



Enabling QoS via Interception in Middleware

Jim Pruyne
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-29
February, 2000

E-mail: pruyne@hpl.hp.com

QoS,
middleware
systems

Middleware systems are commonly used for building distributed systems, but they provide little, if any support for altering the quality of services aspects of an application. We wish to structure middleware so that QoS development can be carried out independently from application development. Separation allows experts in QoS fields to apply their work to any application developed for the middleware. Our approach is based on the notion of interceptors that are dynamically added to running applications. These interceptors are allowed to observe and modify application middleware calls to implement desired QoS functionality. We have developed a programming model for interceptors that supports development of a variety of QoS functionality, and have implemented this model in the context of the e-speak middleware platform.

Enabling QoS via Interception in Middleware

Jim Pruyne (pruyne@hpl.hp.com)
Hewlett-Packard Laboratories

Middleware systems are commonly used for building distributed systems, but they provide little, if any support for altering the quality of service aspects of an application. We wish to structure middleware so that QoS development can be carried out independently from application development. Separation allows experts in QoS fields to apply their work to any application developed for the middleware. Our approach is based on the notion of interceptors that are dynamically added to running applications. These interceptors are allowed to observe and modify application middleware calls to implement desired QoS functionality. We have developed a programming model for interceptors that supports development of a variety of QoS functionality, and have implemented this model in the context of the e` speak middleware platform.

1 INTRODUCTION

Middleware systems have helped to make distributed computing a reality. They succeed in hiding distribution concerns so that developers are able to build distributed applications almost as easily as they do non-distributed applications. This simplification has been the primary goal of middleware, and in particular, it has concentrated on the functional aspects of a distributed application. The functional aspects are those concerned with purely providing a service such as what data must be exchanged between client and server components to carryout a service request. In the widely used, off-the-shelf middleware systems, little effort has been made to accommodate the *non-functional* or Quality of Service (QoS) aspects of developing a distributed application. Because middleware does not explicitly support them, non-functional properties are commonly left to developers to implement within their applications. Therefore, these important properties are often either ignored, or, when they are deemed important enough, the subject of a large,

custom effort to provide the required non-functional properties for a specific application.

Unfortunately, the result is that the QoS aspects end-up being closely tied to the application, and therefore the results cannot be readily re-used in another application.

Supporting QoS demands in middleware has been the focus of many research efforts. This has been a good fit because middleware is logically positioned very close to applications, providing nearly an end-to-end view. Additionally, a large amount of semantic information (such as pairing of request and response messages) is available.

Efforts to support QoS in middleware have largely fallen into one of two categories. The first is modifications or extensions to middleware to enhance specific QoS properties. Examples of these include the TAO [Sch97] real-time CORBA ORB, and OMG specifications for real-time and fault-tolerance services [OMG98], [OMG99a]. The second is extensions to the middleware programming model to make QoS concerns explicit. Projects following this approach include Quality Objects (QuO) [Van98], the Management Architecture for Quality of Service (MAQS) [Bec98], and the Squirrel project [Kra98]. These approaches have similarities with aspect oriented programming. A QoS specification is written in parallel to the application specification, and the middleware merges these to create a QoS aware application.

As an alternative to these approaches, we have developed a programming model for adding explicit support for non-functional aspects to middleware based applications. The model is based on the notion of an *interceptor*, which can inspect every interaction between application components. As it inspects the call, it may alter or *customize* the interactions to satisfy a particular QoS goal. Interceptors are available in some middleware systems, such as CORBA, but as we discuss later, existing interception models are not well suited for implementing QoS related functionality.

By introducing a new programming model, we completely separate development of applications and non-functional logic. This allows us to re-use our QoS implementations with any application

including those previously developed on the middleware platform. The decision to separate applications and QoS functionality will not be universally correct. In some cases, the QoS functionality must be intertwined with applications. However, experience has shown it works in many situations, and has a number of benefits. Separation permits us to defer decisions on including QoS components until run-time when they may be dynamically loaded into applications. We may then customize based on new environments, and as conditions change. We feel these features provide some benefit over other approaches, but also may complement them as a means of implementing and deploying the enhancements developed within the other frameworks.

