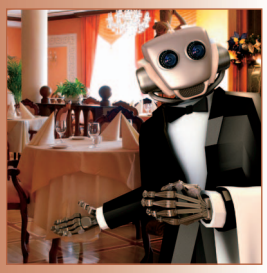

Managing End-to-End QoS in Distributed Embedded Applications

Maintaining end-to-end quality of service (QoS) is a challenge in distributed real-time embedded systems due to dynamically changing network environments and resource requirements. The authors' middleware QoS management approach encapsulates QoS behaviors as software components. Using the Corba Component Model, they build these specialized QoS components and combine them to produce a comprehensive management system that maintains QoS. The authors illustrate their approach by building a real-world medium-scale system with these components. Using this example, they demonstrate the reusability of each component in different contexts.

Praveen Kaushik Sharma, Joseph Loyall, Richard E. Schantz, Jianming Ye, Prakash Manghwani, and Matthew Gillen
BBN Technologies

George T. Heineman
Worcester Polytechnic Institute

Distributed real-time embedded (DRE) systems are increasingly at the core of domains ranging from telecommunications to medicine, disaster response, and e-commerce. These systems are network-centric with real-time constraints and use Internet protocols and principles to communicate. In addition to stringent quality-of-service (QoS) requirements from traditional closed embedded systems, DRE systems have greater end-to-end QoS needs (such as managing resources for all participants and shaping application data attributes throughout an application's life cycle) and are distributed across volatile network environments.

BBN Technologies has been developing a middleware approach to provide dynamic, end-to-end QoS management in DRE systems using encapsulations of QoS-management code segments, called *Qoskets*.[1] (See the "Related Work in Middleware and QoS Composition" sidebar for other work in this area.) In prior work, we demonstrated how Qoskets helped provide dynamic QoS management in DRE object applications.[2] We developed Qosket component instantiations, or *Qosket components* (QCs),[3,4] for the Corba component model (CCM),[5] an avionics domain-specific component model (PriSm),[6] and the Cougaar JavaBean-based component model (www.cougaar.org).

QCs help us demonstrate the feasibility of integrating end-to-end, dynamic QoS management into DRE systems by

 Published by the IEEE Computer Society 1089-7801/06/$20.00 © 2006 IEEE

Related Work in Middleware and QoS Composition

Given the extensive quality of service (QoS) and component literature, we focus on approaches that are most directly related to our efforts, specifically regarding composition of QoS 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 that applications can adapt to available resources. QARMA[2] adds a resource manager and system repository to the available Corba services.

In the Corba component model (CCM) framework, QoS isn't part of the standard specification. We've worked with the developers of Component Integrated Adaptive Communication Environment (ACE) ORB (CIAO; www.cs.wustl.edu/~schmidt/CIAO.html) to define static and dynamic QoS support for CCM within the CIAO framework. The QoS Enabled Distributed Objects (Qedo; www.qedo.org) effort provides QoS to components by integrating data streams (based on their streams for the CCM specification).

Although the middleware framework should be responsible for providing important QoS-related mechanisms, applications must be able to specify policies that work with the middleware to plan for, or otherwise negotiate, QoS needs, as we demonstrate with our Qosket component (QC) approach. Formal models of adaptive QoS-enabled middleware employ a two-level structure (similar to our management QCs) to concurrently execute application activities and services for resource management.[3] Our working system embodies such a structure.

QoS Composition

In the Web services domain, business-to-business (B2B) interactions are formed by combining existing services. One common approach is to design a middleware platform that selects and combines appropriate services from available Web services. Liangzhao Zeng and his colleagues[4] rely on a planner and execution engine that uses integer programming to select optimal plans based on data and execution dependencies. Abdelkarim Erradi and Piyush Maheshwari rely on a lightweight broker architecture to ensure the dependability of Web services.[5] Eric Wohlstadter and his colleagues present an architecture that actively mediates the QoS requirements of clients and servers at runtime.[6]

We could apply our QCs to manage, control, and mediate Web services as we now do with functional components. One reason for our initial concentration on a CCM context for QoS composition is that the middleware area is more advanced for the real-time embedded domains driving our applications of interest. We believe these concepts will eventually emerge in all forms of component and service-integration approaches.

