Providing QoS Customization in Distributed Object Systems*

Jun He[1], Matti A. Hiltunen[2], Mohan Rajagopalan[1], and Richard D. Schlichting[2]

[1] Department of Computer Science, The University of Arizona, Tucson, AZ 85712
[2] AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932

Abstract. Applications built on networked collections of computers are increasingly using distributed object platforms such as CORBA, Java RMI, and DCOM to standardize object interactions. With this increased use comes the increased need for enhanced Quality of Service (QoS) attributes related to fault tolerance, security, and timeliness. This paper describes an architecture called CQoS (Configurable QoS) for implementing such enhancements in a transparent, highly customizable, and portable manner. CQoS consists of two parts: application- and platform-dependent interceptors and generic QoS components. The generic QoS components are implemented using Cactus, a system for building highly configurable protocols and services in distributed systems. The CQoS architecture and the interfaces between the different components are described, together with implementations of QoS attributes using Cactus and interceptors for CORBA and Java RMI. Experimental results are given for a test application executing on a Linux cluster using Cactus/J, the Java implementation of Cactus. Compared with other approaches, CQoS emphasizes portability across different distributed object platforms, while the use of Cactus allows custom combinations of fault-tolerance, security and timeliness attributes to be realized on a per-object basis in a straightforward way.

1 Introduction

Middleware platforms such as CORBA [23], Java RMI [29], and DCOM [2] provide high-level programming abstractions that facilitate distributed object computing, but lack a unified framework for supporting Quality of Service (QoS) guarantees related to fault tolerance, security, and timeliness. The lack of uniformity is apparent in two separate ways, both equally important. First, most existing standardization and research efforts in this area focus on providing a single QoS attribute such as fault tolerance (e.g., [4,6,15,22,24]) or security (e.g., [1]) rather than combinations of attributes. While useful, applications in areas such as financial services and multimedia often need multiple types of guarantees, as well as the ability to control performance and functionality tradeoffs between the different attributes. Second, most efforts provide point solutions for only a single middleware platform rather than a framework that can be used uniformly across different platforms. While sufficient in some cases, it limits applicability and requires

* This work supported in part by the Defense Advanced Research Projects Agency under grant N66001-97-C-8518, and by the National Science Foundation under grant ANI-9979438.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 351–372, 2001.
© Springer-Verlag Berlin Heidelberg 2001

unnecessary reimplementation of essentially similar techniques. Indeed, many of the basic techniques for implementing various QoS attributes can be used with minor changes on any platform supporting a request/reply interaction paradigm.

To address these issues, we have developed CQoS, a platform-independent QoS architecture for distributed object computing. CQoS consists of *CQoS interceptors* and configurable *CQoS service components*. The service components can be customized to provide the desired combinations of attributes, while the interceptors are used to insert CQoS transparently between the application and middleware platform on both the client and server hosts. The service component is implemented using Java version of Cactus, a system that supports construction of highly-configurable network protocols and services [10,12]. CQoS is designed to be easily portable, and in particular, to allow QoS attributes to be implemented in a way that can be used across different middleware platforms. To do this, the interceptor is used to abstract away middleware and application-specific details, providing a standard interface for implementing enhanced functionality.

The primary goal of this paper is to present CQoS as a single unified framework for providing multiple types of QoS attributes across multiple middleware platforms. We do this by first describing the software architecture and then giving examples of how customized fault-tolerance, timeliness, and security attributes can be realized in a platform-independent manner. The mapping of the architecture to CORBA and Java RMI is then presented, along with experimental results using Visibroker 4.1 and JDK 1.3 on Linux. Finally, we evaluate our approach in the context of related work. While issues related to providing enhanced QoS in middleware have been explored elsewhere (e.g., [19,33]), no other approach offers the same combination of support for multiple QoS attributes, platform independence, and the ability to make fine-grain object-specific customizations as does the architecture described here.

2 Software Architecture

2.1 Overview

The CQoS architecture allows the QoS attributes of a distributed object system to be customized transparently to client and server applications. Figure 1 provides a high-level overview of the architecture. In our prototype implementation, the middleware platform can be CORBA or Java RMI, but as noted above, the same approach can be used for any platform that supports a request-reply interaction paradigm, including standard remote procedure call (RPC) systems. For example, it would be feasible to intercept HTTP requests and replies, in which case the TCP socket layer would be viewed as the middleware layer. CQoS can be configured separately for each distributed object application in the system, allowing application-specific customization.

QoS customization can be done on different system levels, including below the middleware, as a modification to the middleware, or as a service built on top of the middleware. Each alternative has its relative tradeoffs, including factors such as transparency, effectiveness of the approach, performance overhead, and the ease of implementing and customizing such service enhancements. In the case of CQoS, implementing on top of the middleware layer has a number of advantages, including the ability to use the higher-level primitives provided by the middleware for locating objects and for performing

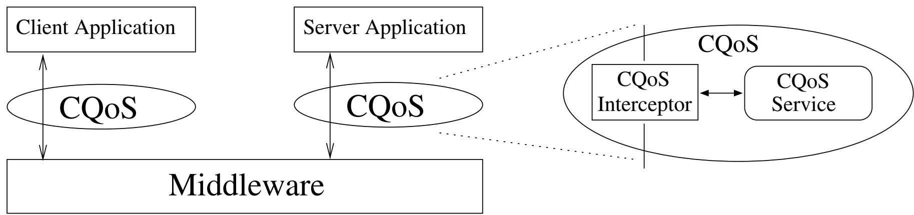

Fig. 1. High-level view of CQoS architecture.

inter-object communication. It also means that CQoS can be inserted transparently to both application objects and middleware, requiring no changes in either. We discuss the relative merits of the different approaches further in section 6.

The key observation enabling this design is that the fundamental techniques for implementing QoS properties such as fault tolerance, security, or timeliness are similar regardless of the specific middleware platform. For example, fault tolerance can be increased by replicating the server and multicasting each method call, independent of whether the middleware is CORBA or Java RMI. The architecture eliminates the need for reimplementing similar techniques for different platforms by allowing QoS properties to be realized in a platform-independent manner.

To achieve portability, CQoS is structured as two components: a middleware- and application-specific *CQoS interceptor* and a generic *CQoS service component* that implements the QoS enhancements. The interceptor provides the service component with the necessary interfaces for manipulating requests and replies to implement the properties. Note that CQoS is needed only to enhance the QoS attributes provided to the application, not to replace or reimplement guarantees that are already provided. For example, if the underlying CORBA ORB provides security services, CQoS can be configured to enhance properties other than security.

