Packaging Quality of Service Control Behaviors for Reuse

Richard Schantz, Joseph Loyall, Michael Atighetchi, Partha Pal
BBN Technologies
{schantz, jloyall, matighet, ppal}@bbn.com

Abstract

Two limitations of the current implementations of adaptive QoS behaviors are complexity associated with inserting them into common application contexts and lack of reusability across applications. What is needed is a way of bundling all the specification for systemic behavior into one place, and making it feasible to insert the collection of related QoS management artifacts into applications as a single, reusable behavior. As a means of addressing these issues, we have developed a new abstraction intended specifically to bundle together the lower level abstractions, specifications and implementations associated with providing adaptive QoS behaviors in a manner which is both easier to package and makes the behavior bundle reusable in a variety of applications and styles of use. In this paper we introduce the concept of "qoskets" as reusable systemic behaviors, lay out elements of our design for qoskets within the QuO framework, and use a case study example to highlight the issues of using qoskets in practice.

1. Introduction and overview

Over the past decade, various technologies have been devised to alleviate many complexities associated with developing software for distributed applications. Their successes have added a new category of systems software to the familiar operating system, programming language, networking, and database offerings of the previous generation. Some of the most successful of these technologies have centered on *distributed object computing (DOC) middleware* architectures, which are composed of relatively autonomous software objects that can be distributed or collocated throughout a wide range of networks and interconnects. Clients invoke operations on target objects to perform interactions, invoke functionality, and transfer data needed to achieve application goals. To support these interactions, a variety of middleware-based services are available off-the-shelf to simplify, integrate and coordinate distributed application development. Aggregations of these simple, middleware-mediated interactions form the basis of large-scale distributed system deployments.

For a number of years, the Quality Objects (QuO) [16] project has been investigating, developing and delivering the components of a layered middleware architecture and framework designed to manage and package adaptive quality of service (QoS) capabilities as common middleware services. Using that framework, we more easily construct applications that can change their behavior based on matching QoS requirements with the current runtime environment in a predictable, controlled manner.

Adaptively managing QoS behavior tends to be spread throughout the whole application, becoming tangled with the program logic. In reaction to that, a key concept underlying the QuO architecture is the decoupling of middleware and application software needed to build effective distributed systems along the following two dimensions:

- *Functional paths,* which are flows of information and control between interacting components that define the content of the applications. In distributed systems, middleware ensures this information is exchanged easily and efficiently between remote peers, at the highest level appropriate to the design of the application. The information itself is largely application-specific, determined by the functionality being provided (hence the term "functional path"). These are the capabilities offered by most DOC middleware platforms today through various forms of ORB technology.
- *QoS attribute paths*, which are responsible for determining how well functional interactions behave end-to-end with respect to key system properties, such as:
  1. The appropriate provisioning of resources to client/server/peer-to-peer interactions at multiple levels of distributed systems to enforce real-time behavior across computational nodes;
  2. The proper application and system behavior if available resources are less than or greater than the expected resources; and
  3. The failure detection and recovery strategies necessary to meet end-to-end dependability and security requirements over varying operating conditions.

The QoS attribute paths are associated with controlling the behavior of an application, not its basic functionality, and are absent from current DOC middleware platforms.

The idea behind this decomposition is that if we can separate the "behavior" part from the "functional" part,

then we can more easily change the behavior part. If we can independently change the behavior part, then we can make those changes be dependent on the current operating conditions of the application, and thus construct run time adaptive applications. Related to this is another key attribute: the separation and smooth transition between reusable, multi-purpose, off-the-shelf, resource management parts of the solution (the middleware), and those which need customization and tailoring to the specific domain or object type or individual user preferences (the application). In this model, the middleware is responsible for collecting, organizing, and disseminating QoS-related meta-information that is needed to:

1. Monitor and manage how well the end to end functional interactions occur, and
2. Enable the adaptive and reflective decision-making needed to support QoS attribute properties robustly in the face of rapidly changing mission requirements and environmental conditions.

There are two complementary aspects toward doing this:

1. Controlling the lower level OS and network infrastructure to best meet current QoS requirements, and
2. Manipulating and/or modifying the application to better fit within the current resource constraints.

To bridge the large gap between the controlling of lower-level resources and application-specific needs and behaviors, we have been developing QoS-centric abstractions and services that are needed to flexibly and adaptively program, organize, and control systems composed of coordinated, rather than isolated, components. Among these abstractions are: QoS *contract objects* to organize and control an application's current operating mode, *system condition objects* to provide standard interfaces for measuring, collecting and accessing information about current characteristics, and *delegate objects,* which are adaptive components that modify the system's runtime behavior. Along with the artifacts of the distributed object middleware paradigm (invocations, callbacks, system services such as naming, etc.) we have successfully used the QuO framework and these QoS abstractions to build a number of predictable, adaptable and flexible applications [10]. In the course of doing so we discovered two limitations of the current implementation. First, the introduction of the various QoS abstractions brought with it with a certain amount of complexity associated with putting the right pieces together and inserting them into common application contexts. Second, while individual artifacts might be reusable, the intended adaptive behavior provided by the organized integration of the QoS artifacts was clearly not very reusable, especially as it depended on capabilities to monitor delegate level operations and integration with application specific operation semantics.

What is needed is a way of bundling all the specification for systemic behavior into one place, whether or not the behavior will ultimately end up in a delegate, contract or system condition object, and making it feasible to insert the collection of related QoS management artifacts into applications as a single, reusable behavior.

As a means of addressing these issues, we have developed a new abstraction intended specifically to bundle together the lower level abstractions, specifications and implementations associated with providing adaptive QoS behaviors in a manner which both makes it easier to package the behaviors within an application, and makes the behavior bundle itself reusable in a variety of applications and styles of use. We call an integrated bundle of adaptive behavior specification and implementation a "*qosket*".

