

# Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration\*

Partha Pal, Joseph Loyall, Richard Schantz, John Zinky, Rich Shapiro, and James Megquier

BBN Technologies

10 Moulton Street

Cambridge, MA 02138

{ppal, jloyall, schantz, jzinky, rshapiro, jmegq}@bbn.com

## Abstract

*Recent work in opening up distributed object systems to make them suitable for applications needing quality of service control has had the side effect of increasing the complexity in setting up, configuring, and initializing such applications. Configuration of distributed applications is more complicated than that of non-distributed applications, simply because of the heterogeneous and distributed nature of the application's components. CORBA and other distributed object middleware simplifies the configuration of distributed object applications, but hides much of the information and control necessary to achieve quality of service (QoS). This paper describes the techniques and tools that we have developed within our Quality Objects (QuO) framework for simplifying the configuration of distributed applications with QoS attributes. We describe a QuO configuration language, as well as the specific configuration needs of particular QoS properties – real-time, security, and dependability – and the support we provide for them.*

## 1. Introduction

The heterogeneous and distributed nature of computing resources increases the complexity associated with developing distributed applications. It also increases the complexity of configuring such applications, since configuration of a distributed application consists of locating, creating, instantiating, and initializing (possibly geographically remote) components written in different languages and executing on different platforms.

Distributed object middleware, such as CORBA, has alleviated much of the complexity associated with developing distributed applications, by hiding implementation details behind functional interfaces. It has also attempted to lighten the burden of configuring such systems by providing

configuration services, such as the naming service and directory services. However, while hiding the complexity and implementation details, distributed object middleware also hides much of the information needed to support the quality of service (QoS) needs of many applications, such as real-time, multimedia, and financial applications. Moreover, in wide-area distributed environments, system properties are more dynamic and unpredictable, and also more likely to change from configuration to configuration. In order to field a distributed application over a wide-area network, the usage patterns, the QoS requirements, and the underlying resources must be carefully measured and controlled.

Over the past few years, we have been developing the Quality Objects (QuO) framework [20], which supports the development of distributed applications with QoS requirements. QuO opens up the implementation of a distributed object application, enabling the specification, measurement, and control of the QoS aspects of an application and the specification and implementation of adaptive behavior in response to changing system conditions. The QuO framework and current CORBA-based QuO prototype includes a *QuO toolkit*, which provides software development tools for developing QuO applications and includes a suite of QoS Description Languages (QDL), a runtime kernel, and reusable libraries of system condition objects for measuring and controlling QoS. The QuO toolkit is used by *QoS developers*, who are responsible for developing the QoS contracts, system condition objects, and adaptable behavior and who supplement the traditional roles of application developers and mechanism developers.

We have developed a *Configuration Setup Language (CSL)*, a component of QDL that supports the configuration<sup>1</sup> of QuO applications. Using CSL, a QuO developer can specify the configuration of QuO applications at a high

---

\*This work is sponsored by DARPA under contract nos. F30602-97-C-0276 and F30602-98-C-0187 and monitored by US AFRL.

---

<sup>1</sup>In the context of this paper, we define *configuring* or *setup* as a sequence of actions that occurs at the beginning of the execution of a program, after which the main functionality of the application can be commenced. By the same token, *configuration* refers to the state of connectedness in which activating the functionality becomes possible.

level. Code generators generate creation, initialization, and setup code and weave it into the runtime components of the QuO and CORBA application. The goal of CSL is to separate the configuration concerns from the main functionality of the application, in the same way other QDL components separate the QoS concerns from the main functionality. This separation of concerns, a staple of aspect-oriented programming [9], eases the programming burden and improves the maintainability and portability of the system.

Specific dimensions of QoS, such as real-time, security, and dependability, often have special-purpose configuration needs, in addition to those of general QuO applications. For example, a real-time application might require locating or creating real-time schedulers and event channels, and might require initializing real-time application attributes such as priority. Furthermore, alternative distributed application topologies, such as mesh or pipeline, require different, possibly more involved, configurations. In many cases, CSL is powerful enough to support the special-purpose configuration of specific QoS properties. In others, such as real-time and security, we have chosen to supplement CSL with a property specific configuration language.

In a series of earlier papers [11], [12], [18], we have described QuO, its runtime architecture and how code generators weave QDL specifications into the runtime components of a QuO application. This paper discusses the newest important component of the QuO framework, i.e., the CSL configuration language. Section 2 gives a brief overview of the QuO framework. Section 3 describes the CSL, how it is used to configure QuO applications, and how QuO currently supports runtime dynamic reconfiguration. Section 4 discusses some of the dimension and architecture specific configuration support that we've examined, including support for setting up the real-time event channels in the TAO real-time ORB [16]. Section 5 describes current open issues and work in progress. Section 6 describes related work. Finally, Section 7 provides some concluding remarks.

## 2. Overview of QuO

We give a brief overview of QuO in this section. More detailed discussion can be found in [20], [11], [18], [12] and [15]. Figure 1 illustrates the QuO framework.

The center of Figure 1 illustrates a client-to-object logical method call. In a traditional CORBA application, this method call includes the client, orb proxy (i.e., the stub), orb, network, orb, orb proxy (i.e., the skeleton) and the remote object. A method call in the QuO framework also includes the following components, illustrated in Figure 1:

- *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.