The remainder of the paper is organized as follows. Section 2 describes how interceptors fit into middleware, and our goals in defining an interception model powerful enough to enable QoS customization. Section 3 provides a detailed description of APIs for performing interception, and how they fulfill our goals of QoS customization. The next section briefly outlines how interceptors might be used to implement some basic QoS properties. Section 5 describes related work in the area of middleware-level interception, and their shortcomings for customization. The final section provides current status of this work, possible future directions, and concluding remarks.

2 OVERVIEW OF MIDDLEWARE INTERCEPTION FOR QoS

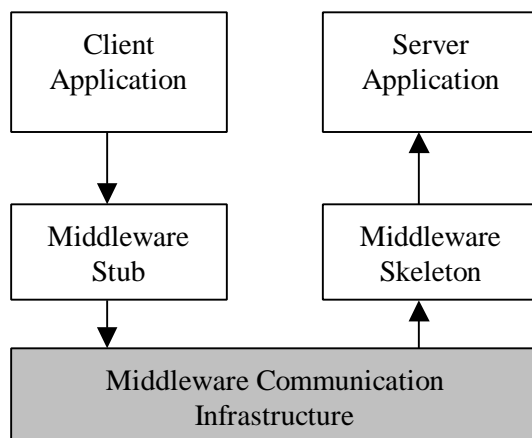


Figure 1 – Component interactions in an RPC-based middleware system

Figure 1 shows the components involved when invoking a service in a typical middleware system. The components provided by the middleware are the stubs and skeletons and the communication infrastructure. The stubs and skeletons are usually generated using an Interface Definition Language (IDL) compiler, and implement the logic needed to distribute the functional aspects of the application. The stub is invoked directly by a client to initiate a remote call. The stub, in turn, marshals the data associated with the request into a message, and transfers the message using a communication infrastructure to the site of the remote service. There, the skeleton takes the data out of the message, and directly invokes the actual service. This structure effectively transfers the functional requests between a client and a server, but it does not directly permit a method of handling the non-functional aspects. There is simply no place in the system where non-functional components can be written without changing the applications or the middleware.

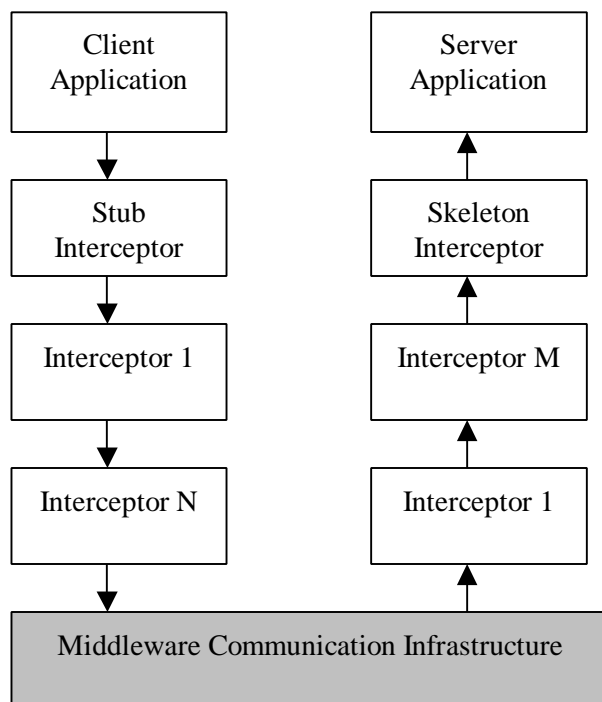


Figure 2 – Component interactions with interceptors

In Figure 2, we show an augmented middleware system that includes interceptors. This structure also contains IDL-generated stubs and skeletons. However, rather than having them pass messages directly two and from the communication layer, we introduce a set of interceptor modules. The interceptors receive the request prior to passing it to the communication system on the client side, and prior to the request being passed to the skeleton that invokes the service. The final interceptor on the client side, and the first interceptor on the server side perform the actual marshalling of a request’s data for transfer across the network. This general structure permits the requests to be inspected by the interceptors, and altered as needed to perform customization.

