End-to-End Quality of Service Management for Distributed Real-time Embedded Applications

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

Abstract

Many of the world's most critical systems are distributed real-time embedded (DRE) systems, with mission-critical quality of service (QoS) requirements. However, because of their nature – heterogeneous nodes and links, shared and constrained resources, and deployment in dynamic environments – providing QoS requires coordinated QoS management throughout the system of multiple end-to-end application streams competing for shared resources. It requires dynamic resource allocation to these end-to-end application streams based on potentially changing mission requirements and shaping application behaviors to effectively use the resources that are allocated. In this paper, we describe the issues involved with providing end-to-end QoS management in DRE systems, an architecture we have designed to support system-wide end-to-end QoS management, and a multi-UAV surveillance and target tracking application we are using to evaluate these technologies.

1. Introduction

In many of today's computer applications, quality of the service provided is as important as functionality, i.e., *how well* an application performs its function is as important as *what* it does. Many of these applications are embedded systems that control physical, chemical, biological, or defense processes and devices in real-time. Increasingly, these embedded systems are part of larger *distributed real-time embedded (DRE)* systems, such as military combat or command and control systems, manufacturing plant process systems, emergency response systems, and telecommunications. DRE systems consist of

- Multiple competing end-to-end streams of processing and information;
- Changing numbers and types of participants, with changing roles and relative importances;
- Heterogeneous, shared, and constrained resources.

Quality of Service (QoS) management is a key element of the design and runtime behavior of DRE systems, but it is often defined in terms of management of individual resources, e.g., the *admission control* provided by network management or CPU scheduling mechanisms or services. While individual resource management is necessary, it is not sufficient in DRE systems because:

- There might be multiple, simultaneous *bottlenecks* (i.e., the most constrained resources) and the bottlenecks might change over time;
- Effective QoS management spans individual resources. That is, the consumer of information determines the QoS requirements, which might change over time, while the information source (which might be remote from the consumer and therefore using different resources) and transport determine the quality and form of information.

QoS management for DRE systems must therefore capture the QoS requirements from the mission requirements, manage all the resources that could be bottlenecks, mediate conflicting demands for resources, effectively utilize allocated resources, and dynamically reallocate as conditions change.

Under the DARPA Program Composition for Embedded Systems (PCES) program, we have been developing technologies to specify and enforce end-to-end QoS in DRE systems. We have been evaluating this end-to-end QoS management technology in the development of a medium-scale, real-world capstone demonstration for the PCES program, a multi-UAV surveillance and target tracking application. The PCES capstone demonstration, illustrated in Figure 1, involves coordinating multiple *unmanned aerial vehicles (UAVs)* performing surveillance, target tracking, and battle damage indication; a *command and control (C2)* node providing theater battle management; and a US Army ground-based weapon system [8].

In this paper, we describe the issues in, and our approach to providing, end-to-end QoS management. The main contributions of the paper are

- An architecture for providing end-to-end QoS management in DRE systems and an approach to realizing this architecture by composition of QoS components.
- An evaluation of the architecture and realization in a representative real-world application, the PCES multi-UAV surveillance and target tracking application.

2. Issues in Providing End-to-end QoS Management

Providing end-to-end QoS requires simultaneously managing resources to effectively allocate them among competing users and shaping application resource and data usage to effectively utilize allocated resources throughout the system. Which resources need to be managed; at what points in time; and what constitutes effective usage depends on the QoS and mission requirements and the resource availability at any instant in time during the system's operation. In other words, effective QoS management relies on recognizing where bottlenecks exist at any given point in time and effectively managing resources to remove the bottlenecks; adapting application functionality to compensate for the bottlenecks and meet system requirements; or both.

Designing and implementing end-to-end QoS management into a system therefore requires building in the control and flexibility to manage changes in resource availability and mission requirements. Such a system must include the following capabilities:

- Coordinated monitoring, control, and management of all resources that are potential bottlenecks during system execution.
- Capturing mission requirements and translating them into policies for effective resource usage.
- Dynamically adapting resource allocations and application behavior.

