Building Adaptive Distributed Applications with Middleware and Aspects[†]

Gary Duzan
BBN Technologies
10 Moulton Street, MS 6/3D
Cambridge, MA 02138, USA
+1 617-873-7676
gduzan@bbn.com

Joseph Loyall
BBN Technologies
10 Moulton Street, MS 6/3D
Cambridge, MA 02138, USA
+1 617-873-4679
jloyall@bbn.com

Richard Schantz
BBN Technologies
10 Moulton Street, MS 6/3D
Cambridge, MA 02138, USA
+1 617-873-3550
schantz@bbn.com

Richard Shapiro
BBN Technologies
10 Moulton Street, MS 6/3D
Cambridge, MA 02138, USA
+1 617-873-2390
rshapiro@bbn.com

John Zinky
BBN Technologies
10 Moulton Street, MS 6/3D
Cambridge, MA 02138, USA
+1 617-873-2561
jzinky@bbn.com

ABSTRACT

Middleware technologies allow the development of distributed applications without explicit knowledge of the networking involved. However, in the face of changing network and CPU conditions across the distributed system, these applications often will need to adapt their behavior to maintain an acceptable quality of service (QoS), which implies a knowledge of these conditions. This adaptation is neither part of the application logic nor part of the distribution middleware, and so represents a separate concern which needs to be addressed.

This paper describes an aspect-based approach to programming QoS adaptive applications that separates the QoS and adaptation concerns from the functional and distribution concerns. To simplify aspect development for these applications, our approach integrates a domain-specific adaptation specification with a novel aspect language which includes distribution and adaptation-specific join points in its join point model. We compare and contrast this with existing aspect-oriented language approaches and illustrate our approach with an example distributed system application.

Categories and Subject Descriptors

C.2.4 **[Distributed Systems]**: Distributed Applications – *quality of service management.*

D.3.3 **[Programming Languages]**: Language Constructs and Features – *aspects, contracts.*

General Terms

Management, Measurement, Performance, Design, Languages.

Keywords

Aspect-oriented programming, adaptive quality of service, distributed objects, middleware, CORBA.

1. INTRODUCTION[†]

1.1 Distributed Objects

The distributed object (DO) model, as embodied in architectures such as CORBA [12], abstracts out object location information, so that a DO programmer has no knowledge of whether a given object is local or remote. This allows the programmer to model the application in terms of objects and focus on program logic instead of the details of communication.

However, remote invocations do in fact exhibit different behavioral characteristics than local invocations. For example, a remote method call can take much longer than a local one, a remote call might fail when a local call would not, and making remote calls can expose security issues which would not arise when making local calls. The degree to which these factors impact the application can also vary significantly at run time. For example, the delay introduced by a remote call can change due to network load, changes in network topology, and load on the remote system.

Conversely, the distributed application's impact on its environment will also vary over time. In order to maintain a stable environment it may be necessary to modify the application's behavior to match the circumstances. For example, under heavy network load conditions it would make sense for applications with

[†] This work was supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under contract F33615-00-C-1694. Approved for Public Release, Distribution Unlimited.

lesser importance to switch to an operational mode that consumes fewer network resources while still providing acceptable results.

The application might be able to deal with some of these quality of service (QoS) issues if it knew about them, but they are hidden by the distributed object model.

1.2 Adaptive Quality of Service

The initial temptation for the application programmer when encountering these issues is to modify his code to examine its surroundings and take the appropriate measures. Unfortunately, this requires the application programmer to violate the distributed object model, and adding the necessary extra code tends to tangle the QoS management logic with the application logic. In addition, without tool support, implementing the monitoring, decision making, and adaptations can be quite complex, affecting the application code in a number of different places. This leads to the observation that QoS management is a crosscutting concern.