2.1 Design Goals for Interceptors

The structure shown above provides a mechanism for performing customization via interceptors, but it does not provide many guidelines on what specifically interceptors must be able to do. We have laid out a set of design goals for interceptors to permit them to implement a variety of non-functional customizations in many environments.

1. **No use of application specifications such as IDL** – Interceptors are most valuable when they are re-useable, and are thus developed independently from applications. Therefore, we cannot assume that any application specific information, such as interface definitions will be available.
2. **A powerful and easy-to-use programming model** – The ability to perform useful customizations comes from the power of the interceptor programming model. It must be general enough to enable a variety of customizations without becoming so complex or specific that it is difficult to use.
3. **Composability of interceptors** – Interceptors should be composable in the sense that multiple interceptions should be applicable to a single application or single interface concurrently without interfering with one another assuming the goals of the interceptors themselves are not contradictory. Determining what interceptors or goals may conflict with one another is a hard problem in general, and is beyond the scope of this paper. This goal is implied by the structure shown in Figure 2, but is called out explicitly to emphasize the point.
4. **Dynamic inclusion of interceptors** – We typically cannot know *a priori* what non-functional properties will be required for particular deployments or instances of services, clients or client-service pairs. We therefore require a method of deploying and inserting interceptors dynamically, while the clients and servers are already running, and perhaps already communicating with one another.
5. **Client/server neutrality** – A single model should be used for developing interceptors for use on either the client or the server side of a distributed interaction. A single interceptor implementation should be applicable on either side, as long as it can practically be applied on either side.

6. **Middleware neutrality** – We want our interceptor programming model to be independent of any particular middleware so that interceptors need not rely on any particular middleware functionality. Ideally, we will provide an abstraction layer on top of middleware so interceptors never access any middleware platform specific functionality. This facilitates building an environment where interceptors can be designed once and applied to applications running in a variety of middleware environments.

3 A PROGRAMMING MODEL FOR MIDDLEWARE INTERCEPTION

In the previous section, we discussed interception in general and our goals for our interception programming model and environment. In this section, we define a specific programming model for interception. This model is based on the goals described previously, as well as particular requirements we've developed through experience using interception to implement non-functional properties in the past. The resulting programming model developed to meet these requirements has three abstractions. The first, the "request", is used to represent a remote invocation made by a client application. This abstraction permits us to inspect a call, and to add to or extract data from the request. The second is an interface definition to be implemented by developers of interceptors. This interface permits developers to manage exceptions, inspect or alter middleware calls, or access other middleware functions. The final, "control," abstraction is used for controlling the use of interceptors such as installing or removing individual interceptors. This interface satisfies our high-level goal of making interceptor inclusion or removal dynamic. In the following sections, we describe interfaces for each of these abstractions.

3.1 The IceptorRequest Class

```
public class IceptorRequest
{
    public IceptorRequest(String interfaceName,
                          String methodName);
    public String getInterfaceName();
    public String getMethodName();
    public Object addParam(String paramName, Object val);
    public Object setParamValue(String paramName,
                                Object val);
    public Object getParamValue(String paramName);
    public Object setReturnValue(Object val);
    public Object getReturnValue();
}
```

The IceptorRequest class is used to represent a method invocation as it passes from the client, through various interceptors, and finally to the server (as show in figure 2). The client stub generates an instance of IceptorRequest for each application level call. Interceptors use the IceptorRequest object to inspect, alter or propagate the application's middleware calls. Each instance of an IceptorRequest is associated with a specific interface and method to be invoked on the server. This information is typically available in the stub because stubs are generated on a per-interface basis in most middleware systems. Each parameter for the call is added by name using the addParam method. Including the name permits interceptors to inspect a call for particular parameters by name, and change their values if needed. Once added to the parameter list for the call, parameter values are inspected or set using the getParamValue and setParamValue methods.

