Component Technology and QoS Management

George T. Heineman, Joseph Loyall, Richard Schantz

BBN Technologies
10 Moulton Street
Cambridge, MA USA
{gheinem, jloyall, rschantz}@bbn.com

Abstract. Component-based technologies make it possible to decompose complex software systems into functional software components. Unfortunately, most component technology does little to address non-functional Quality of Service (QoS) requirements that concern the runtime execution of the software. We are developing a technique to enable one to satisfy QoS requirements of software systems by decomposing and assembling Qoskets similar to the way systems are decomposed into functional components. A qosket is a packaged unit of reusable QoS-related behavior and policy. These qoskets are integrated into a component-based application to enable dynamic, adaptive QoS management. We present our design and describe our initial prototype. We conclude with a set of open questions and directions for future work.

1 Introduction

As software systems grow in size and complexity, it becomes increasingly difficult to ensure both the desired functional requirements (what the system does [2]) and non-functional requirements (how the system should accomplish the desired functionality given “the constraints of a non-ideal world” [3]). Some non-functional requirements (NFRs) are related to the process of software engineering, such as, maintainability, portability, or extensibility. We are concerned with NFRs related to the run-time execution of the software, such as performance or network bandwidth utilization. These concerns are often classified under the larger concept of Quality of Service (QoS), a term first used by the communications and networking community to describe the ability to measure and guarantee transmission rates over computer networks. The QoS concept extends to involve any “constraint” [3] that might impact the execution of software, such as CPU utilization, network bandwidth usage, or power consumption.

Advances in Component Technology [4] address the complexity of software by decomposing software into software components that conform to a specific component model. Components reduce the labor-intensive nature of software development by enabling application developers to package and organize code in a more formal, higher level manner. Components also enable a shift in perception from writing software systems from scratch to assembling systems using pre-existing software components.
Many believe that the only way to ensure end-to-end QoS requirements within a software system is to embed QoS-related statements throughout the software system; this approach leads to non-reusable software and maintenance difficulties. BBN has for a number of years investigated ways to incorporate aspects of QoS management as extensions to standards-based middleware using Quality Objects (QuO) [7]. One key objective of this approach to QoS management is to develop software that can rapidly adapt to the changing run-time conditions often found in distributed, real-time and embedded system environments. BBN is now directly investigating the ability to package QoS-related software into reusable units, called Qoskets, and to make these technologies available off-the-shelf to engineers trained in QoS (also known as qosketeers).

We are concerned about end-to-end QoS management which cannot be localized to a single or even a handful of components. Succeeding in this arena will enable systematic approaches towards the construction of software systems with complex QoS needs. We also are particularly concerned with providing dynamic, adaptive QoS management that will be needed for future systems’ complex (and changing) QoS requirements. We are developing a solution that, in the long term, will enable one to satisfy the necessary QoS requirements of software systems by decomposing and assembling Qoskets, similar to the way systems are decomposed into functional components. Our solution will have immediate benefits.

1. Add capabilities (B1) – Once there is support for dynamic, adaptive QoS management, it will be possible to build software systems that can alter their functional capabilities or resource usage in response to changes in the environment.

2. Reduce effort (B2) – Developers will not need to make as many individual decisions for ensuring QoS. The real pay-off will be the automated support for assembly rather than hand-customization of the individual parts.

We believe that any solution must have the following characteristics:

1. Incorporate standardized component technology (C1) – By using standar-dized component technology, one minimizes the dependence on vendors of proprietary software. Existing analysis, design, or deployment tools for the standardized technology can be used “as is”.

2. Interoperate with non-component software (C2) – One cannot assume that all software elements of the final software system are packaged and deployed according to the agreed-upon component model. There may be substantial legacy software assets.

3. Integrate with system of systems (C3) – Any QoS management of resources, especially end-to-end QoS, takes place within a larger context. There will be no centralized QoS manager; rather, there will be federations of semi-autonomous managed systems.

1.1 Decomposition and Separate Responsibilities

Figure 1 presents our vision that combines the separation of roles (between application assembler and qosketeer) and the ability to decompose problems in each
space. The bold vertical line depicts when the action of satisfying QoS Requirements can be separated from the action of satisfying functional requirements. The dashed horizontal line divides monolithic and compositional development. On the left side of Fig. 1, the functional and QoS requirements are addressed simultaneously; below the dashed line, the software is decomposed into software components. On the right side of Fig. 1, there is a separate effort to ensure QoS requirements; below the dashed line we can decompose our QoS requirements whereas above the line, developers have no choice but to address all QoS requirements simultaneously.