A number of efforts have focused on providing QoS management and control in the infrastructure, e.g., the operating system [6][8] or network [5][20], which controls the QoS on behalf of the applications running within it. However, this has only been partially successful because different applications can have very different QoS requirements. For example, applications such as military systems, air traffic control, telemedicine, emergency response, and industrial production, require strict controls, with little tolerance for delay or degradation. On the other extreme, applications operating in wide-area environments (e.g., the Internet) which only expect "best-effort", may be able to provide better service by adapting to provide predictable, or at least gracefully degraded, service. Applications can compete for more resources than are available for them, and not all applications (even of the same class) are equal. QoS management in the infrastructure is often lacking the application knowledge necessary to make proper QoS decisions.

1.3 Quality of Service as an Aspect

Because managing a system's quality of service in a dynamic, adaptive way requires knowledge of both the applications and infrastructure, *middleware*, which lies between the applications and the infrastructure, is the proper place to insert it. However, it is not sufficient to do this statically since the QoS management will depend upon design, configuration, deployment, and runtime information.

Our approach is to build QoS management as an aspect using specialized tools and weave the result into the boundary between the application and the middleware. The aspect code can then have access to application-specific, middleware, and system information and controls without having to modify either the application logic or the underlying distribution middleware (i.e., the ORB). To help facilitate the development of adaptive QoS aspects, we define an aspect model which includes join points specific to distribution and adaptation that are not available in traditional Aspect Oriented Programming (AOP), and an adaptation model which defines the adaptation strategy.

Our prototype implementation is the QuO Toolkit, illustrated in Figure 1, which has been described in previous papers [9][10][17][21].

The QuO Toolkit provides support for building systems with adaptive QoS, i.e., systems which change their behavior in response to changes in the environment in order to provide the best possible results to the application. There are four main entities in the QuO model: Contracts, which are used to define adaptation policy; System Condition Objects, which are used to monitor the environment; Callbacks, which are used for middleware, system, and out-of-band application adaptation; and Delegates, which are used for in-band application adaptation. Delegates define the aspect-oriented weaving of the adaptive behavior into wrappers around application interfaces, while the other elements define the adaptation itself and the interaction with the surrounding system. When these adaptive behaviors are combined to meet an overall systemic goal, the result is a distributed QoS aspect.

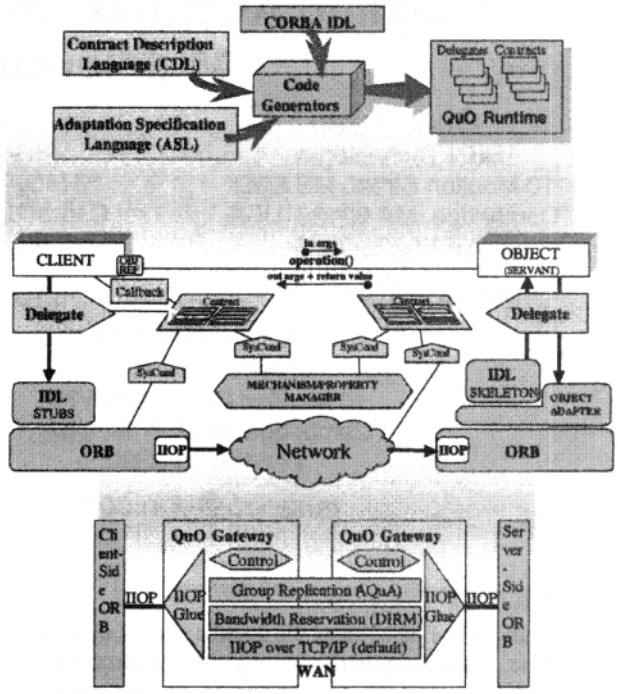

Figure 1: The Open-source QuO toolkit includes aspect languages, code generators, and a runtime system.

2. The QuO Adaptation Model

In order to provide the proper context, we first present the adaptation model, then describe how the adaptation is integrated into the application as an aspect.

2.1 Contracts

The central concept in defining an adaptation policy in QuO is the *contract*. The contract defines a number of *states* or operating *regions* which represent the potential states of the system from a particular point of view, along with criteria for transitions among the states and optional side effects of the transitions. Regions and states may be nested to provide a hierarchical view of system state. A contract can capture a set of statically "negotiated" regions of operations between objects, i.e., the level of service a client object expects to see and the level of service a server object expects to provide, as well as a set of "reality" regions representing the actual levels of service that the system can

encounter at runtime. Contracts are defined using QuO's Contract Definition Language (CDL) [14].

