Abstract-- The Internet is a desirable platform for many new forms of distributed applications in business, science, and education. A side effect emerging from rapid widespread and uneven acceptance and fast paced new extensions of the Internet communication technology base is the varying operating conditions under which a distributed application can be expected to be running. For mission critical applications especially, and for maximum user satisfaction in general, effectively selecting and controlling the operating environment and behavior of these applications adaptively, without introducing significant software complexity, is a problem which ranges from difficult to impossible, depending on how much adaptivity is needed. A key part of the problem is a limited ability to easily identify and react to changes in anticipated operating conditions without damaging the integrity of the application. We are developing a Quality of Service based middleware layer imposed between applications and the Internet as a potential general architectural solution. This middleware makes it significantly easier to develop and control application behavior under variable and changing operating conditions. We describe the background motivating this work, the components of our object computing based middleware solution, and show how components of this middleware are being used right now as part of a test and evaluation in the current Internet environment.

Index Terms-- Middleware, Quality of Service, Adaptive Computing, CORBA, QuO, replicated web access

I. INTRODUCTION

One side effect of the rapid growth and deployment of communication-based distributed applications, particularly Internet-based applications, is the generally recognized need for more attention to system level usability and the accompanying flexibility and control of the communications functions in service of these applications. In these environments it is difficult or impossible to predict ahead of time even approximate configurations or load mixes, making it mandatory to develop approaches that support varying but predictable behavior at different times during an application’s life cycle and adapting to current operating conditions. Quality of Service (QoS) is the term that is being used to organize the loose collection of activities and technology initiatives that have emerged to improve and control network oriented resource management based on mounting experience with distributed, Internet applications.

While the quality of the service provided has always been a factor influencing effective use, it is only recently that Quality of Service as a named attribute has become the focus for directed distributed infrastructure enhancements. This new focus stems largely from previous Internetwork limitations in areas such as unsynchronized media streams in distributed multimedia applications, or in lacking assured communication resources for high priority users during high-demand periods, with the side effect of diminishing available resources to others. QoS concerns have led to a number of implemented and suggested improvements to the distributed computing infrastructure.

Richard Schantz, John Zinky, Joseph Loyall, Richard Shapiro, James Megquier

Adaptable Binding for Quality of Service in Highly Networked Applications

BBN Technologies, A Part of GTE
Distributed Systems Group
10 Moulton St.
Cambridge, Massachusetts 02138 USA
Schantz@bbn.com

Taken in their narrowest form, the QoS improvements are concerned with the capabilities and control mechanisms available within the communications network itself. Recent work has extended the QoS viewpoint to include not only an end to end application picture, but also the integration of other properties affecting quality of service beyond timely communication services. In its largest sense, QoS involves decision making and actions beyond the network itself, as well as a multitude of properties beyond the application-specific functional behavior of a particular distributed application. Examples of these properties include behavior and adaptability under various changing environments to achieve degrees of performance characteristics, dependability, and security. This article is about Quality of Service (QoS) in this larger context as an organizing concept for integrated resource management for distributed computing infrastructures and applications. Much of this material is work in progress, with emphasis on emerging trends and directions.

II. MIDDLEWARE

Requirements for faster development cycles, decreased development effort, and reusable solutions motivate the use of middleware. Middleware is software that resides between applications and the underlying operating systems, protocol stacks, and hardware to enable or simplify how these components are connected and interoperate. The role of middleware, such as Sun's Jini[15], Java RMI[18], and EJB[16] frameworks, Microsoft's DCOM [1], and the Object Management Group’s (OMG) CORBA[8, 9, 10] middleware, is to decrease the cycle-time and level of effort required to develop high-quality, flexible, and interoperable distributed and embedded systems using reusable software infrastructure component services, rather than building systems entirely from scratch for each use. In general, middleware provides the following benefits: (1) it shields software developers from low-level, tedious, and error-prone details, such as socket-level programming [13], (2) it provides a consistent set of higher-level network-oriented abstractions for developing distributed and embedded systems, and (3) it amortizes software lifecycle costs by leveraging previous development expertise and capturing implementations of key design patterns [2, 3] in reusable frameworks, rather than rebuilding them manually for each use.