The upper left box in Fig. 1 represents Case 1, the traditional monolithic software development where both Function and QoS requirements are addressed simultaneously by the software developers. In Case 2, the lower left box, the software is no longer monolithic and is decomposed into components; however individual component developers must be aware of QoS requirements and embed solutions to these QoS requirements within the components themselves. This leads to software that is not robust (in response to changes in executing environment) and cannot easily be modified to guarantee complex QoS requirements. In both these cases, the system is typically unable to support the dynamic adaptation of QoS requirements at run-time.

Starting from the premise that it is both meaningful and possible to separate QoS and functionality, there are two additional cases to consider. The upper-right box in Fig. 1 depicts Case 3 when there is a clear separation of roles between the functional developers and the qoseteers, but the underlying software codebase is still monolithic. The lower-right box in Fig. 1 depicts the ideal Case 4, where functionality and QoS are addressed separately and decomposition occurs independently, that is, whether to produce or assemble components (the small rectangles) or Qoskets (the small hourglasses).

![Role Separation](image)

**Fig. 1.** Dimensions of Development

There are two intermediate solutions before achieving the ideal Case 4. Case 3a matches BBN’s QuO technology for developing QoS modules that can be
composed within a monolithic (i.e., non component-based) application to ensure end-to-end QoS requirements [7, 8]. Such a system could be developed in an object-oriented language but the system remains monolithic without independently deployable components. Case 3b provides a single “point of control” for managing the desired QoS requirements (i.e., the code to support QoS is monolithic) within a component-based application. This latter case is undesirable because it becomes increasingly complex and impossible to reuse QoS support between multiple applications; it also does not scale.

1.2 Motivating Problems

We motivate our work with examples that show the local and global characteristics of QoS management.

![Diagram](image-url)

**Fig. 2.** System of systems

**Local Example.** Assume there is an Unmanned Aircraft Vehicle (UAV) that delivers images to a processing ground station. While the flight software for the UAV is carefully engineered to operate within tightly constrained restrictions (i.e., scheduling, resource usage) for the duration of its operation, more flexibility is allowed for the mission software. Specifically, the designers can identify numerous resource tradeoffs, such as power consumption, on-board CPU utilization, and network bandwidth requirements; there are also application tradeoffs, such as the amount of processing requested, the type of processing requested, and the nature of the data being processed. Instead of fixing these tradeoffs in advance, the UAV could operate differently based upon the current mission mode (an application-level concept) and available resources; for example, during
navigation the highest image quality would be distributed but during an engagement a lower image quality might be acceptable to increase on-board sensor processing.

**Global Example.** We also address QoS management at a global “system of systems” level. Figure 2 shows how the in-flight UAV processing described above is simply part of a larger system composed of Command & Control (C2) ground stations, a Command Activity and Operations Center (CAOC), other remote UAVs, and even some simulated UAVs and C2 stations. The boxes in Fig. 2 represent computers with standard PC operating systems or proprietary UAV hardware that are networked together using Ethernet, radio, or satellite communication links. In this picture, there are three communication paths that must cooperate (and compete) for shared resources on each computer, such as CPU utilization, network bandwidth, or disk usage. We need to develop QoS management solutions to support the (changing) needs of applications over this space, for example, as new applications are added or as mission priorities change.

## 2 Existing QoS Management Approaches

There are a variety of approaches towards enabling QoS management within component-based applications. We briefly summarize these in this section.

**Static analysis/prediction of QoS requirements:** Given an assembly of software components, together with relevant meta data, some researchers propose to analyze or predict statically that a given component assembly will meet its QoS requirements. De Jonge et al. present a technique for predicting the resource consumption for a component (for example, memory usage) by analyzing scenarios of resource usage [9]. Hissam et al. use rate monotonic analysis (RMA) to predict the latency of component execution at assembly time [12]. Such static approaches nicely complement the dynamic, adaptive enforcement of QoS requirements.