2.2 System Condition Objects

System condition objects (*sysconds*) provide interfaces to monitor and control low-level details of the system. The contract uses sysconds as variables in predicates to determine its current state or region. Sysconds can be selectively monitored by the contract so that changes in syscond values will trigger a reevaluation of the contract, which results in a reevaluation of the predicates and the execution of any defined *transition actions* if the resulting state is different from the previous state.

2.3 Callbacks

In some cases it is desirable to have side effects occur when transitioning between contract regions or states. These side effects are encapsulated in *callback objects* which can be invoked by the contract in transition actions. Callbacks may be used to change attributes of the application, in which case they are application specific, or to affect the operation of the OS or middleware, in which case they are application independent.

2.4 Kernel

At run time, contracts and sysconds are contained in a QuO *kernel* object, which provides a CORBA interface to maintain their life cycle. If desired, the kernel can live in a separate process, decoupling it from the application. This is most useful for out-of-band reactive contracts which monitor system conditions and manage system controls in response. More detail about the QuO runtime kernel is available in [19].

2.5 Qosket

A collection of contracts, sysconds, callbacks, and support code, which are designed to meet some overall system goal is called a *qosket*. Qoskets are designed to be reusable, and so are not bound to any particular application or application interface. More detail about qoskets is available in [18].

3. The QuO Aspect Model

In order to deal with the particular needs of distributed applications which require adaptation, we introduce an aspect model which deals with these issues more directly than aspect models from traditional AOP.

3.1 Delegate

In some cases it is necessary to monitor and/or control the behavior of an application as it is executing in order to maintain its quality of service policy. To accomplish this goal in a way that is integrated with the policy, the monitoring or control behavior is described in QuO's Aspect Specification Language (ASL). The ASL is then compiled to produce a *delegate*, which acts as a proxy for calls to an object reference or a servant (the *remote object*), and adds the desired behavior to its invocation. A single ASL specification may define behaviors for multiple interfaces, which makes it a cross-cutting specification. Likewise, code from many ASL specifications can be woven together into a delegate.

Delegates are particularly appropriate for introducing aspects to middleware-based applications since they operate at the boundary between the application and the middleware.

```
behavior RSSBehavior ()
{
 // Instance variable declarations..
 octet[] examples::bette::SlideShow::read (in long gifNumber)
{
  return_value octet[] result;
  local instr::Trace_rec rec;
  after METHODENTRY {
   methodID = "read";
   rec = rssQosket.createTraceRec(methodID);
  }
  inplaceof METHODCALL {
   region slow {
    java_code #{
      iServer =
        (com.bbn.quo.examples.bette.SlideShowInstrumented)
          rssQosket.getInstrumentedServer();
    }#;
    instrumented_result = iServer.read(gifNumber, rec);
    result = instrumented_result.getBytes();
    rec = instrumented_result.getRecord();
   }
  }
};
```

Method

Join Point

Advice

Region

Advice

Figure 2: Example ASL code, with aspect-oriented features noted.

3.2 Advice Model

The advice model in QuO's ASL is illustrated in Figure 2 and described in the following paragraphs.

3.2.1 Method

Advice in ASL is applied to a particular method of a particular interface as specified in CORBA IDL.[1] Delegates can be generated for alternate middleware such as RMI, or using no middleware at all, but currently there must be a CORBA IDL interface (for use by the QuO code generator) that corresponds to the desired target interface. Using a common method signature format does introduce some restrictions on the methods which can be matched (e.g., IDL doesn't support overloaded methods), but it allows ASL to maintain language independence and simplifies the implementation.

3.2.2 Join Points and Pointcut Designators

By referencing a method in the ASL, the method call join point is implicitly expanded to define additional join points, specified by the following primitive pointcut designators, or *places*:

- METHODENTRY
- PREMETHODCONTRACTEVAL
- METHODCALL
- POSTMETHODCONTRACTEVAL
- METHODRETURN

The default behavior at the METHODCALL join point is to invoke the method on the target object normally. The default behavior at the PREMETHODCONTRACTEVAL and

[1] Multiple methods may be specified by the use of wildcards.