Manageable resources include network bandwidth, CPU, power, memory, and other, not so obvious, resources such as screen real estate. In many cases, well defined system context can constrain the set of resources that must be managed to ensure end-to-end QoS. For example, in the PCES capstone demonstration context illustrated in Figure 1, we are initially concentrating on network bandwidth and CPU resources, as these are the most likely bottlenecks. Expanding the context, however, can also extend the set of resources that must be considered. For example, if the system context includes sensor nodes, motes, or smaller UAV vehicles, then power and memory will need to be considered. If the scenario includes dismounted soldiers or rescue personnel with handheld devices, then
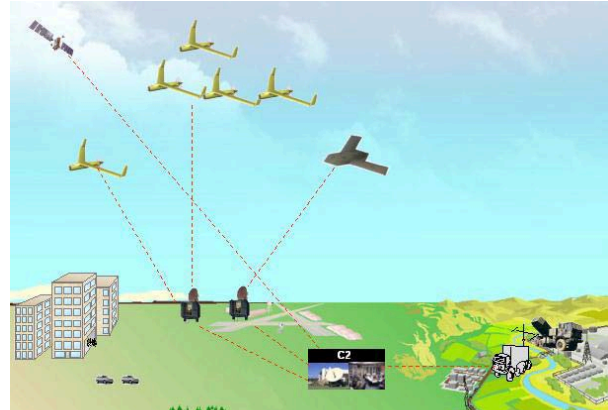

Figure 1. The PCES capstone demonstration concept of operations includes multiple UAVs, combat vehicles, and command centers

screen real estate becomes a potential additional bottleneck. Finally, if the scenario expands to include many more reconnaissance UAVs under control of a single or a few command center personnel, then human perception (i.e., the number of displays a person can simultaneously observe and process) becomes a potential bottleneck.

While managing the resources at a bottleneck might be sufficient at any given time, it is important to understand the consequences of managing the resources only at a specific point. Eliminating a bottleneck by providing additional resources might simply expose a different bottleneck elsewhere that must also be managed. For example, allocating more bandwidth (e.g., using bandwidth reservation, differentiated services, alternate paths, or reducing the other load on the network) might simply expose that there isn't enough available CPU at a node along the end-to-end path to process the now plentiful data.

Furthermore, shaping application or data usage to fit a bottleneck can also have consequences that change, but do not eliminate, the bottleneck. For example, an application facing a constrained network can use data compression to consume less bandwidth. However, in doing so the application is consuming more CPU, which might or might not be available.

So, while managing individual resources is a *necessary* part of end-to-end QoS management, it is not *sufficient*. We take an approach of providing end-to-end QoS management by combining the following:

- Coordinated and dynamic management of all the important resources along the end-to-end path (i.e., those resources that are, or could become, bottlenecks)
- Cooperative allocation of resources among multiple end-to-end streams based upon their participation in system-wide mission requirements
- Adaptive shaping of application behavior and resource usage based upon resource allocation and the mission requirements.

This leads to the following three views of resource management:

Resource view. Resource specific mechanisms exist to control access to a resource. They decide whether and how a request for a resource allocation should be granted. Typical CPU and network resource allocation is based on priorities or on reservations. Resource allocation mechanisms have little or no knowledge of the applications using them or their requirements, although some limited information can be propagated to the resource level in the form of relative priorities, reservation requests, etc. The mechanisms, or resource-specific managers built upon them (such as bandwidth brokers [3] or CPU brokers [5]) define the interfaces for establishing, changing, and accessing the policies attached to each resource.

Application view. An application view of resource management involves acquiring whatever resources are needed to meet the applications' requirements and to effectively utilize whatever resources are available to them. If there are not enough resources available, then an application needs to be flexible enough to adjust its resource needs (with corresponding graceful degradation of its functionality) or it will fail, e.g., throw an exception. Applications greedily acquiring all the resources they need does not scale in dynamic and severely resource constrained environments, but applications cooperating to share resources requires sophisticated coordination and control. Unless this is done carefully, this can lead to static, brittle, or inefficient systems.

System view. At the system level, there is the knowledge of the mission goals, the applications in the system, and the available resources. This is also the level at which there is an understanding of the relative importance of applications to mission goals, resource allocation strategies for each goal, and the policies for mediating conflicting application resource needs. This knowledge can be located at a central point, such as in a knowledge base or at a command and control node, distributed throughout the system, or combinations of these.

