

# Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems

Praveen K. Sharma, Joseph P. Loyall, George T. Heineman,  
Richard E. Schantz, Richard Shapiro, Gary Duzan

BBN Technologies  
Cambridge, MA  
{psharma, jloyall, gheinem, schantz, rshapiro, gduzan}@bbn.com

**Abstract.** Large scale distributed real time and embedded (DRE) applications are complex entities that are often composed of different subsystems and have stringent Quality of Service (QoS) requirements. These subsystems are often developed separately by different developers increasingly using commercial off-the shelf (COTS) middleware. Subsequently, these subsystems need to be integrated, configured to communicate with each other, and distributed. However, there is currently no standard way of supporting these requirements in existing COTS middleware. While recently emerging component-based middleware provides standardized support for packaging, assembling, and deploying, there is no standard way to provision QoS required by the DRE applications. We have previously introduced a QoS encapsulation model, *qoskets*, as part of our QuO middleware framework that can dynamically adapt to resource constraints. In this paper we introduce implementing these QoS behaviors as components that can be assembled with other application components. The task of ensuring QoS then becomes an assembly issue. To do so we have componentized our QuO technology instead of integrating QuO into the middleware as a service. To date, we have demonstrated our approach of QoS provisioning in MICO, CIAO, and Boeing's Prism component middleware. We present experimental results to evaluate the overhead incurred by these QoS provisioning components in the context of CIAO CCM. We use a simulated UAV application as an illustrative DRE application for the demonstration of QoS adaptations using qosket components.

## 1 Introduction

Next-generation distributed real-time embedded (DRE) applications are increasingly at the core of a wide range of domains, including telecommunications, medicine, avionics, command and control, and e-commerce. These complex applications often operate over

---

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. Approved for Public Release, Distribution Unlimited.

shared CPU and network resources and under dynamically varying workload and environment conditions. To be useful, these applications need to provide dynamically adaptive Quality of Service (QoS) attributes.

**Limitations of current approaches for developing DRE applications and managing QoS.** DRE systems are being built using off-the-shelf (COTS) middleware based on the CORBA object model, Java RMI, or Microsoft's COM+. However, COTS middleware currently lacks the support needed for standard ways of packaging, assembling and deploying these applications in a heterogeneous distributed environment. Additionally, COTS middleware also lacks the ability to separate the concerns of functional application behavior from QoS specification, enforcement, and adaptation. These limitations hinder the use of current COTS middleware for building DRE systems in the fashion that component middleware is now often being used to build other systems.

**Emerging component-based middleware.** Component-based middleware has begun to emerge as an extension to object-based middleware and is based on the core abstraction of *components*, reusable software that can be easily configured, assembled, and deployed [8, 19]. However, until recently component-based middleware offerings were mostly proprietary and non-standard.

The release of the CORBA 3.0 standard has introduced several new features – the most prominent and most relevant to this paper being the CORBA Component Model (CCM) [19]. CCM extends the CORBA 2.0 object model to define a component model and shifts the software development paradigm from coding to assembling and deploying reusable software components. The CCM specification provides a standard way of designing components, configuring the connections of these components and their default attributes at assembly time, packaging these components as distributable units, and deploying them over the network. These features make CCM highly suitable for developing DRE systems. There are several implementations of CCM currently available including *MICO* (Mico Is Corba) [17], *CIAO* (Component integrated ACE ORB) [3], *OpenCCM* [22], *K2-CCM* [10] and *EJCCM* (Enterprise Java CCM) [5]. Each implementation is in different stages of development and encompasses different strengths. There are proprietary component-based middleware implementations, such as Boeing's CCM-inspired *Prism* [26], that are also well suited for DRE applications.

**Recent work by other groups in integrating QoS with components.** The CCM specification lacks the notion of QoS and there are ongoing efforts in the Object Management Group (OMG) to standardize QoS support for CORBA components [20]. We have been working with the developers of CIAO to define static and dynamic QoS support for CCM within the CIAO framework. The Qedo (QoS enabled Distributed Objects) effort has been involved in providing QoS to components by integrating data streams (based on their Streams for CCM specification submitted to the OMG) [24]. Fault-tolerant CCM is yet another effort that focuses on providing fault-tolerance to CCM components [2].

**Our approach and summary of the organization of the paper.** In this paper, we describe our research in developing dynamic QoS adaptive support for CCM components. We build upon our previous work in developing QoS adaptive support for objects by encapsulating adaptive QoS behaviors as components. These adaptive QoS components,

called *qosket components*, can be developed separately from functional components, can be configured with the application components using CCM tools, and can adapt the behavior of the system at run-time. This sets the stage for providing QoS behavior to DRE applications by configuring and assembling qosket components. In a recent paper, we presented our design for enabling dynamic, adaptive QoS management using qosket components [9]. In this paper, we describe the first working instantiation of qosket components within three separate component-based middleware implementations, MICO, CIAO, and Prism; for space reasons, we focus on our results for CIAO.

This paper is organized as follows: Section 2 describes our preliminary work that sets the stage for this paper and introduces the distributed UAV image dissemination application that we use to demonstrate and evaluate our approach. Section 3 describes how we encapsulate adaptive behaviors as qosket components. Section 4 presents experimental results of the performance tradeoffs of using qosket components in the UAV application domain. We present future directions in Section 5 and conclusions in Section 6.