POSTMETHODCONTRACTEVAL join points is to evaluate the delegate's associated contract and determine the current region.[2] The default behavior at METHODRETURN is to return the value obtained from the invocation of the target object. The METHODENTRY join point has no default behavior.

Each of these places may be identified as residing on the client side, the server side, or both (the default).

3.2.3 Advice

A piece of advice is defined to execute BEFORE, AFTER, INPLACEOF, or ONEXCEPTION for a particular place. Advice is defined using a simple language which allows variable assignments, method calls, arithmetic expressions, and exception generation, where the types of variables, objects, and exceptions are as defined in CORBA IDL. This allows the same advice to be used in a Java or C++ delegate. When language-specific actions are required, advice may also be specified in native Java or C++ code.

3.2.4 Execution Environment

The advice has access to all the arguments of the method, plus any variables that are defined in the ASL. These variables may be bound by support code to any object with a CORBA interface, which provides access to arbitrarily complex behaviors. Also, the remote object reference and the return value of the call to the remote object may be assigned names so that they can be accessed and/or modified by the advice.

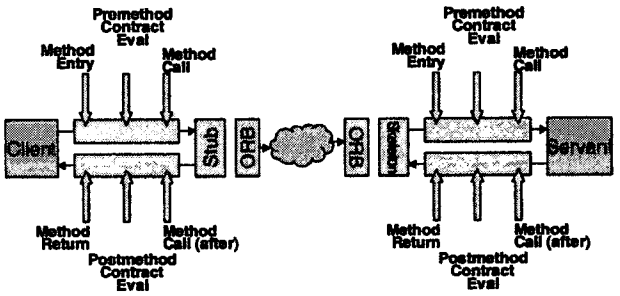

Figure 3: QuO includes join points that are elements of the distribution of objects.

3.2.5 Contract Integration

To produce truly adaptive behavior, pieces of advice may be defined to be conditional on contract regions. The results of the PREMETHODCONTRACTEVAL may be used AFTER PREMETHODCONTRACTEVAL or for METHODCALL advice; the results of the POSTMETHODCONTRACTEVAL can be used AFTER POSTMETHODCONTRACTEVAL or for METHODRETURN advice. This allows advice to be woven into a join point defined by

(a) The method (remote or local) being called;

(b) The step in the remote method invocation process currently being performed; and

(c) The external condition of the system's environment in which the call is being made or serviced (i.e., QoS).

3.2.6 Delegate Templates

QuO supports reusing advice on different methods by the use of *delegate templates*, where the behavior is defined using the same ASL language, but with template arguments substituted for the method signature. The behavior is specialized by referencing the name of the template and a concrete method signature in the body of another behavior, in which case the specialized template behavior is added to the host behavior. Delegate templates may be packaged with the qosket since they are not bound to any particular application interface.

3.2.7 Aspect Weaving

For the case in which multiple unrelated behaviors are desired in the same interface, QuO will accept multiple ASL specifications at once and weave them into a single delegate. If the order of execution of the woven advice is important (e.g., due to interdependencies), it can be defined explicitly in the configuration at the ASL file or advice granularity, using a "weave" statement in a configuration file or at the command line. The default weave order is in the order in which the aspect files are parsed.

3.3 Discussion of the QuO Join Point Model

The QuO join point model is consistent with the definitions in traditional AOP [7]0 in that it defines specific points in the runtime behavior of a program at which cross-cutting advice can be invoked. However, it extends the join point model provided by many instantiations of AOP, which have concentrated their join point models on elements of the *functional* language semantics, such as control flow, data flow, and field accesses. The QuO aspect model goes beyond this to span the distribution of remote method calls. For example, AspectJ [3] inserts advice at points in the execution of a functional program (i.e., method calls and field accesses in Java program execution), ignoring *how* the execution occurs. In contrast, the join points associated with QuO's ASL language associate advice with the distribution of method calls, as illustrated in Figure 3. This complements the use of other AOP languages, which can be used to weave cross-cutting functionality into application functional components, by providing the ability to weave in aspects between these components as they are distributed throughout a system.