When middleware is commonly available for acquisition or purchase, it becomes commercial off-the-shelf (COTS). While it is possible in theory to develop complex systems from scratch, i.e., without using COTS middleware, contemporary economic and organizational constraints, as well as competitive and interoperability pressures, are making it implausible to do so in practice. Thus, COTS middleware plays an increasingly strategic role in software intensive, mission-critical distributed systems, which is why we base our adaptive QoS-centric R&D activities on COTS middleware.

Distributed object computing (DOC) is the most advanced, mature, and field-tested middleware paradigm available today in which to achieve this flexible, adaptable behavior. DOC software architectures are composed of relatively autonomous objects that can be distributed or collocated throughout a wide range of networks and interconnects. Clients invoke operations on target objects to perform interactions and functionality needed to achieve application goals. Within the family of distributed object computing models, we are initially focusing activities on CORBA because it is heterogeneous and because it has a well-established, successful, and open standardization process. Moreover, an increasing number of standards-compliant CORBA implementations [14] are now available for use in mission-critical systems.

Due to earlier constraints on footprint, unavailability of high quality COTS middleware, unavailability of trained personnel, and severe development time constraint for initial operating capability, development of mission-critical distributed systems has historically lagged behind mainstream software development methodologies by a considerable amount. As a result, these types of software systems have been extremely expensive and time-consuming to develop, validate, and optimize, as well as error prone to deploy, maintain, and upgrade. Moreover, they
are often so specialized and tightly coupled to their current configurations and operating environments that they cannot be adapted readily to new market opportunities, technology innovations, new patterns of user behavior or changes in run-time situational environments.

In addition to the development methodology and system lifecycle constraints mentioned above, designers of mission-critical systems have historically used relatively static methods and policies for allocating resources to system components, which often possess competing real-time requirements. For instance, systems with real time constraints [7] have traditionally established the priorities for all resource allocation and scheduling decisions very early in the system lifecycle, i.e., well before run-time. This leads to a reliance on “worst-case” and overconstrained use of resources. Static strategies have been used because (1) system resources were insufficient for more computationally intensive on-line approaches and (2) simplifying analysis and validation was essential to remain on budget and on schedule, particularly when systems were designed from scratch using low-level, proprietary languages, operating systems, interconnects, and software tools. The situation that led to these decisions in earlier generations of distributed systems is rapidly changing with the emergence and availability of efficient COTS middleware as a starting point for new systems operating over high speed internets.

III. TECHNICAL BACKGROUND

In any distributed middleware architecture, the functional path is the flow of application level information between a client's invocation to a remote object and back. The middleware is responsible for exchanging this information efficiently, predictably, scalably, and securely, between the remote entities by utilizing the capabilities of the underlying network and endsystems. The information itself is largely application-specific and determined solely by the functionality being provided (hence the term “functional path”). The functional path deals with the “what” of the client object interaction from the perspective of the application, e.g., what function is to be requested for that object, what arguments will be provided, and what results, if any, will be returned to the client.

In addition to providing middleware that supports the functional path, we are now also concerned with a system path (a.k.a., the “QoS path”) that handles issues regarding “how well” the functional interactions behave end-to-end. Thus, middleware is also intimately concerned with the non-functional aspects of distributed application development. This involves, for example, the resources committed to client object interaction and possibly subsequent interactions, proper behavior when ideal resources are not available, the level of security needed, the recovery strategy for detected faults, etc. A significant portion of the middleware needed to do this is dedicated to facilitating the collection, organization, and dissemination of the information required to manage how well the functional interaction occurs, and to enable the decision making and adaptation needed under changing conditions to support these non-functional “how well” QoS aspects.