A Qosket is defined independently of any object interfaces for remote functional objects. A Qosket is a bundle of specifications and implementations of an adaptive behavior that can be woven together with a functional specification and implementation to generate an adaptive delegate layer, or alternatively can be used standalone to integrate the behavior as part of the operating environment supporting the execution of the application. Some examples of qosket behaviors we have built include dynamic server selection, reserved bandwidth, load balancing, replication management, and intrusion response. In general, we see qoskets as the delivery vehicle for packaging the elements needed for supporting complex QoS attributes in an application independent manner. We envision an expanding, set of prepackaged off-the-shelf behaviors developed by QoS specialists available to be bound in with and perhaps customized for specific application contexts, making it relatively simple to include complex adaptive behavior in a wide array of distributed applications.

A Qosket is each of the following, simultaneously:

- a software component – i.e., a qosket is a set of specifications and classes with an interface for monitoring and controlling systemic adaptation
- a packaging of behavior and policy – i.e., a qosket generally encapsulates elements of an adaptive QoS behavior and a policy for using that behavior, in the form of contracts, measurements and code to provide adaptive behavior
- a unit of behavior reuse, largely focused on a single property – i.e., a qosket can be used in multiple applications, or in multiple ways within a single application, but typically deals with a single attribute (e.g., performance, dependability, security)

In our view, qoskets are a first step towards individual behavior packaging and reuse, as well as a significant step toward the more desirable (and much more complex) ability to compose behaviors within an application context. They are a means toward the larger goal of flexible design tradeoffs at runtime among properties such as real time performance, dependability and security, varying with current operating conditions.

This paper describes our concept for reusable behaviors through qoskets, which is a work in progress. We first

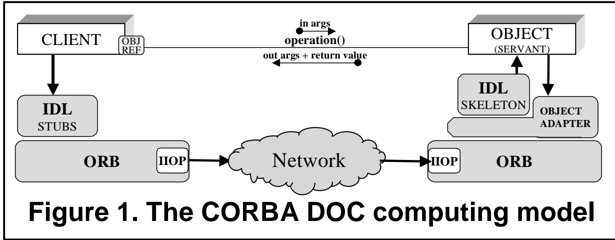


Figure 1. The CORBA DOC computing model

provide a summary of previously reported work, which serves as the building blocks for qoskets. We then describe the qosket concept, followed by a description of a running example, to illustrate the use of qoskets in practice. We then discuss our experience to date with qoskets and some of the key issues we see as next steps. We conclude with a comparison of this activity with related work in the field, and a brief summary of the results to date.

2. Review of the adaptive QuO middleware

Quality Objects (QuO) is a distributed object computing framework designed to develop distributed applications that can specify (1) their 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. By providing these features, QuO opens up distributed object implementations [6] to control an application's functional aspects and implementation strategies that are encapsulated within its functional interfaces.

Figure 1 illustrates a client-to-object logical method call in distributed object computing as instantiated by the CORBA standard. In a traditional DOC application, a client makes a logical method call to a remote object. A local ORB proxy (i.e., a stub) marshals the argument data, which the local ORB then transmits across the network. The ORB on the server side receives the message call, and a remote proxy (i.e., a skeleton) then unmarshals the data and delivers it to the remote servant. Upon method return, the process is reversed.

A method call in the QuO framework is a superset of a traditional DOC call, and includes the following components, illustrated in Figure 2:

- *Contracts* specify the level of service desired by a client, the level of service an object expects to provide, operating *regions* indicating possible measured QoS, and actions to take when the level of QoS changes.
- *Delegates* act as local proxies for remote objects. Each delegate provides an interface similar to that of the remote object stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.
- *System condition objects* provide interfaces to resources, mechanisms, and ORBs in the system that need to be measured and controlled by QuO contracts.

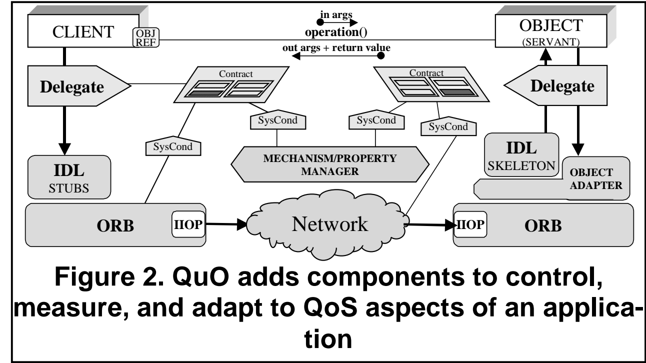
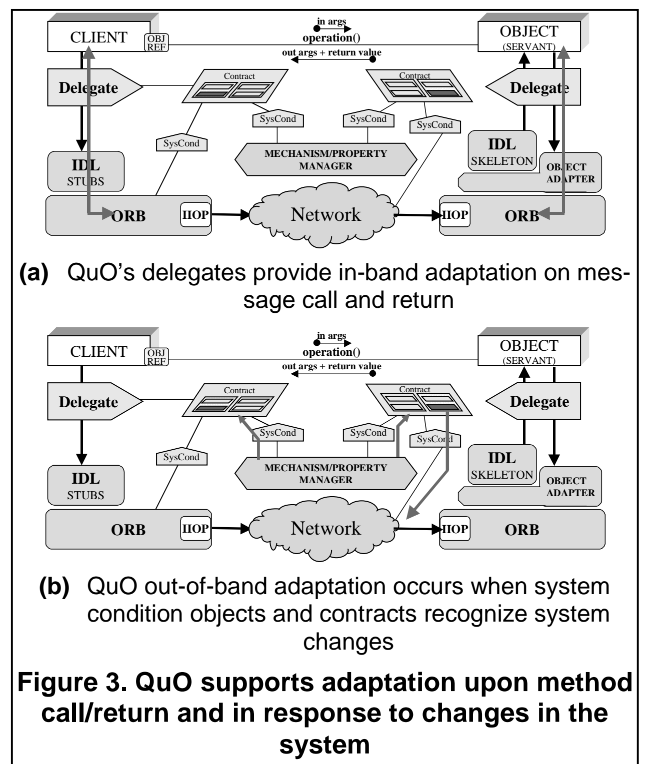


Figure 2. QuO adds components to control, measure, and adapt to QoS aspects of an application

- *Callback objects* provide notification interfaces to clients or objects in the application.

