

Overview of Quality of Service for Distributed Objects

John A. Zinky
David E. Bakken
Richard Schantz

Distributed Systems Department,
BBN Systems and Technologies
10 Moulton St., Cambridge, MA 02138

Abstract ¹

A gap exists between the socket-level Quality of Service (QoS) offered by communications researchers and the object-oriented programming being done at the application level. To bridge this gap, we introduce a framework for handling system properties such as QoS and partial failures.

This framework 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 *system states*, which represent the status of the QoS agreement for an object connection. The application can adapt to changing resources and partial failures by changing its behavior based on the system state of its object connections. Many hooks are provided for measuring and enforcing QoS agreements and for dispatching handlers when the agreements are violated.

1. INTRODUCTION

In recent years the benefits of object-oriented technology have led to its widespread use in all areas of software development. Achieving Interoperability using objects is gaining acceptance with the adoption of the CORBA standard [1,2]. However, the functional interoperability that CORBA offers is only a part of the object-oriented support that many distributed applications need, since they must cope with an environment that is dynamic.

Most distributed applications, other than multimedia-based ones such as video and audio, do not know their object invocation patterns or their delay

requirements. These non-multimedia applications have difficulty coping with changing resource availability. As a result, most of them are hand-tuned with great effort for performance and resource management reasons.

Furthermore, the environments in which distributed applications must operate are far from ideal; anomalies do happen. Unfortunately, few high-level programming environments deal with partial failure or resource management issues [3]. As a result, many real-world distributed applications typically do not deal with anomalous conditions and thus work only in an ideal environment. When they are deployed, these ideal conditions do not exist, so anomaly-handling code is added in an *ad hoc* fashion as anomalies are empirically discovered. In the later stages of the application's life cycle, most of its code (sometimes 90%) is dedicated to anomaly handling. This code is difficult to maintain and evolve; in a word, expensive.

Recent advances in networking technologies offer hope in coping with the above problems. For example, ATM and RSVP offer Quality of Service (QoS) guarantees which let an application specify its requirements *viz.* bandwidth, delay, etc.. Also, fiber optics are becoming more widely deployed, and they offer low bit error rates and low latency in addition to the high bandwidth. Multicast and point-multipoint services available from ATM and elsewhere also offer hope that replication can be more efficiently implemented to improve performance and availability.

Unfortunately, the QoS guarantees of ATM and RSVP are in network terms, e.g. bits per second, not in application terms such as invocations per second. To help overcome this shortcoming, we are extending the CORBA interface to bridge the conceptual gap between network-level guarantees and application-level requirements. To do so, we are introducing a resource management and availability model that is natural to the CORBA layer, which we call Quality of Service for Objects (QuO). QuO will exploit underlying QoS and the multicast support the network offers. It will also enable the QoS actually received to be visible to the application, to enable it to adapt to changing conditions.

We impose two major constraints on QuO to ensure its computability with existing applications, especially collaborative planning ones. First, we will *not* create a specialized programming language to handle QoS, because this would greatly limit the number of applications that would actually use it. Rather, we will use implementation techniques found in CORBA

¹This work is sponsored by Rome Lab Contract No. F30602-94-C-018 "Distributed Computing over New Technology Networks"

To be presented at the 1995 IEEE "Dual Use Technologies Conference," Utica NY. May 23, 1995

environments such as external description languages, language veneers, and code generators. However, we will assume that the application programming language we use with QuO is object-oriented with exceptions. C++ is thus the least common denominator, and Ada 9X, Smalltalk, Python and Common Lisp would also work. We will not consider supporting non-object-oriented languages such as C, Ada, FORTRAN, or Tcl. To use such languages would require that we implement exceptions and inheritance to support the object-oriented QoS management described in Section 2.C and the hierarchical masking of system conditions discussed in Section 2.F.

The second major constraint is that QuO must be implemented above the ORB layer. This helps ensure that QuO can be made portable to and interoperable with multiple ORBs. Also, a layer above the ORB can employ optimizations such as caching to use abundant client-side resources to optimize ORB and network usage.