## 2 Background

The work described in this paper relies on two major antecedents: the Quality Objects (QuO) middleware framework, developed by BBN, and the CORBA Component Model (CCM). We also depend upon an unmanned aerial vehicle (UAV) image dissemination simulation as an illustrative DRE application. In Section 3 we describe how we componentized this UAV application and carried out experiments to measure average latencies to determine the overhead incurred by qosket components, using the CCMPerf [12] benchmark.

### 2.1 Overview of QuO

The Quality Objects (QuO) framework [1, 11, 14, 16, 28, 29, 32] is a QoS adaptive layer that runs on existing middleware such as CORBA and Java RMI and supports the specification and implementation of (1) QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time. To achieve these goals, QuO provides middleware-centric abstractions and policies for developing distributed applications. Key elements provided by QuO, illustrated in Fig. 1 for CORBA, include:

- **Contracts** – The operating regions and service requirements of the application are encoded in contracts, which describe the possible states the system might be in, as well as the actions to perform when the state changes.
- **Delegates** – Delegates are proxies inserted transparently into the path of object interactions, but with woven-in, QoS-aware adaptive code. When a method call or return occurs, the delegate selects an appropriate behavior based upon the contracts' state.

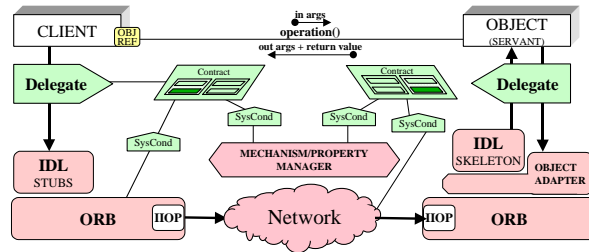


Fig. 1. QuO can control, measure, and adapt QoS requirements.

- **System Condition Objects** – System condition objects are wrapper facades that provide consistent interfaces to infrastructure mechanisms, services, and managers. System condition objects are used to measure and control the states of resources, mechanisms, and managers that are relevant to contracts.

QuO achieved its primary goal of enabling dynamic, adaptive QoS management, but it lacked the ability to quickly compose and configure complex adaptive behaviors and deploy these behaviors in a heterogeneous distributed environment using standard configuration tools and description languages. Our decision to componentize QuO and embrace standards-based middleware [9] directly led to the efforts described in this paper.

## 2.2 Brief CCM Overview

The CORBA Component Model is an OMG extension to CORBA 2.0 that brings ideas from component-based computing into CORBA [19]. Components in CCM are higher-level abstractions that organize collections of CORBA interfaces into dynamically loadable packages that can easily be linked together. CCM extends CORBA's Interface Definition Language (IDL) and provides tools designed to create, assemble, and deploy component packages. The run-time is also extended to include *containers* that create and manage component instances. The core extension to IDL is the Component type itself, which can include one or more of each of the following parts:

- **Facet** – An interface a component provides for use by other components.
- **Receptacle** – An interface a component requires from other components.
- **Event source** – A logical data channel on which a component publishes events.
- **Event sink** – A logical channel from which a component consumes events.

The assembly of a collection of components can be specified as the linkages between facets and receptacles, and between event sources and event sinks. These components are assembled using XML-based descriptions. The assembly description comprises different components, their homes and URL for their executables. This assembly is then deployed into a component server for execution. More details on CCM can be found in [19].

### 2.3 Illustrative DRE Application

In an effort for the US Navy, DARPA, and the US Air Force, we developed a prototype UAV simulation [11] for disseminating sensor information from a set of UAVs to command and control (C2) centers for use in time-critical target detection and identification. This application serves as an Open Experimental Platform (OEP) for DARPA's Program Composition for Embedded Systems (PCES) program [23] in which to explore managing end-to-end QoS requirements for (a) delivering imagery from UAVs to a C2 platform (e.g., a ground station, air C2 node, or shipboard environment) and (b) delivering control signals from the C2 platform back to the UAVs. QoS management includes trading off image quality and timeliness and managing resources end-to-end, with heterogeneous shared resources (e.g., tactical radio and wired IP networks), changing mission requirements, and dynamic environment conditions.

The architecture of the prototype is a three-stage pipeline consisting of three subsystem types: Sender, Distributor, and Receiver. The Sender simulates a remote UAV by sending video or images to a Distributor. The Distributor simulates UAV Ground Stations by distributing data/images to one or more Receivers. The Receiver simulates a C2 center, which displays, analyzes, or otherwise consumes the information. The way in which the imagery is used at the receiver (i.e., the simulated command center) provides the high-level QoS requirements on the image streams. That is, if imagery is used for remote piloting, low latency and smooth video is of primary importance. If imagery is used for targeting decisions, then fidelity of the imagery is of primary importance. If imagery is used for off-line intelligence analysis, then full data content is of primary importance. The communication protocol used for sending images from Sender to Distributor and from Distributor to Receiver(s) uses the TAO Audio/Video (A/V) Streaming Service because of its low overhead streaming capability [18].

## 3 Encapsulating Adaptive Behavior as Qosket Components

As described in [28], we developed a preliminary encapsulation model for adaptive behaviors that packages sets of related QuO contracts, system condition objects, and related code. These packages of reusable behaviors, called *qoskets*, have a run-time equivalent, qosket instantiations, of which there might be several distributed ones making up a logical design-time qosket.