It is necessary to combine the system, application and resource views to effectively manage and provision end-to-end QoS under varying conditions and changing requirements. The system view determines effective allocations; applications choose how to effectively use their allocations; and resource mechanisms manage access to shared and constrained resources.

3. An Architecture for System-Wide End-to-end QoS Management

We have developed an architecture suitable for end-to-end QoS management in DRE systems. It is a multi-layer
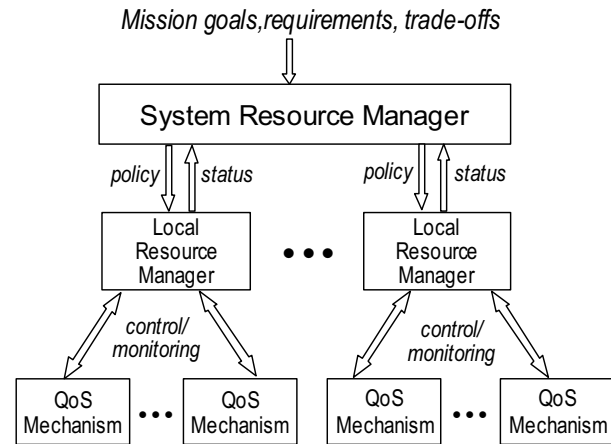

Figure 2. High-level view of multi-layer resource management.

architecture with layers corresponding to the three views, illustrated in a high-level view in Figure 2:

- *System layer* – Captures the system-wide mission requirements and establishes the policies for system-wide resource allocation and cooperation among participants (i.e., subsystems, nodes, or end-to-end streams) in the system.
- *Local layer* – Translates the policy into appropriate actions for individual end-to-end streams or participants in the system.
- *Mechanism layer* – Controls individual resources or invokes QoS mechanisms.

This multi-layer approach is similar to the multi-layer architecture we are using in other research efforts [2]. The differences are in the specific instantiation of the elements and the unique approach we take in PCES to encapsulate functional and QoS elements as components and create the system by composition of these components, described later in this section.

3.1 Elements of the Multi-Layer QoS Management

Figure 3 shows more details about the elements of QoS management in our architecture, described in the following paragraphs.

The *System Resource Manager* (SRM) is a *supervisory controller* responsible for allocating resources among the system participants. It is also responsible for disseminating system and mission wide policies to the local resource managers. These policies include the resource allocation, the relevant mission requirements and parameters, and tradeoffs.

The SRM needs a model of the shared resources, the available resources at any given time, the active participants in the system, and the mission requirements information in order to effectively allocate the resources among, and determine the policy for, the participants. This information is stored in a *System Repository*, which is populated with an initial model of the system and updated with status information gathered at runtime.

A *System Participant* is a part of a distributed system providing a service or performing a certain function. This can be an individual application, an end-to-end application stream, or a subsystem. Associated with each system participant is a local QoS management layer.

The *Local Resource Manager* (LRM) receives the policy from the system resource manager and translates it into local management and control actions. Its primary responsibility is to manage the local resources consumed by an application and to ensure that the correct behavior is chosen for the mission requirements, local information, and allocated resources. The LRM is associated with one or more collocated participants, such as a single application, all the applications on a single host, or all the applications of a single stream. The LRM decides, based on local information, the best way to use the resources allocated to it so that it satisfies the requirements and constraints passed to it as policy from the SRM. As such, the LRM can provide fine-grained control, rapid response, and adaptation within those constraints.

The LRM is a *feedback controller*, using the mission requirements, tradeoff information, and allocated resources part of the policy provided to it to determine which QoS behaviors (e.g., CPU management, network management, data shaping, application adaptation) should be employed, in what order, and to what degree. The LRM also monitors the actual behaviors and adjusts as needed to maintain the QoS level.