References

1. D. Mandato et al., "Handling End-to-End QoS in Mobile Heterogeneous Networking Environments," *Proc. 12th IEEE Int'l Symp. Personal, Indoor and Mobile Radio Communications*, 2001; http://ieeexplore.ieee.org/iel5/7636/20844/00965251.pdf?arnumber=965251.
2. D. Fleeman et al., "Quality-Based Adaptive Resource Management Architecture (QARMA): A CORBA Resource Management Service," *Proc. Int'l Parallel and Distributed Processing Symp.*, IEEE CS Press, 2004, p. 116b.
3. N. Venkatasubramanian, C. Talcott, and G. Agha, "A Formal Model for Reasoning About Adaptive QoS-Enabled Middleware," *ACM Trans. Software Eng. and Methodology*, vol. 13, no. 1, 2004, pp. 86–147.
4. L. Zeng et al., "QoS-Aware Middleware for Web Services Composition," *IEEE Trans. Software Eng.*, vol. 30, no. 5, 2004, pp. 311–327.
5. A. Erradi and P. Maheshwari, "wsBus: QoS-Aware Middleware for Reliable Web Services Interactions," *Proc. IEEE Int'l Conf. E-Technology, E-Commerce and E-Service* (EEE), IEEE CS Press, 2005, pp. 634–639.
6. E. Wohlstadter et al., "GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions," *Proc. 26th Int'l Conf. Software Eng.* (ICSE), IEEE CS Press, 2004, pp. 189–199.

creating specialized QoS management mechanisms using middleware. Encapsulating QoS behaviors as components conceivably lets the middleware layer use any QoS mechanism as long as the QoS developer can provide suitable interfaces to control it. Furthermore, because QoS trade-offs are inevitable in the environments we're targeting, our modular approach lets us identify and expose the configuration and runtime parameters with which we can make dynamic trade-offs.

This article illustrates how to combine individual, off-the-shelf QCs to provide dynamic, end-to-end QoS. We discuss some of our approach's trade-offs, issues, and benefits, focusing on the CCM implementation of QCs.

Motivating Example

We can illustrate the need for combining 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 (http://dtsn.darpa.mil/ixo/appareas.asp?id=126).[7] The full system included multiple subsystems — coded in different languages using various component models — composed of middleware and Web services interfaces. (Full system details are available elsewhere.[7])

In the demonstration, Command and Control (C2) center personnel managed several unmanned air vehicles (UAVs) while engaging a time-critical target — that is, an immediate threat or fleeting target of opportunity. These UAVs operate in a con-