**Run-time enforcement of QoS policies:** Rather than perform complex up-front analysis, it is possible to monitor the run-time behavior of a component assembly to determine if QoS policies have been met. Vecellio and Thomas present an approach (similar to QuO described in Sect. 2.1) that augments the component middleware (in their case, Enterprise JavaBeans) to ensure compliance with separately encoded policies [10,11]. This effort is the most closely related to ours; the difference lies in QuO’s contract definition. Ciuhandu and Murphy have designed a reflective load-balancing service for component-based middleware for ensuring QoS requirements [13].

**Standards-based QoS middleware extensions and QoS Frameworks:** The power of Middleware comes from its ability to simplify applications by providing complex services, such as transactions, persistence, and security. Some believe Middleware needs to be extended to offer QoS services as well. While this is a daunting task, given the complexity of QoS management, there are some successes to date. QoS Enabled Distributed Objects (Qedo) implements QoS
extensions to the CORBA Component Model (http://qedo.berlios.de); they
are interested in support for stream-based communication and negotiation of
QoS contracts between components at design time. There are real-time CORBA
systems (such as TAO, http://www.cs.wustl.edu/~schmidt/TAO.html) that
intend to satisfy performance and deadlines requested by the executing compo-
nents. Our focus on dynamic, adaptive QoS management means we will always be
seeking solutions that go beyond what can be assured at assembly time. Finally,
some researchers propose QoS frameworks within which all QoS requirements
are specified and then supported [14]. We feel such approaches will ultimately
prove too costly to implement and certify.

2.1 Quality Objects (QuO)

QuO is a framework for enabling distributed applications to adapt in response
to a changing environment [7]. In a traditional distributed application, a client
makes a method call on a remote object through its functional interface. The
call is processed by middleware on the client’s host, delivered over the network
to the middleware on a remote host, and processed by the remote object. As
shown in Fig. 3, a QuO application inserts additional steps to this process. The
QuO runtime monitors the state of QoS before each remote invocation through
the use of System Condition objects (or SysConds) that provide a standard
way to measure heterogeneous resources, such as CPU utilization or bandwidth
usage. Delegate objects intercept in-band communication over the middleware
and the local contract decides the appropriate behavior to apply. The contract
is defined by a set of nested regions that describe the possible QoS states in
the system. Each region is guarded by a predicate over SysCond values. Based
upon the current QoS state, the contract could (1) specify additional processing
to perform; or (2) allow the method call to proceed as is; or (3) redirect the
invocation to a different method; or (4) invoke a callback on the application to
alter its execution.

To summarize, QuO involves numerous additional entities:

- delegate objects intercept in-band communication between objects
- system condition objects (SysConds) extract information about the Network
  (such as bandwidth utilization), the Middleware implementation (such as
  access patterns or other reflective information), or the Client and Server
  objects carrying out the communication
- contracts perform the adaptive behavior by monitoring the values of SysConds
- callbacks enable contracts to contact elements of the application layer

These entities are deployed throughout the application, making it a challenge
for packaging or reuse. QuO provides an abstraction of QoS contracts to the ap-
lication assembler so most of these entities are inserted “under the hood” of
the application which makes it a challenge to simultaneously deploy the application
and the QoS mechanisms. Finally, it becomes a challenge to manage sets of
multiple contracts. There are many characteristics of the QuO technology that
cannot be described here for space reasons; full details are found in [7,8] or at http://www.dist-systems.bbn.com. Preliminary results on the use of Qoskets and QuO can be found in [7].

3 Proposed Approach

Figures 4a and 4b show a simplified view of the relevant elements of the problem space. Figure 4a depicts an assembly of functional components using a standards-based component technology such as CORBA Component Model (CCM). The Sender component sends image data generated by Process to a Distributor component on a server which, in turn, distributes the image to multiple remote Receiver components.