Existing limitations of these qoskets led us to design *qosket components* that have all the features of standard components, but that encapsulate the QuO logic needed for adaptation and QoS management [9]. As a result, dynamic adaptive features can be added to an application by assembling the qosket components with the functional components.

A qosket component is a standard CCM component designed such that its interface mirrors the interface of the application components between which it is inserted. Being a CCM component, it can be assembled anywhere and in any number between application

components to provide QoS management. Please notice the difference between a Qosket and its instantiation as a set of qosket components. The former is a reusable unit of encapsulated adaptive behavior and it is, therefore, a generic package of code. The latter concretely realizes the packaged behavior as components specialized to a specific application interface, which can be assembled and executed. To apply qosket components we first converted the UAV application introduced in Section 2.3 into a componentized CCM application. Then we selected two adaptive behaviors as the representative adaptations and designed qosket components to carry out the desired adaptations.

### 3.1 Design of the Component-Based UAV Application

The componentized UAV application consists of three CCM component types: Sender, Distributor, and Receiver. One of the challenges in creating a component version of the UAV application is that there is currently no streams standard for CCM and no A/V streams implementation for CIAO or MICO. In the original prototype, we used TAO A/V streams to transport video images since it was only involved during the construction of the video pipeline, and did not incur the marshalling/un-marshalling overhead of CORBA calls. Though not easy, we could have programmed around the CCM interfaces to continue to use TAO A/V streams. This, however, would bypass the benefits of using the CCM standard. So we decided that an appropriate solution was to use standards-based CCM communication protocols, which led us to consider the following two communication models.

**Model 1: Event Push and Data Pull Model.** In this model, events are used for control flow, to indicate that an image is available, and facet calls are used to transfer image data. As shown in Fig. 2, the Sender acquires images (e.g., from a camera sensor or a file) and emits an event to indicate that the image is available. The Distributor, who has subscribed to this event, fetches the image using a facet call upon receiving the event, makes a local copy, and then publishes its own event indicating that it has imagery ready. Receivers subscribe to this event and, upon receiving the event, fetch and display the data.

This communication model has been used successfully in production avionics systems [26]; however, these systems typically involve tightly coupled processors. This model does not accurately simulate the operational conditions of real UAVs, which push their imagery to the ground station when collected, rather than having the ground station fetch

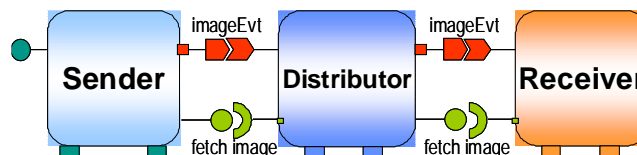
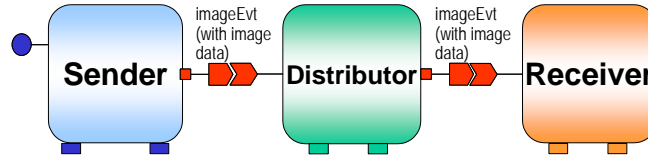


Fig. 2. Model 1: Event Push and Data Pull. Components can be collocated or distributed.



**Fig. 3. Model 2: Data Tagged Event Push. Components can be collocated or distributed.**

the imagery from the UAV.

**Model 2: Data Tagged Event Push Model.** In this approach, events contain an image payload. Since events are a valuetype, a CCM feature, it is possible to send both the event and the image data in the same event push, as illustrated in Fig. 3.

We measure the performance overhead of this approach for both these models in Section 4. Even though the figures for both models show a simplified view with one Receiver, we have successfully run this application with multiple receivers. We have also run the application with the components distributed over multiple hosts.

Above, we presented two models of communication: (1) control push/data pull; and (2) control and data push. Each interaction model has benefits and weaknesses given the context of its use. For instance, in situations of heavy network congestion it may be desirable to decouple the push of a control signal from the attempt to fetch the required data; the component receiving the event can determine whether it is worthwhile or even necessary to fetch the data. By contrast, if the data being transferred is small or is being sent to a remote component, it might be preferable to push the data to avoid costly remote invocations. Thus we foresee that component based applications will contain both interactions types, as appropriate.

### 3.2 QoS Adaptations in the Original UAV Application

The UAV application has complex and stringent QoS requirements commonly found in DRE systems. Meeting the high-level QoS requirements imposed by the mission (i.e., the use of the imagery) requires end-to-end QoS management, managing the CPUs, network links, and application functionality throughout the system.

In the original UAV application, we used the QuO middleware to implement application, network and CPU adaptation strategies by integrating QoS mechanisms all along the pipeline, with CPU management at each node, network management at each link, and in-band and out-of-band application QoS management at several locations. Results of experiments for end-to-end QoS management in the non-component-based UAV application are described in [27].

