Architectural Support for Quality of Service for CORBA Objects

Theory and Practice of Object Systems, 3:1, April 1997, 55–73.

John A. Zinky
David E. Bakken
Richard E. Schantz

BBN Systems and Technologies
10 Moulton Street, Cambridge, Mass. 02138 U.S.A.

Email: jzinky@bbn.com, dbakken@bbn.com, schantz@bbn.com

CORBA is a commercial standard for distributed object computing which shows great promise in the development of distributed programs. Its interface description language (IDL) enables objects to be developed independently of the underlying programming language, operating system, or computer architecture on which they will execute. Programs deployed in a wide-area distributed system encounter conditions which are much more hostile and varying than those operating in a single address space or within a single local area network. In this paper we discuss four major problems we have observed in our developing and deploying wide-area distributed object applications and middleware. First, most programs are developed ignoring the variable wide area conditions. Second, when they do try to handle these conditions, application programmers have great difficulty because they are different from the local objects they are used to dealing with. Third, IDL hides information about the tradeoffs any implementation of an object must make. Fourth, there is presently no way to systematically reuse current technology components which deal with these conditions, so code sharing becomes impractical.

In this paper we also describe our architecture, Quality of Service for CORBA Objects (QuO), which we have developed to overcome these limitations and integrate their solution by providing QoS abstractions to CORBA objects. First, it makes these conditions first class entities and integrates knowledge of them over time, space, and source. Second, it reduces their variance by masking. Third, it exposes key design decisions of an object's implementation and how it will be used. Fourth, it supports reuse of various architectural components and automatically generates others.

This work is sponsored by Rome Lab.

Keywords: CORBA; quality of service, QoS, object oriented programming

1. Introduction

The development and deployment of distributed programs has become increasingly commonplace. Much of this has been made possible by the judicious use of *middleware*, a layer of software above the communication substrate which offers a consistent, higher-level abstraction throughout the network. One increasingly important category of distributed applications is multimedia applications, including video on demand. Such applications demand high- performance communication substrates. Simultaneously, these substrates are offering new features such as quality of service (QoS) and multicast, which these applications could exploit (Leslie et al. 1993; Partridge & Pink 1992; Topolcic 1990; Zhang et al. 1993). For example, QoS allows reservations with guaranteed *system properties*, operational attributes such as throughput and delay. However, these QoS features are offered at the communication substrate level, and new middleware is needed to translate these features to a form appropriate for the application level.

Other important kinds of distributed applications can benefit from QoS support in middleware. For example, a significant new kind of distributed application is collaborative planning (Burstein et al. 1995). These applications typically feature widely-dispersed people collaborating using, for example, a map as shared workspace, a video conference, and expert systems to develop a course of action. These applications are created out of many subcomponents which are integrated with CORBA, a commercial middleware standard for distributed object computing (Object Management Group 1995b). The interactions between these subcomponents feature a much wider spectrum of usage patterns and QoS requirements and operate in a much more hostile and varying environment than typical multimedia applications running across a local area network (LAN) or at most across one or two ATM switches, the worst-case environment assumed in current middleware. In the process of developing and fielding many such wide-area applications and middleware for them over the past twenty years (Akkoyunlu et al. 1974; Cosell et al. 1975; BBN 1981; Schantz et al. 1986; Gurwitz et al. 1986; Anderson & Flynn 1990; Walker et al. 1990; Walker & Dean 1995), we have observed that these systems have great difficulty adapting to the volatile *system conditions*, measured or observed manifestations of a particular system property, and to the relatively scarce resources typical in a wide area network (WAN) environment. They also have difficulty evolving over time to be deployed in different environments, e.g. to be developed on a LAN and migrate to a WAN or from a fixed site to mobile use. The root cause of these problems is middleware's lack of support for handling these environmental variables (system properties) such as QoS.

CORBA's interface description language (IDL) is an important base for developing distributed applications. IDL describes the *functional interface* to the object, the type signature of the operations which the object embodies, independently of the underlying programming language, operating system, and communication medium. Specifying only the functional interface allows distributed applications to be developed rapidly and transparently without regard to the underlying services. CORBA thus uses the traditional model of an interface that hides the implementation details. However, distributed applications based on CORBA's IDL operate acceptably only as long as resources are plentiful. For example, experience has shown that current CORBA implementations work well where objects are either local (in the client's address

space) or within the same LAN as the client, because the system properties of these environments are stable, well understood, and resources are plentiful (Bakken et al. 1996).

However, in wide-area distributed environments, system properties are more dynamic and hostile, 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 dealt with. Unfortunately, these features are precisely what is being hidden behind the functional interface described by IDL. Thus, to make a distributed application perform adequately and be adaptive, we need to specify more of the details of the design decisions embodied in an implementation of an object. Also, the implementation must be "opened up" to give access to the system properties of the CORBA ORB and objects. This enables an inexpensive and easy way to alter the implementation without sacrificing the software engineering gains from object-oriented techniques.

Another aspect of the problem is that current application programmers are not normally trained to deal with system properties such as those embodied in QoS, because the system properties of local objects are ideal. For example, local objects do not fail independently of a client once they are created, for all practical purposes, and the delay is negligible when invoking a method. Further, because of the simple resource model, programming languages mix together the functionality of the code and the optimization of resources. Unfortunately, the system properties of distributed objects are far from ideal: they can fail unexpectedly, and the delay for a method invocation to return may be long and have high variance. As a result, most invocations to remote objects in a typical distributed application are bracketed with extra code to handle errors and performance conditions. This coding is very difficult to do, for reasons we describe below, and it makes it even more difficult for the programs to be used in a different environment than which it was originally hardcoded for. Worse, it negates all of the value of the transparent interface definitions.

This paper describes key issues which must be addressed to support QoS at the CORBA object layer, especially across wide-area and mobile environments. These issues include synthesizing information about system properties, reducing their variance, exposing key design decisions which affect the suitability of an object for a given environment, and providing a framework to support reuse of code dealing with system conditions to make the handling of system properties feasible.

With these issues in mind, we have developed an architecture, Quality of Service for CORBA Objects (QuO), to support QoS at the CORBA layer. QuO extends the CORBA functional Interface Description Language (IDL) with a QoS Description Language (QDL). QDL specifies an application's expected usage patterns and QoS requirements for a connection to an object. The QoS and usage specifications are at the object level (e.g., methods per second) and not at the communication level (e.g., bits per second). An application can have many connections to the same object, each with different system properties. QDL allows the object designer to specify QoS regions, which represent the status of the QoS agreement for an object connection. The application can adapt to changing conditions by changing its behavior based on the QoS regions of its object connections. Finally, QuO provides mechanisms for measuring and enforcing QoS agreements and for dispatching handlers when the agreements are violated. These mechanisms

can help distributed applications be more predictable and adaptive even when end-to-end guarantees cannot be provided.

1.1 Technical Preview

Distributed Applications developed using current technologies are fragile for four major reasons. First, they are developed ignoring system properties or, at best, by hand coding a single set of assumptions about system properties for one environment. Second, programmers have difficulty handling WAN system properties because they are used to the system properties of local objects. Third, there is presently a large barrier to entry for creating even a minimally adaptive system because there is no information available about the intended usage of the object, the QoS it can strive to deliver, and the structure of the implementation. Fourth, programmers cannot create strongly adaptive systems with multiple implementations (each with different tradeoffs) because current technology does not support a framework for reuse which would make the development of multiple implementations and the comprehensive handling of system properties feasible.

QuO solves these problems by:

1. Making system properties first class entities, and integrating knowledge of them over time, space, and source so the application can have a feasible way to be aware of and handle changes in its environment.
2. Reducing the variance of the system properties which the programmer must deal with by masking.
3. Exposing the key design decisions of an object's implementation (which are currently hidden) and of how the object will be used, to help the application to reconfigure adaptively.
4. Supporting reuse of various QuO architectural components at multiple points in the application lifecycle.

We next describe distributed collaborative planning applications, and then discuss the above four items in more detail in the following sections.

2. Observations about Distributed Collaborative Planning Applications

Distributed collaborative planning applications (DCPAs) are an increasingly important category of distributed applications. For example, they have been deployed by BBN and others across WANs to support logistics for Operation Desert Storm and presently in Bosnia. DCPA style systems can be applied to other domains such as, a manufacturer and its suppliers devising a delivery schedule for their virtual corporation, a physician and a specialist jointly analyzing an X-ray, a video customer support link, an electronic relocation bureau, or a remote video teller (Phuah et al. 1996, Sasnett et al 1994).

Collaborative planning applications can be very complex, featuring dozens of people collectively performing many different tasks. As an example, the structure of the different kinds of interactions between just two of the participants in a typical collaborative planning application is given in simplified form in Figure 1. Here, two users are collaborating at different levels involving a video conference, a shared workspace, and scheduling algorithms. The video conference is used to exchange verbal communication, including synchronizing the workspace. The domain- independent shared workspace allows the scheduling algorithms to construct domain-specific graphical objects, and the collaborators to view, manipulate, and annotate these objects. The domain-specific, or even application-specific scheduling algorithms plan a given scenario and display their results on the shared workspace.