In order to determine which QoS behaviors to employ, the LRM needs to predict the effect of employing each QoS behavior and combination of QoS behaviors. In Figure 3, we separately indicate the control and prediction parts of the LRM, the former illustrated as a *Controller* and the latter as a *QoS Predictor*. The effects of some QoS behaviors can be determined analytically, e.g., the results of cropping an image (i.e., the amount of data in the resulting image) or reserving an amount of bandwidth (i.e., the amount of bandwidth available to the application). Other behaviors have no analytical model (or less accurate ones), e.g., some compression algorithms or setting a network priority (the results of which are difficult to determine analytically without global knowledge of many other external factors). With the former, the QoS predictor con-
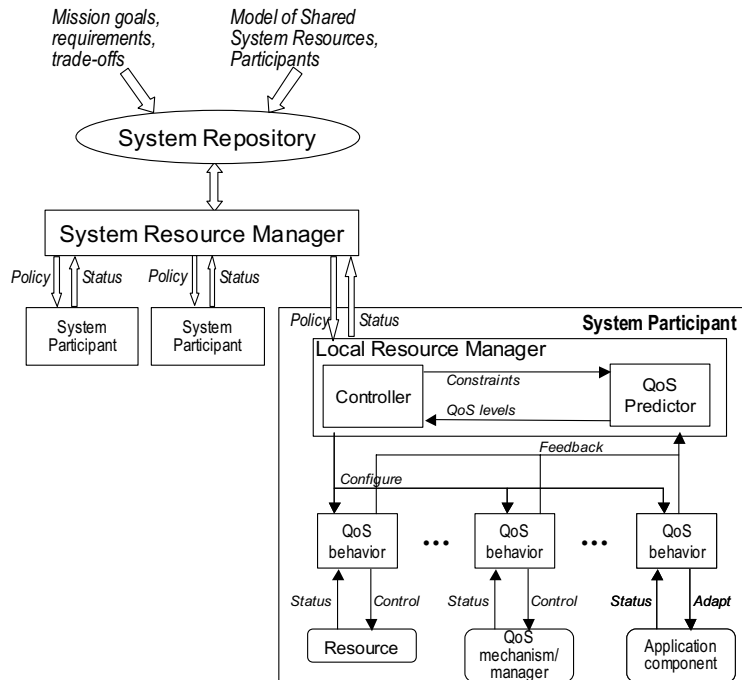

Figure 3. Elements of Multi-Layer QoS Management

tains the model, equation, or formula to predict the behavior. With the latter, the QoS predictor is initialized with experimental data produced in test runs, and updated at runtime with more accurate monitored information.

The QoS mechanism layer consists of encapsulated *QoS behaviors* that control and monitor the following:

- *Resources*, such as memory, power, or CPU, which can be monitored and controlled through knobs exposed by the resource.
- Specific *QoS mechanisms*, such as network reservation or network priority services that expose interfaces to resource monitoring and control; or *QoS managers*, such as bandwidth brokers [3] or CPU brokers [5], that provide higher level management abstractions.
- *Application or data adaptation*, such as changing the rate of tasks, algorithms or parameters of functional routines, or shaping the data used or produced by application components.

There is a layered, recursive pattern of resource management here, where each layer receives policy from the layer above it and status from the layer beneath it, uses the status and policy to make decisions, and then produces policy or control information for the layer beneath it. In the case of this paper, we define and instantiate three layers and illustrate them in a geographically distributed system of embedded participants with a centralized authority, i.e., a battlefield system with distributed UAVs and a command and control center.

3.2 Constructing End-to-End QoS Management Using Qoskets and Qosket Components

In this section, we describe an approach to instantiating the end-to-end QoS management capability we described in the previous section by composing it from reusable QoS management components.

In previous papers, we have discussed how QoS management cross-cuts the functionality of an application [4] and how we encapsulate related code pertaining to a QoS behavior into a reusable package of code, called a *Qosket* [13]. The code encapsulated in a Qosket includes the following:

- Code to *monitor* or *measure* levels of QoS or current conditions in the system.
- Code to *control* or *influence* QoS through QoS interfaces, knobs, mechanisms, or managers.
- Code to *decide* the appropriate control, adaptation, or reaction to invoke.
- Additional code, such as functional routines, helper functions, or library code, useful for QoS management.

Qoskets can be instantiated as sets of *qosket components* [14] that can be assembled with the functional components of an application to integrate QoS management into a component-based distributed system. In many cases, an individual Qosket is realized as many distributed qosket components that perform the monitoring, decision making, and control necessary at the right points in the application.