Software components are assembled together, communicating with each other through explicit and published interfaces. In some cases, the communication is a synchronous method invocation, in others it is the delivery of an asynchronous event. Each component executes within a container (shown as a gray box in Fig. 4) that provides the run-time environment for the component; there may be multiple components co-existing in the same container. Figure 4b shows the middleware and network layers that enable the execution of the distributed real-time and embedded system (as sketched by the directional lines). The containers in Fig. 4a are part of the middleware layer while other elements of the middleware are offered as services to the components in the application layer.
To ensure that end-to-end QoS requirements of the component assembly are met, there are many basic questions that must be addressed; among them are the following:

- Who should take the lead in QoS assurances for a software system?
- Where in Fig. 4b should QoS-enabling technologies be placed?
- What QoS artifacts need to be created?
- Where in Fig. 4a should QoS artifacts (i.e., qoskets) be placed?
- What changes are required for the software component lifecycle to address QoS?
- What happens when Qoskets are composed together?

In this paper we present our initial directions towards answering these questions, and conclude with specific milestones that we envision for future work.

3.1 Initial Directions

To enable the rapid integration of dynamic, adaptive QoS management with existing component technologies, we augment standards-based component middleware technologies with Qoskets. The application assembler is responsible for connecting components together to satisfy the functional requirements of the software system; this effort is supported by tools provided by the standards-based middleware and the resulting component assembly is described by an assembly file. We assume the application is already decomposed into functional components and the qoskeeter must only apply Qoskets to ensure dynamic, adaptive end-to-end QoS management.
A Qosket is a packaged unit of reusable QoS-related software. A Qosket is each of the following, simultaneously:

- A collection of cross-cutting implementations – a set of QoS specifications (and supporting implementations) that are woven throughout a distributed application and its constituent components to monitor and control QoS and systemic adaptation.
- A packaging of behavior and policy – an encapsulation of adaptive QoS behavior and a policy for using that behavior (in the form of contracts and other artifacts).
- A unit of behavior reuse – an element supporting a single property (i.e., performance, dependability, or security).

This initial definition of a Qosket consolidates the various QuO technologies into bundled reusable behaviors. This consists of the following artifacts: QoS Contracts, System Condition Objects (SysConds), Callback Objects, Delegate Templates and Delegate Instances, Glue Code and other helper methods.

Each Qosket has as an element a qosket component that conforms to the component model defined by a standards-based middleware and provides access to the QoS-related technology supported by the Qosket. The qosket component is deployed in exactly the same manner as functional components. We envision the need for a qosketeer tool called QoSTool that processes an existing assembly file (for example, as shown in Fig. 4a) to generate a new assembly file that integrates the qosket components with the functional components (for example, as shown in Fig. 5). The use of QoSTool occurs when the distributed real-time and embedded software system is assembled and deployed; when the software system executes, the QoS technology specified by the Qoskets enforces and ensures the required dynamic, adaptive QoS management at run-time. The long term vision is to enable application assemblers to decompose and assemble software systems from functional components and to simultaneously satisfy the necessary QoS requirements by decomposing and assembling Qoskets.

**Who should take the lead in QoS assurances for a software system?**

The person whose role is qosketeer should be the QoS representative for the software system. We expect that this is already a role that the application assembler plays today, although in an informal capacity. There is no need to put together a new QoS team; far better to train application assemblers in key skills (and technologies) needed to ensure QoS through components. Note that many organizations already have sophisticated development processes in place that address their existing QoS needs. The qosketeer may have to create programming standards or other guidelines to train component developers for certain domain-specific QoS capabilities; in general, however, we expect the individual component designers will be unaffected by the need to ensure QoS.

**Where should QoS-enabling technologies be placed?** Since containers form such a prominent role in standards-based component middleware, our initial
idea was to use off-the-shelf containers “as is” and encapsulate all QoS-related
capabilities in the qosket components; we still believe this is the proper
direction to head. An alternative view is to extend the containers to embed QoS knowledge
within. The use of extended containers will be an attractive alternative for
those attempting to optimize the component middleware to operate on resource
constrained systems, as is common with distributed and real-time embedded
systems; for now, we are avoiding optimization concerns in our discussion. We
also believe this latter approach is more costly to design and implement.

CCM components (both functional components and qosket components) are
deployed according to assembly files that describe the connections between
components. To enable the integration of qosket components with functional components
there needs to be a tool that (1) generates appropriate wrappers to
intercept events going into (sinks) or out of (sources) a component as well as
method invocations into a component (facets) or out of a component (receptacles); (2) creates assembly files to wire the generated wrappers together with
the existing functional components. BBN will design this tool called QoS Tool
from the experience gained in manually carrying out these tasks. QoS Tool will
directly support qosketeeers.