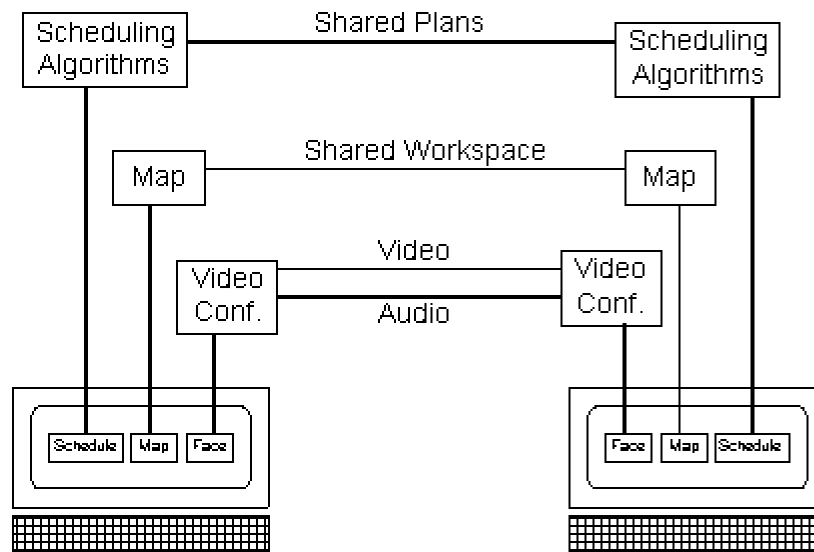

Figure 1: Collaborative Planning Usage Patterns

Each level in this example has its own client usage patterns and adaptivity requirements. At one extreme, the usage patterns of the video conference are knowable *a priori* but little variance can be tolerated if the program is to satisfy the user's expectations. The other extreme involves scheduling interactions, whose usage patterns are difficult or impossible to obtain ahead of time but whose delivery allows for significant variance. In between these two extremes of usage patterns is the shared workspace. These three levels also have different abilities to adapt to changing resources while satisfying the user's expectations. The audio portion of the video conference cannot tolerate a lower bandwidth or higher variance to be useful, while the users can generally tolerate the video portion being sent at a much lower rate or even dropped if resources become more scarce. Updates to the shared workspace can suffer additional delays under adverse conditions, but if the delay gets too long then the users will consider the updates untimely and the program's perceived usefulness will be sharply curtailed. The scheduling algorithms can endure much longer delays in worst-case scenarios, because users often will not be waiting for immediate feedback. Note that all these interactions share the same computational and communication resources, but have radically different usage patterns and system requirements.

2.1 Problems in Developing and Deploying Distributed Applications

DCPAs illustrate many of the problems inherent in using current technology to develop and deploy distributed applications across a WAN, because of their complex interactions. We have observed a number of these problems from developing and deploying collaborative planning and other distributed applications (Burstein et al. 1995; Anderson & Flynn 1990; Walker 1991) as well as with developing the Cronus and Corbus distributed object middleware (BBN 1981; Schantz et al. 1986; Walker et al 1990; Walker & Dean 1995), which was developed for use across WANs and has been used to field many such applications. Distributed applications tend to be fragile, e.g. they initially perform poorly and unpredictably when deployed; they are hard to move from the development environment to the field environment or to a different field environment; and components cannot be reused because they are tailored to a particular environment. This fragility is present largely because distributed applications are designed using functional interfaces, i.e. ones which ignore system properties such as throughput, delay, and availability. This functional interface represents significant progress over the old, tedious way of moving data across a network, but it is only half of the solution that is needed. The necessary other half is a way to describe and manage the non- functional aspects of a client-object interaction.

As a result of this limitation, programmers either ignore system properties altogether or handle them in an *ad hoc* manner with code fragments scattered throughout the program. When they do try to deal with them, programmers also have difficulty handling the system properties which distributed applications face because they are used to programming assuming the system properties of local objects. As a result, they can generally ignore system properties and thus program an invocation to a remote object much like they would one to a local object (one in the same address space; e.g., created with the C++ **new** keyword). Interactions with local objects are typically programmed implicitly assuming that they offer infinite bandwidth, no delay, and no variance. This systemic shortcoming has two facets. First, the base values of the system properties of local objects are better than that of remote ones. This problem can be overcome in special cases where the programmers have a priori knowledge of the worse system properties under which their code will have to operate and these properties remain relatively fixed. For example, programming for satellite communications is an example of an environment with difficult system properties (low bandwidth, high delay, and frequent failures), but one for which code can be programmed without great difficulty. Second, programmers have great difficulty writing code which can operate over a wide dynamic range of system properties, i.e. variance. As a result, distributed applications can not evolve well to new environments. For example, applications developed for a LAN usually perform poorly in a satellite environment because they were not designed to handle the poor system properties encountered there. Conversely, the same application developed for a satellite environment would also not perform well in a LAN because it would contain superfluous specialized code to compensate for poor system properties.

A compounding factor programmers encounter when they try to deal with system properties with current technology is that the information about these system properties is available at different times, in different locations, and from different sources. Programmers make commitments assuming system properties at various times, ranging from design time to invocation time. The knowledge of system properties is available at different locations in a distributed system, from

the client to the object to the communications substrate, and resources connecting them. This knowledge of system properties is produced by different participants, including the object designer, the client designer, the operations staff, and the end user of the distributed application. If these implicit commitments do not match the actual environment in which an object is used, then the application will perform poorly. For example, the object designer commits to an algorithm to implement a method at object design time based on the assumed environment in which the object will operate. However, the application may be deployed in an environment in which another algorithm for implementing the object's method would have been a much better choice.

A final problem with today's technology is that there is no way to reuse code dealing with system conditions. Calls to a functional interface tend to get bracketed with code to handle system properties. This code is almost impossible to reuse, so the task of trying to handle system properties makes software development and meeting deadlines even more difficult.

In summary, distributed applications today are not able to adapt well to changing system properties and resource availability when deployed, and they cannot evolve nearly as readily or easily as is desirable. What is needed is for middleware to support more than just the functional interface to an object, so that system properties can be treated as first class entities and managed accordingly.

2.2 Overview of the QuO Solution

QuO is an architecture we have developed to solve the above problems in four complementary ways. First, it integrates knowledge of system properties over time, space, and source. In order to provide a clean way of reasoning about the system properties of the interactions between clients and objects, QuO employs the concept of a *connection* between a client and object, an encapsulation including the desired usage patterns and QoS requirements specified in the form of a contract. To help simplify the combinatorial problem of dealing with an n-dimensional QoS space in this contract, QuO supports first-class *QoS regions*, which designate regions of operation for the client-object connection. To help the application adapt to different system conditions, QuO supports multiple behaviors for a given functional interface, each bound to the QoS region for which it is best suited. QuO supports different *commitment epochs*, explicit times which different information about system properties is available to be bound.

The second way QuO solves these problems is through the reduction of variance in system properties. QuO masks variance in system properties using *layers of delegate objects* in the client's address space. Each delegate layer embodies knowledge of system properties for one participant in the connection: the client; object; or CORBA Object Request Broker (ORB). System properties are encapsulated into first class objects which we call *system condition objects*. QuO's system condition objects can be used at different levels of granularity, which allows for their aggregation. For example, a single system condition object which measures throughput could be bound to a single method invocation from one client, to any method invocation for a group of objects from multiple clients, or something in-between.

The third way in which QuO solves these problems is by making the key design decisions of an object explicit, while allowing the effort expended in this specification to be tailored to the desired level of adaptivity. The more design decisions that are exposed, the more adaptive the resulting applications. To support this spectrum of adaptivity, the QDL is really a collection of sub-languages which concentrate on different system properties. QDL includes three sublangauges: a *contract description language* (CDL) to describe the contract between a client and an object in terms of usage and QoS; *structure description language* (SDL) which describes the internal structure of an objects implementation and amount of resources they require; and a *resource description language* (RDL) which describes the available resources and their status.

The fourth way in which QuO solves these problems is by providing a framework to greatly reduce the programming effort required to deal with system properties. The programmer has to deal with high-level abstractions involving system properties, which can in many cases be reused. Many of the architectural components are then automatically generated from the high-level descriptions given by the programmer.

2.3 An Example

Throughout this paper, we will use a running example to illustrate the problems with handling system issues and QuO's proposed solution. Our example application is based on a collaborative map which is part of a distributed planning application running over a wide- area network. As part of the example, the network is capable of reserving resources, but the end-user is charged for the reservations to reflect the cost of denying the resources to other users. The challenge for the example is to make an adaptive application that can use reservations wisely, based on the application's expected usage patterns, QoS requirements, and the user's budget. Note that while this example focuses on performance issues, the QuO architecture is general enough to handle other system properties, such as availability, security, or real-time behavior, and other metrics besides cost.

The purpose of our distributed planning application is to help create and execute logistics plans, such as for quickly moving relief equipment to the site of an earthquake. Because of the wide variety of domain and locality information needed, no one person can create the whole plan. The plan itself is represented by CORBA objects which are distributed over a WAN. Each planner is responsible for the content of some subset of the plan objects. In order to resolve conflicts, planners must collaborate with their peers, who are also spread all over the world. One of the tools the planners use to support collaboration is a collaborative map subsystem.

A collaborative map is a geographic map overlaid with icons. Each icon represents the status of one or more domain objects which are being managed by the plan and whose present location is known or can be ascertained. For example, an icon could represent a shipping container whose location moves over time and whose color represents whether it is full or empty. The icon is thus a local view for a remote CORBA object which represents the shipping container, i.e. the icon's graphical characteristics are representations of the attributes of the remote domain object. One map can view and modify many different types of CORBA objects owned by different planners. When a planner makes a change in a plan object, the icons are updated on all the maps which are

displaying a representation of the object. Each map subscribes to the remote objects which it is displaying in order to receive updates.

Subscription implies an on going association between a remote object and a map. The desired system characteristics of that association dictate the kind of network resources which will be required. For example, if many of the objects are changing quickly, a minimum bandwidth is need to service the updates. If this bandwidth is not available, the application must change its behavior, such as flow controlling the updates, throwing away updates, or reserving more resources. A common problem with applications developed on LANs is that they do not address this basic form of adaptability. When they are fielded on a WAN with restricted bandwidth, the update queue overflows and they can literally crash or they pause indefinitely. Thus, the utility of these programs becomes zero, if this bandwidth requirement is not met. Making the application adaptive, softens this hard requirement.