The CQoS interceptors for the client and server sides, called the *CQoS stub* and *CQoS skeleton*, respectively, are automatically generated from the server IDL description (e.g., CORBA IDL) using our Cactus IDL compiler. The generic CQoS service components on each side are implemented as Cactus components that are referred to as the *Cactus client* and *Cactus server*, respectively. The rest of this section describes these components in more detail.

2.2 CQoS Interceptors

Client-side interception is based on replacing the conventional stub used by middleware platforms such as CORBA or Java RMI by the CQoS stub (figure 2). When the client invokes a method on this stub, it creates an abstract request object and notifies the Cactus client. The stub then stores the pending request until the call has been completed. Similarly, server-side interception is based on using the CQoS skeleton as a proxy server for the actual server object. This skeleton overwrites the server object binding with the underlying middleware layer. This causes the incoming requests to be forwarded

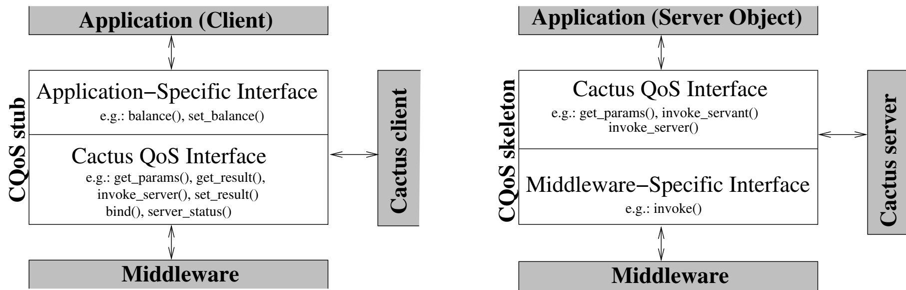

Fig. 2. Structure of CQoS interceptors.

automatically to the CQoS skeleton, which also creates an abstract request object and notifies the Cactus server.

To implement this functionality, CQoS stubs and skeletons provide multiple interfaces for interaction with the application, middleware, and QoS service component. The application interface on the CQoS stub is identical to the original stub and provides operations for each of the server object methods. The middleware interface on the CQoS skeleton provides operations that allow the middleware to pass the request to the CQoS skeleton. The details of CORBA and Java RMI implementations of this interface are discussed in section 4. Finally, the interface for the QoS service component, called the *Cactus QoS* interface, provides methods for the Cactus client and Cactus server to manipulate the requests and connections to servers.

To support request manipulation, the Cactus QoS interface provides an abstract representation of the client request together with appropriate operations. Specifically, the request is represented as a Java class, where the request parameters are represented as a vector of Java objects (`java.lang.Objects`). This interface provides a set of accessor methods to get and set parameters and return values. The implementation of the interface, which is platform specific, takes care of converting the abstract request into a form required by the specific platform. For example, a CORBA implementation that uses DII/DSI converts the abstract request structure into a CORBA request (`org.omg.CORBA.Request`). The request object also provides a field for piggybacking additional parameters onto the request. For example, the Cactus client may include a priority parameter that is used by the Cactus server to determine the order in which requests are to be processed.

The Cactus QoS interface also provides abstract representation of the server objects. Since the implementation of certain attributes such as fault tolerance requires communication with multiple servers, the interface provides operations for creating connections with specific servers (*bind()*), testing the status of a server (*server_status()*), and sending requests to specific servers (*invoke_server()*). The *bind()* operation can also be used to rebind to a failed server after it has recovered. Details related to server names and addressing are hidden by the interface so that the CQoS service component can be independent of the application as well as the middleware platform. In particular, the interface allows the server replicas to be referred to by numbers (1..N) rather than by application

or middleware-specific identifiers. Currently, the *server_status()* operation only indicates if the server is running or failed, but it could be extended to provide information such as the load conditions on the server for load balancing purposes. On the server side, the interface provides operation *invoke_servant()* that the Cactus server can use to actually invoke the method in the server object and the *invoke_server()* operation that the Cactus server can use to communicate with Cactus servers on other replicas. The latter operation is used, for example, to send ordering messages between replicas to implement a consistent total ordering of client invocations.

2.3 CQoS Service Component

QoS attributes are implemented by the Cactus client and Cactus server components. These components are *composite protocols* that are implemented using Cactus, a design and implementation framework for constructing configurable protocols and services [10]. In the following, we briefly introduce Cactus, describe the Cactus client and server components, and outline how these components can be configured to provide custom QoS properties.

Cactus overview. A service or protocol in Cactus is implemented as a composite protocol, with each service property or other functional component implemented as a software module called a *micro-protocol*. A micro-protocol is structured as a collection of event handlers, which are procedure-like segments of code that are executed when a specified event occurs. A customized version of a protocol or service is constructed by selecting the micro-protocols that implement the desired features. The service can also be changed during execution by dynamically altering the configuration of micro-protocols within the composite protocol.

Cactus provides a variety of operations for managing events and event handlers. For example, an operation is provided for binding a handler to an event, with optional static arguments that are passed to the handler on every activation. Events are raised either implicitly by the runtime system or explicitly by a micro-protocol executing an appropriate operation. When an event is raised, all handlers bound to that event are executed. The raise operation also supports a delay parameter, which can be used to implement time-driven execution, and dynamic arguments that are passed to the handlers upon invocation. An event raise may be blocking (synchronous), where the caller is blocked until all the handlers have been executed, or non-blocking (asynchronous), where the caller continues execution concurrently with the handler execution. Other operations are available for unbinding handlers, creating and deleting events, halting event execution, and canceling a delayed event. Cactus also supports data structures shared by micro-protocols in a composite protocol and a message abstraction designed to facilitate development of configurable services.

A number of prototype implementations of Cactus have been completed. These include Cactus/C, a C version that runs on Mach MK 7.3 and Linux; Cactus/C++, a C++ version that runs on Linux and Solaris; and Cactus/J, a Java version that runs on most platforms. The prototypes described here use Cactus/J on Linux.

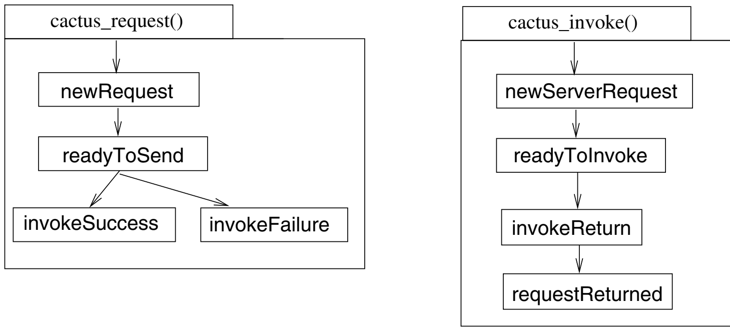

Fig. 3. Cactus events used in CQoS.