We separate the non-functional QoS requirements from the functional requirements for the following two reasons:

1. To allow for the possibility that these requirements will change independently, e.g., over different resource configurations for the same applications; and

2. Based on the expectation that the non-functional aspects will be developed, configured, and managed by a different set of specialists than those customarily responsible for programming the functional aspects of an application.

There are three complementary parts to an advanced adaptive QoS middleware organization:

1. The first part deals with the features and components needed to introduce the concepts for predictable and adaptable behavior into the application program development
An environment, including specification of desired levels of QoS aspects.  

2. The second part deals with providing run-time middleware to ensure appropriate behavior, including collecting information and coordinating any needed changes in behavior.  

3. The third part deals with inserting the mechanisms for achieving and controlling each particular aspect of QoS that is to be managed, including aggregate allocation, scheduling, and control policies.

Integrating these facets and inserting appropriate mechanisms and behavior is a significant development job.

IV. OVERVIEW OF QUO

To make it easier to construct adaptive QoS distributed applications we are developing additional COTS middleware support infrastructure. Certain attributes or parts of a QoS implementation are independent of the particular quality that is being managed. For each part, there are a wide variety of techniques to accomplish QoS objectives in a complex networked environment. In different implementations, each part may be present to a lesser or greater degree depending on the requirements of the particular environment. Typically all are present, although not always accessible or visible to end-users.

Quality Objects (QuO) is an extension of a distributed object computing (DOC) framework designed to support developing distributed applications that can specify and customize (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time. By providing these features, QuO opens up distributed object implementations [4] to control an application’s functional aspects and implementation strategies that are encapsulated within its functional interfaces. To achieve these goals, QuO provides middleware-centric policies and mechanisms for developing DOC applications that can perform the following operations in addition to their functional behavior:

- Specify different operating regions and service requirements for these regions:
  QuO-enabled applications can specify their levels of desired performance or resources (which might change dynamically based upon changes in the environment), operating modes (corresponding to different functional objectives of the application), and operating regions (corresponding to different environment or system conditions, resource availability, etc.).

- Measure environmental and system conditions:
  QuO-enabled applications can insert and use probes in their distributed environment to measure resources, characteristics, and behavior. In addition, these applications can receive information from resource managers, real-time operating systems or ORBs, and other property managers and mechanisms.

- Access to control interfaces:
  QuO-enabled applications can access lower level infrastructure system resource management control interfaces and pass information to resource or property managers to achieve their desired level of service.

- Adapt and reconfigure:
  QuO-enabled applications and systems can adapt to changing conditions at all levels, coordinated through the QuO middleware. For example, in response to changing mission objectives or degraded resources, a QuO-enabled system can respond through adaptation on the part of each of the resource managers, resource control mechanisms, QuO middleware, and application programs.

The functional and system path of QuO illustrated in Figure 2 is a superset of the functional path of CORBA illustrated in Figure 1. The components provided by QuO to support the above operations are defined below.
• Contracts:

The operating regions and service requirements of the application are encoded in contracts, which describe the possible states the system might be in, as well as which actions to perform when the state changes. The possible states are specified as a set of regions, which can be nested. Each region is defined by a predicate over a set of system condition objects, which are described below. A contract also defines a set of transitions between regions, which specify adaptive behavior that is triggered when the system state (as defined by the contract regions and predicates) changes (as represented by a contract transitioning from one valid region to another).

• Delegates:

QuO inserts delegates in the CORBA functional path. Delegates project the same interfaces as the stubs (client-side delegate) and the skeletons (server-side delegate), but support adaptive behavior upon method call and return. When a method call or return is made, the delegate checks the system state, as recorded by a set of contracts, and selects a behavior based upon it.

Contracts and delegates support two means for triggering manager-level, middleware-level, and application-level adaptation. The delegate triggers in-band adaptation by making choices upon method calls and returns. The contract triggers out-of-band adaptation when region transitions occur which can be caused by changes in observed system condition objects.

• System Condition Objects:

These objects provide uniform interfaces to multiple levels of system resources, mechanisms, and managers to translate between application-level concepts, such as operating modes, to resource and mechanism-level concepts, such as scheduling methods and real-time attributes. System condition objects are used to measure the states of system resources, mechanisms, and managers that are relevant to contracts in the overall system. In addition, they can pass information to interfaces that control the levels of desired services.