Our approach is to instantiate the elements of our end-to-end QoS management architecture as qosket components so that they can be assembled with the functional application, as illustrated in Figure 4. The SRM qosket component includes decision making code to decide how resources should be allocated among participants and wrap that allocation into policy, with some monitoring code to determine the number of current participants, the amount and type of shared resources, and other information affecting the policy decision, such as mission states, requirements, and conditions.

The LRM qosket components include decision making code to decide local actions based on the policy, monitoring code to measure the effects of the QoS management, and control code to adjust levels to satisfy the policy. The LRM's control code is typically limited to setting the proper attributes on the QoS behavior qosket components and invoking them in the proper order.

The assembly also includes as many QoS behavior qosket components as necessary. In the example in Figure 4, we illustrate two types of QoS behavior qosket components, one that does data shaping and another that interfaces to a QoS mechanism. These QoS behavior qosket components include mostly control and algorithmic code because their decisions are made by the LRM.

4. An Example DRE System with End-to-End QoS Management

As part of the DARPA PCES program, we have been developing a live flight capstone demonstration based on a scenario of multiple UAVs performing surveillance and target tracking, described in more detail in [8]. The Multi-UAV surveillance and target tracking application that we are constructing for the PCES program includes several UAVs delivering imagery to, and receiving commands from, a command and control center and ground stations. Each UAV is the source of an end-to-end information stream, sending imagery and other information to the C2 node, and performing one of the following roles, each with different mission requirements:

- *Surveillance* – In this role, the UAV is performing surveillance of a designated area. The imagery coming from a surveillance UAV must have sufficient resolution and scan size (i.e., the area covered by the camera) and must be delivered at a rate sufficient to avoid gaps in surveillance.
- *Tracking* – Imagery sent by a UAV in this role is closely examined to track or discern additional details about the area of interest (AOI). It is therefore relatively more important than imagery coming from surveillance UAVs and needs to be of the highest possible resolution and possibly a higher rate, but does not need to cover as large a scan size as long as the images contain the full AOI.
- *Battle damage indication (BDI)* – This role is required after an action has occurred, such as engagement of a target, to provide imagery for review of the effectiveness of the action. High quality imagery is a must, but the rate can be slower relative to the other roles.

The base functionality of each end-to-end imagery stream consists of image capture (i.e., the UAV's camera sensor and associated processing) and image sending (i.e., communicating the imagery off-board) on the UAV; and the image receipt, display, and processing on the C2 node. The image generation rate is a configurable parameter that indicates how often an image should be pushed out, which
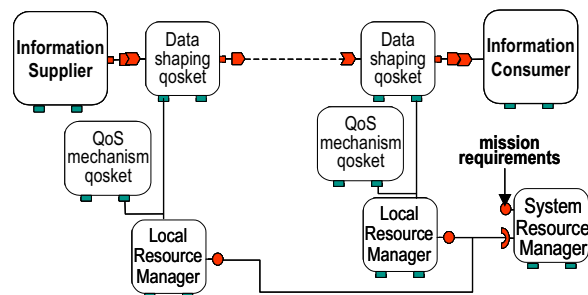

Figure 4. End-to-end QoS management elements are instantiated as qosket components and assembled with the functional components.

is determined by the usage requirements of the imagery, and can be different than the rate at which it is collected. We use both CIAO [15], an implementation of the CORBA Component Model, and PRiSm [11], an avionics specific component model, in the demonstration application.

The full scope of the demonstration system includes a combination of live flight vehicles, ground vehicles, and simulated participants, and is described in [8].

4.2 System Layer Qosket Components

We augment the functional stream with qosket components as described in Section 3 to get end-to-end QoS management. The full assembly for a representative imagery stream is illustrated in Figure 5. There is one SRM component, which we locate at the C2 node, so it is near the receivers and the command authority, both of which provide information needed to determine the mission requirements. In PCES, we use a demonstration driver, which performs theater-wide situation assessment and keeps track of the number and role of participants. This serves as the system repository and when something in the system state changes, such as the number or role of participants, the demonstration driver notifies the SRM component. The SRM uses the relative weights of the roles, the importance of each UAV within a role, the number of UAVs, and the amount of resources available, in order to compute a resource allocation for each UAV. It creates a policy structure for each participant consisting of the following:

- The UAV's role (surveillance, target tracking, or BDI)
- The UAV's importance relative to others in the role

- Allocations of resources (bandwidth, CPU, network priority)
- Minimum and maximum allowable qualities (frame rate, cropping, scale, compression, and CPU reservation)

This policy event is pushed to each of the LRMs.