For the componentized UAV application, we chose two adaptive behaviors for this initial experiment: *scaling* and *spacing of imagery*. Scaling reduces the size of the images by specific factors (e.g., half scale or quarter scale) to address bandwidth constraints during

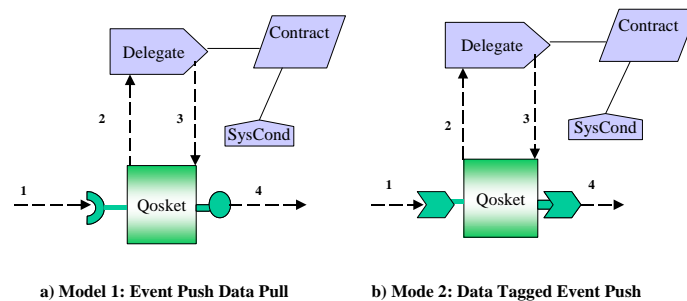
image transfers. Pacing sends images (or tiles when used in conjunction with a tiling qosket) at a steady rate to spread out bandwidth usage, thus better balancing the network load and avoiding jitter.

### 3.3 Architecture of Qosket Components

A qosket component provides adaptation by either intercepting a method call via a facet/receptacle connection or by intercepting an event by subscribing to an emitted or published event. It then grabs a *snapshot* [15] of the current state of QoS in the system, makes a decision about what to do based on the QoS policy encoded in contracts, and invokes the adaptation. This might modify the contents of data received (e.g., scaling an image), provide a modified implementation of the facet interface, or publish an event with the modified payload. Therefore, a qosket component includes an implementation of a facet interface that extends both the CCM executor generated by a CCM IDL compiler and the adaptive code generated by the QuO technology for Model 1 and an implementation of the event interface that dispatches the request to generated adaptive code in Model 2. For example, in Model 1, a qosket component modifies the data it receives on its receptacle using the Delegate (QuO generated code) before making the data available on its facet. (Fig. 4a). Similarly, in Model 2, a Delegate publishes the event that the qosket component was publishing after it alters the event payload, as depicted in Fig. 4b.

### 3.4 Implementing Dynamic Adaptation with Qoskets

A qosket component implements dynamic behaviors, driven by contract-encoded decisions and controlling mechanisms through, or reacting to conditions monitored by, system condition objects. For example, the scaling qosket includes a contract that determines dynamically at runtime whether each image should be scaled and by what ratio. The contract

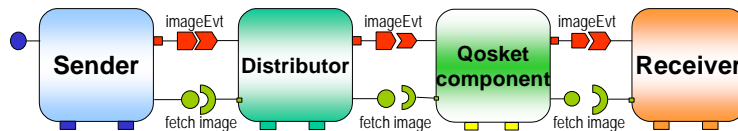


**Fig. 4. Qosket component Architecture for Model 1 (a) and Model 2 (b). Dashed arrows represent the control flow and the numbering presents the sequence of the flow.**

bases its decision on the available bandwidth measured by a system condition object and triggers scaling, when appropriate, through another system condition object. The system condition object that triggers scaling provides an interface to the image scaling routine, an off-the-shelf format-dependent routine. Using contracts supports the making of adaptation decisions dynamically at runtime. Using system condition objects supports abstracting the details of the system conditions that can be measured and the mechanisms that can be controlled through common middleware-layer interfaces.

### 3.5 Adding Qosket Components to the UAV Image Dissemination Application

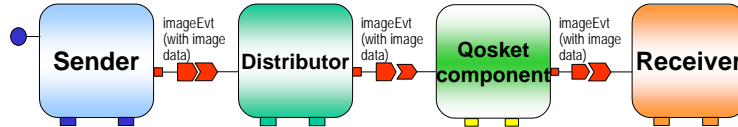
To insert adaptive behavior in the component-based UAV application using the Model 1 communication strategy, we created qosket components with receptacles and event sinks mirroring those of the Receiver and facets and event sources mirroring those of the Distributor. The qosket component is then assembled with the application as illustrated in Fig. 5. The qosket component receives imageEvt events from the Distributor through those interfaces, fetches the image data from the Distributor, and performs the proper adaptation. In the specific case of the prototypes we have built, the qosket component scales the image and adjusts its rate to match the available bandwidth. It then emits the proper event (to be consumed by the proper Receivers) to indicate that the modified image is ready to be fetched.



**Fig. 5.** A qosket component can provide adaptive behavior by intercepting facet/receptacle method calls using the Model 1 communication strategy.

For the Model 2 strategy, the qosket component subscribes to the same event that a Receiver subscribes to and publishes an event similar to that published by the Distributor. As illustrated in Fig. 6, the qosket component intercepts the event from the Distributor, pulls out the data payload, and reconstructs the event with modified data, which it then publishes.

In this componentized UAV application, the components have identical interfaces for their ports (facet/receptacle and event) for communicating with each other. This enables our qosket component to be inserted anywhere. Each qosket component can have as many adaptive behaviors as desired. However, encoding each qosket with one and only one adaptive behavior decouples different adaptive behaviors and increases the reusability of



**Fig. 6.** A qosket component can provide adaptive behavior by subscribing to and emitting events using the Model2 communication strategy.

each and thus would be a preferable approach in the future. So far, we have implemented qosket components that provide the following adaptive behaviors:

- (1) Network resource management: DiffServ Qosket (sets Diffserv code points to prioritize traffic).
- (2) CPU resource management: CPUBroker Qosket (reserves CPU for important processes).
- (3) Data shaping: scaling qosket (scales the size of an image before sending it over the network), compression qosket (compresses to different levels – lossless and lossy), pacing qosket (paces the data over the net to reduce jitter), and cropping qosket (zooms into a particular area of interest).