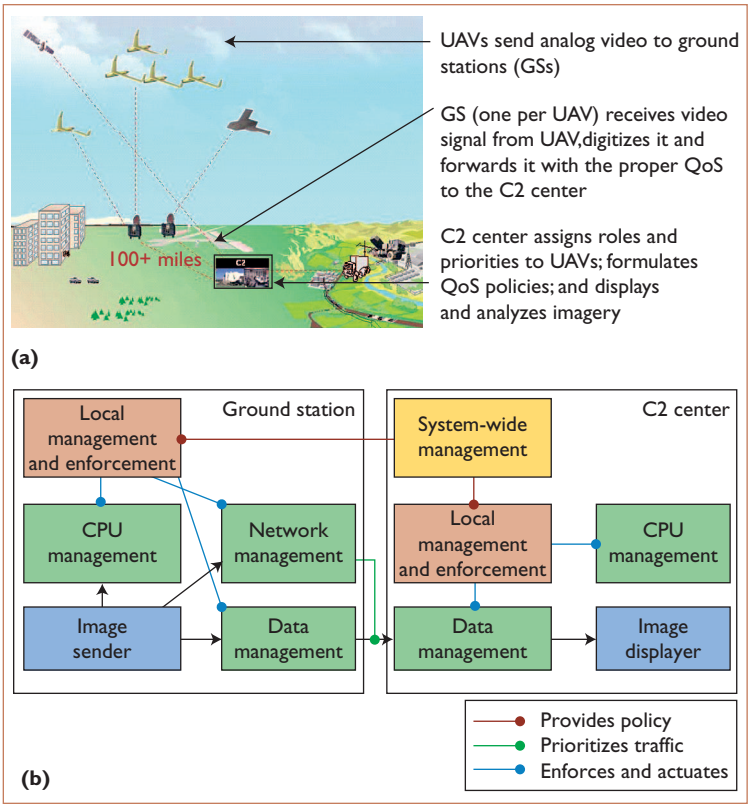

Figure 1. End-to-end QoS management in a live-flight and live-fire demonstration at the White Sands Missile Range. (a) Live unmanned air vehicles (UAVs) send images and receive command and control (C2) signals. (b) The system also includes a system-wide manager located at the C2 node providing QoS policy to local managers, which enforce QoS using resource-control and data-shaping mechanisms. Black arrows indicate the data flow.

strained network with limited CPU resources and are assigned different functional roles such as surveillance, target tracking (TT), and battle damage indication (BDI). During surveillance, UAVs send images of the surveilled area to help commanders determine items of interest. During TT, UAVs track targets and send images or streaming video that let a commander follow unfolding situations. During BDI, UAVs send images at regular intervals to let human operators analyze details. Each role has distinct QoS requirements for application data such as image rate, size, and resolution, and the system dynamically allocates resources based on the available resources and roles of participants. UAVs can change their roles as needed, for example, when a commander identifies a potential threat. The system assigns each role a priority with specific resource requirements — TT has the highest priority, followed by BDI, then surveillance.

As Figure 1a depicts, we used six UAVs (two real and four simulated) with sensors that sent images to the C2 center. 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, located more than 100 miles away and connected to the GSs via a shared fiber-optic network. A virtual LAN, using routers to control and distinguish traffic, provided 80 Mbytes per second (Mbytes/sec) network capacity, which the six UAV-to-C2 image streams, C2 traffic, and other demonstration-specific traffic shared. This system illustrates several challenges:

- managing end-to-end QoS dictated by the user, provided by the application, and delivered by the infrastructure within the specified time constraints;
- dynamically adapting QoS based on changing application requirements, operating conditions, and available resources; and
- cross-layer mapping of the application's QoS requirements (from higher mission-layer concepts such as required fidelity to lower resource-layer concepts such as resource availability and specific mechanisms) to provisioning resources and QoS control.

As Figure 1b illustrates, the system is distributed with dynamically changing participants, resources, and mission goals. The system needs QoS management software to oversee various functions. It must

- allocate resources among the participants and provide system-wide management of all participants, available resources, and QoS and application requirements. The software should be based at the C2 center where the policies are formulated. System-wide managers fulfill this.
- locally manage and enforce the policies by adapting real-time allocations. It needs to be associated with each asset, and it runs on each GS associated with a UAV. Local managers fulfill this function.
- provide local mechanisms for controlling system resources and shaping data streams to meet resource constraints.

To avoid 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 assemble these QCs to create managed end-to-end aggregate behavior.

Qosket Encapsulation Framework

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 more detailed description is available elsewhere.[1]) A Qosket includes code that lets us

- monitor the state of a particular QoS property in the system, which often encompasses items such as available resources, resources used, and the satisfaction of a QoS policy;
- make decisions regarding what's needed to control and deliver QoS to mediate conflicting demands, gracefully degrade and adapt as conditions change, and meet applications' requirements within resource allocations; and
- actuate, provide, enforce, and control 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.