4.3 Local Layer Qosket Components

There is an LRM component associated with the sender assembly and another one associated with the receiver assembly. Each receives the policy sent by the SRM and updates the relevant QoS Predictors with the minimum and maximum cropping, scaling and compression levels. The LRM then queries the QoS Predictors to get the proper levels to set for each of the data shaping components to fit the allocated bandwidth and CPU. The adaptation strategy and tradeoffs for each role is captured in a model of the system [9] and is used to determine the order of assembly and calling of the QoS predictors and the data shaping qosket components. For example, the strategy we use for the surveillance role is to reduce the rate until the minimum (the minimum is the slowest rate that does not cause gaps in surveillance); compress until the maximum allowed compression; and scale the image as a last resort if needed.

The LRM then sets each of the QoS mechanism qosket components with the proper settings from the policy and QoS predictors.

4.4 Mechanism Layer Qosket Components

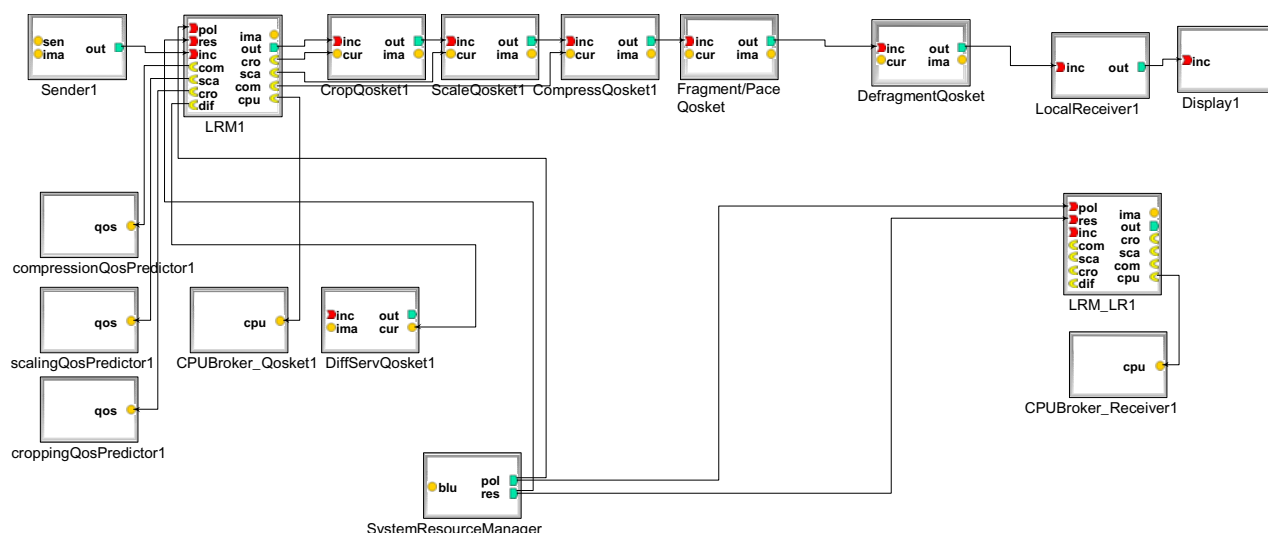The PCES application includes three of the types of

Figure 5. Full assembly for one end-to-end image delivery stream in the PCES Multi-UAV Surveillance and Target Tracking Demonstration

QoS behavior qosket components listed in Section 3.1, QoS mechanism, QoS manager, and data adaptation qosket components.