Return values are handled in much the same way as parameter values. There are explicit calls for setting and retrieving return values. Typically, only skeleton code that actually invokes the server object would call the setReturnValue method, however some sophisticated interceptors may be able to divine a return value without calling the server object, so interceptors are not explicitly prevented from calling this method.

3.2 The Iceptor Class

```
public abstract class Iceptor
{
    public abstract void register(Object params);
    public abstract void unregister();
    public abstract void invoke(IceptorRequest req)
        throws Exception;

    public void invokeNext(IceptorRequest req)
        throws Exception;
}
```

The Iceptor class defines the interface that must be implemented by authors of interceptors.

When an Iceptor instance is inserted into a middleware call-chain, its register method is executed, and is passed data to be used during initialization. As an example, this data might contain a reference to another service this interceptor will need to communicate with. The unregister method is called when the interceptor has been removed from the system, or when the client or server object it is intercepting calls on is removed by the application.

For every application level request made, the Iceptor object's invoke method is called. It receives the IceptorRequest object associated with this call. In the invoke method, the interceptor may inspect the interface and method names, the names and values of call parameters, add new parameters, and do any other processing it desires prior to propagating the call to the service.

When this pre-processing is complete, the interceptor must call invokeNext with the (possibly modified) IceptorRequest object. InvokeNext simply passes control on to the next Iceptor object in the logical chain. The end of the chain is the service object itself. When invokeNext returns, the interceptor can assume that the application's service has been invoked. We also permit the invocation to return an exception of some sort. This is typically used in distributed middleware to indicate a failure or some other unexpected outcome of an invocation. The interceptor may catch this exception and attempt to handle it itself, or it may simply re-throw this exception. If all

interceptors re-throw the exception, it will propagate back to the client just as if no interceptors were installed at all.

3.3 The IceptorControl Class

```
public class IceptorControl
{
    public id addInterceptor(Iceptor iclept,
                           Object params);
    public boolean removeInterceptor(id);
    public void removeAllInterceptors();

    public void invokeFirst(IceptorRequest req);
}
```

The IceptorControl class performs two basic functions. First, it does record keeping on the chain of interceptors associated with a client or server. The addInterceptor, removeInterceptor and removeAllInterceptors methods are used for doing this bookkeeping. The addInterceptor takes an instantiated Iceptor object and inserts it on the end of the current chain of interceptors, and calls its register method with the provided parameter object. Interceptors are removed via the removeInterceptor method by using the identifier returned by addInterceptor. Likewise, the removeAllInterceptors call will remove every interceptor in the call-chain. As presently defined, the IceptorControl interface is available only within a client or server process. In a more complete implementation, this interface could be exposed externally, permitting management of interceptors, and the QoS properties associated with them to be manipulated by a third-party. We briefly discuss issues in this area later.

The final piece of interceptor management handled by the IceptorControl class is executing the interceptor chain. This is accomplished by calling the invokeFirst method with a completed IceptorRequest object. Middleware stubs use this call to initiate processing of an application level request.

3.4 Summary of the Features of the Interceptor Programming Model

The primary value of our interceptor model comes from its ability to permit a variety of non-functional properties to be implemented using it. The features listed below describe specific functionality our programming model provides to enable this. In some respects, these features demonstrate how we satisfy our previously stated goals. They provide specific attributes of the programming model where the previous list gave abstract goals for a middleware system that includes interception.