What QoS artifacts need to be created? QoS contracts form the basis
for specifying QoS behavior, as has already been documented and demonstrated
using BBN’s QuO technology. The various sub-elements that make up a Qosket
all have great reuse potential, which means that qosketeeers developing Qoskets
will not be forced to design and create Qoskets from scratch. Since the qosket
component conforms to standards-based component middleware, there will need
to be tools to help qosketeeers rapidly create these if the Qoskets are to be
applied to different component middleware technologies. At this time, we will
craft the qosket components by hand. The QoS contract for a Qosket will be the
primary way in which the QoS behavior of the Qosket is defined and made visible.
The QuO technology currently provides a proprietary process for ensuring the
Qosket sub-elements are distributed throughout the run-time of the underlying
middleware [7]. BBN is currently upgrading QuO to support CCM which will
enable the Qosket sub-elements to be deployed according to a CCM standard.

To optimize within resource-constrained environments, it is possible to re-
duce the operational footprint of components by more tightly integrating the
qosket components with the functional components. While we do not pursue
this direction further in this document, the option could be added to QoS Tool.

Where should QoS artifacts (i.e., qosket components) be placed? The
distinction between Qosket and qosket component needs to be made clear. A
Qosket is composed of numerous elements while the qosket component provides
a CCM-conforming element that supports the QoS behavior as encapsulated by
the qosket. When a qosketeeer applies a qosket component to an assembly, there
are many sub-elements that must be distributed throughout the system. The in-
sertion of the qosket component occurs at assembly and deployment time and
so the distribution of sub-elements (such as system condition objects for monitoring, callback objects for contacting the application, and delegate wrappers for intercepting method invocations) also occurs at assembly and deployment time. These sub-elements will all be present at run-time (located throughout the distributed system) as already occurs with BBN's QuO.

We envision there will be generic Qoskets that are made “domain-specific” for use within a software system or are contextualized for use within a specific product. This approach also prevents Qoskets from having to be constructed from scratch and enables families of Qoskets to be built around core concepts, such as network bandwidth or CPU management.

**What changes are required for the software component lifecycle to address QoS?** Each component is packaged with meta data that describes relevant information about the component, such as the vendor, its footprint, or system requirements. The set of meta data is standardized according to the component model. We will extend this meta data to include attributes that contain primitive QoS information. For example, the dynamic memory requirements for a component could be known and declared in the meta data (i.e., requires 4K allocated bytes). As we design and package existing QoS-related software as Qoskets we will define standard QoS-related meta data attributes that will enable the QoS management of the components. These “styles of use” will be encoded within QoSTool and will enable downstream design tools to include this information when analyzing component assemblies. Such meta data information will be composed of range values (i.e., requires between 200-475 ms to execute), exact information (i.e., spawns one additional thread), or parameterized values (i.e., requires 32N bytes of memory where N is the number of times function “open” is called).

We also expect that the ability to properly manage the QoS expectations for some components will demand that the component expose functional interfaces (also called knobs). In the same way that we will develop stylized attributes in the meta data, we will define standardized interfaces that prove useful to QoS management. We expect that over time (and certainly with the help of organizational memory) key interfaces could be defined, standardized by the qosketeer, and used by component developers. These interfaces would be “owned” by the qosketeer but the implementations would be part of each component, and thus be owned by the component designers. BBN will make its stylized attributes and knobs available to accelerate the integration of third party components and QoS-related technologies.

In the context of the UAV scenario, imagine a component that makes image information from a camera available as a JPEG file. The JPEG standard allows a variety of control over the size and quality of the image (i.e., whether to use standard or progressive encoding, the scale of compression). If the component exposed these application-specific options through a functional interface (i.e., a knob) then the QoS-management of the component could alter these settings.
Often generic components are produced that must be customized for a particular platform to be used. During customization, the meta data information associated with the component could be further refined based upon exact statistical information known for the platform. Turning to the UAV scenario again, it may be a firm requirement that the image quality never fall below a certain threshold; this value could be encoded in the meta information for each component and be used during QoS management.

Some QoS issues can only be decided when the functional assembly of components has been assembled. Using either sophisticated analysis techniques, simulation, or actual testing, the qosketeer assigns relevant QoS meta data values to the components in the assembly. The qosketeer also designs the relevant QoS contracts and generates qoskets to drop into the assembly.