The qosket components can then be integrated during assembly to compose adaptive behaviors. We have identified some important issues regarding Qosket composition (such as the ordering of qoskets, e.g., compression and decompression) that were discussed in our previous paper [9].

QoS management solutions depend on effective QoS enforcement mechanisms. Each of the qosket components described above include adaptation strategies, monitoring, and control. The control enforces the QoS behaviors either algorithmically (e.g., scaling and cropping) or using off-the-shelf routines (e.g., compression) or mechanisms (e.g., Utah’s CPU Broker [4] and Diffserv). Each of these enforcement mechanisms is part of end-to-end, adaptive QoS enforcement and the qosket components provide middleware level interfaces for using them.

The current architecture to provide QoS adaptations using qosket components does not rely on streaming provided by A/V Streams. When (or if) streaming becomes incorporated within CCM, it will be yet another form of communication model, as discussed in Section 3.1, and will be amenable to dynamic adaptation via qosket components. It will still require QoS management rather than solve all QoS issues.

### 3.6 Assembling and Deploying Functional and Adaptive Behaviors

Both the functional, i.e., UAV application specific Sender, Distributor and Receiver components, and the QoS components were assembled using an XML based configuration file. In the case of CIAO and MICO, we assembled the application using a component assembly descriptor format and deployed the system using the deployment mechanisms

specified by CCM. In the case of Prism, we used Prism's proprietary configuration, which is also XML but is quite different from CCM specifications.

### 3.7 Generality of Component-Based QoS Management

The ability to enable dynamic adaptive QoS behavior by assembling qosket components is an important contribution that should increase the applicability of component-based middleware. First, it makes it possible to manage end-to-end QoS requirements without requiring the middleware to provide such a service. This is important because there is no agreed-upon QoS standard for CCM, although there are proposals actively being considered [6, 20]. Second, by working within the component model supported by the middleware, we are able to use the same assembly, packaging, and deployment tools provided by the middleware for qosket components. Third, we have been able to create qosket components that are independent of the underlying middleware by separating code specific to the particular component model from the generic code required to manage and evaluate the QoS contracts. To date we have handcrafted the specific "boilerplate" code for MICO, CIAO, and Boeing's Prism. Currently, we are working to provide a comprehensive tool chain, as described in [9] using modeling software such as GME [13].

Indeed, while MICO, CIAO, and Boeing's Prism are similar (Prism was heavily influenced by ideas contained in the CCM specification) the implementation of the qosket components are similar for MICO and CIAO but significantly different for Prism. The underlying QuO capabilities, as described in Section 2.1, are used "as is", while the component structure, as demanded by the various component models, is specifically written. The qosket component performs its QoS duties by capturing the current state of QoS in the system using QuO's system condition objects (SysConds). In CIAO and MICO, these SysConds are known to the CORBA ORB through the Portable Object Adapter (POA), thus the qosket component can simply request the values from the ORB. Prism, however, does not expose the POA to the executing components, for time-critical reasons, thus the Qosket component must request values from SysCond components existing within the Prism application.

The only differences in the implementation of qosket components in CIAO and MICO are minor implementation specific details, such as the number of IDL/CIDL specifications to write and the header files to include. Assembling and deploying the qosket and functional components is also similar except for minor differences in XML format. These differences are because the CCM specification does not define all details of component packaging; final implementation decisions are left to the CCM vendors, who are required only to provide tools for packaging, deployment, and assembly. Prism, however, needs a different process for implementing, assembling and deploying qosket components.

## 4 Experimental Results

The componentized UAV application described above demonstrates that adaptation can be inserted into an application by assembling qosket components with the application components. An important question to address is the tradeoff cost of the increased flexibility, power, and capabilities enabled by these higher level-programming abstractions. Previous studies have shown that using CIAO components, a layer of abstraction on TAO CORBA, incurs some overhead over using TAO objects [7]. We expected that qosket components would similarly introduce an additional overhead associated with the QuO adaptation infrastructure and additional adaptation specific algorithms (which should be more than offset by the improvement in resource utilization and QoS obtained). In this section, we take a first step toward measuring this for our CIAO qosket implementation, building upon the CIAO benchmark experiments previously conducted. First we discuss our experimental metrics followed by an explanation and the reasoning behind our experimental design. We then describe our experiments and discuss the results.

**Experimental Metrics.** We measured the average latencies of the UAV application using both models of image transmission, Model 1: Event Push and Data Pull and Model 2: Data Tagged Event Push. We based our experiments on the CCMPerf benchmark [12] in the CIAO implementation, except instead of measuring latency as the response time for a round trip operation, we measured the time elapsed in completing an event propagation from the Sender component to the Receiver (which includes the round trip data pull calls in Model 1). Comparing the results of these experiments for UAV applications with and without qosket components assembled into them provides a representative measure of the overhead associated with using qosket components in DRE applications. We selected the CIAO implementation because of the baseline work that has already been done for comparing CIAO with TAO [7].

We measured the extra latencies introduced by qosket components by running the experiments with and without a qosket component assembled in the UAV application. The qosket chosen for these experiments is one that provides image-scaling adaptation using a scaling algorithm provided by the QT library [25]. It scales the images to one-half or one-quarter scale based on bandwidth constraints specified in a QuO contract. Since our application is distributed, we ran our experiments by deploying one component, the Receiver representing a C2 center, on one host and the rest of the components of the assembly on another host. The qosket component assembled between the Distributor and the Receiver but running on the same host as the Distributor, chooses whether to scale images before transporting them over the network, in response to simulated bandwidth constraints.