Cactus client and server. The Cactus client and server provide a very simple interface for CQoS interceptors. In particular, the client provides a *cactus_request(requestID)* operation that the stub can use to notify it of request arrival, and the server provides an analogous operation *cactus_invoke(requestID)* for the skeleton. Both operations block until the request has been completed. For example, when *cactus_request()* returns, the stub can return the return value from the request structure to the client. The implementations of these operations simply raise the appropriate events newRequest and newServerRequest, respectively, with the actual processing done by various micro-protocols. The above design assumes all client invocations are synchronous, but the implementation could easily be extended to support asynchronous invocations.

Figure 3 illustrates the events used in the Cactus client and server. The arrows between events indicate causal relations between events, that is, an arrow from ev1 to ev2 indicates that some micro-protocol that processes ev1 (i.e., has a handler bound to ev1) raises ev2. Event readyToSend indicates that a request is ready to be sent to the server(s), and invokeSuccess and invokeFailure indicate that an invocation completed successfully or failed. Event readyToInvoke indicates that an invocation is ready to be passed to the server object, invokeReturn that the invocation has returned from the object, and requestReturned that the reply to the request has been sent back to the client side.

Customization. Since CQoS decouples the application from the specification and implementation of the QoS attributes, QoS customization can be done in a variety of ways by end users, system administrators, or application designers. While customization must currently be done using a programming interface, a graphical tool similar to the Cactus-Builder [8] could be developed to facilitate the process. Here, we focus on the underlying mechanisms designed to support this customization.

Cactus supports customization both statically at configuration time and dynamically at execution time. With static customization, the desired set of micro-protocols is specified either by modifying the constructor of the composite protocol to start the appropriate micro-protocols or by using a configuration file that is read by the constructor of the composite protocol. With dynamic customization, the set of micro-protocols is deter-

mined after the Cactus protocols start execution and the micro-protocols are loaded using Java's dynamic code loading features.

Dynamic customization uses two generic micro-protocols developed for Cactus/J: rBoot and rControl. rBoot provides the minimal functionality required to load rControl. In particular, it connects to the object from which the code is to be loaded and accepts a message that contains rControl as a Java archive. rControl loads the actual micro-protocols required in the configuration. It remains active for the duration of the composite protocol, and thus, allows new micro-protocols to be loaded during execution. Using this technique, the object constructor in the composite protocol needs only to start the generic rBoot micro-protocol to support full dynamic customization. The current versions of RBoot and rControl use a separate TCP connection to load the code, but other alternatives are being explored such as using the underlying middleware platform or Jini technology.

While static customization is conceptually simple and easy to use, dynamic customization offers more flexibility. For example, the configurations in statically customized client and server protocols must match for the system to operate correctly, while dynamic customization allows a matching configuration to be loaded at execution time. This ability can be utilized in a number of ways. For example, the client can download the necessary micro-protocols from the server, the server can download micro-protocols from the client, or both the server and client can download micro-protocols from some external configuration service. Each of these options has useful application areas. For example, a client can download a multicast or load balancing micro-protocol that is used by a specific collection of replicated servers, while a server can download the secure communication micro-protocol required by a given client. An external configuration service allows the properties—and thus the configurations—to be defined for all [user,service] pairs without requiring direct manual configuration of protocols. All of these alternatives make it easier to deploy new or updated micro-protocols since the updates only need to be made at the clients or the servers, or the configuration service.

The ability to alter configurations dynamically introduces the need to coordinate these changes on different hosts to maintain consistency. For example, if a micro-protocol implementing consistent total ordering of client invocations is changed at runtime, all the server replicas must perform the change at the same time with respect to the flow of invocations. In our current prototype, dynamic customization is limited to the client side and only at the time when the client binds to a server. Since we do not consider configuration changes at the servers or more general runtime changes, the only coordination required is to ensure that the Cactus client loads the necessary micro-protocols before it passes requests to the Cactus server. We have explored coordination issues in the context of group communication services [3] and intend to apply these techniques in future versions of CQoS.

3 Micro-protocols for QoS Enhancement

Cactus micro-protocols can be used to implement any service property or function, but this paper focuses on QoS attributes related to fault tolerance, security, and timeliness. Other properties and functions such as caching, prefetching, and load balancing could

be implemented in similar ways. Note that none of the specific techniques used here are novel; indeed, all have been used in other CORBA and Java RMI systems, and prior to that, in RPC systems and other systems that use the request/reply paradigm. The novelty of our approach is the way in which they can be configured to realize different combinations of attributes.

3.1 Base Micro-protocols

A Cactus composite protocol typically includes micro-protocols that provide basic service functionality. In CQoS, these micro-protocols are CLIENTBASE at the client and SERVERBASE at the server. CLIENTBASE consists of three handlers:

- *assigner*. Bound to event newRequest, it assigns a server to the request and raises readyToSend.
- *syncInvoker*. Bound to readyToSend, it uses the server determined by *assigner* to issue the request. It checks the server status (*server_status()*), connects to the server if necessary (*bind()*), calls the server (*invoke_server()*), and raises invokeSuccess or invokeFailure depending on the result of the call.
- *resultReturner*. Bound to both invokeSuccess and invokeFailure, it provides the default processing of these events by releasing the waiting client thread when an invocation is completed.

SERVERBASE consists of two handlers:

- *getParameters*. Bound to newServerRequest, it extracts Cactus parameters from the request structure and raises readyToInvoke.
- *invokeServant*. Bound to readyToInvoke, it invokes the server object by calling the *invoke_servant()* method of the CQoS skeleton and raises invokeReturn.

Note that the basic behavior is broken into multiple handlers with events used to pass control from one handler to another. This allows the actual QoS micro-protocols to insert their processing at the appropriate points of the control flow. All the handlers in the base micro-protocols have been ordered to be the last ones executed when its respective event is raised. This makes it possible for other micro-protocols to do additional processing before these handlers are executed or to override them by stopping the event execution before their execution.

3.2 Fault Tolerance

Many fault-tolerance techniques are relatively easy to implement transparently. For example, method calls can be sent to replicated servers to tolerate host failures, and messages can be retransmitted to tolerate transient network failures. To hide the impact of such replication, the mechanisms used also need to eliminate duplicate messages and combine multiple replies. It may also be necessary to ensure that all server replicas receive method calls in the same order and have a consistent view of the server group membership. The current prototype does not support state transfer for either recovering or newly joining replicas. Such a facility could easily be added using standard techniques,

although it would lessen transparency since but it would require that server objects provide operations for getting and setting their state. Also, the current implementation only considers host crash failures, although a similar customizable approach could be used for less benign failures [9]. Currently, we assume the underlying platform handles network failures, but it would be easy to add retransmission micro-protocols if necessary.