We are developing the initial prototype with collaborative planning applications in mind. We hope to help programmers of collaborative applications to better understand their network requirements and enable them to build stable applications without requiring extensive customization or hand-tuning.

2. COLLABORATIVE PLANNING

Distributed collaborative planning programs allow people in multiple locations and with varying and diverse expertise to cooperatively solve a problem. Figure 1 shows an example of this, a collaborative campaign planning system that was demonstrated in JWID '94. There are a number of attributes of this system that are germane to our discussion. First, note that this is not a supercomputer but is rather a loosely coupled collection of widely-dispersed computers. Also, this application is not client-server based. Rather, information flows between numerous entities that are implemented with an object-oriented programming paradigm. Finally, this is not a "workflow" application with information being transformed in a simple assembly line fashion. The information being shared is flowing with varying degrees of synchronization; some information is totally unrelated to and uncoupled with other information in the system, other data is manipulated in an assembly line, while the flow of yet other information falls somewhere in between. Thus, this really represents a knowledge flow system.

In particular, such collaborative planning systems have multiple usage patterns, as depicted in Figure 2. Here, a collaborative planning system is creating shared plans using a map as a shared whiteboard and a video

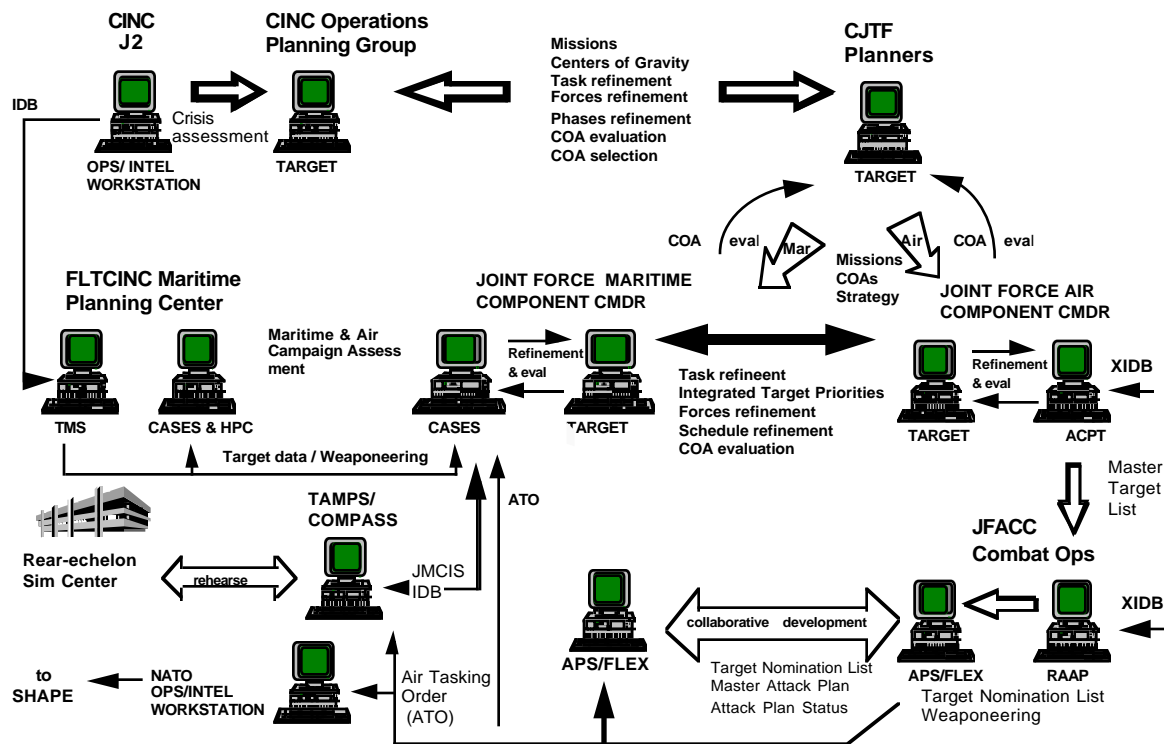


Figure 1 Collaborative Campaign Planning in JWID '94

conference to aid communication. The shared plans are computer-computer communication. The resource requirement patterns may not be easy to describe or ascertain, but there is a relatively large amount of variance allowable in their delivery. It is thus delivered on a best-effort basis. On the other end of the spectrum is the video information.