Suppose that our map needs such a minimum bandwidth path to a remote object server and that reserving resources will be used to met the requirement. A key issue is when to establish a resource reservation and when to tear it down. One simple policy to reduce the cost of a resource reservation is to cancel the reservation when the application is idle. However, canceling network resources must have no effect on the functional part of the application, namely the correct representation for the remote map objects. For a simple example, when a user leaves a workstation, the workstation detects that it is idle and turns on the screen saver. When the user returns, she moves the mouse and the screen quickly appears as if nothing happened. Analogously, the network resource reservation could be canceled and then reestablished when the user starts working again.

Although it is trivial in its scope, implementing a Network ScreenSaver policy will be used to illustrate the dimensions of QuO and to show how the QuO architecture supports adding adaptive capabilities to an application. This policy was chosen for the running example, because it illustrates many of the problems associated with handling system conditions and yet it is easy to understand in isolation. The rest of the paper will show how the QuO architecture address pragmatic problems such as these: How are complexity of reservations hidden from the application? What happens if a reservation fails or is not available? How are the policies associated with making a reservation specified? Can a group of objects be associated with one reservation? Who is responsible for making the reservation code? Can the code dealing with system conditions be reused for many objects?

3. QuO Design Part 1: Integrating Knowledge of System Properties

A distributed program is a complex entity. It is created from multiple components, with many design decisions made about the construction, combination, and behavior of these components. The information on which these decisions are being made is provided by different providers, at different times, and is being consumed by a number of consumers. Consider a simple example involving the implementation of a plan object being used by a map client, and the times at which various commitments are made:

- *object definition time* : the plan object's designer commits to an IDL interface specification for the attributes of a plan object such as its geographic location.
- *object implementation time*: the plan object's designer commits to a specific algorithm to implement the interface specification (e.g., it looks up the object's location in a relational database).
- *client implementation time*: the map client's designer commits to a specific behavior for what the object is expected to do (e.g., it must convert from the geographic location to a pixel location on the client's screen).
- *installation time*: the system administrators commit to specific implementations for the plan object and map client. An alternate implementation of the object or client may be installed without the other knowing that it has changed, so long as the object's interface remains the same.
- *run time*: the user decides which plan objects to view with the map client.

The previous example showed the different times at which the functional properties of an object are bound. However, its system properties must also be bound if the client's expected usage pattern is to be met and if the client and object are to be able to adapt to changing system conditions. In doing so, it helps a distributed application to be adaptive if it can defer these binding decisions as late as possible For example, the client must specify its expected usage patterns and the object must indicate whether it believes it can support the proposed usage pattern. The client and the object must both be informed when the environment changes so they can change their expectations or behavior, for example, when resources are lost or congestion occurs. A client must inform the object when its expected usage patterns change. Similarly, an object must inform the client when the usage pattern it can support (QoS it can deliver) changes.

3.1 Connections

In a traditional client/server architecture, system information is present in three independent places which are difficult to reconcile: the client, the communications substrate, and the object. This tripartite division is driven by functional layering, because the communication layer is a convenient interface between the client and the server. However, if any kind of end-to-end QoS is to be provided, then this disparate system information must be reconciled, and the issue is where is the best place in which to do this.

One way is to have the communication network supply QoS guarantees and to support an external management information base (MIB) to assess whether or not these guarantees are being met; an example is the QoSockets package (Florissi & Yemini 1994). But this approach keeps the system information separated and some external agent must be used to integrate it. A second approach is to have the client take responsibility for end-to-end QoS. For example, current World Wide Web (WWW) architectures feature intelligent browsers managing system properties by prefetching, parallel image retrieval, and caching. But a QoS architecture employing smart clients would put a heavy burden on the client programmer, and we would expect there to be many more inexperienced client programmers than object programmers. We provide a third alternative, which is to make the object responsible for the end-to- end QoS. The object's knowledge of the system conditions is extended and part of the object's implementation is moved

into the client's address space. We accomplish this by the use of a layered stack of delegate objects to implement the abstraction of a client-object *connection* with QoS.

Connections define a boundary where expected usage pattern and QoS requirements between the client and the objects can be agreed upon. In our example, the connection is the place where the current throughput is measured and client idleness is detected. All interaction between the client and the objects pass through the connection. Effectively, the boundary between the client and the remote object is moved into the clients address space by using a delegate object. The client will create these local delegate objects, which will in turn bind to the remote object. The delegate objects support the functional interface of the remote object, and they forward the client's invocations on to it. Delegate objects also provide an API for handling the system properties affecting the client-object connection.

Having the connection boundary reside in the client's address space is beneficial for two major reasons. First, there is essentially no delay or congestion between the client and the connection object. This is a major advantage; for example, with an external MIB there is essentially an additional layer of QoS between the client and the MIB, and the MIB's job is further complicated since it has to account for this layer. The second advantage is, for all practical purposes, the delegate object will not fail independently of the client. It can, however, monitor the availability of the remote object and of any replicas it has. These two advantages combined allow the connection's knowledge of both the remote object's implementation and the communication substrate which connects them to be exploited to provide for better support for QoS and adaptivity. Employing a connection is a heavyweight solution, when compared to a local procedure call, but its costs are negligible compared to a remote procedure call.

3.2 QoS Regions

System conditions will change over time. As a result of this, the usage the client actually generates and the QoS the object actually provides may diverge from their expectations.

QuO handles this divergence by allowing the specification of two levels of system conditions, involving both the client and the object. A *negotiated region* is a named region defined in terms of both client usage and object QoS based on the system conditions within which they will try to operate. A negotiated region is thus defined in terms of the *expectations* which both the client and the object (via its delegate connection object) set. A typical QuO object will support a number of negotiated regions. Within a given negotiated region there may be many *reality regions*, which are named regions defined in terms of the actual client usage and object QoS *measured* by the QuO runtime system.

QuO allows for the specification of handler routines to be invoked in either the client or the object when transitions occur between either negotiated or reality regions. A handler for a reality region transition informs the client or connection object when measured conditions change sufficiently; e.g., when its observed behavior is not meeting its expectations. This allows the client or connection object to either take compensatory action to try to operate within its expectations, or to change those expectations. A negotiated region handler informs the client or object when the negotiated region has changed. Since this is a fairly heavyweight operation,

possibly involving reallocation of lower level resources, we anticipate that QuO applications will normally try to adapt using reality region handlers.

The negotiated (reality) regions may overlap, i.e., the predicates involving the expected (measured) usage and QoS can overlap between negotiated (reality) regions. QuO allows for the specification of a precedence among the regions, which is used to select the current region if the predicate for more than one is true.

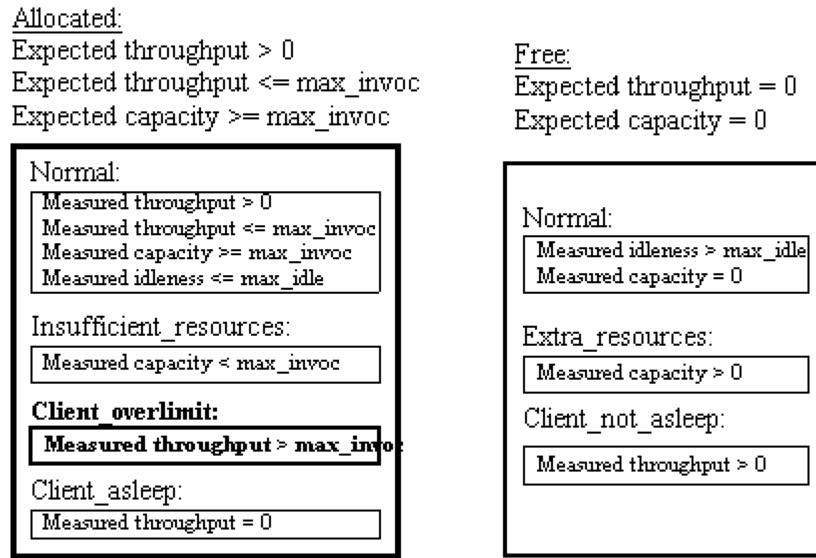

Figure 2: ScreenSaver Contract Regions

Figure 2 shows the ScreenSaver example's negotiated and reality regions. This contract supports two negotiated regions, "Allocated" and "Free". The Allocated region is for the normal mode of operation, while Free is for a mode where the client will be inactive for some period of time. The Free region could be entered because either the client explicitly set its expected throughput to zero (e.g., to indicate its user is going to lunch) or by the QuO runtime system observing no traffic from the client over a reasonable length of time. While in the Free region, the state and resources associated with the remote object and the connection object will remain allocated, but the resources which were previously allocated to support the delivery of invocations to the remote object may be recycled. The Allocated negotiated region is where the client expects to generate a throughput greater than zero but not greater than max_invoc methods per second.

The Allocated region has four reality regions: Normal, Insufficient_resources, Client_overlimit, and Client_asleep. The Normal reality region is where measured throughput and capacity are consistent with the client's and the object's expectations. The Insufficient_resources reality region is when the measured throughput and capacity are inconsistent with the client's and object's expectations. In this case, a handler in the connection object would be invoked to inform it that its capacity was insufficient. The Client_overlimit reality region is when the client is generating more than its expected throughout; in this case the client will be informed that it is exceeding its expected usage. Reality region Client_asleep denotes when the client was observed

to have generated no throughput for some time (as measured by an idleness detector). In this case, a client callback will set its expected throughput to zero, and an object callback will set its expected capacity to zero. This change in expectations triggers a reevaluation of the negotiated region, which will change to Free. (Free is similar to Allocated, and a discussion of its parts is omitted for the sake of brevity.)

3.3 Adaptivity Implies Multiple Behaviors

Distributed applications need to adapt to changing system properties. The ways in which they can adapt include:

1. Finish later than expected, either by simply tolerating its finishing later than originally expected or by rescheduling it for later (when system conditions are presumably better). In our example, if the required bandwidth is not allocated then the map client could simply block until bandwidth is allocated, or it could perform the update later.
2. Do less than expected. For example, the map client could discard some of the updates and thus tolerate lower fidelity in the representation of the plan.
3. Use an alternate mechanism with different system properties. For example, if the throughput goes above the allocated bandwidth then the updates could be compressed before they are sent over the network. This uses less bandwidth at the cost of using more processing cycles.