Our current prototype has two replication micro-protocols: ActiveRep and PassiveRep. ActiveRep implements active replication where the request is sent to all server replicas and all non-crashed replicas reply. ActiveRep consists of one handler *actAssigner* that is similar to the base *assigner* except that it raises readyToSend asynchronously. The constructor of ActiveRep binds *actAssigner* to the event newRequest multiple times, once for each server. When the event is raised, an instance of *actAssigner* is executed for each replica. From this point, execution proceeds as outlined above—each instance of *actAssigner* raises readyToSend, which starts a separate instance of *syncInvoker*. The fact that readyToSend is raised asynchronously means that each instance of *syncInvoker* is executed concurrently by a separate thread and thus, the blocking server invocations (*invoke_server()*) are executed in parallel. The *actAssigner* handlers override the base *assigner* by executing before it and halting further execution associated with the event.

PassiveRep supports passive replication, where a designated primary server replies after forwarding the request to other replicas to keep them consistent. The client side consists of two handlers:

- *pasAssigner*. This handler overrides base *assigner*, and assigns the first non-failed server to serve the request.
- *primarySelector*. This handler overrides base *resultReturner* for event invokeFailure, marks the current primary as failed, and raises newRequest to re-execute the request.

The net result is that the client thread is not released until a proper result has been received or all replicas have failed. The primary server uses techniques similar to those used in ActiveRep to forward the request concurrently to the backup replicas. It also keeps track of requests already received, so that receiving a request again does not corrupt the server state.

The current prototype supports three different *acceptance semantics*, which determine when a request is considered completed and a reply can be returned to the client. ClientBase by default implements a policy useful for the non-replicated case where the first reply (success or failure) to arrive is returned to the client. A second micro-protocol returns the result from the first successful execution and a third returns the majority value from non-failed replicas. Both of these micro-protocols consist of one handler that is executed before the base *resultReturner*.

The TotalOrder micro-protocol ensures that all replicas receive requests from multiple clients in a consistent total order. Our prototype uses a sequencer-based total ordering algorithm, where a coordinator determines the ordering for each request, and multicasts it to the other replicas. Although failure of the coordinator is not currently tolerated, it would be simple to add this using standard techniques. TotalOrder consists of three handlers in the Cactus server:

- *assignOrder*. Bound to readyToInvoke at the coordinator, it determines an ordering for each new request and sends it to other replicas in parallel using technique similar to ACTIVEREP.
- *checkOrder*. Bound to the same event on all replicas, it processes both requests and ordering information and releases any request that becomes eligible for execution.
- *checkNext*. Bound to invokeReturn, it determines if a waiting request can be executed.

3.3 Security

Many security features such as secure communication, authentication, and access control can be implemented transparently. Our current prototype includes provisions for message confidentiality, message integrity, and access control. As an example, DESPRIVACY encrypts and decrypts the request parameters and reply using DES. The client side uses a handler bound to readyToSend to encrypt the request parameters and a handler bound to invokeSuccess to decrypt the reply value. Both handlers are executed as the first handler for these events. The server-side decryption of request parameters is implemented by a handler that overrides the base *getParameters* handler. The server-side encryption of the reply value is implemented by a handler bound to invokeReturn. Note that since the micro-protocol encrypts only the request parameters and replies, the security level provided is slightly less than CORBA Security Level 1, which encrypts the entire request message. Integrity is provided by a signature-based scheme implemented by micro-protocols at the client and server, while access control is implemented by a micro-protocol at the server.

3.4 Timeliness

Providing rigorous timeliness guarantees is difficult and requires control of client admission and request scheduling. However, service differentiation properties that provide more timely service to high priority requests can be implemented as relatively simple micro-protocols. Our current prototype includes three such micro-protocols. The first, PRIORITYSCHED, manipulates thread priorities. It consists of one handler *setPriority* bound to readyToInvoke that sets the priority of the current thread based on the request priority. It is set to execute as the first handler for this event so that it can change the priority as early as possible. The second, QUEUEDSCHED, schedules request execution by queuing low priority requests if high priority requests are executing. This behavior is implemented by three handlers:

- *checkPriority*. Bound to readyToInvoke, it allows a request to either continue or queues it.
- *notifyWaiting*. Bound last to invokeReturn, it raises event requestReturned asynchronously with a low thread priority if no high priority requests are being executed.
- *wakeupNext*. Bound to requestReturned, it releases waiting low priority requests.

Note that the second handler uses a modified raise operation that allows the thread priority to be specified. This ensures that execution of *wakeupNext* does not interfere

with the thread that is returning the high priority request. Finally, the third micro-protocol, TimedSched, uses a similar strategy, except that it keeps track of how many high priority requests have arrived in a time period and only releases low priority requests one at a time when the number of high priority requests in the previous period is smaller than a threshold. Currently, the request priority is simply determined based on client identity, but other techniques could easily be added.

Note that both TotalOrder and the last two service differentiation micro-protocols order request execution, making it possible for the orders to conflict. That is, it is possible for the next request according to TotalOrder to be a low priority request that is queued and thus blocked by the service differentiation micro-protocols. This problem can be solved by including the service differentiation micro-protocols only at the coordinator of the total ordering algorithm. This ensures that the total order assignment respects request priorities.

Two changes were made in the Cactus runtime system to allow these micro-protocols to manipulate thread priorities. The first is the variant of the raise operation mentioned above that specifies the priority of the thread used to execute the handlers. The second preserves thread priorities in event operations. In particular, the handlers associated with a given event are guaranteed to be executed by a thread with the same priority as the thread that raised the event, unless specified otherwise.

3.5 Combining QoS Properties

The Cactus framework allows micro-protocols to be designed so that the composite protocol can be customized to provide different combinations of QoS micro-protocols. In our case, the fault-tolerance, security, and timeliness micro-protocols have been designed to work together in any combination. The fault-tolerance micro-protocols can be used in five different combinations: passive replication (1) or active replication with any combinations of total order and acceptance (4). Overall, a service can be configured with no fault tolerance or any of these five fault-tolerance combinations with any combination of the three security micro-protocols and any of the three timeliness micro-protocols. As a result, even this small set of micro-protocols can be configured in over 100 different ways.