1. **Exception catching** – By explicitly invoking handling of the application call, interceptors can catch exceptions thrown by either the middleware or the application.
2. **Exception throwing** – An interceptor's invoke method may throw an exception, simulating or propagating an application or middleware request, or may introduce new exceptions.
3. **Parameter inspection** – Interceptors receive the request object associated with each middleware call that allows inspection of all the parameters associated with the call.
4. **Inserting and extracting extra data or parameters** – The request object also provides a way for adding and extracting extra data on the call in a manner that does not interfere with the application level call. This permits interceptors to communicate directly among themselves when needed.
5. **Short-circuiting / local handling** – An interceptor can avoid propagating a request forward simply by not calling its invokeNext method. A common sort of customization that may use this facility is one that does client-side caching of results.
6. **Access to other middleware services** – Interceptors are not in any way limited by the programming model from calling middleware services. In our current implementation, we even provide additional middleware information via the IceptorControl interface. This

functionality is largely in opposition to our goal of middleware neutrality, but is important from a pragmatic sense of permitting interceptors to truly perform useful functions.

4 USING INTERCEPTORS TO IMPLEMENT QoS PROPERTIES

We have described our desire to support QoS properties in middleware, and our programming model for doing so. To demonstrate the power of interceptors, and illustrate the importance of the features described above, we provide outlines of interceptors that achieve two properties: availability and admission control. Our goal in these sections is not to attempt complete solutions, but to give a flavor of what must be done to provide this functionality, and how the interception programming model supports these needs. Clearly, complete solutions in these areas are beyond the scope of this article.

4.1 High Availability via Interception

Failures of hardware and software components are inevitable, so the goal of a high availability system is to mask or hide failures as often as possible so that they do not interfere with the tasks users wish to perform. In a distributed system the goal is for remote services to appear to be operating continually even when the reality is that components that make up the service are not functioning. A crucial part of achieving this is detecting when a failure occurs, discovering an alternate server, and continuing operations with this new server. We refer to this process as *failover*. If we assume that servers are stateless between client invocations, and that they have no side-effects (e.g. database interactions), a collection of servers and basic failover may be all that is needed for a service to become highly available. Below is pseudo-code for a client-side interceptor that performs basic failover.

```

public class FailOverIceptor extends Iceptor
{
    public void register(Object params)
    {
        store a "description" of the service
    }

    public void invoke(IceptorRequest req) throws Exception
    {
        for (I = 0; I < retryLimit; I++) {
            try {
                return invokeNext(req);
            } catch (FailedServiceException e) {
                use the "description" to query for a
                replacement service;
                if (query returns a new service) {
                    re-bind the client to the new service;
                } else {
                    re-throw the exception;
                }
            }
        }
        throw FailedServiceException;
    }
}

```

In the register method, we save a description of the current server. This description will be dependent on the middleware system in use. It may, for example, contain a query string to be passed to a trading service of some kind.

In the invoke method, we start by setting a loop for the maximum number of times to re-try a request. In some cases, it truly will not be possible to find a functioning service replica, so we should only try a few times. We then set-up to catch an exception thrown by the middleware indicating that a service has failed. Note that in the normal case, we simply call invokeNext and return without performing any additional processing. However, when the failure occurs (as indicated by a caught exception), we attempt to find a replica using the description stored during initialization to form a new query. If the query succeeds in finding a replica, we re-bind the client to the new server so that all subsequent calls will go to the new server. After re-binding, we loop back to the top to re-invoke the call on the new server. If we fail to find a replica, all we can do is re-throw the exception to the higher layers indicating this unrecoverable failure.

This example makes a lot of simplifying assumptions, but it shows the potential for performing tasks needed to achieve high availability in the middleware. It relies heavily on the ability to catch exceptions, handle them, and reissue a call to a new server. In a more practical example, we would have to concern ourselves with the state of the new server as compared to the old server, and what other side-effects the service call may have. In any case, failover will be one of the crucial components of a high availability solution, and synchronization with a newly selected server could also be handled in the interceptor.