In addition, QuO applications may use property managers and specialized ORBs. Property managers are responsible for managing a given QoS property (such as the availability property via replication management [3] or controlled throughput via RSVP reservation management [15]) for a set of QuO-enabled server objects on behalf of the QuO clients using those server objects. In some cases, the managed property requires mechanisms at lower levels in the protocol stack. To support this, QuO includes a gateway mechanism [12], which enables special purpose transport protocols and adaptation below the ORB.

Besides the traditional application developers (who develop the client and object implementations) and mechanism developers (who develop the ORBs, property



(a) QuO's delegates provide in-band adaptation on message call and return

(b) QuO out-of-band adaptation occurs when system condition objects and contracts recognize system changes

Figure 3. QuO supports adaptation upon method call/return and in response to changes in the system

managers, and other distributed resource control infrastructure), QuO applications involve another group of developers, namely QoS developers or Qoskateers. Qoskateers are responsible for defining QuO contracts, system condition objects, callback mechanisms, and object delegate behavior. To support the added role of Qoskateer, we have been developing a QuO toolkit, described in earlier papers [9,13], and consisting of the following components:

- *Quality Description Languages (QDL)* for describing the QoS aspects of QuO applications, such as QoS contracts (specified by the Contract Description Language, CDL) and the adaptive behavior of objects and delegates (specified by the Adaptation Specification Language, ASL) [9].
- The *QuO runtime kernel*, which coordinates evaluation of contracts and monitoring of system condition objects. The QuO kernel and its runtime architecture are described in detail in [13].
- *Code generators* that weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program. Runtime integration of QDL specifications is discussed in [9].

2.1. In-band and out-of-band adaptation

The QuO architecture supports two means for triggering adaptation at many levels throughout the system, e.g., (property) manager-level, middleware-level, and application-level, as illustrated in Figure 3. QuO delegates trigger *in-band* adaptation by making choices upon method calls and returns. The Delegate intercepts all method calls to sets of remote objects and adapts its interactions to the remote objects based on the current QoS Region given by the Contract. Contracts trigger *out-of-band* adaptation when changes in observed system condition objects cause region transitions. In this way, the contract monitors and controls the system's QoS, but operates orthogonal to and outside the in-band functional behavior of the system. For example, [10] describes QuO's use in an avionics dynamic mission replanning system. In this system, in-band QuO adaptation is used to measure the progress of and change the size of data (using tiling and quality controls) being exchanged between aircraft across a constrained network link. Meanwhile, out-of-band QuO adaptation is used to monitor the available CPU and aid in scheduling of the hard and soft real-time avionics tasks.

While this architecture separates in-band concerns from out-of-band concerns, it only partially addresses the separation of functional behavior from systemic behavior. Functional behavior tends to reside in delegates and systemic behavior tends to reside in contract and system condition objects. But the systemic behavior may need to insert some code in-band, such as monitoring code to measure response time. Also, the functional behavior may need a richer interface for controlling the system than just querying a contract for its current QoS region.

With this as background, we can now discuss the technology and abstractions we have devised to package and reuse behaviors as a complement to the significant work that has previously gone into packaging and reusing functional attributes.

3. Qoskets – QuO Support for Reusing Systemic Behavior

One goal of QuO is to separate the role of QoS (or systemic) programmer from that of application programmer. A complementary goal of this separation of programming roles is that systemic behaviors can be encapsulated into reusable units that are not only developed separately from the applications that use them, but that can be reused by selecting, customizing, and binding them to an application program. To support this goal, we have defined *Qoskets* as a unit of encapsulation and reuse in QuO applications. Qoskets are used to bundle in one place all of the specifications and objects for controlling systemic behavior, independent of the application in which the behavior might end up being used, and whether or not the behavior will be used in-band or out-of-band.

Qoskets encapsulate, as reusable components, the following systemic aspects:

- *Adaptation policies* – As expressed in QuO contracts
- *Measurement and control* – As defined by system condition objects and callback objects
- *Adaptive behaviors* – As defined by ASL specifications, partially specified as templates until they are specialized to a functional interface, and by contract transitions and states.
- *QoS implementation* – Defined by qosket methods.

For example, a video dissemination application we have built includes qoskets that help maintain the QoS of video delivery in the face of limited resource availability, (CPU and network). It includes qoskets that encapsulate

- *adaptation policies* to maintain appropriate timeliness and fidelity tradeoffs for the video;
- *measurement* of video throughput and network and CPU load and *control* of network reservation;
- *adaptive behaviors* that filter the video, reserve bandwidth, and migrate functionality;
- *QoS implementation* that provides setup of the qosket objects and helper methods for filtering specific video formats and fragmenting packets to facilitate network reservation.

3.1. What constitutes a Qosket

A qosket is a component, encapsulating the interfaces, contracts, system condition objects, callback objects, un-

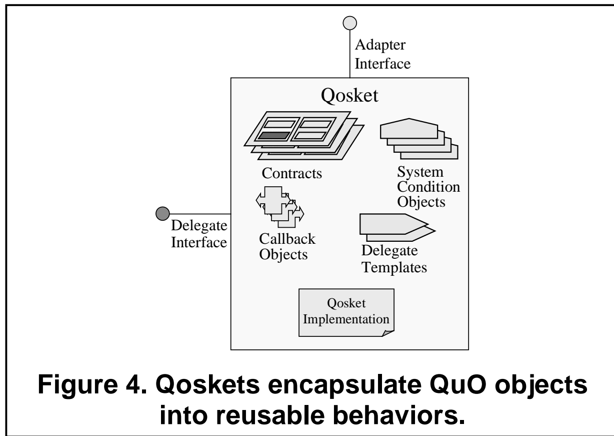


Figure 4. Qoskets encapsulate QuO objects into reusable behaviors.

specialized adaptive behavior, and implementation code associated with a reusable piece of systemic behavior. A qosket is specified by defining the following:

- The contracts, system condition objects, and callback objects it encapsulates;
- ASL template code, defining partial specifications of adaptive behavior.
- Implementation code for instantiating the qosket's encapsulated objects, for initializing the qosket, and for implementing the qosket's defined systemic measurement, control, and adaptation;
- The interfaces that the qosket exposes.