QuO provides mechanisms to support all of these adaptivity schemes. A client and object (via its connection object) may be notified that the reply for an outstanding request will not return to the client with the expected QoS (e.g., within the expected delay). In some cases, the client may be warned that, based on the recent history, any requests in the near future are unlikely to be serviced with the expected QoS. In either case, both the client and the object need to be notified so they can agree upon which of the above options to choose. Also, in the last case, the client and object need to agree at runtime to change their behaviors. Thus, to allow distributed applications to adapt to changing system conditions, the object designer and application programmer must be able to deploy multiple implementations of a given functional unit, ones which implement the same functional interface but with different runtime behavior (e.g., with different design decisions trading off communications bandwidth versus storage versus processing cycles). It also needs to have a way to specify which implementations are valid, or at least best, under which system conditions, and be able to dispatch the best implementation at runtime for the current conditions.

The QuO architecture provides mechanisms to support all these options. The object and client can be warned of a pending request whose expected QoS is not being met using a reality region callback. They can be made aware that a given QoS is unlikely by the negotiated region which they are presently in. And they can change behaviors in two different ways. The first is accomplished by explicitly calling different internal functions in different negotiated or reality regions. The second is by the object designer specifying alternate paths through the connection object and the remote object using a *structure description language*; e.g. to transparently compress the data in the connection object and transparently uncompress it at the remote object, before the invocation reached the target method's body.

3.4 QuO Commitment Epochs

QuO supports object-oriented QoS in a way which allows the application to adapt to changing system properties. It allows the object designer and client programmer to defer binding decisions as late as possible. QuO thus supports a number of binding times, which we call *commitment epochs*. They are:

- *Definition*: define the type of connection using QDL. Also define the structure of QuO regions, as well as hooks to bind handlers to regions to effect different behaviors. Finally, define alternate paths through the object's components (including marshaling and unmarshaling code) for the object, via the SDL. In our example, the connection is bound to the ScreenSaver system behavior at this time.
- *Connection*: create an instance of the connection object, passing in parameters which bind the shape of the structure defined at definition time (e.g., max_invoc above). Bind to the remote object via a particular communication substrate.
- *Negotiation*: agree upon expectations for client's traffic and object's QoS, i.e., choose the expected bounds to which the client and object will attempt to operate within. In other words, decide upon the agreed behavior. In our example, the client can explicitly free the resources by setting the expected throughput to zero. Alternately, the QuO runtime can detect that the client is idle and invoke a map client's callback, which could in turn change its expected throughput.
- *Invocation*: measure actual client usage generated, object QoS delivered, and failures; i.e., observe actual behavior. In our example, the throughput is measured and, if the client is over its agreed upon throughput limit, it can employ flow control to shape its usage.

QuO integrates the information about assumed and actual system properties provided at these times and by the various parties to provide the object-oriented QoS it provides.

4. QuO Design Part 2: Reducing the Variance in System Properties

Programs written using local objects -- ones in the client's address space -- can assume a much simpler model of system behavior than those using remote objects (Waldo et al. 1994). For example, the delay in delivering an invocation to a method's implementation and its reply back to the client is negligible, and the throughput (invocations per second) is high. Thus, the base values of the system properties are much better for local objects than for remote ones.

The variance of these system properties is also negligible in the local case. While it is possible for a client's host to get overloaded, such occurrences are relatively rare now given the proliferation of inexpensive and powerful workstations and personal computers. Thus, resources on the client side are usually much more abundant than on the server side (the remote object) or in the communications link between them. Not only is the base value of the system properties higher for remote objects, the variance of these system properties is also much higher than for local objects. Additionally, local objects do not fail independently of their clients, for all practical purposes, while remote ones can. This is another (extreme) form of variance in system properties. Indeed, in our experience this higher variance of the system properties of remote objects is generally harder for programmers to deal with than worse base values for the system

properties. For example, if there were little variance, programmers could easily deal with higher delays, lower throughput, etc.; in fact their programs might be structured much like those using local objects. However, with current technologies programmers using remote objects are forced to include substantial error handling code to deal with the great variance in the system properties.

A goal of QuO is therefore not only to improve the base value of a remote object's system properties, as observed by the client, but also to reduce its variance. It accomplishes this through the use of negotiated and reality regions and also by layering internal delegate and system condition objects. QuO applications can adapt to some variance by the use of reality regions, which bind different client behaviors (implementations) to different system conditions. This allows the client and object to adjust their behavior to keep the connection within its negotiated region of operation as long as possible, thus reducing variance. Further, QuO employs multiple layered delegate objects on the client side, and each layer is able to both mask out some variance of system conditions as well as improve their base values by employing the algorithms (implementations) most appropriate for the current conditions.

4.1 Masking Variance with Layers of Delegates

Improving system conditions cannot be accomplished at a single time and place, since the systems knowledge which is required to do this is available at different times and in different places. Also, knowledge of how best to use this information is distributed among multiple partners. We layer the delegate objects in the client's address space with this in mind, and an example is given in Figure 3. In this figure, the delegates each use their knowledge of the system information they possess to improve the system conditions seen by the layer above them by masking. Each layer exports a negotiated region to the layer above. It uses various techniques (changing policies, etc.) to mask any changing conditions and maintain the QoS it provides to the layer above. When it cannot maintain the QoS corresponding to the current negotiated region, it propagates this information upward via a handler indicating a change in reality region. Each party tries to adapt (by changing policies, etc.) and if it cannot, it indicates a change in expectations. This triggers a renegotiation of the negotiated region, because negotiated regions are defined in terms of the expectations of the client and object. The changing conditions which can be handled by this architecture include changes in resource availability in the network or on a host (which of course affects the QoS which the object can deliver), changes in a client's usage pattern, and the failure of an object.

For example, in the figure, assume that the client knows it can wait an indefinite period of time for an invocation to the object to complete, even if the object fails and must be restarted. This is indicated in QDL, and the QDL code generator generates a client delegate object for the application to use. The application invokes the functional methods on the client delegate, and the delegate passes this invocation down and handles exceptional conditions resulting from this invocation. For example, if the remote object or the network link to it fails, the client delegate will indicate that the invocation has been interrupted and will attempt to reestablish the link or create another instance of the object. Once this has been completed, it will pass the invocation down again.

The client delegate is optional, and in many cases will not be used. It is available, however, for those clients who do wish to simplify their invocation semantics in a particular way; e.g., to eliminate much application-level exception handling.

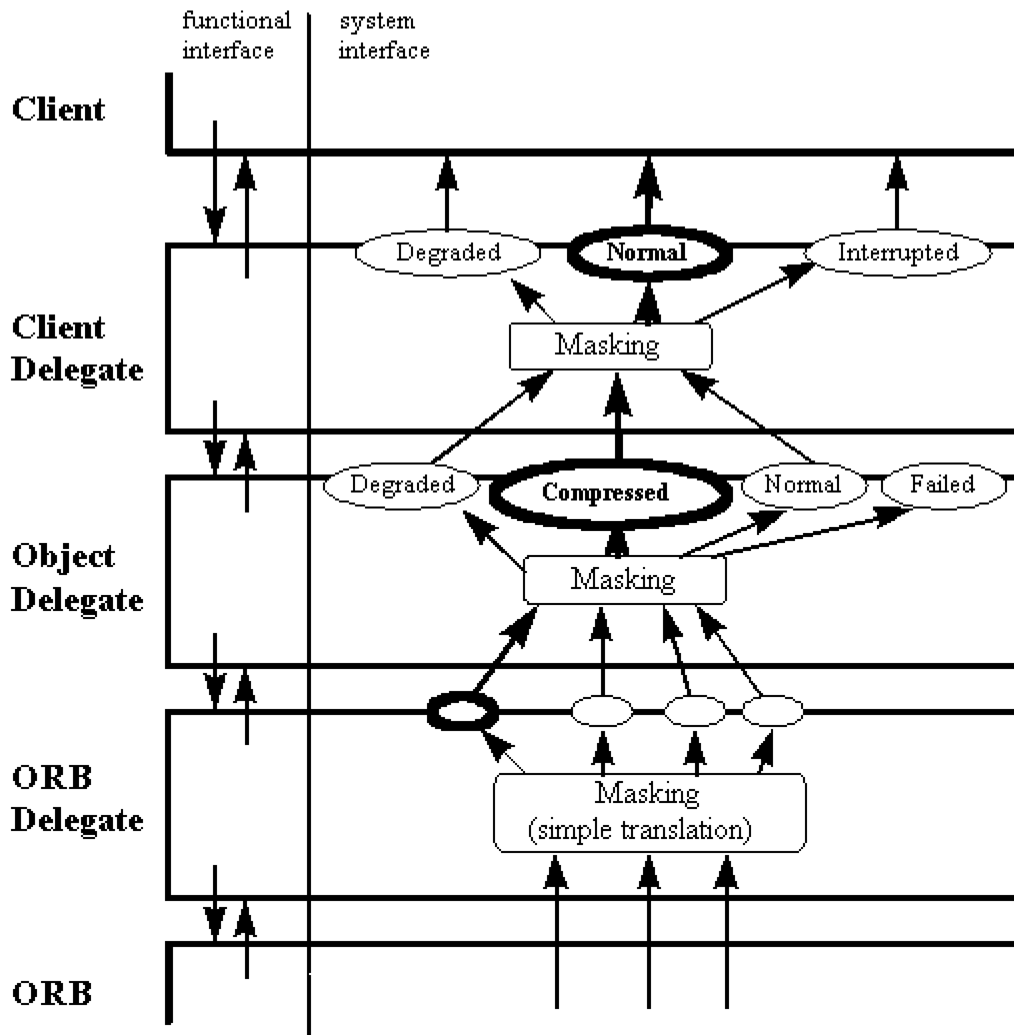

Figure 3: Masking System Properties

The object delegate can take advantage of the knowledge of its different implementations to adapt to changing conditions while still providing a useful QoS to the client. For example, in the figure, the object delegate will automatically use compression if the bandwidth the ORB is delivering is insufficient to meet the client's needs; it will trade off additional CPU cycles to maintain a desired bandwidth. In many cases this can be done transparently to the client. However, while the bandwidth of a compressed region may be the same as a normal one, other system properties such as the delay may be different. A separate reality region (compressed) indicates it no longer expects to be able to meet the conditions for the normal region. But since the client delegate only makes throughput guarantees, it still remains in the normal region and the compression is transparent to the client.