While using Cactus does not automatically guarantee that micro-protocols are composable, it provides flexible mechanisms that allow the micro-protocol designer to maximize composability. One example is the event mechanism that allows handler execution to be ordered as desired, including provisions for overriding existing handlers. These facilities are used, for instance, to ensure that the decryption handler is executed transparently prior to all other handlers. Note, however, that the composability is the result of careful design of the event set and ordering of handler execution.

The current set of micro-protocols could be easily extended with different security, timeliness, and fault-tolerance techniques, and to handle more complicated invocation scenarios, such as an invocation from one replicated server to another. Additional security micro-protocols could also be added using the approach presented in [11], which includes numerous micro-protocols for confidentiality, as well as for other security attributes such as authenticity, non-repudiation, key distribution, and auditing. Each security attribute can also be enforced using combinations of two or more micro-protocols

if desired. Additional timeliness micro-protocols could include admission control and traffic enforcement, while additional fault-tolerance micro-protocols could include request logging, server recovery, and more rigorous failure detection and membership management.

4 Implementing CQoS on CORBA and Java RMI

The CQoS stub and skeleton form the middleware and application-dependent components of this framework, providing a mechanism to abstract system specific details and implement a standard interface by which Cactus protocols can access data in an implementation neutral manner. This section describes highlights of how these interceptors have been mapped to CORBA and Java RMI in the prototype implementation. We emphasize again that the actual implementation of the QoS attributes in the Cactus client and server is independent of the specific platform used.

4.1 CORBA

CORBA is a vendor-independent software architecture designed to facilitate object-based distributed computing. The architecture consists of three major components: an interface definition language (IDL), a communication infrastructure (ORB) supporting remote method invocation, and a variety of supporting services. Examples of services include a naming service, a security service, and an event notification service. The IDL definition is used to generate stubs and skeletons. Stubs marshal the invocation into a request and pass it to the ORB, which is responsible for transmitting the request from the client to the server host. Skeletons unmarshall the request and perform the invocation on the corresponding servant for the object. The result of the invocation is returned in an analogous manner.

Our approach to adding customizable QoS is to insert CQoS stubs and skeletons between the client and ORB and the ORB and the servant. These replace the conventional stubs and skeletons and are responsible for intercepting method invocation on both the client and server sides. The interception is completely transparent and no modifications are required to the client code, to the server code, or to the IDL description of the server interface. Among other things, this allows enhanced QoS to be added to legacy CORBA applications.

Figure 4 illustrates the different system components and their interactions. `Client` and `Servant` are the user-provided CORBA client and servant, where the servant is a Java object that implements the request processing for one or more CORBA objects. `stub`, `helper`, and `skeleton` are standard components generated by the IDL compiler. The standard `stub` and `skeleton` are replaced in our approach by the CQoS stub and skeleton and the `helper` is modified slightly to enable interception. `startup` on the server side is a standard object-specific initialization file modified slightly to enable interception. Dotted lines represent interactions at servant initialization and client bind time, while solid lines represent interactions at invocation time.

A careful naming convention for the POAs (Portable Object Adaptors) and the CQoS skeletons is used to enable the client side to locate the potentially replicated objects.

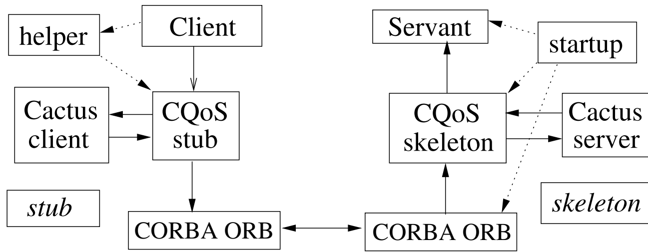

Fig. 4. System components and interactions in the CORBA implementation.

Generally, a CORBA object is represented by an IOR (Interoperable Object Reference) that is created by a POA when the servant that implements the object registers with the POA using the appropriate object identifier. We used a set of POAs in our system, one for each object replica, to create the IORs. The POA for the ith instance of object with identifier "OID" is named "OID_agent_poa_i". Given that the POAs have different names, we can use the same object identifier "OID_CQoS_Skeleton" for the CQoS skeletons on all replicas. Using this convention, the CQoS stub can get the IORs of the object replicas by binding with the correct POA and using the object identifier "OID_CQoS_Skeleton".

Client- and server-side interception is implemented by modifying the `helper` and `startup` files, respectively. On the server side, the `startup` file is modified to start and register the CQoS skeleton rather than the application servant. Specifically, `startup` uses the above naming convention to create a POA for the object and register the CQoS skeleton with the POA. A pointer to the original servant is passed as an argument to the CQoS skeleton. This pointer is used within the skeleton to implement the *invoke_servant()* operation by making a native Java call to the servant object.

On the client side, a line in the standard `helper` file is modified to specify the CQoS stub as the stub to be used by this client. As described above, this stub has a method corresponding to each server object method, which converts the method call into an abstract request structure that is passed to the Cactus client using *cactus_request()*. The CQoS stub also intercepts a client's *bind()* operations. In our current implementation, the *bind()* operation simply creates a CQoS stub and returns it to the client, with the actual binding done at the time the client issues its first request. The CQoS stub may create multiple bindings (e.g., for replication), and the Cactus client can access the status of bindings and request rebinding through the Cactus QoS interface described above in section 2.2. Note that the CQoS skeleton uses identical techniques to establish connections between server object replicas when necessary.

Finally, the implementation of the *invoke_server()* operation provides communication between the CQoS stub and skeleton, and between CQoS skeletons, using CORBA facilities. We have completed two implementations based on two alternative CORBA interfaces defined for the IDL to Java mapping [25]. The first uses the Dynamic Invocation Interface (DII) at the client and the Dynamic Skeleton Interface (DSI) at the server, while the second uses CORBA streaming APIs. In the DII/DSI approach, the CQoS stub constructs a CORBA DII request object using a delegate for the CQoS skeleton and

sends the request using the *invoke()* operation provided by the request object. The CQoS skeleton provides an analogous *invoke()* operation that is called by the POA when the request object arrives. This operation uses DSI facilities to extract the method name and parameters, including any extra Cactus parameters added to the request.

In the streaming approach, the CQoS stub uses the streaming APIs to construct a CORBA portable output stream, write the request parameters to the output stream, invoke the server delegate with the output stream as a parameter, and read the result from the resulting input stream. The CQoS skeleton provides an *invoke()* operation in a manner similar to the DII/DSI approach, but in this case it uses stream API operations to extract the method name and parameters from the input stream. Regardless of the method used to extract the parameters, the CQoS skeleton creates an abstract request object that is passed to the Cactus server by calling *cactus_invoke()*. The experimental results in section 5 demonstrate the performance benefits of this approach relative to the use of DII/DSI.