A qosket can be instantiated and used in either or both of two ways:

1. As a component generally usable by the application, e.g., to provide out-of-band adaptation and QoS control, through an adapter.
2. Used in conjunction with a QuO delegate, i.e. specialized to a particular functional interface to provide in-band adaptation and QoS control.

Therefore, qoskets expose two interfaces corresponding to these two use cases:[1]

- The *adapter* interface – An application programmer interface. This provides access to QoS measurement, control, and adaptation features in the qosket (such as the system condition objects, contracts, and so forth) so that they can be used anywhere in an application.
- The *delegate* interface – An interface to the in-band method adaptation code to define functions to make writing the delegate adaptation easier. Adaptation code can be encapsulated into qosket methods that are exposed through the delegate interface and then used in the ASL code that defines the delegate's adaptation strategy. This simplifies the adaptive descriptions written in QuO's ASL language as well as encapsulating adaptation helper functions.

[1] Although, like general component models, qoskets aren't limited to exposing only these two interfaces. Nor are both interfaces required to be non-empty.

The general structure of qoskets, objects they encapsulate, and interfaces they expose are illustrated in Figure 4.

3.2. Creating Qoskets

We follow the lead of the OMG CORBA specification in our QuO and Qosket programming model. In CORBA, a CORBA object programmer writes an interface in the CORBA interface definition language (IDL) and an implementation in the object's native target language, e.g., Java or C++. An IDL compiler generates the stubs and skeletons implementing infrastructure support for making inter-object remote method calls, marshalling and unmarshalling data, and tying interfaces to implementations.

In a similar way, QuO components, including qoskets, are defined by programming specifications and implementations, and a set of code generators generate the native language glue and infrastructure code necessary to support them. Figure 5 illustrates the programming model for qoskets implemented in Java. C++ qoskets are similar. Ovals represent the hand-written specifications written in IDL, CDL, or ASL, while rectangles represent native language code (Java or C++). Dotted arrows represent code generation – the target of the arrow is generated from the source – while solid arrows represent class dependencies – "extends" and "implements" in Java and inheritance in C++. Italicized notes in the figure indicate which code is hand-written and which is generated.

Defining a reusable qosket. A reusable qosket is defined by programming the objects that comprise the qosket as follows:

- Contracts are specified using QuO's Contract Description Language (CDL), described in [11]. QuO's code generator generates the implementation of the contract.
- System condition objects and Callback objects are defined similarly to general CORBA objects. The QuO

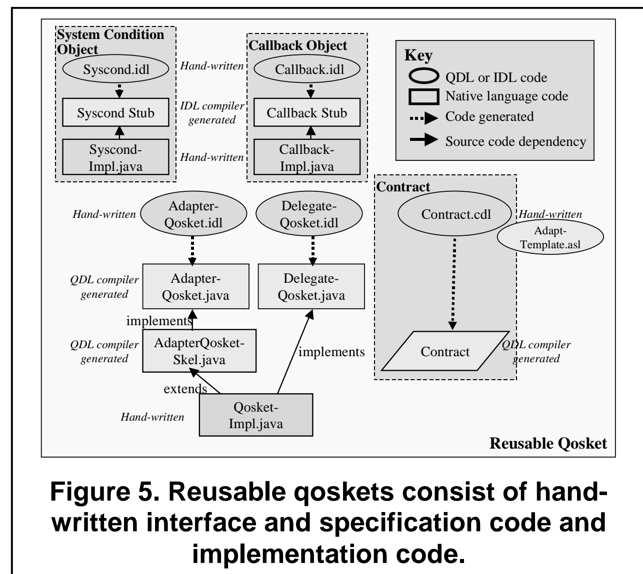


Figure 5. Reusable qoskets consist of hand-written interface and specification code and implementation code.

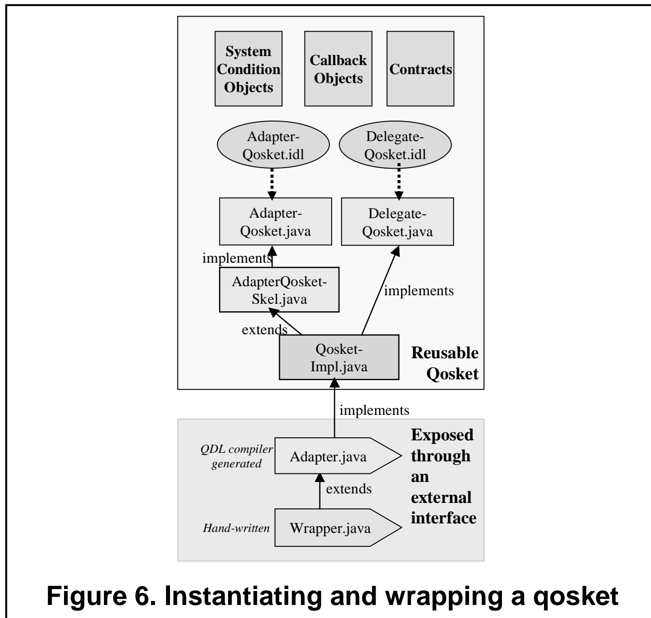


Figure 6. Instantiating and wrapping a qosket

programmer specifies the interfaces for the objects using CORBA IDL and the implementation for the objects using native language (e.g., Java), hooks them up with the other QuO objects, and registers them with the QuO kernel [11].