We demonstrated QoS provisioning using Qoskets in distributed object applications in a previous work.[2] This article concentrates on using Qoskets as components in a component-based software engineering model.

Qosket Components

QCs consist of Qosket code wrapped inside standards-compliant components that we can assemble and deploy using existing tools and infrastructures.[4] QCs expose interfaces, letting us integrate them 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 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 combined with other QCs.

DRE systems require various types of QoS management software. To address this, we developed and classified QCs into *managerial*, *enforcement*, and *mechanism* QCs, based on the QC's scope and role.

Managerial QCs have a system-level view of an application's functional and system components. They maintain specifications for QoS requirements and domain-specific application requirements that are translated by the managerial QCs into QoS policies. Relatively speaking, managerial QCs are decision makers, with only high-level monitoring and little or no actuation.

For our WSMR demonstration, we developed a *system resource manager* (SRM) managerial QC that was hosted at the C2 center. The SRM QC uses weighted utility functions to dynamically allocate system resources based on the number of system participants, relative priorities, and total available resources. It creates QoS policies that include the roles, priorities, and allowed quality ranges, and sends those policies and allocations to the enforcement QCs on each host.

Enforcement QCs locally manage a node or group of nodes. They receive policies from managerial QCs, translate them into actions, and enforce them. They also decide which resource and application controls (in the form of mechanism QCs, which we define in a moment) can best 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 mechanism QCs' control interfaces.

We developed a *local resource manager* (LRM) enforcement QC that manages resources and trade-offs on each host. It configures specific mechanisms to control resources, shape data, and alter the application behavior to satisfy any resource allocations and QoS policies the SRM QC sends it.

Mechanism QCs access system controls and interfaces to adapt to 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, for example, for selecting the best compression algorithm to match the traffic format, but they generally have less decision-making capability than enforcement or managerial QCs.

For the WSMR demonstration, we used the following mechanism QCs:

- network prioritization DiffServ QC, which prioritizes outgoing IP traffic;

- CPU reservation mechanism QC, which reserves a percentage of the CPU;
- encrypt and decrypt security QC, which encrypts and decrypts data;
- pacing and depacing QC, which minimizes network jitter; and
- application-adaptation or data-shaping QCs, including a compression QC (to compress images), cropping QC (to crop images by removing pixels from each side), and scaling QC (to scale images to either half or quarter size).

We then combine these QCs to provide QoS management.

Composing QCs

We assembled the WSMR demonstration system by combining the QCs we described in the last section with the application's functional components (for example, the image sender, image receiver, and processing components) through the component and QC interfaces. During the UAV demonstration, the SRM QC at the C2 center used the number of demonstration participants, priorities, and roles to create a resource allocation and policy for each participant whenever the mission, roles, or available resources changed. The SRM QC pushed the allocation and policy to all LRM QCs, each of which then selected, configured, and activated the correct set of mechanism QCs to satisfy the policy within the allocated resources.

There was an LRM QC at the UAV GS and one at the C2 center for each information stream. Each LRM consisted of a set of mechanism QCs that it configured and activated to satisfy the policy with the allocation the SRM provided.[8]

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

- *Hierarchical composition* (see Figure 2a). In this pattern, managerial QCs are layered on enforcement QCs, which in turn are layered on 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* (see Figure 2b). When QCs receive data simultaneously and perform their QoS behavior independently, they can run 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 (for example, at the data's GS source and the C2 destination) in parallel because 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* (see Figure 2c). Often, we must tightly integrate a set of QCs to ensure that a set of QoS behaviors operates 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 data rate, altering (by cropping, scaling, or compressing) the data, and finally pacing it — that is, sending it in pieces over time to control jitter.

While integrating the QCs, we came across several issues that can impact QoS management.