In addition, QuO elements, e.g., contracts and sysconds, provide the ability to effectively weave code between the application and the system in which it is executing. This allows aspect-oriented insertion of cross-cutting behaviors affecting the interactions between the program execution and the execution of the rest of the elements of the system, environment, and infrastructure in which the program is executing. To the extent that some of these interaction points are in the execution path of the program, they could be considered additional join points.

Some of these interactions are outside the standard join point model definition, since the interactions can be asynchronous from the execution path of the program. For example, QuO contracts and sysconds could recognize and respond to overload in the system, regardless of the current state of the program execution. Furthermore, the response could affect the future execution of the program, regardless of where in the execution the program is currently. This combination of systemic programming and the

[2] Contract evaluations may be omitted if the QuO code generator determines that the results are not required.

aspect weaving described above enables the separate programming of QoS adaptive behaviors that cross-cut the functional dominant decomposition of the program by influencing the application behavior only, the system only, or both the application and the system.

```
module Document {
  typedef string Query;
  typedef string DocumentData;
  typedef sequence<octet> Image;

  interface Server {
    DocumentData get_document(in Query q);
    Image get_image(in Query q);
  };
};
```

Figure 4: Document Server IDL

4. Example Application

The following is an example of how to use the QuO Toolkit to build a QoS adaptive aspect.

4.1 Scenario

As an example of how to add adaptive quality of service to a distributed object application, let's imagine a document server with the interface in Figure 4 specified in CORBA IDL. The `Server` interface allows the client to query the server for document or image data using a textual query language, e.g., in XML, and receive a text document or binary image in response. Our challenge is to provide reasonable service under a variety of network and CPU load conditions. Our basic strategy will be to monitor the network and the CPU, operate normally if sufficient network bandwidth is available, compress images if the network is loaded and we have CPU time available for the compression, and give up if neither resource is sufficiently available.[3]

4.2 Adaptation Definition

The first step in defining a QoS adaptive aspect is to determine an adaptation policy which describes these conditions, as described in the contract in Figure 5 using QuO CDL. The sysconds in the contract reflect the environment in which the application is operating: the `avail_network` syscond will contain a value which reflects the network bandwidth available to the application at this point in time[4], the `heavy_net_load` syscond reflects the network load which is considered "heavy", and so on. The region definitions then define the policy in terms of these syscond values. Note that the regions are nested, so the `CriticalLoad`, `Compress`, and `NoCompress` subregions are only taken into account if the `HeavyLoad` region is active. In addition, the transition definition causes the `sysop` callback object to be invoked automatically when the contract enters the `CriticalLoad` region.

4.3 Aspect Definition

Once the adaptation policy is in place, we need to bind it to the application using an aspect. An aspect to bind an image compression policy to the application is defined in QuO ASL in Figure 6. First, we declare a `qosket` object, which provides services for use by the resulting delegate, and an instance variable (`ivar`) which refers to the `allow_compression` syscond in the contract on the server. Then we specify that we want the

```
contract DocServerResources (
  syscond quo::ValueSC quo_sc::ValueSCImpl avail_network,
  syscond quo::ValueSC quo_sc::ReadOnlyValueSCImpl heavy_net_load,
  syscond quo::ValueSC quo_sc::ReadOnlyValueSCImpl critical_net_load,
  syscond quo::ValueSC quo_sc::ValueSCImpl avail_cpu,
  syscond quo::ValueSC quo_sc::ReadOnlyValueSCImpl compression_cpu,
  syscond quo::ValueSC quo_sc::ValueSCImpl allow_compression,
  callback SystemOperator sysop )
{
  region Normal ((long) avail_network > (long) heavy_net_load) {}
  region HeavyLoad ((long) avail_network <= (long) heavy_net_load) {
    region CriticalLoad ((long) avail_network <= (long) critical_net_load) {}
    region Compress ((boolean) allow_compression and
                     (long) avail_cpu > (long) compression_cpu) {}
    region NoCompress (true) {}
    transition any -> CriticalLoad { synchronous { sysop.notifyCritical(); } }
  }
}
```