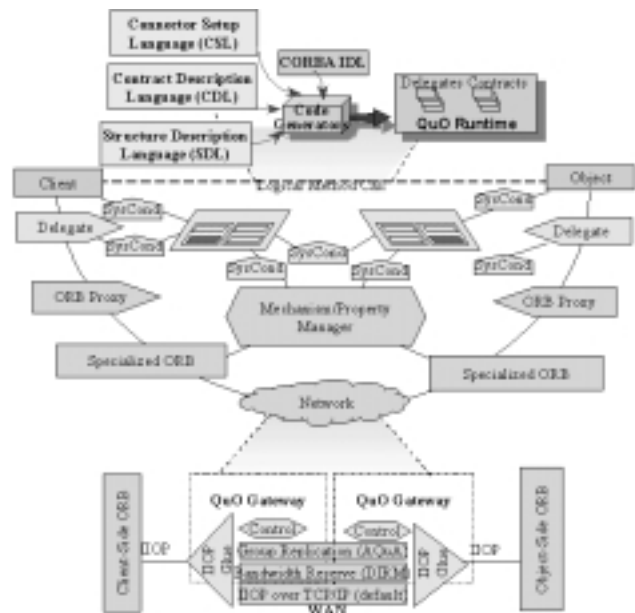


Figure 1. Architecture of the QuO Framework

- *Delegates* act as local wrappers for remote objects. Each delegate provides an interface identical to that of its remote object, but provides locally adaptive behavior based upon the current state of QoS in the system.
- *System condition objects* interface to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO contracts.

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 [1]) 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 in place at lower levels in the protocol stack. To support this, QuO includes a gateway mechanism [15], illustrated in the magnified portion at the bottom of Figure 1, that enables special purpose transport protocols and adaptation below the ORB.

In addition to traditional application developers (who develop the client and object implementations) and mechanism developers (who develop the ORBs, property managers, and other distributed infrastructure), development of QuO applications involve another group of developers, namely QoS developers. QoS developers are responsible for developing QuO contracts, system condition objects, call-back mechanisms, and object delegate behavior. To support the added role of QoS developer, we are developing a QuO toolkit (described in earlier papers such as [11],[18] and [12]), illustrated in the magnified portion at the top of Figure 1, 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 Structure Description Language, SDL). CDL and SDL are described in [11, 12].
- 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 [18].
- *Code generators* which weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program. Runtime integration of QDL specification is discussed in [11].

In previous QuO releases, QDL did not provide help for hooking together the various components of a QuO application. Instead, the configuration and setup steps were written manually in the application code. The following sections introduce the latest members of the QDL suite that are designed to address the configuration and setup issues of QuO adaptive distributed applications. CSL (Connector Setup Language) is the general purpose QDL component that is used by QoS developers to specify the start up configuration of a QuO application. In addition, depending on the specific property and underlying mechanism that is of interest to the QuO application at hand, QoS developers may need additional QDL components to specify the property specific parameters required for the initial set up. So far, we have encountered this need for the realtime and security dimensions. We have been developing a Realtime Setup Language (RSL) and are working on integrating TIS's DTEL++ access control specification language. For other properties managed by QuO, like dependability and bandwidth, the general purpose QDL languages have been sufficient.

### 3. Using QDL to Configure QuO Applications

A simple CORBA application consisting of a single client and a single remote object, is typically set up using several configuration steps, such as the following, before a remote call can ever be made:

- On the server side: initialize ORB, obtain an Object Adapter (POA or BOA), create the object, register the object with the object adapter, and publish its IOR (e.g., by writing it to a file).<sup>2</sup>
- On the client side: initialize ORB, obtain the IOR, resolve the IOR and narrow the resolved object into a local pointer.

<sup>2</sup>There are alternate ways to configure a CORBA application, such as using a naming service rather than publishing the IOR.

These operations must be carried out in the proper order, which in some cases is obvious (e.g., the ORB should be initialized before it is used) and in others is not so obvious (impl\_is\_ready should be called after object\_is\_ready).

A QuO application requires the configuration of several additional application components:

- Instantiating system condition objects that measure or control relevant QoS parameters
- Instantiating the contract(s) describing the possible states of the system
- Instantiating the delegate(s) associated with each stub and skeleton
- Hooking these up with the client and object.

The following sections describe how a QoS developer uses the CSL component of QDL to specify the configuration and setup of QuO applications and how the SDL component of QDL is used to specify strategies for runtime reconfiguration of QuO applications as part of their adaptive behavior. More detail about the syntax and semantics of CSL, as well as an example CSL specification and the Connector code generated from it are in Appendix A. The remainder of this section will use and refer to the code in Appendix A as an example. For a more complete description refer to the CSL documentation [2].

#### 3.1. Configuration and Setup of the Client Side



**Figure 2. Example configuration of the client side of a QuO application**

Figure 2 illustrates a representative client side configuration. The QuO client obtains a reference to a QuO delegate, which presents the same signature as the remote object stub, but can choose the specific behavior to invoke (encoded using SDL) based upon the current state of the system (as represented by one or more contracts) whenever a method call

or return occurs. Delegates can be stacked, as illustrated in Figure 2, with each delegate corresponding to a contract representing a different aspect of QoS in the system. For example, one contract might measure the round trip response time of method calls while another might represent the security requirements of the application and the security capabilities of the system components. Each contract uses system condition objects to measure and control system resources, conditions, and properties. These system condition objects can be nested (e.g., to compute higher level measures from lower level ones) and can be shared by contracts (e.g., more than one contract might need data from the system clock).

Therefore, the client side of a QuO application is configured in the following way:

- The client has a reference to the topmost delegate.
- Each delegate has references to delegates beneath it or remote object stubs (if it is a bottom-most delegate).
- Each delegate has a reference to the contract with which it is associated.
- Each contract has references to the system condition objects it uses to measure and control the QoS parameters in the environment.
- Each contract has references to callback objects used to inform the client about contract region transitions.