Factors Affecting QC Composition

A complete treatment of all the issues related to the formalisms, tools, and analyses that can guide effective composition is beyond this article's scope. However, we can briefly discuss several important issues. For example, our demonstration system showed that the composition order of some QCs is important to the feasibility or effectiveness of end-to-end QoS management. We crop before compressing files because the cropping QC can only process noncompressed image formats. As another example, a QC adjusting the rate of imagery should be executed before other data-shaping QCs to avoid wasting time and resources by compressing or cropping an image that won't be sent.

Some QCs have to combine their behavior with corresponding "undo" behaviors. In our demonstration, for example, each compression QC on the source side requires a corresponding decompression QC on the receiver side. The "undo" QCs usually must execute in the reverse order of their paired QCs.

There are dependencies between QCs that assist with system integration and those that restrict the circumstances in which we can usefully combine them. For example, in QC interfaces, the cropping QC works 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 instance, some encryption and compression algorithms don't integrate well because encrypting might restrict our

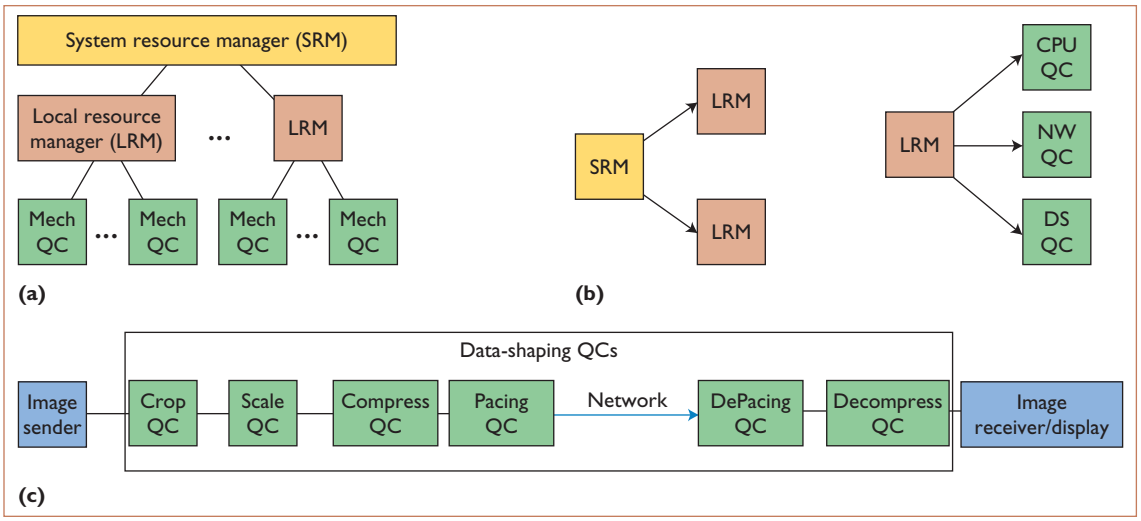

Figure 2. Composition patterns. We used these to construct the demonstration system from managerial, enforcement, and mechanism Qosket components (QCs). (a) In the hierarchical composition, each layer manages a set of QCs below it, pushing policy and control down and receiving status up. (b) In the parallel composition, a set of QCs must be invoked in parallel. (c) In the sequential composition, a set of QCs form a chain of combined QoS behavior.

ability to compress data, and compression might produce data that we can't properly encrypt. We're still investigating ways to incorporate this type of information so that automated tools can verify appropriate integration requirements.

The host boundaries on which we deploy QCs can play a crucial role in the aggregate QoS management's effectiveness. Placing data-shaping QCs closest to the data source makes the most sense unless multiple clients are using the data and demanding different qualities. Some QCs are only effective when separated by host boundaries. Compression can only reduce network traffic, for example, 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 QC. A peer-to-peer or ad hoc system, however, might need a different number and placement of managerial QCs. We're still experimenting with how to express the relationships between QC placement and the outcome of the end-to-end compositions.