Figure 5: QuO Contract in CDL, defining resource based adaptation policy

advice to be associated with the `get_image` method, as defined in the IDL, by presenting the method's signature and enclosing the advice in a subsequent block, much like a method definition in C++ or Java. We need to be able to refer to the value being returned by the server in the advice on this method, so we declare

[3] This is a simplified version of strategies that we have prototyped that utilize combinations of CPU, network, and data management to deal with variations in resource availability and contention. Giving up in extreme conditions helps prevent overload conditions from becoming worse by eliminating load from relatively unimportant activities or from futile activities, i.e., those that are destined to fail.

[4] The available network and CPU values are updated dynamically by some external mechanism or by specialized sysconds.

```
behavior CompressImage ()
{
  qosket CompressImageDelegateQosket qk;
  ivar onserver quo::ValueSC allow_compression;

  Document::Image Document::Server::get_image(in Document::Query q) {
    return_value Document::Image rval;

    inplaceof PREMETHODCONTRACTEVAL onclient {}
    before METHODCALL onclient {
      q = qk.add_allow_compression(q);
    }
    after METHODENTRY onserver {
      allow_compression.booleanValue(qk.get_allow_compression(q));
    }
    after POSTMETHODCONTRACTEVAL onserver {
      region NetworkLoad {
        region CriticalLoad {
          throw CORBA::NO_RESOURCES;
        }
        region Compress {
          rval = qk.compress_image(rval);
        }
        region NoCompress {
          throw CORBA::NO_RESOURCES;
        }
      }
    }
  }
};
```

Figure 6: QuO Aspect in ASL, defining application adaptation

a `return_value` variable, which specifies the name of the variable used to hold the return value.

The `inplaceof PREMETHODCONTRACTEVAL onclient` advice prevents the contract from being evaluated before the method call, since we don't check the contract state on the client.

The `before METHODCALL onclient` advice passes the query string to the qosket object, which adds additional information on the compression requirements to the query string and returns the modified string. By rebinding the parameter variable, `q`, the call to the server will now pass the new value.

The `after METHODENTRY onserver` advice passes the query to the qosket object, which extracts and returns the compression requirement, which in turn is stored in the `allow_compression` syscond. This value, which originated in the client advice, is used in evaluating the contract on the server. Since the server has advice both before and after the method call, there are contract evaluations generated for both, as well. The pre-method contract evaluation doesn't have any associated regions in the ASL, so it is only evaluated for its side effects, namely notifying the system operator when transitioning to the `CriticalLoad` region as specified in the CDL.

The post-method contract evaluation, on the other hand, is used to select the behavior in the `after POSTMETHODCONTRACTEVAL onserver` advice. The regions specified in the ASL are matched against the regions in the CDL, including the nesting, and if the region of the most recent contract evaluation matches the advice region, that advice is executed. If no ASL region matches the current contract region, then the default behavior is executed. So in this case, if the contract is in the `Normal` region, no advice is matched, and the default action (in this case, "do nothing") is executed. If the contract is in the `CriticalLoad` or `NoCompress` region, the advice throws an exception. If the contract is in the `Compress` region, the image data to be returned is passed to the qosket, which compresses it and returns the compressed image. The result is used to rebind the return value variable, which causes the new value to be returned to the client.

Other aspects can be defined in a similar way and combined with this one. For example, under critical network load conditions, the server could strip all image references from a document before returning it to the client to prevent future image requests. This would be defined in a separate ASL file since it is a separate aspect, and may be applied independently of the aspect in Figure 6. The existing contract would be used in both cases since it already includes the desired load policy.

4.4 Tools

Once the IDL, CDL, and ASL are defined, they are run through the QuO code generator to produce the contract, kernel, client and server delegates, and an adapter to tie the delegates to the contract. These combined with implementations for the qosket and callback objects result in object wrappers which can be dropped into the client and server to add the defined behavior.