The QuO programmer specifies this configuration using CSL. The QDL code generator generates C++ or Java code, implementing a *connector* class. A connector is essentially a delegate with enough knowledge to hook up with the other objects that it needs. The connector class is a subclass of the class for the topmost delegate. It has a *connect* method that implements the required configuration/setup process. The client programmer simply has to replace the code that would normally find the object stub (in a non-QuO CORBA application) with code to create a connector instance and call connect. The following is the client code corresponding to the Connector specified in Appendix A:

```
CounterClientConnector c = new CounterClientConnector();
c.connect();
```

In the above code fragment, CounterClientConnector is the class name of the connector object. In this simple example, the client application does not initiate an ORB, it relies on the connector object to supply one if needed. In many cases, a client application instantiates an orb and passes it to the connector as an argument to the connect call, as illustrated in Section 3.2. Basically, specifying a Connector in CSL consists of specifying the following:

- The IDL, CDL, and SDL files containing the specifications of the CORBA interfaces, system condition

objects, callback objects, contracts, and delegates involved in the configuration.

- How to locate or instantiate these objects.
- How each of these objects are related, i.e., hooked up.
- A set of function or method calls to invoke after all the objects have been instantiated and hooked up.

CSL offers a number of options for instantiating or locating objects, including creating new instances, reading IORs from files, and passing object references into the Connector as arguments. In addition, CSL allows the programmer to export an instance once it is created, e.g., by writing its IOR to a file. This enables Connectors to share objects such as system condition objects.

Hooking up objects is specified by passing one object as an argument in the creation of another. For example, a CDL specification for a contract class takes the contract's external system condition objects as arguments to the contract constructor. Thus, specifying that a contract uses a system condition in CSL consists of instantiating (or obtaining a reference to) the system condition object, then instantiating the contract with the system condition object as an argument. The following code fragment provides an example in which a system condition object and a callback object are created and hooked up with a contract that uses them:

```
/* Instantiate System Condition Objects */
ValueSC countDirection = new ValueSCImpl ( ... );

/* Instantiate Callback Objects */
CounterCallback theCallback =
    new CounterCallbackImpl ( ... );

/* Instantiate Contracts */
quo::Contract contract1 =
    new CounterContract ( "Counter Contract" , ... ,
        countDirection , theCallback );
```

The following code illustrates how a toplevel delegate is created. Other, nested delegates are created similarly, but using the *makedelegate* command.

```
returndelegate thisDelegate(thisRemoteObj, contract1);
```

### 3.2. Configuration and Setup of the Server Side

On the server side things are slightly different. The server side delegate is a legitimate CORBA object with its own IOR. However, it still needs all the QuO objects, similar to the client side delegate, in order to be able to perform QoS monitoring and adaptation. A QuO server application creates an instance of the connector class, which is a subclass of the server side delegate, and then calls connect on it, followed by the appropriate call to *obj\_is\_ready* and *impl\_is\_ready*, as shown in Figure 3.

Notice that, in Figure 3, the server implementation initializes the ORB and passes it into the Connector. CSL

```

ORB orb = ORB.init();
BOA boa = orb.BOA_init();
...
CounterServerConnector co =
    new CounterServerConnector();
co.connect(orb);
boa.obj_is_ready(co);
...
boa.impl_is_ready();

```

**Figure 3. Use of server side connector**

provides features to allow the programmer to specify the parameters to the connect method, as follows:

```

connectparams (in CORBA::ORB o)
use o as orb

```

The *connectparams* declaration specifies the signature of the connect method generated from the CSL specifications. If there is no *connectparams* declaration, then the generated connect method has no arguments (as in the client side example). The *connectparams* declaration consists of a parameter list whose syntax follows that of CORBA IDL parameter declarations. The *use* declaration specifies if the argument is to be treated specially by the CSL processor. For example, the *use* declaration above indicates that CSL processor should not generate code to initialize an ORB instance and instead should use the one that is passed in.

The body of the generated *CounterServerConnector* class is similar to the *CounterClientConnector* class, except that this class extends a server side delegate. The CSL specification is also similar to the CSL specification of the client side connector, except for the following differences:

- The CSL attribute that specifies the target for the generated code specifies “server” instead of “client”
- The server side delegate is instantiated as an implementation of an IDL interface. Therefore the delegate specification statement takes one fewer argument – it doesn’t require the argument which, on the client side, identifies the stub (or lower level delegate).

The implementation (specified by SDL) of the server side delegate might still deliver a message to alternate remote objects. If it does so, the CSL specification specifies these remote objects.

### 3.3 Extending CSL

CSL is intended to simplify the configuration of QuO applications. We have not tried to make it a full-fledged programming language, but a useful specification covering the most common patterns of QuO application configuration. This doesn’t limit its power, however, since the capability of a connector can be extended beyond what is generated from CSL. This is done by creating a subclass of the generated connector class that performs the additional functionality desired. For example, the QoS developer can define a

new connect method with a different signature or additional functionality than the generated one. The application calls the connect of the subclass, which performs its additional functionality and calls the superclass, i.e., generated, connect to perform the configuration.

It is also possible to use specialized application or property specific initialization mechanisms via CSL. One way is to instantiate other CORBA objects in the CSL specification or pass objects in as arguments to the connect method. The last section of the CSL specification can then specify methods to invoke on these objects. The objects don’t have to be hooked up to any other objects in the CSL specification. As an example, the following passes in an instance of a Java class *DataBaseServer* and calls *init* on it, even if nothing else in the CSL specification refers to the instance.

```

/* in the Connector Attribute Spec. Section */
connectparams(in DataBaseServer sc)
...
/* in Funccall Section */
sc.init();

```

## 3.4 Runtime Reconfiguration

So far we have seen how to set up the initial configuration. But during the execution of a QuO application the configuration may change in various ways as part of QoS driven adaptation. In this section we assume some familiarity with SDL and CDL.