Higher-level system condition objects can interface to other, lower-level system condition objects, forming a tree of system condition objects that translate mechanism data into application data. System condition objects can be either observed or non-observed. Changes in the values measured by observed system conditions trigger contract evaluation, possibly resulting in region transitions and triggering adaptive behavior.

Observed system condition objects are suitable for measuring conditions that either change infrequently or for whom a measured change can indicate an event of notice to the application or system. Non-observed system condition objects represent the current value of whatever condition they are measuring, but do not trigger an event whenever the value changes. Instead, they provide the value upon demand, whenever the contract needs it, i.e., whenever the contract is evaluated due to a method call or return or due to an event from an observed system condition object.
Observed system condition objects can measure frequently changing system conditions by coding the system condition object to smooth out continuous changes. For example, a system condition object measuring the load on a host can be observed. However, it can be programmed to only report periodic events showing average load over some time duration. Likewise, it can be programmed to only report events when the load crosses a certain threshold.

There are other pieces of the QuO framework (e.g., instrumentation, object gateways, quality description languages used to automate much of the software engineering involved in constructing and configuring the adaptive middleware pieces), but they are beyond the scope of the work reported here. See [5, 6, 11, 12, 17, 19] for more information on the design and implementation of QuO.

V. THE WEB PAGE REDIRECTION PROJECT

During the course of developing our Quality Objects middleware concepts, we were presented with an opportunity to utilize some of these emerging ideas and some of the software on which it is currently based. Although it was a simplified “application” from what we have been considering as a model for the QuO toolset and abstractions, it was nonetheless an opportunity to work out details for and validate some of the inner parts within real world constraints and performance considerations.

The needed development was for a replacement load balancing content distribution capability for Web browsing, where the relevant Web material was predeployed to a number of Web hosting sites throughout the world. There was a current capability, but it suffered from many of the software engineering difficulties mentioned earlier of somewhat ad hoc solution software driven to immediate use by market forces. The problem to be addressed was a transparent replacement with a well engineered, flexible underlying load balancing capability for web browsing applications which initially performed the same load balancing algorithms as the current one but eventually would be enhanced with more sophisticated approaches, as well as support for other functionality beyond load balancing (e.g., better fault tolerance capabilities and security).

So in contrast to building new, network aware adaptive applications, one of our first uses of the QuO middleware concepts turned out to be using a subset to develop an application transparent extension of the prevailing network infrastructure. Its value derives from inserting performance boosting and load equalizing adaptive behavior into existing applications (e.g., web browsing) which by requirement would be unaware that the new behavior was in place and uninvolved in resource management decision making. The product name for this collection of adaptive middleware applied to this problem is Hopscotch.

A. Summary of Hopscotch Services

Hopscotch is a traffic distribution service, which is intended to deliver improved web page access to its end users. For high traffic Web sites the actual quality of service a Web server provides to end users typically depends on two parameters – network-transfer speed and server-response time/availability. Server response time will degrade when the server becomes overloaded with HTTP requests and resource intensive processes (CGI, streaming media, etc.). They can also become unavailable as a result of a crash or being brought down for routine maintenance. Network transfer speed can slow down because routers at the peering points become overloaded or backbones become congested, or simply because the client is geographically far away (in terms of number of routers or speed of connections) from a particular server.

Hopscotch provides load distribution and failover by distributing users to one of multiple, replicated Web servers based upon criteria such as the speed of the underlying network infrastructure, server load, and server availability. When an end user requests a web page from a replicated web site, Hopscotch redirects the end user to an available

1 Hopscotch is a registered trademark of Genuity Inc.
Web server that will offer the assumed best performance for that user based on information collected about current operating conditions. For this application, the existing HTTP middleware protocols and interfaces served as the development context.