4.2 Java RMI

Java Remote Method Invocation (RMI) is a communication architecture aimed at integrating the distributed object model into the Java programming language while maintaining its original semantics [29]. Overall, the RMI architecture is relatively similar in structure to the CORBA architecture in figure 4, with automatically generated stubs hiding the lower-level communication details from the applications. However, the RMI architecture since JDK 1.2 does not use server-side skeletons and does not require a separate helper file. The RMI specification supports multiple underlying protocols—the default JRMP (Java Remote Method Protocol) and CORBA IIOP—and custom stubs may be generated for each of these using the `rmic` stub compiler. The architecture also provides the RMI registry, which is a naming service used to bind remote objects with generic names. Clients use the registry to locate remote objects through methods provided in the `java.rmi.Naming` class.

Server-side interception for RMI is modeled on our approach on CORBA. However, RMI is simpler than CORBA and does not have concepts such as POA and DSI, which affects implementation details. Since Java no longer supports server side skeletons, we introduce the CQoS skeleton as a proxy object that is modeled on a typical RMI (JDK 1.1) skeleton. We use a naming convention for the skeletons at the object replicas. Specifically, the skeleton for the ith replica of object with identifier "OID" registers with the Java naming service using name "OID_CQoS_Skeleton_i". The client side CQoS stubs then bind to these skeletons, rather than the original objects, and thus, all client requests are automatically delivered to the skeletons. A mechanism similar to DSI is simulated in RMI by having the skeleton export only a generic *invoke()* method (`java.lang.Object invoke(java.lang.Object[])`). The Java request structure containing the actual method to be called is passed as a parameter to this method. The actual invocation of the server method is done through a native Java call similar to the CORBA implementation.

Client-side interception is based on simply replacing the standard stub with a CQoS stub with the same name. The stub provides one method for each of the server methods and when a method is called, the stub creates the abstract request structure and notifies

the Cactus client. When the Cactus client wants the request to be sent to the server, it calls the *invoke_server()* method of the stub with the request as a parameter. Similar to the CORBA implementation, the stub binds to the server when the client issues its first request. We are working on extending the Cactus IDL compiler to automatically generate the CQoS stubs and skeletons for Java RMI.

Note that while Java RMI currently supports both JRMP and IIOP, the default is currently in the process of being changed to IIOP, which allows RMI objects to access CORBA objects if they conform to a small set of restrictions. These RMI-IIOP systems can be customized using the CQoS on CORBA interception mechanisms described above. To achieve this, RMI-IIOP stubs are simply replaced with customized CQoS stubs for CORBA. This interception approach can be extended to any RPC-like client/server communication model.

5 Experimental Results

The performance of the approach was tested using a simple BankAccount object that provides operations for setting and retrieving the balance of a bank account. Tests were conducted on a cluster of 600 MHz Pentium III PCs running Linux 2.2.14 connected by a 1 Gbit Ethernet. The CORBA tests were conducted using Visibroker 4.1, while the Java RMI tests use Java 2 SDK version 1.3 for Linux.

The first set of experiments focused on measuring the overhead imposed by our approach. We first measured the response time using the standard middleware platform (CORBA or Java RMI) and then ran the same experiments for different combinations of CQoS components. Each test run measured the time to execute 10000 pairs of *set_balance()* and *get_balance()* operations, and was run multiple times. The client and server objects were on different machines. The results are given in table 1, where each line adds one more CQoS component into the configuration. Note that in the CORBA case, adding the CQoS stub and CQoS skeleton does not simply add them to the baseline test, but replaces the original stub and skeleton generated by the standard IDL compiler. The CORBA tests in the table were performed using the stream API implementation. The Cactus client and server were configured with only the base micro-protocols. The column labeled "ohead" indicates the overhead of the added component compared to the previous configuration, while the subsequent column gives the cumulative overhead compared to the baseline.

Overall Java RMI performance, both the baseline and the CQoS enhanced, appears to be better than CORBA. We speculate that this is because Java RMI is a lighter weight middleware without the need, for example, to support multiple programming languages. The larger CQoS overhead with CORBA is due partially to the fact that the stub must first convert a request into the abstract form and then transmit it on the output stream one parameter at a time, whereas the Java RMI version simply transmits the abstract request object. A number of optimizations have been used in both implementations to improve performance, such as reuse of the request data structures to avoid object creation. The cost of the Cactus client and server have also been reduced by optimizing the Cactus runtime system. For example, use of a thread pool for event handling reduced overhead considerably.

Table 1. Average response times (in ms)

Configuration	set + get	one call	ohead	cum ohead
Original CORBA	2.74	1.37	0	0
+ CQoS stub	3.01	1.51	0.14	0.14
+ CQoS skeleton	3.06	1.53	0.02	0.16
+ Cactus client and server	3.44	1.72	0.19	0.35

Configuration	set + get	one call	ohead	cum ohead
Original Java RMI	2.19	1.10	0	0
+ CQoS stub	2.21	1.11	0.01	0.01
+ CQoS skeleton	2.27	1.14	0.03	0.04
+ Cactus client and server	2.61	1.31	0.17	0.21

We also tested the DII/DSI-based implementation and performed other tests to identify the impact of Java garbage collection and compiler optimization on the CORBA implementation. In the first case, we determined that the DII/DSI numbers were consistently larger. For example, the time for one call with just the CQoS stub was 1.64 ms, while the time for one call with all components was 2.16 ms. Thus, the overhead was approximately 10%-20% despite our attempts to optimize performance such as reuse of the name-value list objects required by DII/DSI. Java garbage collection also impacted performance results, especially the variance between runs. To measure this impact, we increased the heap size and set JVM parameters so that garbage collection was unlikely during a test run. This resulted in considerable performance improvement in all cases and a reduction in the variance. For example, the average time for one standard CORBA call was reduced to 1.29 ms, while the time for one call including all CQoS components was reduced to 1.48 ms. Finally, the impact of the optimizations performed by Sun's standard "HotSpot" compiler was determined to be significant. For example, the average response time for one call with all components was 3.04 ms when optimization was disabled.

The second set of experiments (table 2) illustrates the response times for different QoS configurations. Each micro-protocol increases the response time by using more CPU time (e.g., encryption), sending more messages (e.g., replication and total ordering), or both. In tests with multiple object replicas, the client and each replica are all on separate machines. The CORBA numbers are again based on the stream API implementation. As expected, the numbers indicate that adding functionality that introduces additional messages or that is CPU-intensive is relatively expensive, but simple functionality costs relatively little. Note that the increase in response time when the DesPrivacy micro-protocol is used is greater for CORBA than for Java RMI. We attribute this to the fact that the entire request object is encrypted and transmitted as a single CORBA parameter of type `Any`, whereas the arguments of calls in configurations without encryption can be transmitted as simple data types. In contrast, the request object is transmitted as a single entity in every case with the Jave RMI implementation, so that the only change is the actual CPU overhead associated with execution of the cryptographic algorithms.