It is human-human communication and as such is real-time information that permits little variance in its delivery. However, its resource requirement patterns are fairly easy to know *a priori*. In between these two extremes is the shared workspace. Its information does not have to be delivered in real time, though it must be delivered in a reasonably timely fashion since it is used for humans to collaborate with. Its delivery thus must be predictable. While QuO can handle all three kinds of usage patterns, for our first prototype we will concentrate on supporting the predictable shared workspace information.

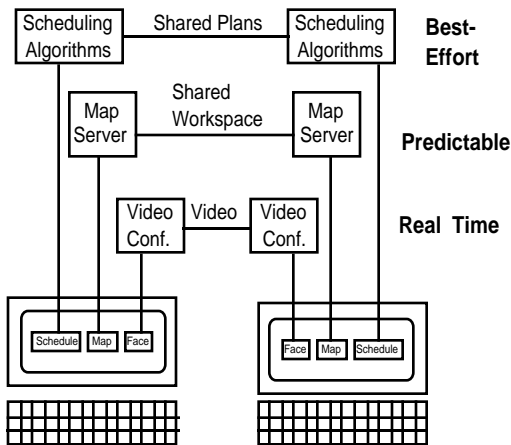


Figure 2: Collaborative Planning Usage Patterns

3. QUALITY OF SERVICE FOR OBJECTS

A. End-to-End QoS and Traffic

In communications networks, QoS is typically specified in terms of host-to-host parameters. However, QuO defines an object-oriented QoS that must take an end-to-end perspective, from the client to the object and back. For example, the components of the delay a user of an object sees is given in Figure 3. Here, the delay includes the following components:

1. Parameter marshaling at client
2. Network delay for parameters
3. Parameter unmarshaling at server
4. Method execution
5. Results marshaling at client
6. Network delay for results
7. Results unmarshaling at client

The network delay specified by ATM's QoS, for example, is for #2 or #6, while the client of the object sees all steps of the delay. Thus, QoS between an object and its client must account for changing resources and partial failures involving all seven steps.

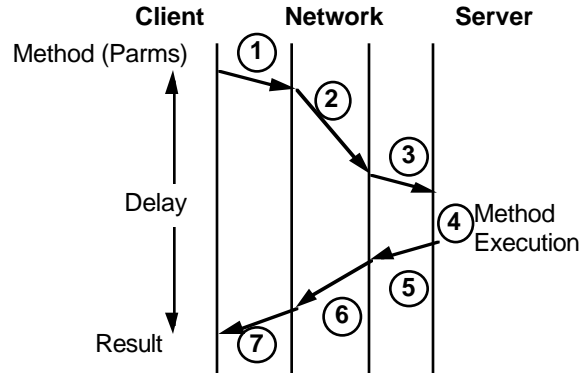


Figure 3: End-to-End QoS Components

An end-to-end QoS contract between an object and its client must consist of two parts: the QoS the client requires from the object and the usage pattern the client promises to abide by. The usage pattern must account for at least the following items:

- Rate: methods/sec, which methods (read/write)
- Parallelism constraints: how many outstanding calls (for asynchronous invocations)
- Priority

The QoS requirements that the object client requires from QuO involves at least the following:

- End-to-End delay (invocation to result)
- Capacity (max methods/sec)
- Availability (how often is object usable?)
- Ordering constraints (can asynchronous invocations be executed by the object or return to the client out of order?)
- Error rate (can I handle an invocation never returning?)

Consider a typical object-oriented program running in a single address space. It operates under the following *de facto* QoS requirements and usage patterns:

- Rate: 0 to infinite methods/sec. The client can deliver any rate from zero to unbounded. But if the object were distributed, this lack of constraints can have adverse interactions with other parties.
- Parallelism constraints: none. The client does not receive any parallelism in the form of asynchrony. But a distributed application may need parallelism just to overcome communication delays, for example, window-based protocols.
- Delay: 0. The client assumes no overhead for method calls. But the overhead for a remote call is at least an order of magnitude slower than an in-core call.