We collected values as an average over five separate runs of 50 iterations each for each experiment. We ran an extra set of 10 iterations at the start before collecting any data to avoid including connection and other startup overhead in the latencies.

**Experimental Design.** Our experimental design consisted of the assembly of the following three configurations:

**Sender-Distributor-Receiver:** This component assembly consisted of the three components, Sender, Distributor and Receiver, where the Receiver was distributed on a remote host. This configuration served as the baseline case. The latency measurement included the time used by all three components for processing an image end-to-end. For example, the Sender spent time to load the PPM images in memory from a file and to send one image per event. The Distributor needed the time to cache an image and forward it to the receiver, again one image per event. The Receiver needed time to display the images. All of these operations were included in the end-to-end latency measurement.

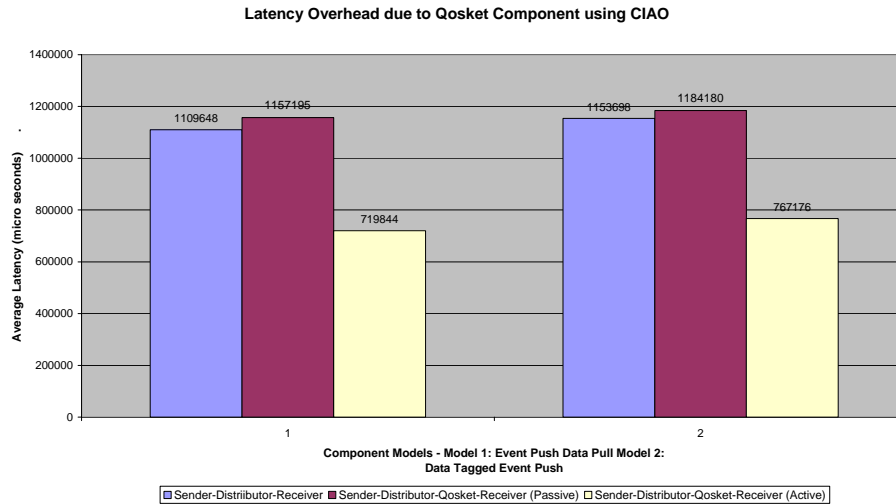
**Sender-Distributor-Qosket-Receiver (Passive Adaptation):** This component assembly consisted of the baseline configuration (above) with a qosket component assembled between the Distributor and Receiver. The qosket performed no adaptation, but simply passed the image through. This passive adaptation exhibits the latency overhead associated with the qosket component instantiation. This configuration is important as it clearly separates the qosket component overhead cost from the latency cost associated with specific adaptation algorithms and strategies that might be applied.

**Sender-Distributor-Qosket-Receiver (Active Adaptation):** This component assembly is similar to the Passive Adaptation configuration except that the qosket component performs active adaptation. The qosket component scales images in response to changes in SysCond object values monitoring bandwidth specified in a QuO contract. In the experiment, we changed the system condition object values every ten iterations to force contract evaluations and thus scaling of images by a different factor. The images were scaled full size to half-size to quarter-size, back to half-size, and then back to full-size. This added the cost of scaling to the latency as a representative adaptation.

**Experimental Testbed.** The testbed consisted of two machines linked by 10 Mbps Ethernet. Each of the machines was a 2.4 GHz Intel Pentium IV laptop with 512 KB RAM and were running the RedHat Linux 9.0 Operating System and ACE 5.4, TAO 1.4 and CIAO 0.4. The clocks on the machines were synchronized with a central server using ntp. In our experiments, we deployed the Receiver component on one machine and the rest of the components on the other machine. We transferred one PPM image of size 1.79 MB during each event from a set of 10 images that we cycled through.

**Experimental Results.** In the graph in Figure 7, there are two sets of results. The first three columns represent the Model 1: Event Push and Data Pull model and the second three columns represent the Model 2: Data Tagged Event Push Model, both implemented in CIAO. The first column in each set represents our baseline assembly of a Sender, a Distributor, and a Receiver. The second column in each set represents the second assembly with a Sender, a Distributor, a qosket component, and a Receiver when the qosket is used passively. The third column in each set is the same configuration as the second column except the qosket includes adaptive behavior that scales the images, alternating every 10 iterations from full size to half-size to quarter-size to half-size to full size again.

As shown in Fig. 7, we observe that the insertion of a qosket component did not significantly increase average latencies of the application when the qosket component was assembled passively (a 4.2% increase in latency for Model 1 and a 2.6% increase for Model 2). This increase in latency can be attributed to the overhead of having an extra



**Fig. 7. Average Latency Overhead of CIAO Qosket Components**

component (and the associated extra calls) through which the imagery is transmitted. When active adaptations were involved, we observe 35.1% and 33.5% decreases in average latencies from the baseline and 37.8% and 35.2% declines in latencies from the passive configuration. This decline is significant because it not only contains the processing time for a qosket component, such as generating the event and making method calls, but also the time required to scale the image and the additional cost incurred in marshalling and unmarshalling of images when transported across the network from Distributor to Receiver. This illustrates that carefully chosen adaptations can more than make up for the extra overhead involved in inserting the adaptations into assembled applications. Furthermore, we note that in this experiment, we scaled the imagery after the Distributor component right before sending the imagery over the network (from Distributor/Qosket to Receiver). We anticipate that the latency improvement would have been even more significant had we scaled the imager earlier, in the Sender, especially if the Sender had been distributed on a remote host too, due to the extra decrease in marshalling, copying, and network bandwidth.