Thus, the Hopscotch Project involved the creation of distributed adaptive middleware, which extends the HTTP middleware with additional system path concerns. The problem is to adaptively bind what amounts to logical resource (web page) requests to appropriate physical instantiations based on the current operating conditions. Clients send HTTP or DNS requests to one of many Hopscotch Redirectors, which redirects the request to a specific web server. Redirection decisions are based on both server factors, such as CPU load and memory usage, and network factors, such as delay and drop rate from the server to the client. The web servers are physically spread throughout the world and several servers can typically meet a request. Each server has a monitor process which can collect information about the server’s current system properties and actively participates in measuring the current network latency from the server to the client.

Figure 3 illustrates the concept being developed. In it web browser software operates without modification requesting Web pages either directly via a URL or indirectly via a Domain Name Server (DNS) lookup. Hopscotch uses two methods by which it can redirect users to the appropriate servers. The first is HTTP temporary session redirection (HTTP status code 302). The second is DNS mode where Hopscotch acts as the primary Domain Name Service (DNS) server for a specific domain. The Hopscotch subsystem itself provides responses to these DNS queries.

These requests are intercepted by the redirection software system, which uses various measurement and status checking daemon processes to collect information on the current operational situation. After appropriately collecting and distributing the status and load information, a decision is made as to which site to redirect that Web browser to use. From that point, it looks to the Web server software at the designated (replicated) Web page site like a normal request designated specifically for it. The mechanisms to achieve this level of application transparency, while interesting, are not germane to the subject of this paper and will not be discussed further.

B. Requirements for the Needed Capability

The project was a rewrite of an existing fielded implementation with the goal of increasing the redirection performance by a factor of 10 and scalability of the number of servers by a factor of 50. In particular, each redirector needed to be able to handle in excess of 500 redirects per second, with a total elapsed time of 2.5 seconds per interaction (under normal conditions), and there should be no single points of failure or centralized inline components to limit scalability. The project had an extremely aggressive schedule of only two months, from start to fielding. Other important issues included robust installation and maintainability of the code.

The requirements for this deployment included the need to keep both the web client and the web server software interfaces unchanged from their non-load balanced forms (i.e., HTTP based). This, plus the fact that the Internet applications being developed were not distributed object based, meant that parts of the developing QuO adaptivity machinery would not be useable, at least for early versions of the project. What would be available (under the covers) were the DOC support components for constructing the application transparent, distributed load balancing
capability itself, along the lines outlined above for delegates changing behavior and system condition objects acquiring current status information on which to base the runtime behavior which was sensitive to the network’s current operating capability.

Another key requirement for the project was a very rapid development cycle for the initial operating capability (similar to that which led to a number of earlier ad hoc, non-middleware implementations of this and other complex network centric software projects). The first version of the software needed to be integrated with current network management and trouble reporting subsystems, validated, introduced to operational personnel and be operational for a small set of current users within 8 weeks. This aggressive schedule reinforced our COTS middleware based approach to constructing the load redistribution system. In particular, we already had QuO designs and interfaces compatible with a CORBA ORB based infrastructure, and operational experience with a particular ORB, TAO [13,14], which was well suited to the high performance transaction rates needed for inline interactive Web browsing applications on the Internet. This meant that there would be significantly less software to be built by the project than for the previous incarnations of this functionality, and that there would be simple, high level approaches to interfacing to the legacy systems involved, with only small parts of the overall code being the difficult to debug and maintain low level routines to connect the redistribution system with the existing infrastructure.

Given these constraints, a power development environment was necessary to supply libraries of functionality and operating environment. The ACE/TAO CORBA middleware system was chosen for the development platform. ACE offers an OS independent system for creating high performance multi-threaded C++ applications. ACE comes with rich libraries that handles many of the data structures and design patterns needed to make a high-speed HTTP and DNS Redirector. In addition, TAO, a real-time CORBA ORB built on ACE, was used to connect the Redirectors with the Server Monitors. CORBA simplifies passing data structures between remote processes and has many services to help fielding and operating distributed applications, such as Naming and Event Channel. Using ACE/TAO saved much development effort because much of the necessary functionality came prepackaged and well integrated.