### 3.4.1 Logical Reorganization

Even though a QuO client application has a reference to a single delegate, it is still possible that the client’s remote method invocations will be serviced at multiple objects and/or different objects at different times. This can be done without having the client stop and restart with a different configuration. The client will always pass its remote invocation to the delegate, but the delegate may have multiple remote objects to dispatch to. The following SDL declaration shows a delegate representing the interface named *Inventory* with an alternative remote object that implements the interface *SecuredInventory*:

```

Delegate behavior for interface Inventory and
contracts InventoryContract is
SecuredInventory alternateRemoteObj bind;

```

As a result of the *bind* declaration, the delegate class will have a class member variable named *alternateRemoteObj* and a *set* method with the following signature:

```

void set_alternateRemoteObj(SecuredInventory x);

```

that sets the *alternateRemoteObj* variable. The following CSL statements ensure that the delegate is properly hooked up with the alternative remote object during the initial set up:

```

/* obtain the remote object */
Inventory remoteObj = fileior("inventoryobj.ior");
/* obtain the alternative remote object */
SecuredInventory altRemoteObj =
    fileior("securedinventoryobj.ior");
...
/* set up the delegate representing the remote
object of type Inventory */
returndelegate thisDelegate(remoteObj, ...);
...
/* set up the alternative remote object in
the delegate */
funccall thisDelegate.set_alternateRemoteObj(altRemoteObj);

```

The following SDL fragment is an example of redirection. If the `add` method (a method defined by the `Inventory` interface) is invoked while the system is in the `Access_Suspect` region, it will be dispatched to the alternate object implementing `SecuredInventory` rather than the normal object that implements `Inventory`.

```

Call add:
Region Access_Suspect:
    pass to alternateRemoteObj;

```

The following SDL fragment shows how multiple dispatch is specified. `Normal` is the name of a contract region, `add` is a method defined in `Inventory` interface, and `register` is a method in `SecuredInventory` interface. The SDL specifies that when `add` is invoked in the `Normal` region, the delegate does two dispatches: first it invokes the `register` method on the alternate remote object implementing `SecuredInventory`, passing in the string "add" followed by all the other actual parameters of the call, and then it invokes the `add` method on the remote object it represents (implementing `Inventory`).

```

Call add:
Region Normal:
    pass to alternateRemoteObj by register
        with this_method_params.prepend("add");
    pass_through;

```

### 3.4.2 Physical Reorganization

In some cases, a QuO application will want to alter its configuration at runtime. For example, an intrusion aware application which detects that a server object may have been attacked may want to break the connection to that server and locate a server that has not been compromised.

We currently support runtime reconfiguration in SDL as part of adaptation. For instance, assume that a delegate starts up connected to a remote object named *A*. The following SDL specification specifies that when the contract region indicates an `Abnormal` condition, the delegate should throw away its connection to *A* and connect instead to a different remote object named *B*. When the contract indicates that things are `Normal` again, the delegate rebinds itself again to *A* and starts dispatching calls there.

```

Call read:
Region Normal:

```

```

rebind with name A;
pass_through;
Region Abnormal:
rebind with name B;
pass_through;

```

In the context of managed bandwidth, an application could start using a reserved channel if the contract indicates that available bandwidth is lower than desired. This run time reorganization makes use of the QuO gateway and RSVP/DIRM bandwidth management mechanisms and is described elsewhere [1]. Similarly, in the context of the dependability property, an application could start using a different number of replicas on a different set of machines when the contract indicates that the desired number of replicas cannot be sustained in the original set of hosts. This example makes use of the QuO gateway and the Proteus dependability management mechanism and is also described elsewhere [3], [14].

## 4. Property and Architecture Specific Configuration and Set up

### 4.1. Dependability Using Reliable Group Communication

The AQuA project [3] is providing dependability mechanisms and policies for the QuO framework. One of the main goals of the AQuA project is to provide support for a wide spectrum of dependable systems. Towards this end, it is supporting the development of many different operating points trading off performance versus dependability among others. Specifically, the AQuA architecture supports a spectrum of choices across the following dimensions:

- Replication style: active (all replicas receive messages and send replies), semi-active (same as active, but only leader replies), and passive (checkpoint and restart)
- Voters: both the location (client or server side) and the algorithm (pass up first, wait for majority etc.)
- Intra-group communication style: leader for the group, or symmetric (no leader)
- Group to group communication style: how to send messages from the client group to the server group (ordering, send to all or some etc.)

To support all these, AQuA uses the Maestro/Ensemble [10] reliable group communication framework. We have developed a QuO dependability gateway [15] for AQuA that hides much of the complexity of the Maestro/Ensemble facilities and services. Setting up a dependability oriented QuO application requires the following steps:

- Specify the dependability requirement by means of contracts and system conditions that interact with the Proteus dependability manager [14], define adaptive behavior, and use the QuO code generators to generate the contracts, delegates and connectors.
- Start up the Ensemble gossip server (a directory service used by Ensemble). Indicate which hosts to use and which objects should be replicated and start the Proteus dependability manager. Start Proteus factories on the indicated hosts. These are done off line.
- The application obtains a gateway and then requests a replicated object.