**What happens when Qoskets are composed together?** To achieve the premise of Qosket decomposition, QoS Tool must have sophisticated logic to detect when the application of multiple Qoskets requires special handling. The interaction between Qoskets is similar to the feature interaction problem that has long been studied in telecommunications software and software in general. We will not provide a solution for the general case of Qosket composition; rather, we will identify families of Qoskets (i.e., those that manage CPU, those that manage network bandwidth) and provide algorithms for valid compositions (or heuristics if the algorithms prove unwieldy). To provide a specific example of why Qosket composition matters, consider a compressor Qosket that compresses data at the expense of extra processing and a manager Qosket that manages the application's use of the CPU. By applying the compressor Qosket, the cycles available to the manager Qosket are reduced. Other compositional issues arise when one considers the hierarchical composition of Qoskets; these issues will be addressed in the future.

4 Current Status

Given the Sender/Distributor/Receiver example earlier, consider how one could add the following capabilities to the system:

1. Higher priority receivers must receive data before lower priority receivers
2. Ensure that all receivers receive an image within a fixed time window; the image can be scaled to accommodate available resources
3. Enable receivers to specify a minimum acceptable image quality
4. Enable sender to perform opportunistic processing when resources allow

One could easily imagine adding any number of similar characteristics to an application. Using our Qosket solution, Fig. 5 shows how we can introduce various qosket components (shown as hourglasses) into the application assembly. We omit many details for brevity; the assembly pictured here is currently being designed and prototyped.
The Sender component generates reconnaissance images; based upon available resources, additional “features” can be added to the image, such as GPS location or automatic target recognition (shown with an outline). The Distributor component delivers this image to numerous Receivers, perhaps scaling the image to suit the available resources of the client receiver. The original Sender, Distributor, and Receiver components exist without change, and all desired QoS behaviors are encapsulated within the qosket components.

5 Future Direction

This paper presents initial results in our efforts to componentize units for QoS management. Moving forward, we know there are many questions yet to be answered, each of which will lead to fruitful research discussions.

5.1 Qosket vs. qosket component

There is a distinction between Qoskets, an idealized reusable unit of QoS behavior, and qosket components, the executable units enabling the QoS management that conform to a specific component technology. To date, we have developed qosket components for Mico CCM and are currently developing qosket components for CIAO CCM and Boeing’s BoldStroke [1]. We expect to be able to design an abstraction (much like language bindings in CORBA) to enable these qosket components to be generated from executable specifications of the different
component technologies. We can also build on the templates already developed by QuO [7].

5.2 Coordinating Multiple Qosket Components

While decomposing QoS requirements into Qoskets is essential to manage complexity, we have yet to provide a comprehensive solution for the coordinating the efforts of multiple Qoskets. At one level we need to ensure that the information gathered by SysCond objects are shared and used by the Qoskets that need them. Currently, QuO contracts interact with each other in an ad hoc manner; we desire a standardized approach to qosket interaction.

5.3 Composing Multiple QoS Behaviors

Once Qoskets are interacting with each other, we need to ensure that one can compose QoS behaviors by describing the interaction patterns between qoskets. We will not attempt to ensure arbitrary composition of Qoskets; rather, we are targeting specific QoS categories (CPU utilization, network bandwidth) and specific policies (i.e., reservations, prioritizations). Within this focused domain we should be able to accurately model and analyze the compositions to determine whether the QoS requirements are met.

5.4 Investigate QoS Mechanisms

Qoskets are ideal for providing standardized interfaces to complex QoS services. To date, BBN has investigated numerous services, such as CPU prioritization, scheduling, multi-level resource management, fault tolerance, and security. Rather than waiting for the underlying middleware to support the desired characteristics, we are pursuing numerous investigations towards encapsulating the desired behaviors within Qoskets.

Acknowledgments

We would like to thank Gary Dusan and Praveen Sharma of BBN for their help in integrating BBN QuO technology with Mico CCM. This work is funded by a DARPA contract from the Programmable Composition of Embedded Software Program, http://dtso.mil/ixo/prograndetail.asp?progid=37. George Heineman is on sabbatical from Worcester Polytechnic Institute.

References