C. The Design of the Hopscotch Middleware

Our design for the scalable, componentized Hopscotch adaptive middleware which was needed to augment the HTTP based Web client server middleware involved developing a QuO like distributed, adaptive information collection, dissemination and redirection decision making infrastructure tailored to the Web client/Web server environment and the specific Hopscotch requirements. Figure 4 illustrates our design concept. It consists of HTTP and DNS Listeners interfacing between the Web clients and the Hopscotch subsystem, Redirectors which make resource management binding decisions based on dynamically collected status information and pluggable resource management policy modules, status collectors, and distribution of status information via channels. All of these are constructed as decoupled individual components and put together as individually replaceable pieces of a fast and scalable whole subsystem. In the remainder of this paper we look at the design considerations for one of the important pieces of reusable QoS infrastructure utilized in the Hopscotch project, the status dissemination event channel.
D. The Status Dissemination Event Channel

In the Hopscotch system, the Web servers are remote from the binding decision making components, and information about the status of these servers must be sent from the Web Server to the Redirectors. Each Web Server has a Server Monitor, which collects information about the server’s system properties. Also, each Data Center has a Network Latency Estimator, which actively measures the network latency from the server to the client, using an active pinging strategy. Transferring the status information from the server location to the Redirector presents several issues, depending upon the type of data being transferred.

The main issue for the Server Monitors is scaling as the number of Web Servers grows into the 100’s. The information collected by the Server Monitors can be collected and sent to the Redirectors using a push pattern. All the Redirectors need up-to-date status data about each Web Server. So instead of each Redirect polling for the information and incurring the polling delays, the Server Monitors can push status changes to the Redirector only when a significant change in status has occurred. But with large numbers of Servers Monitors and Redirectors, managing the connections between them is a burden. It would simplify matters if each entity would just publish or subscribe to the status information and require no specific knowledge of the other parties.

The CORBA Event Service is a standard push service that enables decoupling suppliers from consumers of events. The Server Monitor can publish its status into an Event Channel and the Event Channel will deliver the status message to all the appropriate subscribing Redirectors. Each Server Monitor only has to know about the Event Channel, and Redirector consumers can come and go without any interaction with the suppliers. Likewise, the Redirectors are also isolated from Server Monitors. While an Event Channel is logically one object, it can be implemented as a distributed mesh of servers. Different event channel server topologies permit the Event Channel Service to have different reliability, message efficiency, or capacity properties. The Event Channel topology is transparent to its suppliers and consumers, so the topology can also change without changing the supplier/consumer configuration. For example, for a small number of Server Monitors, one central Event Channel Server can be used. To increase the capacity of the Event Channel Service as the number of Monitors increase, this can be changed to a mesh with one Event Channel Server per Data Center. Or to increase the reliability of the Event Channel Service, two central servers can be used at the cost of receiving and filtering two (redundant) status updates.

One problem with the traditional Event Channel implementation is that the events are queued and received in order. But, what happens when a consumer cannot receive at the normal event rate, because of network capacity or consumer capacity problems? The queued up events may be out of date, with the event at the front of the queue being superseded by events later in the queue. The policy for dropping messages for handling flow control issues should lie in the Event Channel and should have knowledge of the semantics of the data being sent. Some of these issues are addressed in the CORBA Notification Service, but it deals mainly with event filtering and is not powerful enough to handle status information. To get the special policies needed for status dissemination, we have created a CORBA Typed Event Channel specialized for status information transfer. Because the API for the Status Channel is exactly the same as the standard Event Channel, they can be swapped without affecting the Redirectors or Server Monitors. The only difference is that the new service is more resilient to variations in network and server load. Using our CORBA Event Channel Status Service will allow the Hopscotch subsystem to scale to 100’s of servers in the future, adaptable to a variety of operating environments.