- The QuO adapter interface and the delegate interface (either or both of which can be empty, if no methods are to be exposed) are specified using CORBA IDL and implemented (*QosketImpl.java*) in a native language. The QuO code generators generate the native language interfaces and glue code.
- An optional, partially specified ASL adaptive behavior template may also be in the qosket definition.

In addition, the Qosket implementation will need to include code to instantiate the contract, system condition, and callback objects; to connect up these objects; and to implement methods defined in the adapter and delegate interfaces. Once all these pieces have been programmed, the result is a reusable piece of adaptive behavior that can be used in-band or out-of-band in an application.

3.3. Using Qoskets in an application

If an application simply wants to use the qosket to provide out-of-band measurement, adaptation, or control, a programmer can simply instantiate the qosket object and use it directly like any other object. However, QuO also provides a more convenient way of binding a reusable qosket so that it can also be used for in-band and/or out-of-band adaptation. Figure 6 illustrates this process of specializing a qosket as an object that can be used throughout an application, through the adapter interface. In order to do this, the QuO programmer uses the QuO code generator to generate an adapter object that combines the qosket and delegate (if one exists). While the adapter can, in general, be instantiated and used directly, the adapter object usually requires some initialization, such as locating and setting remote servers and initializing QuO objects, that are convenient to localize in a wrapper class. Instances of this wrapper class then become the interface to adaptive behavior (in the case of *out-of-band* adaptation) and replacements for the remote objects in the application (in the case of *in-band* adaptation). For example, the BWSelect qosket described in Section 4 can be used directly by an application (for out-of-band measurement or adaptation) simply by instantiating an instance of the BWSelectQosketImpl class described in Section 4.3. The application would then use the instance like a regular object. However, the initialization and customization of this object (initializing the QuO objects and setting the server objects) is wrapped into a new "connect" method on the wrapper class.

While the programming and instantiation of the adapter wrapper is sufficient to use a qosket if there is only out-of-band adaptation, measurement, and control, a qosket that is used in-band needs to also be specialized to its functional interface. As illustrated in Figure 7, the QoS programmer writes an adaptive behavior specification using QuO's Adaptation Specification Language, ASL, possibly specializing a behavior template in the qosket. The ASL description describes the actions that are taken in the path of remote method calls and returns (specified in *Functional.idl*) when the system is in particular states (described in *Contract.cdl*). The ASL language is described in detail in the QuO reference manual [11] and includes support for invoking alternative methods; adding functionality before, in place of, or after method calls; catching exceptions; and more. ASL templates can be written completely independent of functional IDL, using keywords in place of the methods and interfaces to which

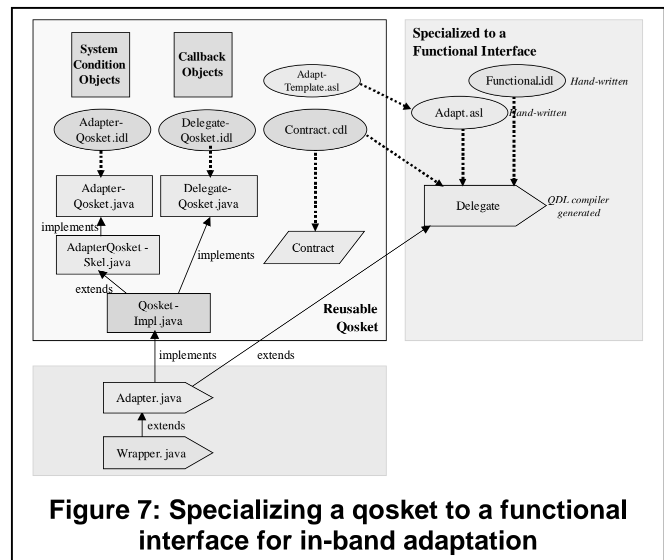


Figure 7: Specializing a qosket to a functional interface for in-band adaptation

```
contract BWSelect
  (
   syscond quo::ValueSC quo_sc::ValueSCImpl
Server1Status,
   syscond quo::ValueSC quo_sc::ValueSCImpl
Server2Status,
   syscond quo::DataSC quo_sc::data::DataSCImpl

Server1ExpectedNetworkCapacity,
   syscond quo::DataSC quo_sc::data::DataSCImpl

Server2ExpectedNetworkCapacity )
{
  region NoServers (Server1Status == 0 and
                    Server2Status == 0 ) {}
  region SomeServer(true)
    select (MAX) {
      region UseServer1
(Server1ExpectedNetworkCapacity * Server1Status)
       {}
      region UseServer2
(Server2ExpectedNetworkCapacity * Server2Status)
       {}
    }
};
```

Figure 8. The BWSelect contract defines the region space in a hierarchy – availability of servers and availbility of bandwidth

they get bound later, as illustrated in Figure 10 for the BWSelect example. Instantiated delegates are bound to a functional IDL, i.e., the QuO code generator uses all three specifications – the IDL interface, the CDL contract, and the ASL behavior – to create the object delegate. The object delegate exposes the same interface as the object interface, but *intercepts* method calls and returns, checks the current state of the contract, and invokes the appropriate adaptive behavior.

The delegate's adaptive behavior, as specified in ASL, has access to call any of the methods defined in the qosket's delegate interface. This has two advantages:

- ASL has access to the full functionality of the native language, while maintaining the simplicity of a specification language. Adaptive behavior is encapsulated into methods that are invoked in the ASL code.
- Adaptive behavior is more reusable. Encapsulating the code supporting a behavior into methods on qoskets enable it to be associated with the reusable packaged behavior component. Any functional-interface-specific information can be passed in through the delegate interface at runtime or when the qosket is specialized to a functional interface. In contrast, coding the adaptive behavior directly into the ASL description ties it more closely with a functional interface and removes it from the reusable qosket package.

For example, a video dissemination application uses frame filtering as an adaptation mechanism to compensate for a constrained network resource. Frame filtering, however, is fairly complex and requires significant computational and datatype support – as provided by native languages – to implement. We implement the frame filtering as a method exposed through the qosket's delegate interface and call the method in the delegate's ASL specification when frame filtering is needed.