The ORB delegate translates from CORBA system exceptions and other system information available from the ORB into QuO negotiated regions. In doing so, it can take advantage of knowledge it has of the ORB's implementation, since there is one ORB delegate class created for each different ORB which QuO supports.

The client, object, and ORB delegate objects are automatically generated by the QDL code generator. At connection time the client passes in references to objects to handle the callbacks (e.g., to be notified of a change in the current reality region) as specified in the QDL. It can optionally pass in one or more system condition objects to be used by the delegate objects as specified in the QDL, e.g., for aggregation. If the client passes in no condition object reference then the QuO runtime system will create one at connection time.

4.2 Integrating System Knowledge from Different Sources

Figure 4 shows how the QuO architecture integrates knowledge from the different parties into one cohesive framework. In this example, a functional interface named inv is being extended to include system properties. This stack is set up when the client connects to the remote object. The participants with different systems knowledge are:

- *Client designer*: Knowledge of how to adapt to changing reality and negotiated regions, given the different usage patterns it may generate as well as the different implementations with which it can provide its part of the application program. This knowledge is encapsulated in the client delegate object.
- *Object designer*: knowledge of its different implementations, as encapsulated in the object delegate object.
- *QuO designer*: knowledge of how the ORB which QuO has been ported to is implemented, as encapsulated into the ORB delegate object.
- *ORB designer*: a very simple model of system conditions, encapsulated into CORBA system exceptions.
- *Operations staff*: knowledge of resource availability, resource access permissions, administrative domains, etc., encapsulated in the environment delegate objects.

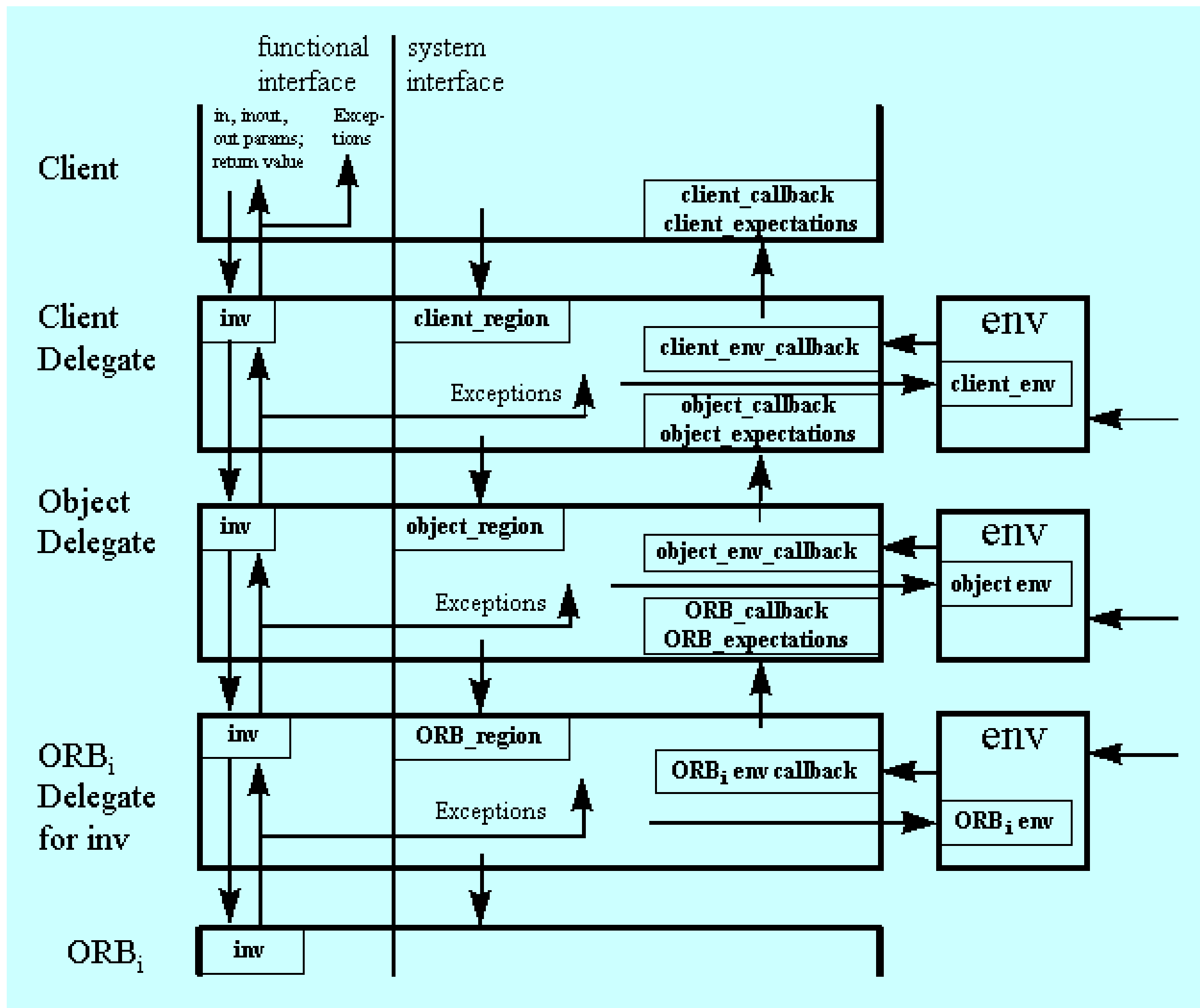

Figure 4: Delegate Object Layers

The delegate objects support different standardized interfaces to facilitate the integration of the knowledge from these varied sources. Each delegate object in the stack implements various interfaces. This includes the functional interface, so that a client's invocations can be passed down towards the remote object. It also includes a callback interface based on the contract of the delegate object below it, for use by reality and negotiated region callbacks, as well as an expectations interface which the lower delegate object can use to query the expectations of the layer above it. It implements a negotiated region interface for use by the delegate object above it; this interface indicates the current negotiated region of the delegate object. Finally, it implements an environment callback interface so it can be informed of changing environmental conditions.

4.3 Complex Delegates with System Conditions as First Class Objects

In the preceding discussion, note that delegate objects are translators. They translate from lower-level system conditions with higher variance (and often worse base values) into more desirable system conditions with lower variance (and often better base values), i.e. system properties closer to that of local objects. In this scheme, there are two flows of information: functional and

system. The functional flow passes on the client's functional invocation, and also updates the appropriate system condition concerning this invocation (e.g., ones which measure client throughput). Some system information propagates independently of the functional information; for example, system condition objects which detect client idleness or use capacity information from a network management system.

A delegate object is not one monolithic object with many attributes and code to implement the functional and system translators, as implied by Figure 4. Logically, the delegate object acts as one object, but physically the system portion of the delegate is really implemented as a web of sub-objects as shown in Figure 5. Some of the objects implement the contract (region objects) while other objects represent system properties (system condition objects).