The *Diffserv qosket component* is a *QoS mechanism qosket component* responsible for setting DiffServ code-points (DSCPs) on component containers. The LRM uses the network priority from the SRM policy to configure the Diffserv component and the container and ORB ensure that all packets going out have their DSCP correctly set. Routers configured to support Diffserv ensure that the packets get queued according to their DSCP priorities.

The *CPU Broker qosket component* is a *QoS manager qosket component* responsible for reserving CPU cycles over a period of time for a component container. The LRM uses the minimum and maximum CPU reservation and the relative importance from the SRM policy to configure the CPU Broker component. The underlying CPU mechanisms (CPU Broker [5] and TimeSys Linux) guarantee that the container gets the minimum CPU cycles it needs. In the case of CPU contention, no more than the maximum CPU cycles are allocated to the container. Note that at any given point in time, a container might be consuming less than the minimum CPU, if the resources aren't needed, or more than the maximum CPU, if more than the maximum resources are available.

Data Shaping Qosket Components. Once the available CPU and network resources have been allocated across UAV streams, each stream must shape its data to use the allocated resources effectively. We assemble several data shaping qoskets that the LRM uses to accomplish this.

Fragmentation, Pacing, and Defragment qosket components are combined to reduce jitter in the network by spreading the transmission of data evenly over the interval specified by its rate. The LRM configures them with the allocated bandwidth and a fragment size (the maximum transmission unit of the network is a logical choice). The fragmentation component breaks an incoming image into fixed sized fragments and the pacing component sends the fragments over the network at regular intervals. Fragmentation on the sender side, of course, must be accompanied by assembly (or *defragmenting*) on the receiver side. The defragment component receives fragments and, once it has received all the fragments of an image, reconstructs the image and pushes it out.

The *Compress qosket component* is responsible for compressing an image. The level of compression is set by the LRM as specified by the QoS Predictor. In the current prototype, we have three levels of compression: no compression (PPM format), lossless compression (PNG format), and lossy compression (JPEG format).

The *Crop qosket component* removes a specified amount of the image from a set place in the image. The amount that the image is cropped is set by the LRM as specified by the QoS Predictor. In the current prototype,

we crop from the center of the image. However, with additional image processing or identification of the area of interest in the C2 command information, the image could be cropped around the AOI no matter where it is in the image. We have five levels of cropping in the current prototype. Our algorithm crops images uniformly around edges, e.g., Crop20Percent removes 5% of the pixels from each edge, reducing the image to 81% of its original size.

The *Scale qosket component* reduces the size of an image. The LRM sets the amount that the image is scaled as specified by the QoS Predictor. Our current prototype supports three scaling levels, NoScale (factor 0), Half-Scale (factor 2), and QuarterScale (factor 4).

4.5 Putting It All Together

Figure 5 illustrates a full logical assembly of one end-to-end QoS-managed UAV image stream. The Sender component generates a PPM image and sends it out. The LRM takes this image, examines the available resources, consults the QoS predictors to determine a suitable set of adaptations, updates the QoS levels on each qosket component, and pushes the image out. The processed image comes out from the Compress qosket component in the suitable size and format to fit the available resources. The Fragmentation and Pace qosket component splits the image into fragments and streams them out with the proper DSCP header, while making sure it is not consuming more bandwidth than it is allowed. On the other side of the network, the Defragment qosket component assembles the incoming data fragments into an image and sends it out. The LocalReceiver component receives the adapted image, decompresses it into a PPM image, and sends it out for display by the Display component.

The SRM, with knowledge of the number of end-to-end UAV image streams and the relative importance of each, divides the available shared resources between them and provides each LRM with a policy including the amount of resources it is allowed and the role it is playing in the full system. Assuming there are enough resources, the SRM ensures that every end-to-end image stream gets at least the minimum it needs and that the more important streams get the majority of the resources. In the cases where there are not enough resources for all the streams, the SRM ensures that the most important streams get the resources that are available. The LRMs make sure that the allocated resources for each end-to-end stream are used most effectively for the UAV's role in the system.