4.2 Admission Control via Interception

Admission control is used in computer networks to limit congestion within a network. By denying some packets admittance to the network, we prevent them from interfering with other packets, and possibly degrading performance for other users. A similar technique can be used in higher-level computer services. By denying access to the service for some users, we can improve the QoS delivered to others. Here, we sketch an approach for implementing server-side admission control via an interceptor. In practice, we may wish to have a complementary interceptor on the client-side that delays requests to reduce the number that are rejected by the server.

```

public class AdmitControlIceptor extends Iceptor
{
    void register(Object params)
    {
        Use the params information to determine user groups
        and their permitted request rates;
    }

    boolean invoke(IceptorRequest req) throws Exception
    {
        extract the identity of the client from the request
        or other middleware-provided information;
        compute new request rate for client's group;
        if (request rate < allowed request rate) {
            return invokeNext(req);
        } else {
            throw RequestRejectedException;
        }
    }
}

```

In this example, we assume that a description of user groups will be provided during initialization. This would be a list that would permit the server to categorize each user as a member of one of these groups. We also receive an allowed request rate for each group.

On each call to the invoke method, we start by determining the identity of the client. This information is available directly from some middleware systems, but in others we could introduce a client-side interceptor that adds identity information (perhaps including a digital signature) into the request. We use this identity to determine the client's group membership, and to compute the current rate of requests for that group. Then, we simply compare the observed rate to the permitted rate, and process the request if it is acceptable. If it is not, we reject the invocation by throwing an exception.

As with the availability example, this is a greatly simplified view of the problem. However, it once again shows that the interception model permits us to implement a useful customization to an arbitrary service. It is also worth noting that these customizations could be used in tandem. It is entirely possible to have both the availability and the admission control customizations in use at the same time without interference.

5 RELATED WORK

The notion of intercepting application-level operations, whether middleware related or not is well known. Indeed, various interception techniques have been used in the past. Typically, they involve modifying an application executable on disk or in memory such that procedure calls are routed to newly inserted code rather than the original procedure entry address. These techniques have been used for a variety of purposes including debugging, application instrumentation and monitoring and so forth. More recently, middleware systems have embraced interception, and exposed it directly. Here, we describe basic methods used for interception in middleware, and how they relate to our goal of using interception for QoS customization.

5.1 Callbacks

One of the common methods for providing a form of interception is via a “callback.” Callbacks are typically established by informing a lower layer of a system, such as middleware, of a procedure that should be called when a particular event occurs. For middleware, the events are usually messages being sent or received. Callbacks are used in both CORBA and Microsoft’s DCOM for monitoring messages between clients and servers.

In CORBA [OMG99b], two interfaces are defined for interceptors: one for “request level” interception, and one for “message level” interception. The request level interception interface consists of two callback functions, one called prior to the service executing, and one called after the service has been executed. Each of these is provided with a request object similar to the one we described. The message level interceptors are provided with an array of bytes into which the request data as been marshaled prior to the message being sent, or just after it has been received but not yet unmarshaled into the request object that represents the call.

DCOM [Edd98] provides a similar callback mechanism referred to as “channel hooks” that are very much like the message level interceptors in CORBA. A channel hook registers with DCOM by providing a unique identifier for itself. When the hook is invoked, it may add extra data into a

message, and that data is tagged with the hook's unique identifier. When a DCOM message is received that contains data with that tag, it is provided to the channel hook in a callback.

Callbacks prove to be difficult to use in many customization scenarios. Consider the high-availability example above. In it, we maintain some state (the retry count), catch exceptions, throw exceptions, and re-issue failed calls. Each of these things is difficult to perform in a callback environment. State must be maintained in a globally accessible location so that it can be used in the multiple-callbacks. Exceptions cannot be thrown directly to a callback, so it is not possible to catch them in a callback-based interceptor. It may be possible to generate an exception in the callback, though the existing models do not explicitly support it. Finally, with callbacks, there is no possibility of re-issuing an application level call. The middleware itself performs these calls when the callback returns. A further complication of this model is that it splits the logic of the interceptor. The pre-call and post-call logic are placed in separate procedures, so there's no single place where all of the logic that makes up the interceptor can be seen. This complicates debugging, and other issues generally related to code maintenance.