Additionally, we note that Model 2 had a marginal increase in latency of about 3.9% over Model 1 for the active case. We think that copying of values in eventTypes is the primary cause of this increase. Though we observed a significant improvement in performance of the application due to the qosket component when it was actively used and only a marginal contribution to latencies overall, we acknowledge that this improvement depends on the adaptation algorithm used, the size of application data, and other factors. We cannot extrapolate such improvements to every adaptation and/or every DRE applica-

tion. This is not surprising, as the proper choice of algorithms and strategies affects the performance and correctness of every aspect of an application, including functionality and QoS. Additionally, the choice of model for disseminating the images has performance tradeoffs. Model 1 involves an extra CORBA call to retrieve the image data, which could be significant when the application is distributed. Likewise, in Model 2 every published event copies the image payload, which could be significant when the payload is large (e.g., in our experiments images were 1.79 MB each). As discussed later in Section 5.3, a more stream-friendly capability for CCM could alleviate these problems.

The experimental results presented in this Section are not a detailed analysis of performance tradeoffs of qosket components; this work is in progress. We plan to gather both black-box metrics such as latency, throughput, and jitter and white box metrics such as functional path analysis and look up time, for each component and for all three component-based middleware platforms, CIAO, MICO, and Prism. We plan to run more widely distributed applications, with all components deployed on different nodes to get an accurate measure of the impact of both network and CPU cost of qosket components.

## 5 Future directions

This paper presents the first realization of the design for providing component based dynamic QoS that we discussed in [9]. This is work in progress and there are many future questions to answer as well as challenges to overcome. We will discuss some of the challenges that we are attending immediately.

**Providing QoS as a Service.** While we presented one way of providing QoS adaptation to CCM based component middleware, we realize that our approach has its limits. A qosket component can alter the data content or event payload flowing through it, or the control flow of the application; however it is more difficult for the qosket component to directly impact an application component's CPU utilization, memory utilization or thread control. Hence, we are also currently exploring a way of exposing QoS interfaces in the container as we discussed in [9, 30].

**Scalability of Our Approach, Especially with Respect to Qosket Composition.** In a complex application, where several qosket components will be involved performing different adaptations, we need a way to ensure the compatibility of multiple qosket components. For example, one could mistakenly assemble a decompressing qosket component before a compressing qosket component. This mirrors the difficulties associated with assembling functional components, which, despite the presence of assembly tools, still requires domain expertise to do correctly. We partially address this challenge with our larger notion of design-time Qoskets, which can represent several related, distributed qosket components, with constraints on their assembly (e.g., that compression must be accompanied by a corresponding future decompression).

**An Integrated Tool Chain for Developing QoS Adaptive DRE Applications.** Encapsulating QoS behaviors as components provides one important benefit exhibited in this

paper, namely that CCM's lifecycle support and associated tools can be used to assemble and deploy QoS components in a similar manner in which they are used to assemble and deploy functional components. We are exploring how to add tool support for more epochs in the application lifecycle: design, implementation, assembly, deployment, and runtime. There is an opportunity to build upon the work at the intersection of Model-Integrated Computing (MIC), component-based middleware, and QoS adaptive middleware to create such a toolchain [31].

**Managing Crosscutting Concerns.** The functional decomposition of an application and its QoS decomposition crosscut one another in a fundamental way. CCM supports the functional dominant decomposition. QuO includes a design time version of Qoskets, an idealized reusable unit of QoS behavior, which is realized at implementation time as multiple qosket components that crosscut the functional components to provide an end-to-end QoS behavior. This paper describes the qosket technology that we have demonstrated in MICO, CIAO, and Prism middleware. There is still significant work remaining to capture the proper high-level abstractions defining an end-to-end QoS behavior, mapping it onto the qosket components that implement the behavior, using assembly tools to integrate the qosket components throughout a functional application, and coordinating their runtime behavior to manage QoS.

**Streaming and Pipeline Support.** The CCM standard doesn't currently include a streaming protocol, such as TAO A/V Streams, that supports a low-overhead pipelined way of sending streaming data (e.g., imagery or video) between objects. There is a need for ongoing research and standards activities to help make component technology suitable for DRE systems [21]. The QoS management work described in this paper will still be a necessary part even when streams become included in the CCM standard.

## 6 Concluding Remarks

DRE applications have stringent QoS requirements and need standardized support for assembling, packaging and deploying. Recently emerging component-based technologies, CCM ORBs specifically, provides the latter but has no QoS provisioning ability so far. Our QuO technology has already been demonstrated to provide dynamic QoS but lacks the ability of composing and configuring supported by CCM ORBs. In this paper, we have described our approach of providing QoS adaptations by encapsulating adaptive behavior in qosket components. QoS provision hereby becomes an issue of assembling of these components along with functional component.

We have presented qualitative arguments for the generality of our approach. Based on our preliminary experimental data on the two communication models in CIAO, we conclude that qosket components do not result in a significant increase in latency when used passively and, in fact, can increase the performance of an application when used actively in the context of DRE applications.