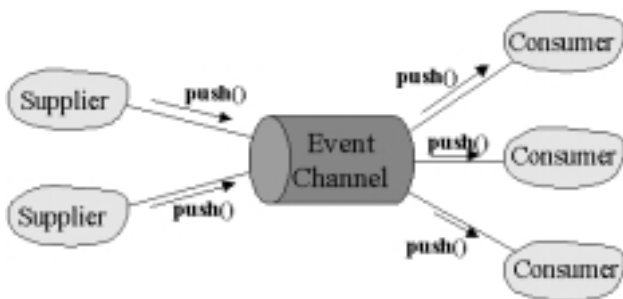
The gateway returns an object reference to the application and manages the IIOP to Ensemble translation. After the initial set up, the contract and system conditions dictate how many replicas are available and where they are located via the Proteus dependability manager.

## 4.2. Realtime

Our initial support for real-time QoS has concentrated on integrating with, and supporting, the TAO real-time ORB [16]. TAO provides support for real-time scheduling of events using a real-time event channel model.

### 4.2.1 The Real-time Event Channel Model

The CORBA Event Service is a CORBA Object Service (COS) that provides a model for asynchronous communication among objects that is an alternative to CORBA's oneway invocation model. In the CORBA Event Service the traditional CORBA client-server model can be seen as a supplier-consumer model, where one or more suppliers generate events that are delivered to one or more consumers asynchronously, without the suppliers and consumers having to be aware of one another.



**Figure 4. The event channel push model**

The TAO Event Channel extensions to the CORBA Event Service add features for supporting real-time applications, including event filtering, event correlation, and pe-

riodic event processing. Whereas the CORBA Event Service supports both the push and pull models, the TAO Event Channel model, described in [5, 7] and illustrated in Figure 4, concentrates on the push model. In the push model, suppliers initiate the transfer of events to consumers. Suppliers generate events, push them to an event channel, which in turn delivers them to consumers. Because of this, our integration efforts have concentrated on the push model and on the supplier side of the event channel hookup.

The event channel (EC) model has similarities to the group communication multicast, except that the EC is anonymous (i.e., the clients do not have references to, nor really know how many server objects) and supports heterogeneous objects (i.e., non-replicas) on each end of the EC. Likewise, the EC performs asynchronous message delivery, similar to the CORBA oneway call. However, oneways are still one-to-one and the client must have a reference to the server object stub, whereas ECs allow multicast (really many-to-many), anonymous communication.

### 4.2.2 Configuring a Real-time Event Channel

Configuring a real-time event channel (EC) in TAO consists of the following steps:

- creating a servant to implement the scheduling service
- creating the EC and registering it with the name service
- defining the events that each supplier will generate and their associated real-time attributes
- defining the events that each consumer receives and their associated real-time attributes
- connecting the suppliers and consumers to the EC

After this configuration is done, the suppliers and consumers generally enter processing loops: the supplier generating events and delivering them to the EC (by calling a *push* method on the EC) and the consumer receiving events (through a *push* method called by the EC). The real-time scheduler schedules event delivery according to the real-time attributes associated with each supplied event, e.g., an event of higher priority will be delivered before one of lower priority in a rate monotonic scheduling (RMS) strategy.

Real-time QoS attributes of events, for both supply and consumption, are specified by populating an *RT\_Info* structure. *RT\_Info* contains the following fields:

- Criticality - a measure of priority used by the scheduler
- Worst Case Time - the maximum execution time needed by an operation
- Typical Time - the time that an operation normally requires

- **Cached Time** - provided in case an operation can provide a cached result in response to service requests
- **Importance** - a lower order measure of priority used to break ties in scheduling
- **Period** - the minimum time between successive iterations of the operations
- **Quantum** - the maximum time that an operation can run before preemption if operations are time-sliced

The current version of TAO requires that values for all of these attributes be specified for each supplied event and for each consumed event, even though only a subset of the attributes are used for the supplier and for the consumer in each scheduling strategy (i.e., RMS, maximum urgency first, minimum latency first, etc.). For example, in RMS, the scheduler uses the consumer’s criticality and importance to schedule event delivery. The supplier must also supply the criticality and importance attributes of RT\_Info, however those values are not used. The consumer’s value for criticality, worst case time, typical time, cached time, and importance are used by the scheduler in different scheduling strategies, while the supplier’s period is used. Currently, quantum is not used by any scheduling strategy.

In addition, in the current version of TAO events must have unique names on the supplier and the consumer side. The consumer must correlate a supplied event with a consumer event, and with a method that needs to be invoked.

We saw an opportunity to move the programming of much of the real-time event channel configuration up to a higher level. Much of the code to create and set up the scheduler and event channel is common from one application to another. In addition, the association of events with suppliers, consumers, and QoS information is lower level than an application developer is necessarily interested in, requiring the specification of unused RT\_Info data on the supplier and the consumer side and requiring the creation of multiple unique names for each event.

We have been developing a real-time specific component of QDL, called real-time specification language or RSL. RSL allows a QoS programmer to specify the following:

- A set of events
- A set of event suppliers, the events each will generate, and the period at which the event will be generated (or that the period will be specified at runtime)
- A set of event consumers, the events each consumes, and any or all of the criticality, worst case time, typical time, cached time, and importance values (or that the values will be specified at runtime).