- Availability: 1. In-core objects never fail unless the whole process fails. But distributed objects may fail while their clients continue.
- Ordering constraints: Totally Synchronous, the order of operations is well-defined in a single process. But messages can get out-of-order when traveling to distributed objects.
- Error rate: 0. Results are never corrupted for in-core operation. But remote messages are exposed to noise.

As you can see, changing resources and partial failures are virtually absent in the single address space, yet are a fact of life in a distributed environment. The costs of supporting In-core QoS requirements for a distributed object-oriented application would be extremely expensive. Thus, if distributed applications ignore QoS constraints altogether it would result in fragile programs, wasteful use of resources, or both. With the application program and object designer specifying QoS requirements and usage patterns the system will have more options with which to efficiently help the program adapt to changing system conditions.

B. QoS Management: Client-Server versus Object Oriented

Distributed objects provide benefits in helping manage end-to-end QoS, as shown in Figures 4 and 5. Figure 4 shows QoS management in a client/server system. An end-to-end QoS requires coordination among the client's runtime system, the socket-level QoS the OS can negotiate, and the server's runtime system. This is a very difficult task; one way to implement it is with an external agent such as the one provided in the Qual language [4].

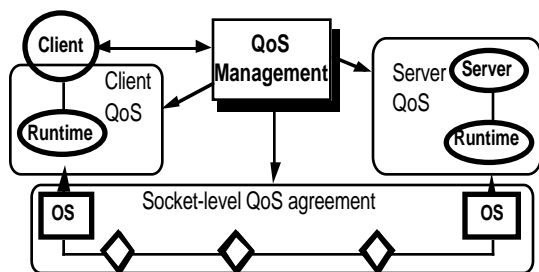


Figure 4: Client/Server QoS Management

An object-oriented QoS simplifies QoS management, as shown in Figure 5. Here, pieces of the object's code are executing at all points where QoS information needs to be obtained and monitored. The object is thus responsible for managing its QoS. This scheme has significant advantages over a socket-level QoS agreement where the external agent must coordinate agreements from many sources outside its control such as the object manager, the client's runtime system, and the socket-level QoS.

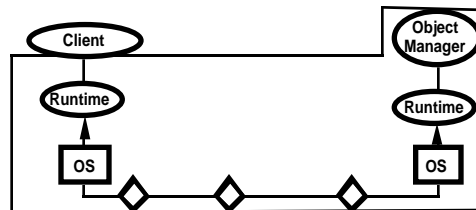


Figure 5: Object-Oriented QoS Management

C. Proxy Objects Blur Location

Proxy objects are a flexible mechanism present in many CORBA implementations. They allow the object designer to place a representative or proxy -- some of the object's code -- into the client's address space. A pointer to the proxy is returned transparently to the client when it obtains a reference to a remote object. The proxy is an instance of a child class for the remote object. As a child of the base class it can override some or all of the base class's method's. For example, it can cache data or invoke one of multiple implementations of the object. Finally, multiple layers of smart proxies may be deployed as ancestors of a base class, as demonstrated in Section 3.F.

A benefit of using proxy objects is illustrated in Figures 6 and 7. The first generation of distributed object implementations was client-server based. As a result, the server is often located on a separate host, Figure 6. The fact that the server has a *location* can affect the client. For example, if the server's host fails, the client must take explicit steps on a per-object basis (with knowledge of the object) to adapt to this failure.

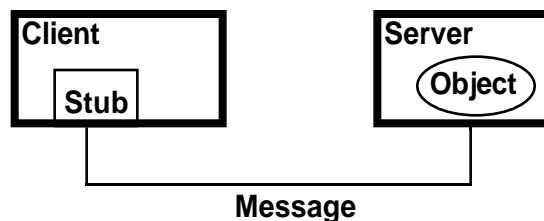


Figure 6: Client/Server Objects Have Location

Contrast this with the proxy object-oriented scheme in Figure 7. Here, the object's state and code are spread throughout the address spaces of clients of that object via smart proxies. The object's proxy code functions as a representative of the object in the client's address space. This allows the server object's designer to take action to compensate for the failure of the remote object state transparently to the client, or at least with better failure semantics. The object designer can handle the failure of its object better than the client can since the designer has detailed knowledge of the object. Having this detailed knowledge in the client's address space enables the client to have better failure handling, since if the client has not failed, then the proxy in its address

space also will not have failed, even if the object's remote representation has done so.