Table 2. Response times for different configurations (in ms)

Configuration	num servers	CORBA set + get	CORBA one call	Java RMI set + get	Java RMI one call
DesPrivacy	1	45.92	22.96	8.57	4.29
PassiveRep	3	6.51	3.26	5.56	2.78
ActiveRep	3	6.50	3.25	4.40	2.20
+ MajorityVote	3	7.98	3.99	4.77	2.39
+ TotalOrder	3	11.23	5.62	8.14	4.07
ActiveRep+TotalOrder	3	8.50	4.25	7.40	3.70
+ DesPrivacy	3	76.70	38.25	13.63	6.84

The last set of experiments in table 3 illustrates the behavior of the TimedSched service differentiation micro-protocol alone, and when combined with other micro-protocols. For these tests, we statically designated some clients as high priority and others as low priority. In these particular tests, we used one high priority client and varying numbers of low priority client. The results indicate that the TimedSched micro-protocol provides relatively good service differentiation and protects a high priority client well from the impact of low priority clients.

6 Related Work

As already noted, the specific fault-tolerance, security, and timeliness techniques used in CQoS are standard approaches that have been used in many other systems. Therefore, we focus here more directly on research related to adding QoS guarantees to CORBA, Java RMI, and other distributed object platforms. We will also shortly discuss how the Jini technology relates to our approach.

The QoS work in distributed object systems can generally be classified into one of several approaches depending on how the QoS functionality is added to the system. These include the service approach, the integrated approach, the interception approach, and the gateway based approach.

The service approach implements QoS enhancements as separate services transparently to the middleware platform, while the integration approach directly modifies the platform to provide the enhancements. Both approaches can be made transparent to the application. The integration approach typically provides better performance since low level optimizations are possible, but interoperability becomes an issue. While the service approach is better for portability and interoperability, it is difficult to make any guarantees for communication between the client and service, which makes it impossible to guarantee end-to-end timeliness or security. A good discussion of the tradeoffs between this approach and the previous one is presented in [7]. The service approach has been used for fault tolerance [6,22], and standard CORBA services such as the security service can be viewed as examples of this approach. The integration approach has been used to enhance fault tolerance [15,17] and timeliness properties [28], and could naturally be used to enhance any other QoS attribute.

Table 3. Average response times (in ms)

Configuration	num servers	num low pri. clients	CORBA high pri.	CORBA low pri.	Java RMI high pri	Java RMI low pri
TIMEDSCHED	1	1	1.74	3.61	1.36	3.31
	1	2	1.85	4.45	1.39	3.53
	1	3	1.92	5.19	1.43	3.60
	1	4	1.97	5.33	1.48	4.27
+ ACTIVEREP	3	1	3.45	6.97	2.33	4.83
	3	2	3.45	8.39	2.33	5.30
	3	3	3.46	10.22	2.34	6.20
	3	4	3.47	12.10	2.36	7.27
+ MAJORITYVOTE	3	1	4.20	8.62	2.51	5.28
	3	2	4.22	9.98	2.62	7.30
	3	3	4.23	12.05	2.71	8.23
	3	4	4.24	13.95	2.77	9.37
+ TOTALORDER	3	1	5.65	11.18	4.10	8.49
	3	2	5.60	13.41	4.14	10.09
	3	3	5.55	15.14	4.17	12.00
	3	4	5.56	18.18	4.17	14.17
ACTIVEREP+TOTALORDER	3	1	4.39	8.70	3.68	7.38
	3	2	4.38	10.01	3.86	8.35
	3	3	4.35	12.18	3.85	9.64
	3	4	4.39	13.60	3.87	11.56

The interception approach, as the name suggests, works by intercepting middleware messages or requests. These systems are classified based on whether the interception takes place above or below the middleware. With CORBA, approaches that operate above the ORB have used mechanisms such as smart stubs and interceptors [31], delegates [16], and reflection [13]. The CORBA 2.2 standard also provides a limited interception mechanism, but it has a number of limitations [26,32]. Approaches that operate below the ORB intercept the IIOP messages sent by the ORB before they are passed to the operating system [18,19,21]. Similar approaches have also been used with Java RMI [20] and other distributed object-based systems [5,14,26]. The interception approach has been used for fault tolerance (e.g., [5,19,21]), security (e.g., [5,21,31]), and timeliness (e.g., [14,16, 18,21]).

Interception approaches that operate above the middleware provide a higher level of abstraction for implementing the QoS enhancements. For example, existing middleware services can be used to locate, communicate with, and detect the failure of servers. Other services such as CORBA security services can also be used if desired, while approaches that do not build on top of the platform must use lower-level facilities to accomplish these tasks. However, interception below the middleware can provide very good performance since it can utilize efficient custom transport protocols. Our approach can be classified as interception above the middleware.

In the gateway approach, a gateway component inserted at the transport layer between the clients and servers is responsible for implementing the QoS enhancements. The gateway may be a single component [1] or may have separate client and server sides that interact [27]. The gateway must be able to intercept IIOP messages. This is done in [1] by using a firewall to forward all IIOP messages to the gateway. The QuO distributed gateway [27] uses a client-side proxy to direct requests to the client-side gateway, which can then interact with the server-side gateway using any mechanism, e.g., a group communication service. This approach does not require any modifications to the ORB and could be used to implement any type of enhancement. However, since the gateway may reside on a different host than the client and server, it may not be possible to provide strict end-to-end QoS guarantees. Moreover, the extra communication steps introduce a performance penalty.

The main features that separate CQoS from other work in this area are its design for portability across different distributed object platforms and its support for fine-grain application-specific customization. The novel architecture in CQoS with two components, one middleware specific and the other generic, provides a beneficial separation of concerns. With this architecture, researchers can focus on developing generic improved algorithms for QoS enhancements, while the work on developing better interception mechanisms proceeds independently. A similar goal of middleware neutrality is presented in [26], but no implementation on CORBA or Java RMI is described, nor any performance results.