5. Discussion and Related Work

5.1 The Aspect Oriented Nature of QuO

Opinions vary as to what exactly makes a system aspect oriented. For the purposes of this discussion, we adopt the definitions and discussion points presented by Filman and Friedman [4] which present AOP as a combination of *quantification*, or application to multiple points in the code, and *obliviousness*, meaning that the base code (or other aspects) can be written without awareness of the aspect code.

In QuO there are two places where weaving takes place: the weaving of aspects to form a delegate and the weaving of a delegate into the base code. The former is statically quantified

over calls to methods defined in IDL interfaces and a number of "places" in the execution of the method, and may be dynamically quantified over contract states. In this case the weaving is oblivious, with the selection of aspects and the weave order for a particular delegate being determined by an external configuration mechanism. The latter is statically quantified over objects with the appropriate IDL interface. In this case the weaving is only partially oblivious, since the base code must be modified to add a delegate wrapper to the base object, but the users of the wrapped object are still oblivious to the existence of the delegate once it is in place since it inherits from the same class or interface as the original. Future versions of QuO may utilize more advanced techniques to weave the adaptive behavior into the application, making its integration more properly oblivious to the application programmer.

Another criterion that is often given for Aspect Orientation is *cross-cutting*. In QuO, the weaving of the aspects into delegates is cross-cutting since it allows specification of behaviors for multiple classes or interfaces in a single aspect. While the weaving of the delegate into the base code is local to a particular object, this can be viewed as an implementation mechanism since the real behaviors are specified in the ASL.

QuO also includes the notion of a Qosket, which is an encapsulation of code which implements some QoS management behavior. The code is then distributed to various points in an application system (using weaving or other composition patterns) in such a way as to produce a coordinated result. This code is typically applied to different classes, processes, and nodes, so the Qosket is clearly cross-cutting across the implementation, composition, and deployment epochs. As a result, a Qosket encapsulates a higher level, systemic, aspect and all the code needed to implement it.

5.2 Composition Filters

QuO delegates are similar to Composition Filters [1][2] in that they both weave code into the application by wrapping the target object and intercepting messages to it. Composition Filters allow the developer to filter method calls and process them by creating a declarative filter specification which is applied to the reified call. QuO delegates only match method calls by name; however, the actions to be taken can be selected in a declarative way by referencing contract regions in QuO ASL, allowing the delegate to easily change the method's behavior based on QoS state. (The delegate can examine the method parameters if necessary, but this must currently be done in procedural code within the ASL.) Weaving multiple ASL specifications into a single delegate is roughly analogous to composing filters; both build more complex behaviors from separately specified, simpler behaviors.

5.3 AspectJ/AspectC++

QuO does not modify application code to introduce an aspect as AspectJ [3] does, choosing the simpler object wrapper model provided by delegates. However, the way advice is woven into the delegate is similar to AspectJ and it uses a method signature syntax similar to that of AspectJ. Like AspectJ, each join point can have advice woven BEFORE, INPLACEOF (i.e., around), AFTER, or ONEXCEPTION (similar to after() throwing()). Each method has five static join points:

- METHODENTRY
- PREMETHODCONTRACTEVAL
- METHODCALL
- POSTMETHODCONTRACTEVAL
- METHODRETURN

QuO ASL also allows the developer to distinguish between the client and the server side. So for each AspectJ join point corresponding to a method call, QuO introduces ten join points corresponding to places in the distribution of the method call. In addition, region designators can be viewed as introducing dynamic join points since the choice of whether to execute the advice depends on run time state. Advice can be expressed in a simple domain-specific language, native Java, or native C++. QuO delegates currently have no facilities for directly adding members to classes as AspectJ and AspectC++ do, though in some cases it can be simulated by wrapping the objects and maintaining additional state in the wrapper.

5.4 ACT

The Adaptive CORBA Template [16] effort from Michigan State University improves the weaving of QuO delegates into applications by using CORBA Portable Interceptors [12]. The result is that the application becomes oblivious to the particulars of the delegates being introduced. There is still some application code required to add the ACT interceptor, but it is only applied once in the main program. The ACT approach utilizes a CORBA-specific feature, and so is not applicable to other middleware such as RMI.