The instantiated wrapper now functions as both a remote object delegate and a qosket object, depending on where it's used: places referencing the remote objects used in the functional objects of the application, or new uses for the qosket elsewhere in the application.

4. An Example: A Qosket for Dynamic Server Selection

In this section we develop and discuss a single running example of a qosket to illustrate its use in practice, and highlight the practical issues and the relationships of the parts described abstractly in the previous section.

In systems deployed today, selecting from among multiple servers offering the same service is either provided in a non-automated fashion, e.g., web pages frequently offer users a list of mirror servers from which to download, or the selection process is tightly bound to a narrow set of QoS attributes, e.g., in CPU load balancing. We can use the qosket concept to provide an automated, adaptive server selection capability to a wide variety of applications, each with their own factors and weighting to use in the selection process.

The *BWSelect* qosket described in this section is intended as an example of an adaptive policy for adding a *system property* to an application. The reusable behavior of this qosket is to *select servers based on whether they are available and what the bandwidth between client and server is.* First, we describe the components of the qosket and then describe how it is bound and used in a CORBA-based image retrieval application. Since we want the example to demonstrate primarily the core components of qoskets, we have simplified it to the minimum needed to illustrate the interacting parts of the qosket and how they are inserted into programs. Obviously, an effective, operational qosket for a system property such as this would need to be more detailed and would need to take into account other system parameters beside network bandwidth.

In the rest of this section, we show how the qosket serves to integrate the elements comprising the server selection property into a single, reusable behavior that can be applied and customized to a variety of applications.

4.1. The BWSelect Qosket's Contracts and System Condition Objects

Figure 8 shows the BWSelect contract, written in CDL, which we use for our running example. The body of the contract describes how we might organize the region

```
module qosket {
 module bw_select {
  interface BWSelectAdapterQosket {
   void setServer1 (in java::lang::Object
server);
   void setServer2 (in java::lang::Object
server);
  };
  interface ExceptionHandler {
   void handle_exception (in java::lang::Object
x);
  };
  interface BWSelectDelegateQosket {
   java::lang::Object getServer (in long i);
   void noServers ();
   qosket::bw_select::ExceptionHandler
makeHandler (in java::lang::Object server);
  };
 };
};
```

Figure 9. The adapter interface exports methods for initializing the qosket and the delegate interface exports methods for using the qosket for in-band adaptive decisions.

space for server selection, based on current QoS factors. Contract regions can be nested in a hierarchical way, and this example uses a two level hierarchy of availability (first level) and preferential selection among the available servers based on available bandwidth criteria (second level). This contract uses four system condition objects as data sources. The first two – Server1Status and Server2Status – indicate whether the servers are currently available or not. The second two – Server1Expected-NetworkCapacity and Server2ExpectedNetworkCapacity – provide information about the current network connectivity between the client and server hosts.

The contract in its illustrated form doesn't scale very well to larger numbers of servers, as each new server X makes it necessary to add a corresponding region Use-ServerX. To handle such situations, CDL has a means of specifying arrays of regions [11]. However, we have omitted that detail for the sake of simplicity in understanding the role qoskets play in integrating the parts of the adaptive QoS behavior.

4.2. The BWSelect Qosket's Exported Interfaces

As described in Section 3, a qosket not only encapsulates QuO objects, such as the contracts and system condition objects described in Section 4.1, but is also an object itself. As such, it exposes the two interfaces described in Section 3.1 – an adapter interface and a delegate interface. Figure 9 shows the adapter and delegate interfaces exposed by the BWSelect qosket. IDL is used to describe the qosket interfaces and QuO uses these IDL specifications to generate the qosket code (but not CORBA stubs

```
template behavior <method METHOD, interface
SERVER_TYPE>
BWSelectBehavior ()
{
 qosket qosket::bw_select::BWSelectDelegateQosket
bwQosket;
 ivar SERVER_TYPE server1;
 ivar SERVER_TYPE server2;
 ivar qosket::bw_select::ExceptionHandler
s1Handler;
 ivar qosket::bw_select::ExceptionHandler
s2Handler;

 METHOD.signature {
  return_value METHOD.return_type result;

  inplaceof METHODCALL {
   region NoServers {
    // This will throw a runtime exception so no
need to set the
    // result.
    bwQosket.noServers();
   }
   region UseServer1 {
    server1 = bwQosket.getServer(1);
    result = (METHOD.return_type)
server1.METHOD.name(METHOD.arguments);
   }
   region UseServer2 {
    server2 = bwQosket.getServer(2);
    result = (METHOD.return_type)
server2.METHOD.name(METHOD.arguments);
   }
  }

  onexception (exception) METHODCALL {
   region UseServer1 {
    s1Handler.handleException(exception);
   }
   region UseServer2 {
    s2Handler.handleException(exception);
   }
  }
 }
};
```

Figure 10. ASL Template for the BWSelect qosket

and skeletons). BWSelect's exposed interfaces consist of the following:

- The BWSelectAdapterQosket interface is used to initialize the qosket when it is used in an application. The setServer1 and setServer2 methods are used to initialize two servers whose selection is managed by the qosket.
- The BWSelectDelegateQosket interface is used to provide access to methods needed by the delegate's in-band adaptation, e.g., "noServers" and "getServer" used by the ASL specification in Figure 10.

In addition to these two interfaces, the BWSelect qosket's interface includes an ExceptionHandler interface that is used with ASL to do exception handling in the delegate's adaptive behavior. By having a well-defined interface, programs using the qosket can implement this

```
behavior ServerSelect ()
{
  BWSelectBehavior <slide::SlideShow::read,
slide::SlideShow>;
};
```

Figure 11. Instantiation of the BWSelect ASL template to fully specify a behavior

interface and integrate qosket exception handling with their custom exception handling code.