The RSL code generator generates the code to create the scheduling service servant, code to create the event channel and register it with the name service, and routines to initialize each server and consumer. These routines will generate unique names for the events, encode dummy values for any RT\_Info attributes not specified in the RSL specification (including the unused attributes), specify the values for attributes specified in the RSL, and put the runtime specified attributes into the signature of the routine.

```

RT_SPECS QuO_RT_Class {
  EVENTS { event1, event2, event3 }
  EC_SUPPLIER supplier1_setup_routine {
    GENERATES { event1, event2 }
    APPLICATION_SPECIFIED { period }
  }
  EC_CONSUMER consumer1_setup_routine {
    CONSUMES { event2 }
    APPLICATION_SPECIFIED { criticality, importance }
  }
  EC_CONSUMER consumer2_setup_routine {
    CONSUMES { event1 }
    APPLICATION_SPECIFIED { }
    CONSTANT { criticality = very high;
               importance = very low; }
  }
}

```

**Figure 5. Example RSL specification**

For example, the RSL specification in Figure 5 generates a class named QuO\_RT\_Class that contains (along with the scheduler and event channel setup code) three methods. The first method, `supplier1_setup_routine`, sets up a supplier that generates event1 and event2 (although it will create unique names for them) at periods that will be passed into the method when it is called at runtime. The second method initializes a consumer that consumes event2 with a criticality and importance passed in at initialization time. The third method initializes a consumer that consumes event1 with a very high criticality and a very low importance<sup>3</sup>. Each of these three methods will encode dummy values for all the other attributes of RT\_Info.

### 4.3 Dataflow and Pipeline Applications

We have been investigating applying QuO in the context of pipeline or data flow architectures, where information is processed at various processing elements (PE) as the data flows through them. Initially, our focus has been limited to applications with PEs such that the quality of work in one pass through a PE depends on how much resource the PE spends during that pass (like CPU power, bandwidth, or even time). Image filtering or signal processing systems are potential candidates: a quick pass produces a bad quality picture or fails to eliminate most of the noise, whereas a good filtering is generally expensive. Our goal is to employ such a system to an incoming stream of varying quality

<sup>3</sup>TAO recognizes criticality and importance values of VERY\_HIGH, HIGH, MEDIUM, LOW, and VERY\_LOW.

(e.g., frames may have different image quality or different levels of noise) and control the level of effort expended by the PEs using QuO. For example, if one incoming frame has a low level of noise then the PEs will spend less effort trying to filter it, but if the next frame has high noise, the PEs will start spending more effort. This could be done without QuO, but only with the measurement and control of the quality aspect of the application coded in the system in an ad hoc manner. Alternatively, if the measurement and variation is not encoded in the application, the PEs will always do a fixed amount of work aiming to produce images of a consistent quality. This seems akin to a system with one single operating region: if the resources are not sufficient for operating in that region, the application will simply fail. With QuO, we have the ability to easily define various modes of operation depending on the availability of resources, so that the application can still operate when resources become scarce. Various realtime command and control applications have similar stream oriented architectures. It is our goal to investigate how QoS-driven adaptivity inter-operates with realtime responsiveness of such systems.

In our initial experiments we are using a source and a sink with PEs between them. The flow of data is carried out by a sequence of oneway calls that originate from the source and terminate at the sink. QuO middleware is inserted in the middle of each such oneway call and depending on the quality of the flow, it is either fed forward or fed back. Initial set up and configuration is carried out by individually setting up the source, PEs, and the sink. Each such individual setup is similar to traditional QuO client or server set up.

The pipeline model is similar to the realtime event channel described in the previous section, except that the EC is like a pipeline of length one, or a single link in a pipeline.

## 5. Open Issues and Work In Progress

There are various open issues that we are still investigating, a few of which we will discuss here.

### 5.1. Fundamental Issues

The QuO framework provides a programming language reference to a connector and mimics a reference to an object implementing a particular interface. In reality, a connector is an entry point to an interconnected structure that may contain other entry points which may or may not be externally accessible. Each such entry point presents some facet of QoS awareness of the objects below it in the interconnected structure. The interesting questions for us involve the management of the facets of QoS awareness that are encoded in a connector: What is the identity of an entry point? What happens when the holder of a reference representing

an entry point passes it to another application? What happens if that application does not know anything about QuO or the context of the entry point? If we dump the IOR of a connector and resolve it, how much of the interconnected structure does it refer to? Answers to these questions depend on many of the difficult issues of naming, contexts, persistence, and sharing of structures. The approaches that we choose to take will impact how easily QuO applications will be configured and set up, how much QoS internal structure is dynamically sharable, and how QuO applications will cooperate and coexist with non-QuO applications.

### 5.2. Mechanism Specific Issues

Mechanism specific configuration and setup needs are addressed at the QDL level only in the context of realtime and security. Both are works in progress with different degrees of maturity. In the realtime context we actually have an initial version of a QDL component which separates the realtime aspects from application programming and maps between the application's view of realtime event channel setup to the lower level mechanisms of TAO. However, there is still work to be done in investigating the proper attributes and controls to project up to the application level for different scheduling strategies and for other real-time models (beyond event channels) such as end-to-end or round-trip real-time requirements.

In the security area, we use COTS languages and tools such as DTEL to specify security aspects such as access control, and produce code that we link in with the application. We also use CSL features to incorporate components specifically initialized for security (for instance an ORB with SSL interceptors) in QuO applications. Ultimately we aim to devise a QDL component to specify security aspects.

Use of replicated objects further complicates the general issue of the identity of the connector. In addition, there is another issue involving contracts in the presence of replication, are server side contracts replicated along with objects?

### 5.3. Architecture Specific Issues

One issue with dataflow/pipeline applications involves the localized scope of CSL. CSL describes one connector, which may be sufficient for simple applications where QuO is used only on the client or server side. With QuO on both sides we need two CSL specifications. In the data flow or pipeline context there will be a proliferation of CSL specifications. Another issue in this context involves measuring the quality of work performed by the PEs. So far the QoS aspects of an application measured and monitored by QuO are ambient qualities like security, bandwidth or dependability. We see the need to be able to specify and monitor the quality of outputs produced in the system as well.