5.2 Wrappers

Another approach to interception is to provide "wrappers" in the middleware that receive calls prior to the application. This approach is very similar to ours in that new code receives a call prior to the application. Unfortunately, none of the standard middleware seems to provide an extensible infrastructure for writing wrappers. That is, the wrappers are built into the middleware, and third parties cannot directly write new wrappers.

The most widely used system based on this wrapper approach is the Microsoft Transaction Server (MTS) [Ree97]. MTS provides a wrapper around server-side DCOM applications to generate a transactional context around these objects. When the server is invoked, the call is intercepted by MTS, and a transaction is started against a database. The call is then forwarded to the server, and the transaction is then either committed or aborted based on a status set by the server.

While this approach is similar to ours at a basic level, it is not suitable for doing QoS customization simply because there is no way to insert interceptors other than the one built into MTS. Also, the DCOM/MTS system works only on the server-side of an interaction. There is no support for interception on the server side. Microsoft is, however, evolving their use of these techniques in newer versions of their middleware (COM+) to support integration with message queuing and other enhancements, so it may become general and open enough to be a basis for QoS customization in the future.

6 STATUS, FUTURES AND CONCLUSIONS

We are using the middleware interception approach described here as a basis for our on-going work in customization of distributed systems. To support this work, we have implemented our interception programming model in the e` speak [HP99] system. E` speak was designed at Hewlett-Packard, and has been released in open-source as a platform for brokering, composing and performing services on the Internet. By including interceptors in the e` speak open-source, we hope to provide a platform for others to do QoS related work in middleware as well as make e` speak a more robust platform.

Our current focus is on the deployment of interceptors. We are striving for a deployment method that is flexible, scalable and secure. Flexibility implies that we want to remotely add or remove interceptors from clients or servers based on administrative or policy based control. E` speak is intended to run in Internet scale environments, so we must also make deployment of interceptors work on this scale. Finally, because interceptors have complete knowledge of client-server interactions, including access to the data in these messages, we must make deployment secure to insure interception does not become a method easily used to compromise application level data.

Middleware provides an attractive opportunity for customizing non-functional properties of systems. Middleware is close enough to applications so that a large amount of semantic information, such as send/reply message pairs, and the layout of the data in those messages, is

available. This permits us to introduce a variety of interesting algorithms fairly easily without changing the functional behavior of the applications. By providing a powerful interception programming model, we hope to introduce QoS properties in environments that otherwise would not be possible simply because it is not possible to modify the applications or the lower level parts of the system such as the operating systems or networking infrastructure. This, in turn, will make use of QoS mechanism more widely used, and lead to better overall performance of the distributed applications we rely upon.

References

- [Bec98] C. Becker and K. Geihs. Quality of Service - Aspects of Distributed Programs. International Workshop on Aspect-Oriented Programming at ICSE'98, Kyoto/Japan (1998).
- [Edd98] G. Eddon and E. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.
- [HP99] Hewlett-Packard Corporation. *E` speak Architecture Specification*. November 1999. Available from www.e-speak.net.
- [Kra98] T. Kramp, R. Koster. A Service-Centred Approach to QoS-Supporting Middleware, Work-in-Progress Paper, Middleware '98, September 1998, The Lake District, England.
- [OMG98] Object Management Group, *CORBA Real-Time Service*, OMG, 1998.
- [OMG99a] Object Management Group, *CORBA Fault Tolerance Service*, OMG, April 1999.
- [OMG99b] Object Management Group. *CORBA 2.3.1 /IIOP Specification*. OMG Document 99-10-07 (1999).
- [Ree97] D. Reed, T. Trewin and M. Tomsen. Microsoft Transaction Server Helps You Write Scalable, Distributed Apps, *Microsoft Systems Journal*, August 1997.
- [Sch97] D. Schmidt, D. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker, *Computer Communications Journal*, 1997.
- [Van98] Vanegas R, Zinky JA, Loyall JP, Karr DA, Schantz RE, Bakken DE. QuO's Runtime Support for Quality of Service in Distributed Objects, in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (Middleware'98), 15-18 September 1998, The Lake District, England.