5. Related Work

Other research projects have tackled the issues of end-to-end QoS management. Many of these concentrate only on network QoS, where end-to-end means managing the

reservations or queues along network paths [1][12]. A few, however, have moved toward the ideals that we lay out in this paper, that end-to-end QoS means coordinated management of all resources that are shared, constrained, and can become the bottleneck at any given time with shaping of the application resource usage to the available resources. The BRENTA architecture [10] describes in different ways some of the concepts we are realizing in PCES. It concentrates on contract-based negotiation of network QoS, but with the addition that applications should adapt to the available resources, even while realizing that some legacy applications might not have that flexibility. QARMA [6] includes architectural elements similar to those we described, including a resource manager and system repository, provided as CORBA services. Li et al [7] propose a task control model approach in which they add a monitoring task and an adaptation task for each functional task in the system. The monitoring task recognizes QoS violations and the adaptation task adjusts application behavior to compensate.

6. Conclusions

We have presented a multi-layered architecture for dynamic end-to-end QoS management for distributed, real-time embedded systems. The architecture captures the mission requirements and system configuration, and distributes it as policy to local QoS management capabilities that control and enforce QoS management. The multi-level architecture is more scalable than centralized QoS management systems, more mission-aware than resource-centric management systems, and more appropriate for loosely coupled DRE systems such as the distributed multi-UAV application. Using our qosket component middleware gives us the flexibility to encapsulate and assemble the resource managers, monitors, and QoS mechanisms appropriately for the deployment environment, system design, and distribution.

Acknowledgements

References

[1] H. Bai, M. Atiquzzaman, W. Ivancic. "Achieving End-to-End QoS in the Next Generation Internet: Integrated Services Over Differentiated Service Networks," NASA/TM-2001-210755, March 2001.

[2] R. Campbell, R. Daley, B. Dasarathy, P. Lardieri, B. Orner, R. Schantz, R. Coleburn, L. Welch, P. Work. "Toward an Approach for Specification of QoS and Resource Informa-tion for Dynamic Resource Management," 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES), Toronto, Canada, May 25, 2004.

[3] B. Dasarathy, S. Gadgil, R. Vaidyanathan, K. Parmeswaran, B. Coan, M. Conarty, V. Bhanot. "Network QoS Assurance in a Multi-Layer Adaptive Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework," Real-time and Embedded Technology and Applications Symposium (RTAS), San Francisco, CA, March 2005.

[4] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, J. Zinky. "Building Adaptive Distributed Applications with Middleware and Aspects," Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, March 2004.

[5] E. Eide, T. Stack, J. Regehr, J. Lepreau. "Dynamic CPU Management for Real-Time, Middleware-Based Systems," 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Toronto, ON, May 2004.

[6] 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, Santa Fe, NM, April 2004.

[7] B. Li, D. Xu, K. Nahrstedt, J. Liu. "End-to-End QoS Support for Adaptive Applications Over the Internet," SPIE Proceedings on Internet Routing and Quality of Service, Boston, Massachusetts, November 1-6, 1998.

[8] J. Loyall, R. 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), San Francisco, CA, March 2005.

[9] J. Loyall, J. Ye, S. Neema, N. Mahadevan. "Model-Based Design of End-to-End Quality of Service in a Multi-UAV Surveillance and Target Tracking Application," 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES), Toronto, Canada, May 25, 2004.

[10] D. Mandato, A. Kassler, T. Valladares, G. Neureiter. "Handling End-To-End QoS in Mobile Heterogeneous Networking Environments," International Symposium on Personal, Indoor and Mobile Radio Communications, October 2001.

[11] W. Roll. "Towards Model-Based and CCM-Based Applications for Real-Time Systems," 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), Hokkaido, Japan, May 14-16, 2003.

[12] V. Sander, W. Adamson, I. Foster, A. Roy. "End-to-End Provision of Policy Information for Network QoS," 10th IEEE Symposium on High Performance Distributed Computing (HPDC), August 2001.

[13] R. Schantz, J. Loyall, M. Atighetchi, P. Pal. "Packaging Quality of Service Control Behaviors for Reuse," International Symposium on Object-Oriented Real-time distributed Computing (ISORC), Washington, DC, April 2002.

[14] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, G. Duzan. "Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems," Distributed Objects and Applications (DOA), Agia Napa, Cyprus, October 25-29, 2004.

[15] TAO and CIAO, http://www.cs.wustl.edu/~schmidt/TAO.html