## 6. Related Work

Our research in the QDL suite of languages has been based upon the concepts of aspect-oriented programming [9] and open implementation [8]. Other researchers in QoS have seen the need for QoS languages and concepts like contracts for QoS [19, 4]. In addition, there are other groups looking at providing real-time QoS control in distributed object systems [17] and looking at the issues of dependability and replication [13].

The QDL languages, and especially CSL, are similar in objectives to scripting languages [6]. That is, they are higher-level languages, but are simpler and therefore do not have the learning curve and development overhead of typical compiled, programming languages, such as C++ and Java. In many ways, CSL is reminiscent of Unix Make. While it is certainly true that parts of CSL's job could be performed using languages like Perl or Make, the distributed object, adaptive nature of QuO applications is not necessarily a natural fit to the text processing and file-based nature of these languages.

## 7. Conclusion

As part of our ongoing research in Quality Objects, we have investigated the difficult problem of configuring and reconfiguring adaptive, distributed object applications with QoS control. We have developed a configuration language, CSL, which simultaneously supports the specification of the components, configuration, and initialization of a QuO application, and simplifies the development of QuO applications by coordinating and driving the QDL code generators. We have used CSL to build real-time, security, bandwidth management, and dependability examples. A few of these, including real-time and security, have also required the development or use of mechanism specific languages.

## 8. Acknowledgements

The authors would like to gratefully acknowledge the contributions of Michael Atighetchi, of Bill Sanders, Michel Cukier, Jennifer Ren, and Paul Rubel of the University of Illinois, and of Doug Schmidt, Fred Kuhns, Chris Gill, David Levine, and Carlos O'Ryan of Washington University, St. Louis, to the research presented in this paper.

## References

- [1] BBN Distributed Systems Research Group, DIRM project team. Dirm technical overview. Internet URL <http://www.dist-systems.bbn.com/projects/DIRM>, 1998.
- [2] BBN Distributed Systems Research Group, QuO Project Team. The connector setup language reference manual. Part of QuO 2.0 release, Internet URL <http://bbn.dist-systems.com/Projects/QuO>, 1999.
- [3] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. E. Schantz. Aqua: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, October 1998.
- [4] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society Press.
- [5] C. Gill, T. Harrison, and C. O'Ryan. Using the real-time event service. Internet URL [http://www.cs.wustl.edu/schmidt/events\\_tutorial.html](http://www.cs.wustl.edu/schmidt/events_tutorial.html).
- [6] T. Griffin. Scripting languages. *Computer Bits*, 9(5), May 1999.
- [7] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA '97*. ACM, October 6-7 1997.
- [8] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 1996.
- [9] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.
- [10] S. Landis and S. Maffeis. Constructing reliable distributed communications systems with CORBA. *Theory and Practice of Object Systems*, 3(1), 1997.
- [11] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. Karr, R. Vanegas, and K. R. Anderson. Qus aspect languages and their runtime integration. *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components*, May 1998.
- [12] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of The 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98)*, April 1998.
- [13] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, April 1996.
- [14] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. Sanders, D. Bakken, and D. Karr. Proteus: A flexible infrastructure to implement fault tolerance in aqua. In *Proceedings of the IFIP International Working Conference on Dependable Computing for Critical Applications*, January 1999.
- [15] R. E. Schantz, J. A. Zinky, D. A. Karr, D. E. Bakken, J. Megquier, and J. P. Loyall. An object-level gateway supporting integrated-property quality of service. In *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.
- [16] D. C. Schmidt, D. Levine, and S. Mungee. The design of the tao real-time object request broker. *Computer Communications Special Issue on Building Quality of Service into Distributed Systems*, 21(4), April 1998.

- [17] E. Shokri, P. Crane, and K. Kim. An implementation model for time-triggered message-triggered object support mechanisms in corba-compliant cots platforms. In *Proceedings of ISORC '98 (IEEE CS 1st International Symposium on Object-oriented Real-time distributed Computing)*, pages 12–21, Kyoto, Japan, April 1998.
- [18] R. Vanegas, J. A. Zinky, J. P. Loyal, D. Karr, R. E. Schantz, and D. E. 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.
- [19] L. Welch and B. Shirazi. DeSiDeRaTa: QoS management tools for dynamic, scalable, dependable, real-time systems. In *Proceedings of IEEE WMDRTSS, 1997*.
- [20] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 1(3):55–73, April 1997.

## A CSL Syntax, Semantics, and Example Code

### A.1 CSL Example Code

The following is an abridged CSL specification. Statements relating to C++ code generation have been removed.

```

/*****
 * Connector Attribute Specification Section
 *****/

class          "CounterClientConnector"
interface "CounterClientConnectorInterface"
codeloglevel   "high"
target         "client"

/*****
 * Include Section
 *****/

include "qdl/CounterContract.cdl"
include "qdl/CounterDelegate.sdl"
include "idl/Counter.idl"
include "idl/CounterCallback.idl"

/*****
 * Object Definition Section
 *****/

/* Define Remote Objects */
Counter thisRemoteObj = fileior ("Counter.ior" );

/* Instantiate System Condition Objects */
ValueSC countDirection =
    new ValueSCImpl ( "Count Direction" ,
                    "com.bbn.quo.ValueSC" ,
                    "com.bbn.quo.ValueSCImpl" ) ;

/* Instantiate Callback Objects */
CounterCallback theCallback =
    new CounterCallbackImpl ( "CountDirCallback" ) ;

/* Instantiate Contracts */
quo::Contract contract1 =
    new CounterContract ( "Counter Contract" ,
                        "com.bbn.quo.examples.simple.CounterContract" ,
                        countDirection , theCallback ) ;

```

```

/* Instantiate Delegates */
returndelegate thisDelegate ( thisRemoteObj ,
                             contract1 ) ;

/*****
 * Function Call Section
 *****/

dumpior countDirection "CountDirection.ior" ;

```

### A.2 CSL Syntax and Semantics

CSL descriptions consist of four sections, described in the following paragraphs.