This scheme also allows proxies for the object to be deployed where needed. For example, clients that invoke just read-only methods can have one proxy for their read-only connection, while clients that invoke methods which modify the object's state can be given different proxies to implement the distributed modification of that object's state.

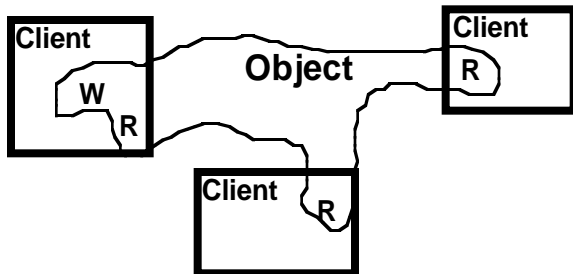


Figure 7: Smart Proxies Distribute Object Location and State

D. Object Connection States

Distributed programs often are required to operate under many different sets of conditions. For example, the same collaborative planning system that works well on a LAN may be required to provide some reduced-capability service to a user connected by a low-bandwidth satellite link when the user is mobile or during recovery from a communications failure.

QuO thus allows the object designer to specify the *system states* for an object connection. These states as well as the usage and QoS for them are specified at connection time with QDL as described below. The usage pattern specifies the traffic constraints the client of the object agrees to abide by when the connection is in the given state, and the QoS specifies the QoS that will be required while the object connection is in the given state.

System states can have handlers attached to them. The state of a connection is monitored, and handlers are called automatically when the state of a connection changes.

Figure 8 gives an example set of states for an object connection. Here the connection usually operates in the Normal state. However, if resources become scarcer the connection will switch to a degraded state; the client will be informed so it can adjust its behavior appropriately. If resources get even scarcer, below what the application can operate on, then the connection will go to an underflow state and the object client will have to take other action.

Similarly, if the client's usage is beyond what it promised for the Normal state, the connection will transition to the over-limit state. If the object client

goes beyond the bounds of this state's promised usage then it will move to the overflow state.

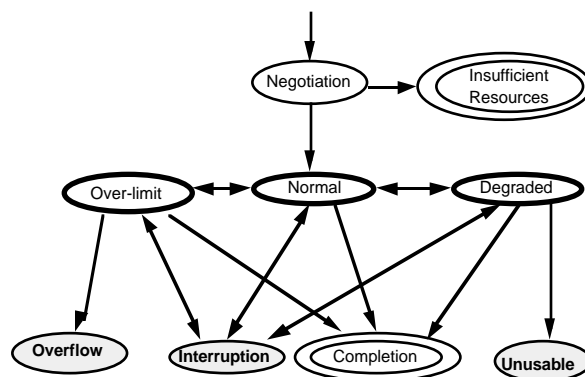


Figure 8: Example Object Connection States

E. Adaptivity Implies Multiple Behaviors

QuO provides an object developer's toolkit that enables alternative implementations of an object to be used depending on system conditions. This circumvents CORBA's early binding where the object designer commits to one implementation for an object at design time. QuO provides a mechanism for multiple implementations to be provided and automatically switched between as system conditions change.

There are three aspects to supporting this mechanism. The first involves the system states of the object. These states are specified at design time through a QoS description language (QDL), which is described below. The exact range of operation for a particular state can be customized at object connection time by parameters supplied by the client. Also, the conditions that define the states must be monitored to know which one is currently active. This is accomplished by the QuO proxy layers, and the state is changed as conditions warrant without any intervention by the client.

The second aspect in supporting multiple behaviors involves allowing the object client to take action when a state changes. This is accomplished by binding handlers to state transitions at connection time via QDL. These handlers are then invoked when the state changes, thus allowing action to be taken to adapt to the change in state.