In the future, we plan to use Model Integrated Computing for developing templates for generating much of the code required for these qosket components. We would actively seek to integrate streams as it becomes available for CCM components. We are also working on alternative approaches of providing QoS.

## Acknowledgements

We acknowledge Arvind S. Krishna, PhD student at Vanderbilt University, for his valuable discussions on conducting our experiments using CCMPerf. George Heineman is on sabbatical from Worcester Polytechnic Institute.

## References

1. BBN Technologies, QuO - Quality Objects, [quo.bbn.com](http://quo.bbn.com).
2. Thomas D. Bracewall, Dave Sperry, Maureen Mayer and Priya Narasimhan. Fault Tolerant CCM. CCM Workshop, Vanderbilt University, Nashville, TN, Dec 10, 2003.
3. Component Integrated ACE ORB (CIAO), <http://www.cs.wustl.edu/~schmidt/CIAO.html>.
4. E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic CPU Management for Real-Time, Middleware-Based Systems. Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), Toronto, ON, May 2004.
5. EJCCM -Enterprise Java CORBA Component Model, <http://www.cpi.com/ejccm/>.
6. Fraunhofer Institute FOKUS. Quality of Service for CORBA Components RFP Initial Submission. Version 1.1, March 29, 2004.
7. Chris Gill and Nanbor Wang. Configuration and Codesign of Low-Level Infrastructure Framework in CCM. CCM Workshop, Vanderbilt University, Nashville, TN, Dec 10, 2003.
8. George T. Heineman and William T. Councill. Component-Based Software Engineering: Putting the Pieces Together, Addison Wesley, June 2001.
9. George T. Heineman, Joseph P. Loyall, and Richard E. Schantz. Component Technology and QoS Management. International Symposium on Component-based Software Engineering (CBSE7), Edinburgh, Scotland, May 24-25, 2004.
10. ICMG:K2-CCM, <http://www.icmgworld.com/>.
11. David Karr, Craig Rodrigues, Joseph P. Loyall, and Richard Schantz. Controlling Quality-of-Service in a Distributed Video Application by an Adaptive Middleware Framework. Proceedings of ACM Multimedia 2001, Ottawa, Ontario, Canada, September 30 - October 5, 2001.
12. Arvind S Krishna, Jaiganesh Balasubramanian, Aniruddha Gokhale, Douglas C. Schmidt, Diego Sevilla and Gautam Thaker. Empirically Evaluating CORBA Component Model Implementations. OOPSLA 2003, October 26-30, 2003.
13. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. WISP, Budapest, Hungary, May 2001.
14. 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 Applica-

- tions. Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, April 16-19, 2001.
15. Joseph Loyall, Paul Rubel, Michael Atighetchi, Richard Schantz, and John Zinky. Emerging Patterns in Adaptive, Distributed Real-Time, Embedded Middleware. OOPSLA 2002 Workshop, Patterns in Distributed Real-time and Embedded Systems, Seattle, Washington, November 2002.
  16. Joseph P. Loyall, Richard E. Schantz, John Zinky, and David Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), Kyoto, Japan, April 1998.
  17. Mico Is CORBA, The MICO project page: <http://www.mico.org>, <http://www.fpx.de/MicoCCM/>
  18. Sumedh Mungee, Nagarajan Surendran, and Douglas C. Schmidt, The Design and Performance of a CORBA Audio/Video Streaming Service, Proceedings of the 32nd Hawaii International Conference on System Systems (HICSS), Hawaii, January, 1999.
  19. Object Management Group, CORBA Component Model, V3.0 formal specification, <http://www.omg.org/technology/documents/formal/components.htm>.
  20. Object Management Group. Quality of Service for CORBA Components, Request for Proposal. OMG Document mars/2003-06-12, June 6, 2003.
  21. Object Management Group. Streams for CORBA Components, Request for Proposal. OMG Document mars/2003-06-11.
  22. OpenCCM – The Open CORBA Component Model Platform, <http://openccm.objectweb.org/>.
  23. Program Composition for Embedded Systems Program, DARPA, <http://dtsn.darpa.mil/ixo/programdetail.asp?progid=69>.
  24. Qedo -QoS Enabled Distributed Objects, [www.qedo.org](http://www.qedo.org).
  25. QT Library <http://www.trolltech.com/>.
  26. 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.
  27. Richard Schantz, Joseph Loyall, Craig Rodrigues, Douglas Schmidt, Yamuna Krishnamurthy, and Irfan Pyarali. Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 2003.
  28. 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, Washington, DC, April 29 - May 1, 2002.
  29. Rodrigo Vanegas, John Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, and David Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), The Lake District, England, September 1998.
  30. Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall and Richard E. Schantz. Total Quality of Service Provisioning in Middleware and Applications. The Journal of Microprocessors and Microsystems, Elsevier, vol. 26, number 9-10, January 2003.
  31. Jianming Ye, Joseph Loyall, Richard Shapiro, Sandeep Neema, Nagabhushan Mahadevan, Sherif Abdelwahed, Michael Koets, and Denise Varner. A Model-Based Approach to Designing QoS Adaptive Applications. IEEE International Real-Time Systems Symposium, December 2004.
  32. John Zinky, David Bakken, and Richard E. Schantz. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, April 1997.