Benefits

There are numerous benefits associated with building a system via embedded QCs instead of a custom stove-piped system — that is, a legacy system, consisting of tightly bound interrelated elements, that must be maintained until it can be entirely replaced with a new system. For example, many of the QCs we used in our UAV demonstration were reused or easily adapted from earlier contexts and remain part of our QC library. Part of our ongoing work involves exploring the trade-offs associated with decoupling a QC from functional interfaces (thereby increasing its reusability in different contexts) but increasing the work associated with using it in a specific context.

Using the embedded QC approach, providing QoS management in a DRE system ultimately becomes more of a configuration issue than a programming exercise. Hence, we can assemble the components required for QoS management into an existing or developing component-based distributed application. In our demonstration system, this lets us 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.

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:

- At configuration time, we can set QC attributes' default values. For example, we can set the attributes of an LRM QC to define the default strategy for selecting and activating a QC.

- At assembly time, we can combine 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.

Beyond these benefits, we still encountered some challenges while integrating QCs.

Challenges

Although we've had success in developing and combining QCs to create DRE systems, such as the UAV demonstration system, we still have many issues left to investigate on the path to common practice and operational use. From our recent experience with putting these ideas into practice, we can identify a few of the most important issues.

Data-specific QCs. There is a trade-off to be made in developing a QC that's specific to a particular data format. A more generic QC should be more widely reusable, but this might not always be feasible. For example, an attempt to develop a format-neutral compression QC leads to the following pitfalls:

- 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.
- Including various algorithms that work with many different formats might create an unwieldy QC that's too heavyweight for any specific context.
- We could use format-neutral compression algorithms, such as gzip, but data-specific algorithms are more useful in many cases. JPEG compression, for example, is more useful for imagery because it compresses efficiently and comes with display software. Over time, emerging standards are likely to help alleviate some of these issues, if for no other reason than to reduce the number of acceptable choices.

In much of our work, we made QCs as format-neutral as possible, even while continuing to work on additional solutions, such as QC interface templates. This can lead to a data incompatibility problem in which data emitted from one set of QCs might not be compatible as input to another set of QCs. Currently, we have no way of specifying this or annotating 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 doesn't propagate to application code because each QC that alters the output data is paired with a QC that undoes the alteration, as we described earlier.

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. Some solutions are to use only more controlled subsets, emphasize traffic-shaping techniques instead, or use a reactive approach that adjusts to the provided QoS, even in a best-effort environment such as the Internet.

Maintaining QoS. It's challenging to provide end-to-end QoS dynamically when the QCs need to interoperate with other middleware services such as the Joint Battlespace Infosphere (www.rl.af.mil/programs/jbi/), a publish–subscribe-oriented service, or the Corba Notification Service (www.omg.org/technology/documents/formal/notification_service.htm). Although these middleware services are individually compliant with standards, no uniform protocol exists for communication among them or for maintaining QoS using these services.

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. At the same time, we're promoting and helping develop QoS management awareness and capabilities for these other common middleware services.

The Qosket and QC work we've described is ongoing, and this article only discusses a portion of our research. Specifically, our work devel-

oping QoS engineering models and model-driven tools is still in its early stages.[9] In the future, we plan to offer better support for encapsulation, reuse, and composition with the goal of producing better software engineering support for dynamic, end-to-end QoS management capability and developing a generally useful model for and theory of integrating QoS behaviors.

Acknowledgments

This work was supported by Darpa (which has approved it for public release, distribution unlimited) and the US Air Force Research Laboratory under contracts F33615-00-C-1694 and F33615-03-C-3317.