6. Conclusions

Introducing distribution to an application also requires introducing adaptation to maintain the quality of service requirements of the application and the overall system. This adaptation is a separate concern, so it is appropriate to implement it as an aspect. This particular type of aspect needs to introduce code in places defined by distribution and adaptation, so having join points which correspond to these places simplifies the aspect development. Since adaptation is a common pattern and can be difficult to implement, separating it from the aspect code and implementing it using specialized tools simplifies the aspect implementation.

The QuO Toolkit is a prototype implementation of these ideas that has been used to introduce adaptive quality of service to a variety of distributed applications.

7. Future Directions

Efforts are underway to bring the advantages of adaptive QoS to component-based development as a complement to the current distributed object support. The primary target will be the CORBA Component Model [13], but alternatives such as Boeing's Bold Stroke component model [15] will be investigated as well. This effort will likely explore the weaving of QoS components into an application's component assembly specification in an aspect oriented way.

There is also an effort to move towards more model-based development, which would allow models of applications to have QoS annotations and provide for transformations of the QoS model into adaptive QoS code. Some of the existing efforts in aspect modeling may be useful in this area.

One possible enhancement to the QuO tools would be to use an existing advanced weaving technology (e.g., AspectJ/AspectC++) to introduce the adaptive behavior into the application. That would allow us to maintain our higher-level aspect model while requiring fewer changes to the application.

8. Acknowledgements

Thanks to Michael Atighetchi for his comments and technical assistance in preparing this paper.

9. REFERENCES

[1] M. Aksit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. ECOOP '92, LNCS 615, Springer-Verlag, pp. 372-395, 1992.

[2] M. Aksit, B. Tekinerdogan, and L. Bergmans. Achieving Adaptability through Separation and Composition of Concerns, in Special Issues in Object-Oriented Programming. M. Muhlhauser (Ed.), dpunkt verlag, pp. 12-23, 1996.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18-22 June 2001. http://citeseer.nj.nec.com/kiczales01overview.html

[4] R.E. Filman and D.P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf

[5] IETF, An Architecture for Differentiated Services, http://www.ietf.org/rfc/rfc2475.txt

[6] Michael B. Jones, Daniela RoÒu, MarcelC tlin RoÒu, CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, In Proc. of the 16 th ACM Symposium on Operating System Principles, St-Malo, France, pp. 198-211, Oct. 1997.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In ECOOP'97-Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220-242, 1997.

[8] C. L. Liu and James W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM (JACM)*, Volume 20 , Issue 1 (January 1973), 46-61.

[9] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, K. Anderson, 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 (LCR98)*, Pittsburgh, Pennsylvania, 28-30 May 1998.

[10] J. Loyall, R. Schantz, J. Zinky, D. Bakken, Specifying and Measuring Quality of Service in Distributed Object Systems, *Proceedings of The 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98)*, 1998.

[11] Submitted by I-Logix Inc., THALES, and Tri-Pacific Software Inc. UML(TM) Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms: Initial Submission. http://cgi.omg.org/cgi-bin/doc?realtime/02-09-01.pdf

[12] Object Management Group. The Common Object Request Broker: Architecture and Specification, 3.0 edition, June 2002.

[13] Object Management Group. CORBA Components. OMG Document formal/2001-11-03, 2001.

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

[15] W. Roll. Towards Model-Based and CCM-Based Applications for Real-Time Systems, *Proceedings Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '03), Hokkaido, Japan, May 14-16, 2003.*

[16] S. M. Sadjadi and P. K. McKinley. ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation. Technical Report MSU-CSE-03-22. ftp://ftp.cse.msu.edu/pub/crg/ACT-techrep.pdf

[17] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, J. Loyall, "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.

[18] R. Schantz, J. Loyall, M. Atighetchi, Partha Pal, "Packaging Quality of Service Control Behaviors for Reuse, *Proceedings of the 5th IEEE International Symposium on Object-Oriented distributed Computing, (ISORC 02)*, Washington DC, April 29-May1 2002.

[19] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, "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.

[20] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network*, September 1993.

[21] J. Zinky, D. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems, vol. 3, num. 1, 1997.