4.3. The BWSelect Qosket's Implementation Code

As in CORBA and Java, the qosket interfaces in Figure 9 must be implemented using native code. In addition to implementing the methods in the interface, the qosket implementation code generally takes care of object instantiation and initialization chores. It is frequently the case that the QoS programmer is confronted with situations in which he or she needs extra functionality to use or invoke in the delegate to implement the desired adaptive behaviors. Because the QDL and IDL languages are specification, rather than programming, languages, it is necessary to code this functionality in the target programming language. This functionality is programmed in the qosket implementation code and made available to the delegate (i.e., the ASL description of the delegate) through the qosket's delegate interface, for the reasons previously cited in section 3.3.

In the BWSelect example, there is a BWSelectQosketImpl class (not illustrated) that contains the following:

- Two member variables for the two servers, plus setter and accessor methods for them (the setter methods are implementations of the two methods exposed through the adapter interface);
- Implementations of the delegate interface methods, *getServer*, *noServers*, and *makeHandler*;
- Constructor and initialization methods;
- Helper methods, including *handleException*.

4.4. The BWSelect Qosket's Delegate Template

Since the delegate cannot be completely specified without the specification of the functional interface that it is delegating, a qosket cannot contain fully specified delegate behaviors (or it would not be reusable). However, as described in Section 3.1, qoskets can include partially specified adaptive behaviors using ASL Templates. Figure 10 illustrates the BWSelect qosket's ASL template containing logic to dispatch method calls to the appropriate server (based on the regions in the BWSelect contract) and to handle exceptions. This template is parameterized over the interface and method signatures for the actual business object, which will be filled in when the qosket is bound to a functional interface. ASL keywords are in boldface and include where the adaptive behavior is to be inserted (e.g., *inplaceof METHODCALL*) and the contract *regions* that trigger adaptation. *METHOD* and *SERVER_TYPE* are parameters that are dependent on the functional IDL and will be filled in when the template is instantiated (in Figure 11). These parameters can be used anywhere that makes sense in the template.

The template is parameterized by the functional method being delegated and the type of the servers from which it is choosing. The normal method call is replaced (*inplaceof METHODCALL*) by behavior that throws an exception when the contract indicates that there are no servers (*region NoServers*) or routes the call to one of the two servers as indicated by the contract (*region UseServer1* and *region UseServer2*). Code for handling exceptions is specified in the *onexception METHODCALL* block, which delegates the exception information to the exception handlers.

In general, the delegates provide an in-band mechanism to augment the behavior specified in the qosket with behavior tailored to the semantics of the operation it is encapsulating. In our example, the BWSelect qosket uses contracts to combine information about the system through system condition objects, and provides through the regions UseServer1 and UseServer2 a policy about which server to choose according to availability and network capacity. The qosket, through the contract mechanism, therefore contains the main logic of server selection behavior. This ASL description is where any desired additional adaptive behavior would be included, such as logging or measuring remote method calls to the respective servers; or shaping data – filtering, tiling, compression, etc. – going to the servers.

4.5. Binding the BWSelect Qosket to an Object

This part describes how dynamic server selection is added to an application called SlideShow by binding the BWSelect qosket to it. The SlideShow application consists of a client requesting and displaying server images.

The use of the BWSelect qosket in the SlideShow application is a client-side in-band use case: the server selection process occurs in the client every time it requests a new image from the servers. A client side delegate responsible for doing this adaptation is instantiated by the code in Figure 11 – which simply instantiates the ASL template in Figure 10. However, additional functional interface-specific adaptation could be included in the ServerSelect ASL in Figure 11, such as choosing alternative methods, logging or instrumenting method calls, or changing the characteristics of method parameters. Any such additional behavior that is not functional interface-specific could be included in the template in Figure 10.

5. Ongoing Work

The QuO software system includes a library containing reusable qoskets. Currently it is sparse but does contain sample qoskets for dynamic server selection, for setting up and tearing down RSVP network reservations, and others for instrumentation and collecting a variety of status information to feed to adaptive delegates. We continue to expand this library and to develop qoskets for particular QuO applications. For example, the qoskets developed to support the embedded QuO applications used in the avionics systems and the shipboard systems described in [10] include qoskets that support video filtering, image tiling, IntServ and Diffserv network management, and CPU allocation. The QuO applications for survivable systems described in [14] include qoskets that support intrusion detection, data integrity, access control, and dependability.

The development and use of these qoskets have helped us develop and refine the qosket idea and implementation. In the context of encapsulation of reusable behaviors – supporting the naming and objectifying a set of adaptive behaviors – qoskets have succeeded in the context of the examples in which we've applied them, including those mentioned above. However, there is still significant work to be done in terms of evaluating the reuse capabilities of qoskets, their relation to other component models, and composition models built around qoskets.

Consistency with Emerging Standards. One area that we are exploring is the relation between qoskets and other component models, e.g., the CORBA Component Model (CCM) [2]. We are exploring how to unify qoskets and a subset of CCM as it matures and implementations become available. In addition, the ability to name and encapsulate adaptive behavior opens the door to identifying recurring patterns of behavior in adaptive applications.

Composition and Reuse. Qoskets provide a rich facility for speaking about composition and reuse of adaptive behaviors. We have identified a number of patterns of composition of qoskets that are possible:

- Composition through interfaces – Code in a qosket calls the methods exposed through the adapter interface of another qosket.
- Composition through shared objects – Two or more qoskets can share sets of system condition objects or contracts.
- Composition through a manager qosket – A manager qosket M is created which has references to instances of qoskets A and B. M exposes the interfaces used by applications and directs calls to A and B as appropriate. M can mediate conflicts between the behaviors of A and B and even add behavior if necessary.
- Composition through use – A single delegate, or a chain of delegates, can use more than one qosket, effectively layering them or composing them, although their behaviors are independent.

We are just beginning to explore and enumerate the composition possibilities and the issues involved and do not believe that this list is either complete or the only dimension along which to enumerate the composition possibilities. In addition, we realize that there are composition issues among the objects inside a qosket, e.g., contracts and system condition objects that present interesting research areas also.