References

1. R.E. Schantz et al., "Packaging Quality of Service Control Behaviors for Reuse," *Proc. 5th IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC), IEEE CS Press, 2002, pp. 375–385.
2. J.P. Loyall et al., "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," *Proc. 21st IEEE Int'l Conf. Distributed Computing Systems* (ICDCS-21), IEEE CS Press, 2001, pp. 625–634.
3. G.T. Heineman, J.P. Loyall, and R.E. Schantz, "Component Technology and QoS Management," *Proc. Int'l Symp. Component-Based Software Engineering* (CBSE7), Springer, 2004, pp. 249–263.
4. P.K. Sharma et al., "Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems," *Proc. Int'l Symp. Distributed Objects and Applications* (DOA), SpringerBerlin/Heidelberg, 2004, pp. 1208–1224.
5. *Corba Component Model,* v. 3.0, formal specification, Object Management Group; www.omg.org/technology/documents/formal/components.htm.
6. W. Roll, "Towards Model-Based and CCM-Based Applications for Real-Time Systems," *Proc. 6th IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC), IEEE CS Press, 2003, pp. 75–82.
7. J.P. Loyall et al., "A Distributed Real-Time Embedded Application for Surveillance, Detection, and Tracking of Time Critical Targets," *Proc. Real-Time and Embedded Technology and Applications Symp.* (RTAS), IEEE CS Press, 2005, pp. 88–97.
8. P. Manghwani et al., "End-to-End Quality of Service Management for Distributed Real-Time Embedded Applications," *Proc. 13th Int'l Workshop on Parallel and Distributed Real-Time Systems* (WPDRTS 2005), IEEE CS Press, 2005, p. 138a.
9. J. Ye et al., "A Model-Based Approach to Designing QoS Adaptive Applications," *Proc. 25th IEEE Int'l Real-Time Systems Symp.*, IEEE CS Press, 2004, pp. 221–230.

Praveen Kaushik Sharma is a staff scientist in the Intelligent Distributed Computing Department at BBN Technologies. Her research interests focus on provisioning dynamic and real-time QoS in distributed systems using software engineering technologies. Sharma has an MS in computer science from Iowa State University. She is a member of the IEEE and the ACM. Contact her at psharma@bbn.com.

Joseph Loyall is a division scientist and lead of the Distributed Systems Technology Group in the Intelligent Distributed Computing Department at BBN Technologies. He has been the PI for several projects dealing with QoS management, distributed real-time embedded systems, and adaptive middleware. Loyall has a PhD in computer science from the University of Illinois. He is a senior member of the IEEE and a member of the ACM and the AIAA. Contact him at jloyall@bbn.com.

Richard E. Schantz is a principal scientist at BBN Technologies, where he leads research efforts toward developing and demonstrating the effectiveness of middleware support for adaptively managing real-time, end-to-end QoS and system survivability. Schantz has a PhD in computer science from the State University of New York at Stony Brook. He is a fellow of the ACM. Contact him at schantz@bbn.com.

Jianming Ye is a staff scientist in the Intelligent Distributed Computing Department at BBN Technologies. His research interests include dynamic resource management and runtime environments for distributed real-time embedded systems, QoS-aware middleware development, and model-integrated computing. Ye has an MS in computer science from the University of Rhode Island. He is a member of the ACM. Contact him at jye@bbn.com.

Prakash Manghwani is a scientist in the Intelligent Distributed Computing Department at BBN Technologies. His research interests include large-scale, distributed real-time embedded systems and sensor networks. Manghwani has a BE in computer science from the University of Pune. He is a member of the ACM. Contact him at pmanghwa@bbn.com.

Matthew Gillen is a staff software engineer at BBN Technologies. His research interests include distributed systems and fault tolerance. Gillen has a BS in computer science from Ohio University. Contact him at mgillen@bbn.com.

George T. Heineman is an associate professor of computer science at Worcester Polytechnic Institute. He has a PhD in computer science from Columbia University. Heineman is the coeditor of *Component-Based Software Engineering: Putting the Pieces Together* (Addison-Wesley). Contact him at heineman@cs.wpi.edu.