The *Connector Attribute Specification* section describes programmer specified attributes about the generated Connector code. Example attributes include *class*, *interface*, *codeloglevel* and *target*. The *class* attribute specifies the (programming language) class name of the generated connector class. The *interface* attribute specifies the name of the interface that the connector class implements. Both the connector class and the interface are produced by the code generator. *Codeloglevel* is a debugging flag, specifying the level of debugging output in the running connector, either *none*, *low*, or *high*. Finally, *target* is a way to specify whether connector code is being generated for the client side or the server side. In the example CSL above, the value of target is *client* since it specifies a client side connector.

The next section, the *Include* section, specifies the IDL and QDL files that need to be processed by the QuO code generator to produce the contract, delegate, and connector code.

The third section, the *Object Definition* Section, is the section in which you specify the CSL objects that need to be instantiated, how they are initialized, and how they are hooked together. For a client side connector the following CSL objects (and their interconnection) are normally required:

- the delegate(s) and the remote object the delegate represents;
- the contract(s) used by the delegate(s); and
- the system conditions and callbacks that are needed by the contract(s).

In the example CSL above, the first statement in this section declares that an object of type *Counter* is to be located by reading its CORBA IOR from the file “Counter.ior”. The local variable *thisRemoteObj* will refer to the object during the remainder of the CSL file. The next statement declares a system condition object called *countDirection*. *ValueSC* and *ValueSCImpl* are an IDL type and a Java class, respectively, provided by the QuO libraries. The next statement creates a callback object. *CounterCallback* and *CounterCallbackImpl* are an IDL type and a Java class specific

to this application. The next statement describes the QuO contract used in this example. All contracts are of type *quo::Contract* but are instantiated using the name specified in the CDL file, in this case *CounterContract*. A contract instantiation always takes two strings as arguments, plus the system condition objects and callback objects described in the CDL code. The first argument identifies the contract as named as “Counter Contract”. The second argument specifies the Java class of the contract. The third and fourth arguments hook the system condition object, *countDirection*, and the callback object, *theCallback*, which we created above, into the contract. The final statements of the *Object Definition* section identifies the delegates that need to be created, the contracts they use and the CORBA objects they represent (or dispatch to). In this example, we create a single delegate that represents the *thisRemoteObj* remote object and uses the contract named *contract1*.

The final section of the CSL code is the *Function Calls* Section. This section specifies a set of function or method calls that are invoked after all the objects are created and instantiated. This gives the opportunity to perform some initialization on the objects as part of the connector code. In the example CSL, the call to the CSL primitive *dumpior* will write the IOR of the *countDirection* system condition object to a file.

```

com.bbn.quo.ValueSCHelper.narrow(
    com.bbn.quo.Connector.instantiateSysCond(
        quo_kernel,
        "CountDirection",
        "com.bbn.quo.ValueSC",
        "com.bbn.quo.ValueSCImpl"
    )
);
com.bbn.quo.Connector.writeIOR(
    countDirection,
    "CountDirection.ior"
);
com.bbn.quo.examples.simple.CounterCallback
theCallback = new CounterCallbackImpl(
    "Count Direction Callback"
);
com.bbn.quo.SysCond[] quo_contract1_sysConds =
    new com.bbn.quo.SysCond[1];
com.bbn.quo.Callback[] quo_contract1_callBacks =
    new com.bbn.quo.Callback[1];
quo_contract1_sysConds[0]=countDirection;
quo_contract1_callBacks[0]=theCallback;
com.bbn.quo.Contract contract1 =
    quo_kernel.bindContract(
        "CounterContract",
        "com.bbn.quo.examples.simple.CounterContract",
        quo_contract1_sysConds,
        quo_contract1_callBacks
    );
super.quo_remoteObj = thisRemoteObj;
super.quo_CounterContract = contract1 ;
}
}

```

### A.3 Java Code Generated from the Example

The Java class generated from the above specification is shown below.

```

package com.bbn.quo.examples.simple;
import java.io.*;
import com.bbn.quo.*;

public class CounterClientConnector
    extends CounterCounterContractDelegate_client
    implements CounterClientConnectorInterface
{
    public void connect()
    {
        org.omg.CORBA.ORB quo_orb;
        if(com.bbn.quo.Connector.orb == null)
        {
            String [] quo_args = new String[1];
            quo_args[0] = "dummyLocalOrb";
            java.util.Properties quo_props = new java.util.Properties();
            quo_props.put("ORBdisableLocator","true");
            System.err.println("calling orb.init ");
            quo_orb = org.omg.CORBA.ORB.init(quo_args, quo_props);
            com.bbn.quo.Connector.orb = quo_orb;
        }
        else
            quo_orb = com.bbn.quo.Connector.orb;
        com.bbn.quo.QuoKernel quo_kernel;
        quo_kernel =
            com.bbn.quo.QuoKernelHelper.narrow(
                com.bbn.quo.Connector.readIOR("QuoKernel.ior"));
        quo_kernel.setDebug( com.bbn.quo.QuoKernel.DEBUG_NONE);
        com.bbn.quo.examples.simple.Counter
            thisRemoteObj =
                com.bbn.quo.examples.simple.CounterHelper.narrow(
                    com.bbn.quo.Connector.readIOR("Counter.ior")
                );
        com.bbn.quo.ValueSC countDirection =

```