation into encapsulated dynamic QoS behaviors and managers, rooted in middleware and component standards. Demonstrations such as the one described in this paper illustrate the practical effectiveness of our approach. We also believe our approach offers an exceptional vehicle to continue to explore the challenges and issues associated with separating and decomposing QoS management as a practical reality for real-world systems.

The Qosket and QC work we have described is ongoing, and this paper provides only a view into part of it. For example, work on developing QoS engineering models and model driven tools are in its early stages [13]. In the future, we plan to offer better support for encapsulation, reuse, and composition, with the goals of producing better software engineering support for dynamic QoS management, and developing a generally useful model for and theory of the composition of QoS behaviors.

References

- [1] Richard E. Schantz, Joseph P. Loyall, Michael Atighetchi, and Partha Pal. Packaging Quality of Service Control Behaviors for Reuse. ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing, April 29 - May 1, 2002.
- [2] Joseph P. Loyall, J. M. Gossett, Christopher Gill, Richard E. Schantz, John Zinky, Partha Pal, Richard Shapiro, Craig Rodrigues, Michael Atighetchi, and David Karr. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21), April 16-19, 2001.
- [3] George T. Heineman, Joseph P. Loyall, and Richard E. Schantz. Component Technology and QoS Management. International Symposium on Component-based Software Engineering (CBSE7), May 24-25, 2004.

- [4] Praveen K. Sharma, Joseph P. Loyall, George T. Heineman, Richard E. Schantz, Richard Shapiro, Gary Duzan. "Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems," *International Symposium on Distributed Objects and Applications (DOA)*, October 25-29, 2004.
- [5] Object Management Group, CORBA Component Model, V3.0 formal specification, <http://www.omg.org/technology/documents/formal/components.htm>
- [6] Wendy Roll. Towards Model-Based and CCM-Based Applications for Real-time Systems. 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), Hakodate, Hokkaido, Japan, 2003, 75-82.
- [7] Cougaar – component model extended from Java Bean component Model: <http://www.cougaar.org/>
- [8] Program Composition for Embedded Systems Program, DARPA, http://dtsn.darpa.mil/ixo/ixo_FeatureDetail.asp?id=126.
- [9] J. Loyall, R. E. Schantz, D. Corman, J. Paunicka, S. Fernandez, "A Distributed Real-time Embedded Application for Surveillance, Detection, and Tracking of Time Critical Targets," Real-time and Embedded Technology and Applications Symposium (RTAS), pp. 88-97, March 2005.
- [10] Prakash Manghwani, Joseph Loyall, Praveen Sharma, Matthew Gillen, and Jianming Ye. [End-to-End Quality of Service Management for Distributed Real-time Embedded Applications](#). The Thirteenth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2005), Denver, Colorado, April 4-5, 2005.
- [11] Joint Battlespace Infosphere, <http://www.rl.af.mil/programs/jbi/>
- [12] CORBA Notification Service, http://www.omg.org/technology/documents/formal/notification_service.htm
- [13] Jianming Ye, Joseph Loyall, Richard Shapiro, Richard Schantz, Sandeep Neema, Sherif Abdelwahed, Nagabhushan Mahadevan, Michael Koets, Denise Varner. "A Model-Based Approach to Designing QoS Adaptive Applications." The 25th IEEE International Real-Time Systems Symposium, Lisbon, Portugal, December 5-8, 2004.

Related Work

Given the extensive QoS and component literature, we focus on approaches that are most directly related to our efforts, specifically regarding composition and middleware.

Middleware. Many researchers seek to coordinate shared resource access to support dynamic, end-to-end QoS management. BRENTA [1] concentrates on contract-based negotiation of network QoS by assuming applications can adapt to available resources. QARMA [2] adds a resource manager and system repository to the available CORBA services. In the CCM framework the notion of QoS is not part of the standard specification. We have worked with the developers of CIAO [3] to define static and dynamic QoS support for CCM within the CIAO framework. The Qedo (QoS enabled Distributed Objects) effort provides QoS to components by integrating data streams (based on their Streams for CCM specification) [4]. While the middleware framework should be responsible for providing important QoS-related mechanisms, there needs to be a way for applications to specify policies that work with the middleware to plan for, or otherwise negotiate, QoS needs, as we demonstrate with our QC approach. Formal models of adaptive QoS-enabled middleware employ a two-level structure (similar to our management QCs) to manage the concurrent execution of application activities and services for resource management [5]. Our working system embodies such a structure.

QoS composition. In the Web Services domain, Business-to-Business (B2B) interactions are themselves formed by composing existing services together. One common approach is to design a middleware platform that selects and composes appropriate services from available Web Services. Zeng et al. [6] rely on a planner and execution engine that uses integer programming to select optimal plans based on data and execution dependencies. Erradi and Maheshwari rely on lightweight broker architecture to ensure dependability of Web services [7]. Wohlstadter et al. present an architecture that actively mediates the QoS requirements of clients and servers at run-time [8]. It is certainly feasible to apply our QCs to manage, control, and mediate between Web Services as we now do with functional components. One reason for our initial concentration on a CCM context for QoS composition is the more advanced state of that middleware for the real-time embedded domains driving our applications of interest. It is our view that eventually, these concepts seem destined to emerge in all forms of component and service integration approaches.

- [1] D. Mandato, A. Kassler, T. Valladares, G. Neureiter. "Handling End-To-End QoS in Mobile Heterogeneous Networking Environments," Proceedings, 12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, October 2001.
- [2] D. Fleeman, M. Gillen, A. Lenharth, M. Delaney, L. Welch, D. Juedes, C. Liu. "Quality-based Adaptive Resource Management Architecture (QARMA): A CORBA Resource Management Service", International Parallel and Distributed Processing Symposium, April 2004.
- [3] Component Integrated ACE ORB (CIAO), www.cs.wustl.edu/~schmidt/CIAO.html.
- [4] Qedo -QoS Enabled Distributed Objects, www.qedo.org.
- [5] N. Venkatasubramanian, C. Talcott, G. Agha, "A Formal Model for Reasoning About Adaptive QoS-Enabled Middleware", *ACM Transactions on Software Engineering and Methodology*, 13(1), pp. 86-147, 2004.
- [6] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, H. Chang, "QoS-aware Middleware for Web Services Composition", *IEEE Transactions on Software Engineering*, 30(5), pp. 311-327, 2004.
- [7] A. Erradi, P Maheshwari, "wsBus: QoS-Aware Middleware for Reliable Web Services Interactions", *IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE)*, pp. 634-639, 2005.
- [8] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu, GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions, Proceedings, 26th International Conference on Software Engineering (ICSE), 2004.