For the Network Load Estimators, the issue is that the vast size of the potential queries makes pushing the network data impractical. Potentially, any host on the Internet can request a Web Server, and this represents a huge to/from matrix between Web Clients and Web Servers. But the
requests for any specific pair will be infrequent, so using push technology for the whole matrix is impractical. Also, a specific Web Client will tend to continue to use the same Redirector, so there will often be no need to share information between Redirectors. For network information, the standard CORBA invocation query interface can be used. The problem is that 100’s of queries may be outstanding at the same time, which presents threading issues. We used CORBA oneway messages for the request and the reply, so that the threads could be reused while awaiting responses. TAO/ACE have several patterns for handling the threading issues which we applied. The details of this implementation approach are beyond the scope of this paper.

VI. HOPSCOTCH MIDDLEWARE IN THE GENERAL QUO CONTEXT

In the course of the Hopscotch project we verified and refined a number of the adaptive feedback middleware concepts, which are under development with our more general QuO middleware infrastructure work. Additionally, we used a number of the building block software implementation components from the QuO development environment while constructing a few others, which will become standard QuO middleware components.

The Hopscotch Redirector is really a specialized version of a QuO delegate, inserted into the HTTP Web client/server environment instead of the CORBA method invocation/invocation response environment. The Hopscotch Monitoring and Status reporting mechanisms served as a closed form version of the QuO System Condition concept connected to information suppliers via CORBA event channel compatible mechanisms. The whole Hopscotch subsystem implementation utilized in common with QuO a distributed object computing COTS CORBA implementation base, consisting of the TAO ORB, its underlying ACE portability layer and selected CORBA services. Additionally, these standard middleware components provided the integrating mechanisms and capabilities for interfacing with the existing non-Hopscotch environment, including Web applications, network monitoring systems and software trouble reporting services.

Largely because of the use of off the shelf designs and components, the project was completed successfully and on time despite an extremely aggressive set of requirements and schedule. At the first deliverable, it was completed with fewer than 10K additional lines of code, beyond the off-the-shelf CORBA components. But more importantly, it validated the QuO adaptive middleware approach and some key infrastructure components as implementationally feasible even in demanding high throughput environments.

The requirements of the initial project did not necessitate the use of the QuO contract and operating region specification and control machinery for two important reasons. First, there was no immediate requirement for application level choice of service tradeoffs, and second, there was no immediate requirement for runtime adaptivity in the selection strategy employed. For this version of the product, the definition of “best” was fixed at design time, and served for all users under all conditions. This may change over the product lifecycle, and we anticipate the increased need for QuO operating region based dynamic decision making capabilities.

VII. CONCLUSIONS AND FUTURE DIRECTIONS

Using a standards based middleware solution allowed for the rapid development, easy fielding, and simple extension of a complex distributed application requiring adaptation to current operating conditions. Demanding performance and schedule requirements were met, and a new product is being fielded. These developments partially validated the implementation base and the design approaches being taken within the QuO advanced, adaptive distributed object computing environment, in a real world, high performance Internet based production environment.

In the long run, the Internet Service Provider fielding Hopscotch has access to vast quantities of information about the status of network traffic and routing that can be used to support better runtime adaptivity for Web server load balancing. This information could replace the current
ping scheme and use indirect network measurements and prediction models to estimate the network characteristics between Web Clients and Web Servers. The current middleware design anticipates these improvements. Using the CORBA Event Channel this information can be published to the Redirectors using the infrastructure we have developed. In addition, future extensions may also combine application specified adaptation control mechanisms, to complement the ISP provided infrastructure based adaptation which is provided within the current development.

VIII. ACKNOWLEDGEMENTS

We would like to acknowledge the contributions to the redirection project reported here from three groups whose work forms a critical part of the overall system: the Hopscotch group at Genuity Inc., the TAO Real Time ORB group at Washington University in St. Louis, and other members of the QuO group at BBN.

IX. REFERENCES

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, Reading, MA , 1995
[16] Anne Thomas, "Enterprise JavaBeans Technology",
l, Dec. 1998
[17] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz and David E. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects", Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing, September 1998