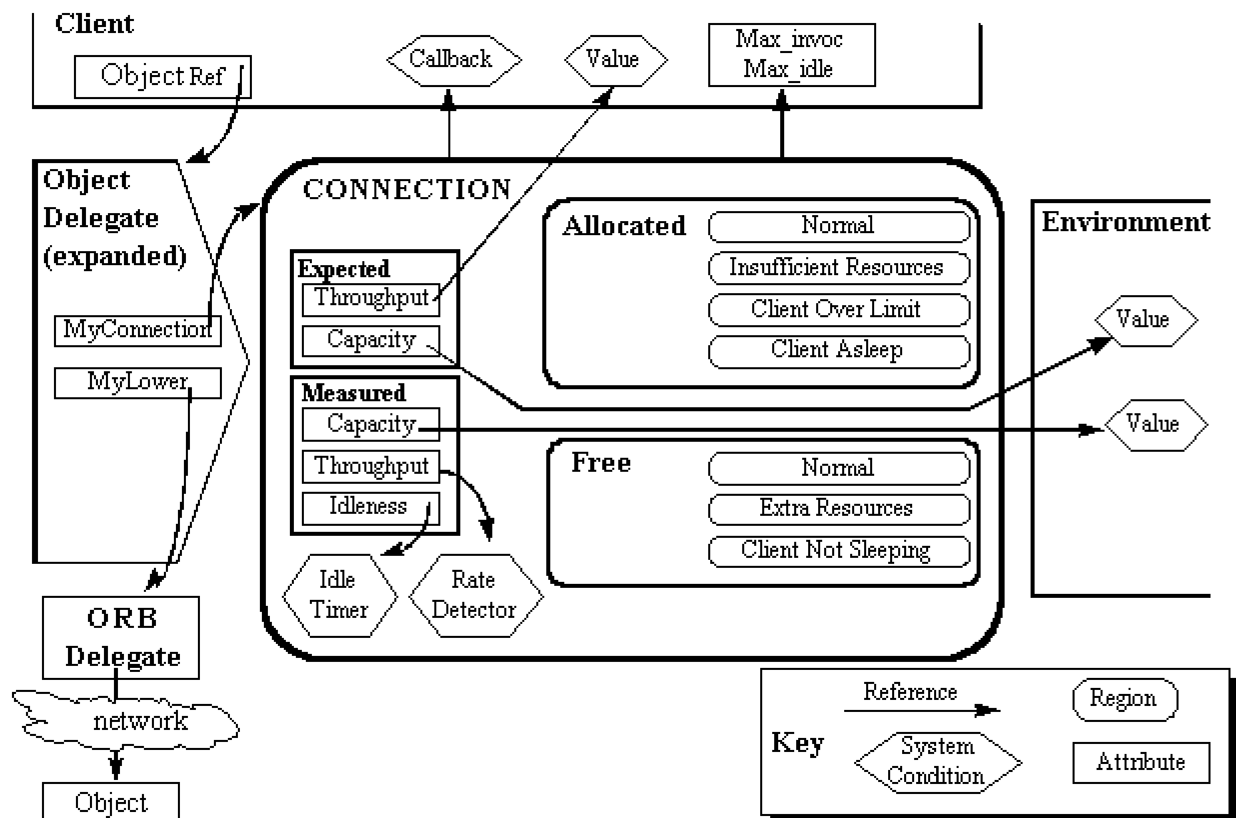

Figure 5: Web of System Condition Objects for Implementing System Translators

Having separate system objects provides two main advantages. First the system objects can run independently (asynchronous) to the functional part. For example, while the functional part is blocked waiting for a response, a system condition object which represents the size of a response can be updated. The change in system condition could trigger an action, such as putting up a dialog box asking if the size of the response is too big to send across a WAN. Second, the aggregate behavior of a group of connections can be represented by sharing a system condition. For example, the measured throughput to a group of objects can be tallied by using one system condition that is updated whenever any object in the group is called. The measured throughput

for a group can be used to flow control interactions to objects that share a common resource such as a server process.

Note that the functional part of a delegate cannot be shared. The functional delegate represents the binding between the functional translator (e.g. a call to remote object) and the system translator (connection object). Thus, the functional delegate must contain at least this binding in its state, i.e. at minimum functional delegates must have a pointer to the next lower delegate object and a pointer to its contract object.

The structure of the web of system objects is defined using QDL. Specifically, the Contract Description Language defines the form of the regions, predicates and system conditions that define the regions (Figure 2). The form of the contract only defines slots for specific types of system conditions. Binding these types to specific instances happens at connection time. Thus, setting up the connection defines the web of interconnected system objects. Connection setup is a fairly heavyweight operation which is done once, before any invocation on the object. We expect the connection operation to be a more comprehensive version of the current "bind" operation or trader lookup which is used to get a local reference to a remote object.

Notice that the interface between the layers of delegates is really a collection of system condition objects that are maintained (or contained) by one layer and read by the other. Reads and callbacks are directed at these interface objects and hence affect their container object indirectly. Also, the interface objects do not have to be co-located with their container, allowing the interface to be spread over several devices.

The interface to the system conditions are described in IDL and system condition instances can be local or distributed. For example, a system condition that represents the failure status of a remote host can be implemented several ways, but would have one IDL interface. One implementation could actively poll for remote status, either on demand or periodically. Alternatively, the status object could be an interface to a remote status service that detects and disseminates device status.

Because system conditions can be updated asynchronously from a method invocation on the object, there is the flexibility to make a tradeoff regarding how much work should be done on demand (at invocation time) and how much should be event driven (when a system condition changes). For example, when should a predicate that defines a region be updated? Only when there is an method call? Or whenever a system condition used by the predicate changes? The system conditions themselves define when to propagate changes and when to cache them. For example, if system conditions propagate their change, this could cause a forward chain of events that changes the active regions and calls a handler, independent of a call to the functional part of the delegate. Alternatively, when the delegate is invoked, the system conditions could backward chain until a cached value was found.

Aggregates can be grouped by objects, a set of methods, or even invocation on a single method. For example, all the methods that modify the object state could be grouped as a "writers group". If the system condition for the expected throughput for the writers-group were set to zero, then

the usage of this object expected in the near future would be considered read-only and replication mechanisms optimizing for this case would be employed to best service the readers of the object.

5. QuO Design Part 3: Exposing Key Design Decisions

Large distributed systems developed by the DoD and by others have grown to staggering levels of complexity. To manage this complexity, large systems are built using layers of abstraction. Each layer provides services to the client layers above it. Each abstraction provides an interface which provides the service but hides the details of the underlying implementation. While a new implementation of an abstraction can replace an old one, there is typically only one implementation of that abstraction operating at any one time.

Unfortunately, a layered abstraction based on exposing only a functional interface, such as CORBA's IDL, is not enough. Distributed systems are fundamentally different than self-contained, stand-alone programs (Waldo et al. 1994). Because of the wide range of resources and usage patterns, a single implementation of a component cannot provide everything needed by all possible clients. Fundamentally, the most appropriate design and implementation for a black box module cannot be determined without knowledge of how the client intends to use the module and the available resources. Looking at distributed application code, it is clear that handling system issues now dominates development. (In the past, moving data structures across the network and providing graphical user interfaces (GUIs) dominated, but now these problems have been greatly reduced by recent advances in commercial software such as CORBA ORBs and GUI builders.)

The inadequacy of the functional-only interface approach has led to *ad hoc* work arounds, either through programming between the lines (Kiczales 1994) or by simply rewriting large parts of the module to tailor it to a specific environment. Programmers are acutely aware of the target hardware, the type of network, and client usage pattern. The resulting application is fragile and will not perform adequately outside its intended environment. For example, one of our programmers spent one month experimenting with 11 different caching schemes for accessing collaborative map objects. Some of the implementations worked well over LANs but poorly over WANs, and others worked well over WANs but unacceptably on LANs. However, in the end he was forced to choose an implementation that worked tolerably on both LANs and WANs. Because he was restricted to a single implementation, he was forced to commit to one specific set of tradeoffs, which may not be appropriate for a different environment for the same application.

Open implementation (OI) techniques (Kiczales et al. 1991; Kiczales 1994) help augment the black box functional interface, allowing the designer of an object to expose key design decisions affecting the object's performance and reliability. An open implementation provides disciplined, object-oriented access to the implementation of the functional abstraction. This allows the application developer to alter the behavior of the application by choosing the implementation of a component best suited to the application. The *meta-data* which describe an implementation are specified separately from the functional aspects of the implementation, so they can be changed more easily. A *meta-level architecture* thus allows a system to reason about itself and change its behavior based on both the self-knowledge and of the current system properties. An object

developed using OI techniques is more general because it can handle a wider range of system properties by supporting multiple implementations.

A monolithic, all-encompassing meta-level architecture would likely overwhelm a typical application programmer, who wishes that system issues would just go away. Thus, to be practical, a meta-level architecture needs a low barrier to entry, i.e. as the programmer is willing and able to provide more meta-data, the system becomes more adaptive. However, the programmer must understand that handling system properties can not be completely automated, only simplified. Programmers must be aware of the tolerances of their application and make them explicit by supplying meta-data. The tradeoff is how much meta-data to supply.

The QuO architecture supports a wide spectrum of adaptive techniques. QuO focuses on supplying the meta-data about system properties needed to make adaptive systems. QuO does not offer (nor mandate) specific adaptive mechanisms; instead it offers a framework to exploit the rich collections of existing point-solutions. Some mechanisms exist in the QuO run-time libraries, but these will expand and change over time.

QuO uses open implementation techniques to expose meta-data on an as-needed basis. The Quality Description Language (QDL) is really made of several independent sub-description languages that allow for the structured specification of meta-data about system properties. If all of QDL is used, the application should fully support comprehensive adaptive applications with multiple implementations. But not all description languages need to be specified; default policies or assumptions will be used if they are not. This allows the granularity of the adaptability to be tailored to the level of effort. For example, CORBA allows for a market in generalized "business objects" which, in order be effective, must adapt to many different environments. Thus, we would expect the experienced object programmers to expend a lot of effort describing meta-data in order to spare the less focus client programmer from the burden, because business objects will be judged by how well they adapt.

QDL uses the following description languages to expose system conditions. Also, some examples are given for the level of adaptability to be achieved with this meta- data.

Contract Description Language (CDL) defines the expected usage patterns and QoS requirements for a connection to an object. As described in a previous section, the contract defines regions of system conditions and allows actions to take place when the region changes. At connection time CDL can be used to help bind to the most appropriate object, using the specified expected usage patterns and QoS requirement used to select the policy to find the object. This augments current CORBA bind mechanisms and envisioned trader services (Object Management Group 1995a). At invocation time, the reality regions can be used to dispatch to different behaviors (implementations) in the object delegate based on the current measured conditions. Also, handlers specified in CDL can be programmed to be called when the contract detects a change in the current region. These handlers can clean up problems both synchronously and asynchronously with calls to the object.

Resource Description Language (RDL) abstracts the physical resources used by the object. These abstract resources can be bound to real devices at run time. The QuO runtime thus models use of

resources that the application is running over along with their current status. At connection time, this resource model helps to choose the most appropriate resource binding for an object. The policies can be based on the current status of groups of resources instead of just general policies, such as "optimize for low network traffic". For example, a policy might be bind to the server with the highest capacity, based on CPU type, load average, and network capacity. At invocation time, the status of resources can be used to route the message to the best active server.

Structure Description Language (SDL) defines the internal structure of an object and how it consumes resources. SDL is a data flow description of the internal processing, communication, and object state. Each method would have its own SDL description. For example, some methods may not change the objects state while other only write it. The placement of a method's implementations relative to the object state it accesses is flexible, allowing the object's state to migrate between the object an client address space, or any where in between.

In conclusion, QuO support a large spectrum of adaptive techniques. QuO "opens up" a distributed object's implementation by using QDL to specify meta- data about system properties. In order to create distributed applications which are not fragile, programmers must address system issues. QuO offers low barrier to entry by allowing the programmer to specify only the meta-data judged to be worth the effort in the current round of development. The more meta-data that is specified, the more adaptive the distributed application. In order to make handling system properties tractable, QuO depends on "reusing" system knowledge, which is the topic of the next section.