Usability. We believe that qoskets simplify the construction and reuse of QuO application components. However, we still have work to do to evaluate the usefulness and reusability of qoskets. We are currently concentrating in this area in providing documentation, examples, and by doing more with code generation, thereby reducing - not increasing - the programming burden using qoskets. We are also exploring the use of modeling techniques [5] as higher level means of specifying adaptive behaviors with synthesis of qoskets or qosket templates.

6. Related Work

The term component in a middleware context generally refers to reusable and composable software artifacts (usually larger than programming language constructs like functions or classes) that have their own internal architecture, providing a clean separation of concern between the application's functional and "plumbing" aspects. Examples of such middleware components are EJB [4] or CCM [2]. Qoskets differ from these general purpose middleware components in at least two ways: a) qoskets encapsulate adaptive behavior that augments the normal behavior of a distributed application and b) in order to define and control the adaptation, qoskets deal with QoS issues associated with the plumbing. In this sense, one can think of the relationship between qoskets and adaptive distributed systems to be analogous to that between EJB or CORBA components with distributed applications.

Componentization of specific aspects of adaptation, reflection and QoS control has been an active area of middleware research. In the open ORB project [1], componentization and reflection are used to build a next generation middleware platform that enables applications to respond to changes in the environment and to meet the needs of changing platform design. Open ORB aims to build a new middleware whereas our approach has been to augment existing middleware with the capability to insert components of adaptive behavior. This is similar in spirit with the dynamic TAO work, where (proxy code) components (fetched by an automatic reconfiguration service) are plugged in to achieve dynamic reconfiguration. The automatic reconfiguration service is part of an architecture proposed in [7] for managing dependencies in a distributed component-based system that supports automatic

reconfiguration and resource management. While the focus of their adaptation (reconfiguration) is more on inter-component connection and dependency, qoskets attempt to componentize the adaptive behavior itself. The decentralized resource management they propose may offer services that can actually be encapsulated in qoskets for using in a QoS aware adaptive distributed application. Component oriented ideas can also be seen in the QualProbes and Agilos systems in the form of Configurators, Adaptors, Qualprobes, Observers and Negotiators [8]. In some sense, an unbounded qosket corresponds to the application neutral Adaptor interconnected with the Observer(s) and Qualprobe(s) that it needs for operating.

7. Conclusion

As part of our ongoing research in the QuO adaptive, QoS-aware middleware, we have developed a method for encapsulating adaptive, QoS-aware behaviors using *qoskets*. Qoskets support instantiation of behaviors in two different ways – in-band and out-of-band – as well as reusing behaviors in different application contexts. We have built a growing library of qoskets and have used them in a number of documented contexts – wide-area environments, embedded applications, heterogeneous languages, and different DOC middleware systems. We have begun to explore composition of qoskets, usability of qoskets, and interactions with other component models. QuO software and the examples described in this paper are available open-source at http://www.dist-systems.bbn.com/tech/QuO.

Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency (DARPA) under contracts F30602-98-C-0187 and F33615-00-C-1694. We would like to thank Dr. Gary Koob and Dr. Doug Schmidt for their participation in and support of the programs and projects investigating adaptive quality of service. These activities are initiating significant changes in the expectations we have about how our systems can and should behave in unpredictable operating environments. We also thank other members of the QuO research group, especially Dr. John Zinky and Rich Shapiro, who worked out many of the details and some of the terminology for the initial implementation of qoskets.

References

[1] Blair G, Coulson G, et al. "The Design and Implementation of Open ORB 2," *IEEE Reflective Middleware*, Vol. 2, No 6, 2001.

[2] CORBA Component Model, OMG Technical Document no orbos/99-07-01.

[3] Cukier M, Ren J, Sabnis C, Henke D, Pistole J, Sanders W, Bakken D, Berman M, Karr D, Schantz R. "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[4] DeMichiel L, Yalcinalp L, Krishnan S. *Enterprise Java Beans Specification Version 2.0*, Sun Microsystems, August 2001.

[5] Gray J, Bapty T, Neema S, Tuck J. "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM,* Vol. 44, No. 10, October 2001.

[6] Kiczales G. "Beyond the Black Box: Open Implementation," *IEEE Software*, January 1996.

[7] Kon F, Yamane T, Hess C, Campbell R, Mickunas M. "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems," *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems,* San Antonio, Texas, February 2001.

[8] Li B., Nahrstedt K. "Qualprobes: Middleware QoS Profiling Services for Configuring Adaptive Applications," *Proceedings of the IFIP/ACM Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, pp 256-272, Springer-Verlag, LNCS 1795.

[9] Loyall J, Bakken D, Schantz R, Zinky J, Karr D, Vanegas R, Anderson K. "QoS Aspect Languages and Their Runtime Integration," *Lecture Notes in Computer Science*, 1511, Springer-Verlag. *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers,* Pittsburgh, Pennsylvania., May 1998.

[10] Loyall J, Gossett J, Gill C, Schantz R, Zinky J, Pal P, Shapiro R, Rodrigues C, Atighetchi M, Karr D. "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems,* Phoenix, Az., April 2001.

[11] QuO Toolkit Users' and Reference Guides. http://www.dist-systems.bbn.com/tech/QuO/release/

[12] Schantz R, Zinky J, Karr D, Bakken D, Megquier J, Loyall J. "An Object-level Gateway Supporting Integrated-Property Quality of Service", *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.

[13] Vanegas R, Zinky J, Loyall J, Karr D, Schantz R, Bakken D. "QuO's Runtime Support for Quality of Service in Distributed Objects," *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.

[14] Webber F, Pal P, Schantz R, and Loyall J. "Defense-Enabled Applications," *Proceedings of the Second DARPA Information Survivability Conference and Exposition (DISCEX II)*, Anaheim, CA, 12-14 June 2001.

[15] Zhang L, et al. "RSVP: a New Resource Protocol," *IEEE Network*, 7(6), September 1993.

[16] Zinky J, Bakken D, Schantz R. "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems* 3(1), 1997.