The fine-grain configurability supported by Cactus provides the flexibility needed to customize guarantees independently for each application across a broad spectrum of QoS attributes. Although previous work on CORBA often encompasses different QoS attributes (e.g., [18,21,27,31]), to our knowledge, no other approach provides a comparable ability to implement custom combinations of different QoS attributes. Customization of multiple QoS attributes has been addressed in other types of object-based systems, however. The metaobject-based FRIENDS [5] architecture is the closest analogue to CQoS. Metaobjects are roughly equivalent to Cactus micro-protocols as used here, and both approaches emphasize a clear separation of concerns between mechanisms and the application of QoS enhancements. A major concern for any reflection-based approach, however, is the need for a language that supports reflection, the level of reflection supported, and performance overhead. In contrast, CQoS depends on middleware-based interface definitions to introduce QoS enhancements using interceptors working in conjunction with generic QoS components. Another related approach presents an architecture in which multiple interceptors can be stacked to provide combinations of attributes [26]. However, as noted above, the architecture has apparently not yet been implemented.

Finally, Sun's Jini technology [30] is largely complementary to CQoS. Jini is a set of specifications that enables services to discover one another and cooperate in a distributed system. One of its novel features is that a client does not need to know *a priori* where the services are located or how to access them. Rather, a client uses the Jini lookup service to locate services and then dynamically loads a proxy object (i.e., a stub) that implements communication with the service. It would be possible to use these Jini facilities to locate and load micro-protocols for CQoS services. On the other hand, the fine-grain customization and ability to change behavior at runtime provided by CQoS

could be applied to construct configurable and adaptive Jini proxies. CQoS could also be used to enhance transparently the QoS properties of existing Jini services.

7 Conclusions

This paper has presented the CQoS architecture as a single unified framework for providing customizable combinations of multiple QoS attributes across multiple middleware platforms. This novel architecture provides separation of concerns between the algorithms that implement QoS attributes and the details of the underlying middleware platform. This is accomplished by dividing CQoS into two components, each of which addresses one of these concerns and provides the appropriate interfaces for interaction between the components. We presented the architecture and interfaces, described the implementation of this architecture on both the CORBA and Java RMI platforms, and showed preliminary performance results.

Future work will concentrate in several areas. In the near term, these include adding to the collection of available micro-protocols, investigating further performance optimizations, and experimenting with more realistic applications. In the longer term, our goal is to incorporate results from other Cactus-related research involving adaptive behavior and mobile systems to provide similar support for distributed object applications.

Acknowledgments. G. Townsend implemented the Cactus/J system used as the basis for this work. Also, R. Gruber and A. Puder provided helpful comments and suggestions.

References

1. T. Benzel, P. Pasturel, D. Shands, D. Sterne, G. Tally, and E. Sebes. Sigma: Security and interoperability for heterogeneous distributed systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 308–319, Jan 2000.
2. N. Brown and C. Kindel. *Distributed Component Object Model Protocol—DCOM/1.0*. Microsoft Corp., Redmond, WA, Jan 1998. Network Working Group Internet Draft.
3. W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 635–643, Mesa, AZ, Apr 2001.
4. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, Oct 1998.
5. J.-C. Fabre and T. Perennou. A metaobject architecture for fault tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems*, 47(1):78–95, Jan 1998.
6. P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 150–159, Niagara-on-the-Lake, Canada, Oct 1996.
7. P. Felber, B. Garbinato, and R. Guerraoui. Towards reliable CORBA: Integration vs. service approach. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 199–205. Verlag, 1997.

8. M. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, Oct 1998.
9. M. Hiltunen, V. Immanuel, and R. Schlichting. Supporting customized failure models for distributed software. *Distributed Systems Engineering*, 6:103–111, 1999.
10. M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, Jun 1999.
11. M. Hiltunen, R. Schlichting, and C. Ugarte. Enhancing survivability of security services using redundancy. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001)*, Gothenburg, Sweden, Jul 2001.
12. M. Hiltunen, R. Schlichting, and G. Wong. Cactus system software release. http://www.cs.-arizona.edu/cactus/software.html, Dec 2000.
13. M.-O. Killijian and J. Fabre. Implementing a reflective fault-tolerant CORBA system. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 154–163, Nuremberg, Germany, Oct 2000.
14. R. Koster and T. Kramp. Structuring QoS-supporting services with smart proxies. In *Middleware 2000*, pages 273–288, 2000.
15. S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. In *Theory and Practice of Object Systems*. John Wiley, NY, 1997.
16. J. Loyall, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, J. Gossett, and C. Gill. Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 625–635, Mesa, AZ, Apr 2001.
17. S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies,*, Monterey, CA, June 1995.
18. P. Melliar-Smith, L. Moser, V. Kalogeraki, and P. Narasimhan. Realize: Resource management for soft real-time distributed systems. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 281–293, Hilton Head, SC, Jan 2000.
19. L. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. Eternal: Fault tolerant and live upgrades for distributed object systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition (Discex'2000)*, pages 184–196, Hilton Head, SC, Jan 2000.
20. N. Narasimhan, L. Moser, and P. Melliar-Smith. Interception in the Aroma system. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 107–115, Jun 2000.
21. P. Narasimhan, L. Moser, and P. Melliar-Smith. Using interceptors to enhance CORBA. *Computer*, 32(7):62–68, Jul 1999.
22. B. Natarajan, A. Gokhale, S. Yajnik, and D. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, Sep 2000.
23. Object Management Group. *The Common Object Request Broker: Architecture and Specification (Revision 2.4.1)*, Nov 2000.
24. Object Management Group. Fault tolerant CORBA. OMG Technical Committee Ducument orbos/2000-04-04, Mar 2000.
25. Object Management Group. *IDL to Java Language Mapping, version 1.1*, Jul 2001.
26. J. Pruyne. Enabling QoS via interception in middleware. Technical Report HPL-2000-29, HP Labs, 2000.
27. R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquire, and J. Loyall. An object-level gateway supporting integrated-property quality of service. In *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint-Malo, France, May 1999.

28. D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A high-performance end system architecture for real-time CORBA. *IEEE Communications Magazine*, pages 72–77, Feb 1997.
29. Sun Microsystems. *Java Remote Method Invocation Specification, Rev. 1.50*. Sun Microsystems, Inc., Mountain View, CA, Oct 1998.
30. Sun Microsystems. *Jini Technology Core Platform Specification, Version 1.1*. Sun Microsystems, Inc., Mountain View, CA, Oct 2000.
31. V. Thomas, S. Kareti, W. Heimerdinger, and S. Ghosh. Mediators and obligations: An architecture for building dependable systems containing COTS software components. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nuremberg, Germany, Oct 2000.
32. M. Wegdam and A. v. Halteren. Experience with CORBA interceptors. In *Proceedings of Workshop on Reflective Middleware (RM 2000)*, Apr 2000.
33. J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.