6. QuO Design Part 4: Supporting Code Reuse and Generation

The QuO architecture described in previous sections supports the development of applications which are adaptive and exhibit more predictable behavior. However, these tangible benefits will in practice never be realized if it requires substantial additional coding. Indeed, many software projects today are over budget and late when trying to deliver just one implementation of each component, so the thought of having to worry about system properties and provide multiple implementations seems excessive.

However, programmers *do* have to deal with system properties if their programs are to work across the wide area. The issues are how much effort it takes to handle them, and can this effort be amortized across multiple projects or configurations through reuse. Also, it is not necessary to make all objects adaptive, only the critical ones, much as experienced programmers tuning a program's performance first observe which pieces are the bottlenecks and then spend time reimplementing only those few. Further, additional implementations can be added later after the new uses are identified and after the value of the application has been proven. It is this knowledge that we can evolve the implementation where necessary which allows us to de-emphasize the initial implementation choice, especially for rapid prototyping disciplines.

The issue of which system property definition or implementation to use may be a hard one for programmers to deal with. Indeed, a whole subfield of the networking community has protracted

debates about which algorithm should define a given system property's measurement and how to translate metrics, so even an experienced programmer who is not part of this community may be baffled by the seemingly endless debates. Further, there is no CORBA standard for instrumentation, so the programmer can find no guidance here. QuO helps this dilemma by providing a collection of implementations of system properties. QuO is extensible and does not mandate any particular algorithm for a given system condition. We anticipate in some cases providing more than one choice for a given system property, albeit with different characteristics (e.g., for measuring throughput both leaky token and recursive filter algorithms are provided).

To reduce the burden of handling system properties to the programmer, QuO supports the reuse and automatic generation of much of the code involved with a given application. This is conceptually similar to using IDL for functional interfaces. In this section we discuss the different ways in which QuO supports these practical software engineering and software management issues, organized by development cycle times from the programmer's perspective. We show the steps a programmer goes through to use QuO, highlighting at each step what can be reused, what will be generated by QuO, and what the programmer must write. We will use the ScreenSaver contract shown in Figure 2. The result of these steps will be the creation of the objects for this contract, shown in Figure 5, as well as the client which uses them.

6.1 Overall Development Process

QuO adds steps to the CORBA development process and defines a new role for a QoS Designer (Figure 6). In the traditional CORBA development process, the Object Designer defines an IDL interface for an object and uses a code generator to make object stubs and client-side proxy code. The Object Designer adds the object functionality to the object stubs and compiles it to make the runtime server. The Client Designer writes code that uses the client-side proxy code and compiles it to make a runtime client.

QuO allows for a new role of QoS Designer who designs custom QuO connections and system condition objects. System conditions are developed like any other CORBA object and capture the QoS Designer measurement expertise. Connections are specified in QDL and a code generator is used to make the region code and delegate stubs. To support adaptive code the Client Designer must add functionality to the callback stubs and the Object Designer must add adaptive behavior to the delegate stubs. These generated stubs can be used without modification, but the resulting application will only be QoS aware and will not adapt, e.g. the throughput will be measured and changes detected, but no action will be taken when the change occurs. Both system conditions and connections are independent of the functional parts of the code. Hence, we expect QoS Designers to create libraries of this code and make them available to the Object and Client Designers to add to their applications.

Figure 6: CORBA and QuO Development Process

6.2 Developing a Contract

The QoS Designer writes the IDL for the objects for the client's callback and expectations objects, such as the **client_callback** and **client_expectations** in Figure 7. The former is what will be used to receive notifications of transitions, and the latter is an object to maintain the client's current expectations.

The QoS Designer writes the CDL for the contract. Figure 7 gives the CDL for the negotiated regions for the ScreenSaver contract, and Figure 8 gives the CDL for one of its reality regions. Note that nothing in this CDL is specific to a particular CORBA interface, so it may be reused with objects of other (functional) types. The function interface is defined by the Object Designer and is needed to generate the delegate.

```
// Forward declarations for classes used in the connection's parameters.
interface ScreenSaver_client_callback;
interface ScreenSaver_negotiated_region;
interface ScreenSaver_client_expectations;

connection invScreenSaver(
    // 3 Parameters required for every QDL connection
    in ScreenSaver_client_callback cl_call,    // for client_callback
    in ScreenSaver_client_expectations cl_exp, // for client_expectations
    out ScreenSaver_object_expectations ob_exp,// for object_expectations
    // Parameters specific to this connection, which can be used
    // in predicates for negotiated and reality regions.
    in double max_invoc m_p_s,
    in double max_idle sec) is

    client_callback interface ScreenSaver_client_callback
    object_callback interface ScreenSaver_object_callback
    client_expectations interface ScreenSaver_client_expectations
    object_expectations interface ScreenSaver_object_expectations

// Meta-level interfaces
    contract ScreenSaver is  // CDL

        negotiated regions are

            Allocated:
                when client_expectations.throughput > 0 m_p_s  and
                when client_expectations.throughput <= max_invoc  m_p_s   and
                when object_expectations.capacity >=   max_invoc m_p_s

            Free:
                when client_expectations.throughput == 0 m_p_s and
                when object_expectations.capacity == 0 m_p_s

            transition callbacks are
                Allocated -> Free:
                    object_callback->client_asleep()

                Free -> Allocated:
                    object_callback->client_awake()
                    client_calllback->now_allocated()

            end transition callbacks

        end negotiated regions

        reality regions for Allocated are separate
        reality regions for Free are separate

    end contract ScreenSaver // CDL

// RDL, SDL, etc. go here
end connection invScreenSaver
```

Figure 7: CDL for ScreenSaver Negotiated Regions

```
separate reality regions for ScreenSaver::Allocated:

    Normal:
        when QuO_condition.measured_throughput > 0 m_p_s and
        when QuO_condition.measured_throughput  <= max_invoc m_p_s and
        when QuO_condition.measured_capacity >= max_invoc m_p_s and
        when QuO_condition.measured_idleness <= max_idle secs

    Insufficient_resources:
        when QuO_condition.measured_capacity  < max_invoc m_p_s

    Client_overlimit:
        when QuO_condition.measured_throughput > max_invoc m_p_s

    Client_asleep:
        when QuO_condition.measured_idleness > max_idle sec

    // precedences tell which reality regions are chosen if
    // more than  one predicate is true
    precedence     Normal, Client_asleep, Client_overlimit, No_resources

    transitions callbacks are

        Normal -> Insufficient_resources:
            // Warn the client that there isn't enough capacity, even though
            // we're in negotiated region Allocated and thus there is supposed
            // to be capacity.
            client_callback->warn_no_resources()
            // Tell the object to allocate more capacity (or lower
            // its expectations)
            object_callback->allocate_capacity(max_invoc)

        Insufficient_resources -> Normal:
            // Let the client know that it doesn't have to hold its breath
            // any more
            client_callback->warn_enough_resources()

        any -> Client_overlimit:

            // Let the client know it is exceeding its negotiated promise
            client_callback->warn_overlimit(max_invoc)

        any -> Client_asleep:

            // Let both the object and the client know that the client has gone
            // asleep. One or both may reset their expectations (e.g., the
            // client's throughput or the object's capacity),
            // which could cause a renegotiation.
            client_callback -> warn_sleeping()
            object_callback -> client_asleep()

    end transition callbacks

end separate reality regions ScreenSaver::Allocated
```

Figure 8: CDL for Reality Regions for ScreenSaver Negotiated Region Allocated

6.3 Generate Delegate and System Objects

The QoS Designer runs the CDL code generator to generate the objects in Figure 5. The delegate objects deal with getting the invocation from the object delegate to the remote delegate, while the objects to the right of the delegates deal with implementing the contract governing this client-object connection. The Object Designer adds adaptive behavior to the delegate stubs.

6.4 Implement Client's API for Contract

The Client Developer writes the implementations of the **client_callback** and **client_expectations** objects to handle negotiation time issues. In many cases where an existing contract can be reused there will be multiple versions of the **client_callback** implementations which are candidates for reuse; each implementation would implement a different policy. The **client_expectations** is very simple and is often little more than one CORBA attribute (read and write methods) for each condition in a client's expectations clauses in the CDL.

6.5 Write Client

The Client Designer must write code that interfaces to the delegate object at connection time and invocation time. Note that the QuO API the client uses consists of the CORBA IDL language mappings (e.g., IDL to C++) to three IDL interfaces specified in the QDL: **functional_interface**, **client_callback**, and **client_expectations**.

The Client Designer writes code to call the connection manager, which will create instances of the delegate, contract, and system condition objects shown in Figure 5. The ORB delegate will also bind to the best implementation of the remote object using the knowledge at hand. The policies which choose the best implementation when setting up the connection can be reused and in some cases provided by QuO. These policies can make their decisions using any structural information provided by the programmer, and using information about the status of remote resources which QuO provides. Indeed, the policy which chooses the best remote object to connect to can itself be chosen based on the kind of structural information provided by the programmer.

The connection manager is passed pointers to the binding arguments, the client's arguments to shape the contract (e.g., *max_invoc* in the ScreenSaver contract), and the **client_expectations** and **client_callback** objects. The connection manager returns a pointer to the top-level delegate. It also returns (via reference arguments) a pointer to the object's expectations object (should the client need to query it) and a hook to allow the client to force an evaluation of the regions independent of an invocation to the object.

The Client Designer use the pointer to invoke a method on the remote object. It is used exactly like an Orbix client uses the pointer to the local proxy object returned by *_bind()*, for example, or like a C++ program uses a pointer to a (local) object returned by the **new** keyword.

7. Related Work

A few efforts have recently begun to bring QoS concepts to the CORBA layer. The ODP Trader architecture provides a means to offer a service and to discover a service which has been offered (Object Management Group 1995a). It does have the concept of a "property" which might be used to specify non- functional characteristics of the service being offered. However, the properties are not defined but rather are general placeholders. Further, the Trader architecture seems to be intended not for direct implementation but as a general architecture on which to create more detailed standards with which to implement.