The third aspect of supporting multiple behaviors is to assist the object designer in writing handlers for a given state or state transition. This will be aided by a QuO toolkit that provides libraries for instrumentation, replication, and other functionality often required by the programmers.

This framework supports three major concerns for designers and users of distributed applications:

- **Adaptivity.** The framework allows the distributed application to adapt to changing resources and usage patterns.
- **Software Engineering.** The framework permits the work of many programmers or even disparate applications to be integrated in a framework that not only handles a functional interface, but also handles system issues.
- **Evolvability.** The definition and automatic detection of states plus, the clean separation of code for each state and state transition makes it much easier for the application to handle new system conditions.

F. QoS Description Language

CORBA's IDL is a functional interface specification language. QuO builds on IDL by adding a QoS description language (QDL) that concerns system properties. The QDL specifies for a given connection the system states for that connection as well as the promised usage and required QoS for each state. QDL also specifies handlers to be called automatically when an object connection's state changes.

QDL will be processed by a code generator to generate layers of smart proxies, as shown in Figure 9. These proxy layers will then use the Orb's proxies (as generated by the IDL compiler) to connect with remote parts of the object.

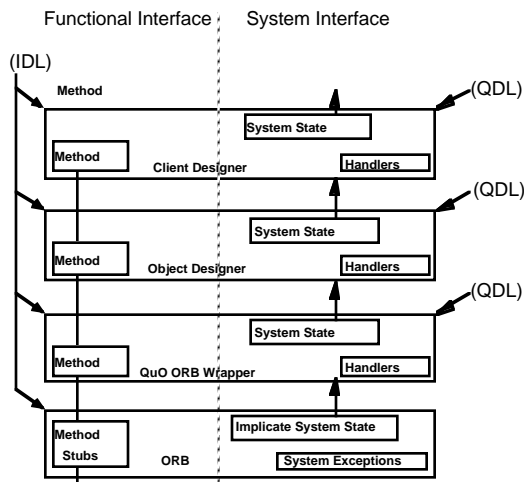


Figure 9: Layers of Smart Proxies

G. Multiple Object Connections

A client can open up multiple connections to an object, each with a different set of system states and associated QoS and usage patterns. For example, in the following code:

The client opens up two connections to the same object, one with a low delay and the other with high throughput. At connection time, the proxy layers and state transition handlers specified in the connection

contract are bound to the client's address space. The different connections are then used for different invocations. For example, the high throughput connection would be used in a loop, while low delay would be used in part of a control setup.

Begin

```
OLowDelay <- Connect(ObjectID,
                    ConnectTypeDelay, RTParams)
```

```
OHighThrp <- Connect (ObjectID,
                    ConnectTypeThrp, RTParams)
```

```
OLowDelay.Method (Params)
```

Loop

```
OHighThrp.Method (Params)
```

Endloop

End

Figure 10: Multiple Connections to an Object

4. CONCLUSIONS

This paper shows the need for extending Quality of Service from the communication-layer to the object-layer. QuO provides a framework which extends CORBA functional interfaces to also support system properties, such as QoS. QuO will allow applications to adapt to changing usage patterns and underlying resources. Future papers will explore QuO through examples and examine its implementation mechanisms in detail.

5. ACKNOWLEDGMENTS

Many thanks the COTR Jerry Dussault.

Also, thanks to Allan Doyle, Mary Trvalik, Tom Mitchell and Ray Tomlinson of the MATT Project for being a testbed for QuO.

6. REFERENCES

- [1] Soley, Richard M. ed. *Object Management Architecture Guide*, Object Management Group, 1993.
- [2] Nicol, John R., Wilkes, C. Thomas, and Manola, Frank A. *Object Orientation in Heterogenous Distributed Computing Systems*. IEEE Computer, 26:6, June, 1993. p. 57-67.
- [3] Bal, Henri E., Steiner, Jennifer G. and Tanenbaum, Andrew S. *Programming Languages for Distributed Computing Systems*. ACM Computing Surveys, 21:3, September, 1989, p261-322.
- [4] Florissi, Patricia and Yemini, Yechiam, *Management of Application Quality of Service*, DSOM 94, Oct 94