One such specialization is the TINA effort (Leydekkers et al. 1995). TINA is an ongoing effort by telecommunications providers and computer vendors to enable the rapid deployment of telecommunication services, with a focus on real-time multimedia applications. It does involve a contract specifying both the client's usage and the object's QoS provided. However, the set of system properties on which the contract is based is limited. Further, it is not adaptive. QuO permits the specification of multiple regions of operation for a contract, both negotiated and reality regions. QuO provides this because we long ago realized that for current and foreseeable wide-area distributed applications contracts will inevitably be broken, and it is important to provide an orderly way for the application to handle this. TINA has no such provisions for adapting to the breaking of a contract. While this is probably a very good assumption and a proper optimization for a video server delivering a movie a few miles over ATM, it is a poor assumption over wide area and mobile environments.

The open implementation idea is an outgrowth of research in the meta-object protocols (MOPs), which is part of the Common Lisp Object System (CLOS). CLOS is an object oriented language implemented in terms of meta-objects and protocols between them. A programmer can extend the behavior of the language by providing his own meta-classes. The MOP approach is now being applied in more general contexts (Kiczales & Lamping 1993 Masahura et al. 1995; Kiczales et al 1991; Kiczales 1994; Anderson 1995a; Anderson 1995b; Lortz et al. 1994). Application areas include fault tolerance (Fabre et al. 1995), distributed objects (Chiba 1995; Chiba & Masuda 1993), and operating systems (Kiczales & Lamping 1993).

8. Conclusions and Future Research

Distributed applications must become adaptable to cope with system properties which are far from desirable and which vary greatly over time, if they are to be acceptable for mission critical use and cost effectively evolve. QuO develops a cohesive framework for constructing adaptable applications by introducing the concepts for quality of service for object access, and by providing the mechanisms which can be used to integrate these concepts into emerging applications. The initial focus of this work has been on adapting to and managing network communication resources. However, the approach taken also lends itself to a broader definition of quality of service which incorporates other system properties such as fault tolerance, security, and end to end performance. The QuO concept integrates the currently dispersed knowledge about system properties into a cohesive framework, so that both the base value and the variance in these system properties can be improved. This is accomplished using techniques from

traditional operating systems concepts where the runtime software is used to provide an easier to use and more manageable abstract resource than is provided by the existing infrastructure. To maximize the utility and widespread use for immediately applying these results, we have chosen the leading commercially available standards based distributed object computing environment as context for the work.

The QuO architecture presented in this paper is a work in progress. We have prototyped many of the components of the QuO architecture, and are currently in the middle of developing a full prototype implementation. This will include the QDL code generator (with CDL fully implemented), a collection of system condition objects, tools for tracing the invocation patterns of clients, and other components. We believe it is important to provide early, widespread access to these intermediate results in keeping with the best-practices approach to successful R and D activities. Since the concepts being prototyped are based on a number of prior projects which have experienced and dealt with these problems in isolation, we are very confident of the overall approach. The main points of the prototyping activity are to assess the fit of these ideas within the evolving CORBA standards in general, and the particular CORBA ORB and language products in particular, and to obtain engineering estimates of the cost and appropriate granularity of these mechanisms in a current, but modern computing environment.

The information about the existing and expected system properties for distributed objects provides a rich base of information. This can help a distributed application be configured to run under different system properties much easier than if this information were absent or, worse, dispersed throughout the code and the system. This paper just scratches the surface of how different implementations can be deployed at runtime to help the application cope with changing resource availability and partial failures. Our first goal was to put in place the "plumbing" and the concepts needed to organize, acquire and utilize this information at the various times and from the various participants in the software life cycle where they are most pertinent. QuO today does this. In the future we hope to develop a family of QDL languages with both different levels of detail hidden from the programmer and also with domain-specific features. Finally, we hope to broaden the scope of the QoS which QuO offers from communications QoS to include security, partial failures, and other types of system properties.

Acknowledgments

We thank Ken Anderson, Natasha Cherniack, Allan Doyle, Jerry Dussault, J. P. LeBlanc, Dave Pitts, Ray Tomlinson, and Tom Wilkes for their feedback on QuO and on this paper. Also, the comments from the anonymous referees helped improve this paper greatly.

References

Akkoyunlu, E. & Bernstein, A. & Schantz, R. (1974). Interprocess communication facilities for network operating systems. *IEEE Computer*, June, 1974.

Anderson, B. and Flynn, J. (1990). CASES: A system for assessing naval warfighting capability. In *Proceedings of the 1990 Symposium on Command and Control Research*. SAIC Report 90/1508, June 1990.

Anderson, K. (1995a). Freeing the essence of a computation, *ACM Lisp Pointers*, 8(2), 1995.

Anderson, K. (1995b). Compiling a metaobject protocol, in Preparation. To be submitted to Reflection '96.

Bakken, D. & Schantz, R. & Zinky, J. (1996). *QoS issues for wide-area CORBA-based object systems. In Proceedings of the Second International Workshop on Object-Oriented, Real-Time Dependable Systems (WORDS 96)*, IEEE, February 1996.

BBN (1981). Cronus system/subsystem specification. *BBN Systems and Technologies Corporation Report No. 5884*, June 1981.

Burstein, M. & Schantz, R. & Bienkowski, M. & desJardins, M. and Smith, S. (1995). The common prototyping environment. *IEEE Expert*, **10**(1), February 1995, 17-26.

Chiba, S. (1995). A metaobject protocol for C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, ACM, 1995.

Cosell, B. & Johnson, P. & Malman, J. & Schantz, R. & Sussman, J. & Thomas, R. & Walden, D. (1975). An operating system for computer resource sharing. In *Proceedings of the Fifth Symposium on Operating Systems Principles, ACM Operating Systems Review*, **9**(5), November 1975.

Chiba, S. & Masuda, T. (1993). Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the European Conference on Object- Oriented Programming (ECOOP '93)*, 1993.

Fabre, J. & Nicomette, V. & and Tanguy, P. (1995). Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, IEEE, Pasadena, California, June 1995, p.489-498.

Florissi, P. & Yemini, Y. (1994). Management of application quality of service. In *Proceedings of the Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1994.

Gurwitz, R. & Dean, M. & Schantz, R. (1986). Programming support in the cronus distributed operating system. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May, 1986.

Kiczales, G. & J. des Riveres, J. & Bobrow, D. (1991). *The art of the metaobject protocol*. The MIT Press, 1991.

Kiczales, G. (1992). Towards a new model of abstraction in the engineering of software. In *Proceedings of the Workshop on Reflection and Meta- level Architectures (IMSA '92)*, 1992.

Kiczales, G. ed. (1994). Workshop on open implementation '94. *Internet Publication* (URL http://www.parc.xerox.com/spl/projects/oi/workshop-94/).

Kiczales, G. & Lamping, J. (1993). Operating systems: why object-oriented", In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOS '93)*, IEEE, 1993.

Leydekkers, P. & Gay, V. & Franken, L. (1995). A computational and engineering view on open distributed real-time multimedia exchange. In *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '95)*, Boston, USA, April 1995.

Leslie, I. & McAuley, D. & Tennenhouse, D. (1993) ATM Everywhere? *IEEE Networks*, **7**(2), March 1993.

Lortz, V. & Shin, K. (1994). Combining contracts and exemplar-based programming for class hiding and customization. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, ACM, 1994, 453- 567.

Lee, A. & Zachary, J. (1995). Reflections on metaprogramming. *IEEE Transactions on Software Engineering*, **21** (11), November, 1995, 883-893.

Masahura, H. & Matsuoka, S. & Asai, K. & Yonezawa, A. (1995). Compiling away the meta-level in object- oriented concurrent reflective languages using partial evaluation, *Proceedings of the Conference on Object- Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, ACM, 1995, 300-315.

Object Management Group (1995a). *ODP Trading Function*. OMG Document 95-07-06, June 1995.

Object Management Group (1995b). *The common object request broker: architecture and specification*. OMG Document 96-03-04, July 1995.

Partridge, C. & Pink, S. (1992). An implementation of the Revised Internet Stream Protcol (ST-2). *Internetworking: Research and Experience*, March 1992, 27-54.

Phuah, V. & Nicol, J. & Gutfreund, Y. (1996). ATM to the Desktop: Impacting Modern Business Communications with Broadband Technology. *Telematics and Informatics (Special Issue on Multimedia Technologies, Systems, and Applications)*, to appear.

Schantz, R. & Thomas, R. & Bono, G. (1986). The architecture of the Cronus distributed operating system. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May, 1986.

Sasnett, R. & Nicol, J. & Phuah, V. & Gutfreund, S (1994). Experiences Developing Distributed Business Multimedia Applications. In *Proceedings of the First International Conference on Multimedia Computing and Systems*, IEEE, May 1994.

Topolcic C. (1990). An experimental internet stream protocol: Version 2 (ST-II). *Internet RFC 1190*, October 1990.

Walker, E. & Dean, M. (1995). The migration of Cronus to CORBA. In *Proceedings of the Fifth IEEE Dual Use Technologies and Applications Conference*, May, 1995.

Walker, E. & Floyd, R. & Neves, P. (1990). Asynchronous remote operation execution in distributed systems. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, June, 1990.

Walker, E. (1991). Decision aids for crisis action planning. In *Proceedings of the AFCEA Mission Planning Symposium*, September 1991.

Waldo, J. & Wyant, G. & Wollrath, A. & Kendall, S. (1994). A note on distributed computing. *Report SMLI TR-94-29*, Sun Microsystems Laboratories, November 1994.

Zhang L. & Deering S. & Estrin D. & Shenker S. & Zappala D. (1993). RSVP: a New Resource ReSerVation Protocol. *IEEE Network*, **7**(6), September 1993, 8-18.