Cover Feature

Using Interceptors to Enhance CORBA

By using interceptors—nonapplication components that can alter application behavior transparently—you can enhance a CORBA framework or application without modifying either.

Priya Narasimhan
Louise E. Moser
P.M. Melliar-Smith
University of California, Santa Barbara

The integration of distributed computing and the object model leads to distributed object computing, in which objects rather than processes are distributed across multiple computers. A well-established standard for distributed object computing is the Common Object Request Broker Architecture (CORBA).[1] Distributed object frameworks like CORBA have many attractive features but provide little support for alternative protocols, profiling and monitoring, security, or reliability.

Previously, you would have had to create—and enable the application to use—the components that provide such additional capabilities. Even if the components that provide these capabilities already exist, using them requires substantial effort, as well as specialized knowledge and understanding of problems outside the application domain.

With the advent of *interceptors*—nonapplication components that can alter application behavior—you can enhance CORBA applications at runtime with components whose operation is transparent to both the application and the CORBA framework; this means that you can modify application behavior without having to modify the application or the CORBA framework.

The Object Management Group—the CORBA standards body—recognizes the value of interceptors.[2] Other distributed object computing standards, such as Microsoft's Distributed Component Object Model (DCOM), provide for interceptor-like components through *custom marshaling mechanisms*,[3] which enable an application to bypass the standard communication mechanisms and to use custom ones.

Interceptors can also allow you to chain together multiple components (each with its own functionality) to achieve new functionality transparently. Over the past few years, we have developed a system—called Eternal—that exploits interceptors transparently.[4-6] The Eternal system enhances unmodified CORBA applications with reliability by using an interceptor to chain together protocol, monitoring, scheduling, and replication management components at runtime.

HOW CAN CORBA BE ENHANCED?

Since the OMG's adoption of CORBA 1.0, the standard has become increasingly popular for building distributed object applications. The key component of CORBA is the object request broker (ORB), which acts as an intermediary between the client and the server. The ORB supports language transparency (allowing clients and servers to be programmed in different languages), location transparency (allowing clients to contact servers with no knowledge of the server's location), and interoperability (allowing clients and servers to communicate across different platforms).

Alternate protocols

At present, CORBA objects cannot interact with other kinds of distributed objects over protocols that do not conform to its General Inter-ORB Protocol (GIOP). Every conformant ORB must implement the standard mapping of the GIOP specifications onto TCP/IP, also known as the Internet Inter-ORB Protocol (IIOP). To enable CORBA applications to communicate using protocols other than IIOP, application programmers must expend effort in rewriting the ORB's transport-level mappings.

A better approach is to divert the GIOP messages sent by CORBA objects—transparently and without modification to the ORB—to components that map GIOP onto other protocols. In this way you can build protocol adapters that take advantage of CORBA's capabilities without modifying the application or the ORB.

Profiling and monitoring

CORBA does not currently include components for profiling or monitoring. While non-CORBA compo-

0018-9162/99/$10.00 © 1999 IEEE

nents may exist that provide such services, they cannot readily be used within the CORBA framework. To use them, you would first have to understand the mechanisms of the component and then incorporate the necessary hooks into the application or the ORB.

A better approach is to use existing profiling components that can trace an application and its messages transparently, without the need to embed any special code into the application or the ORB. You can use such components for monitoring the system and measuring its performance.

Scheduling

CORBA provides support for many of the commonly used multithreading models. However, for some applications it is desirable to employ a custom thread-dispatching policy that the ORB does not support. To do this, you would need to rewrite the ORB's dispatching code.

A better approach is to use an interceptor to introduce a scheduling component before the incoming requests reach the target objects. By controlling the order in which the threads and the requests are released to the application, the scheduling component can override the ORB's—or the application's—thread and scheduling policies.

Reliability

The fault tolerance mechanisms that CORBA currently provides are rudimentary, consisting mostly of exceptions returned if an object or a processor fails. To meet the reliability needs of the applications, you would have to implement the necessary fault tolerance mechanisms within the application.

A better approach is to use an interceptor to add components that handle object replication, fault detection, and fault recovery in a manner that is transparent to both the CORBA application and the ORB. These components, once configured, operate independently of the programmer, the application, and the ORB.

INTERCEPTOR MECHANISMS

Current OSs provide hooks that can be exploited to develop components such as interceptors. Unix provides at least two possible implementations.

- *System calls.* The /proc-based implementation provides for interception at the level of system calls.
- *Library routines.* The library interpositioning implementation provides for interception at the level of library routines.

While the techniques differ, the intent and use of the interceptor in both cases is identical and requires no modification to the intercepted CORBA objects, the ORB, or the OS.

System call interception

The mechanisms of the /proc-based implementation, developed in the context of global file systems,[7] extend the functionality of standard OSs at the user level. In this approach, an interception layer transparently attaches itself to an executing process in order to monitor and control its behavior. Neither the client nor the server needs to be modified or recompiled to exploit this approach.

In the Solaris and Linux OSs, the /proc file system provides access to the internals of each running process on the computer. Each entry in the /proc directory, also called a process's image, is a file whose name corresponds to the Unix process identifier. The files in the /proc file system, and thus the processes they represent, can be manipulated via a standard interface that allows each process's system calls to be captured. The arguments and the return values of the intercepted system calls can be extracted and even modified. The appropriate patching of these system calls can alter the behavior of the intercepted process.

As Figure 1a shows, each CORBA client or server is viewed as a process that can be controlled via the /proc interface. Thus, each CORBA object can be monitored during its lifetime for system calls related to many different activities, including memory management, network communication, and file access. An interceptor can be designed to watch for specific system calls made by CORBA objects when they communicate over IIOP.

Library routine interception

The library interpositioning implementation exploits the facilities of the Unix runtime linker,[8] which allows shared objects to be added to a process as it

Compression

An interceptor can be used to add a compression component that transparently modifies the messages exchanged among the objects of a CORBA application, while leaving the behavior of the application unchanged. Most data is exchanged between application objects via IIOP messages. Thus, the interceptor's interface is targeted at the communication of IIOP messages.

At the sender, the interceptor captures the outgoing IIOP message and passes it to a compression component, which compresses the data and hands the compressed data back to the interceptor. The interceptor, in turn, passes the IIOP message containing the compressed data back to the ORB, which sends it out using TCP/IP. At the receiver, the interceptor captures the IIOP message that it receives from TCP/IP. It passes the IIOP message to a decompression component, which produces the original IIOP message. The interceptor hands the IIOP message back to the ORB, which delivers it to the target object.

The sender (receiver) object is completely unaware of the presence of the interceptor and the compression (decompression) component. Moreover, the ORB never sees the compression or decompression components and handles the IIOP messages containing the compressed data just as it would handle IIOP messages without compressed data.

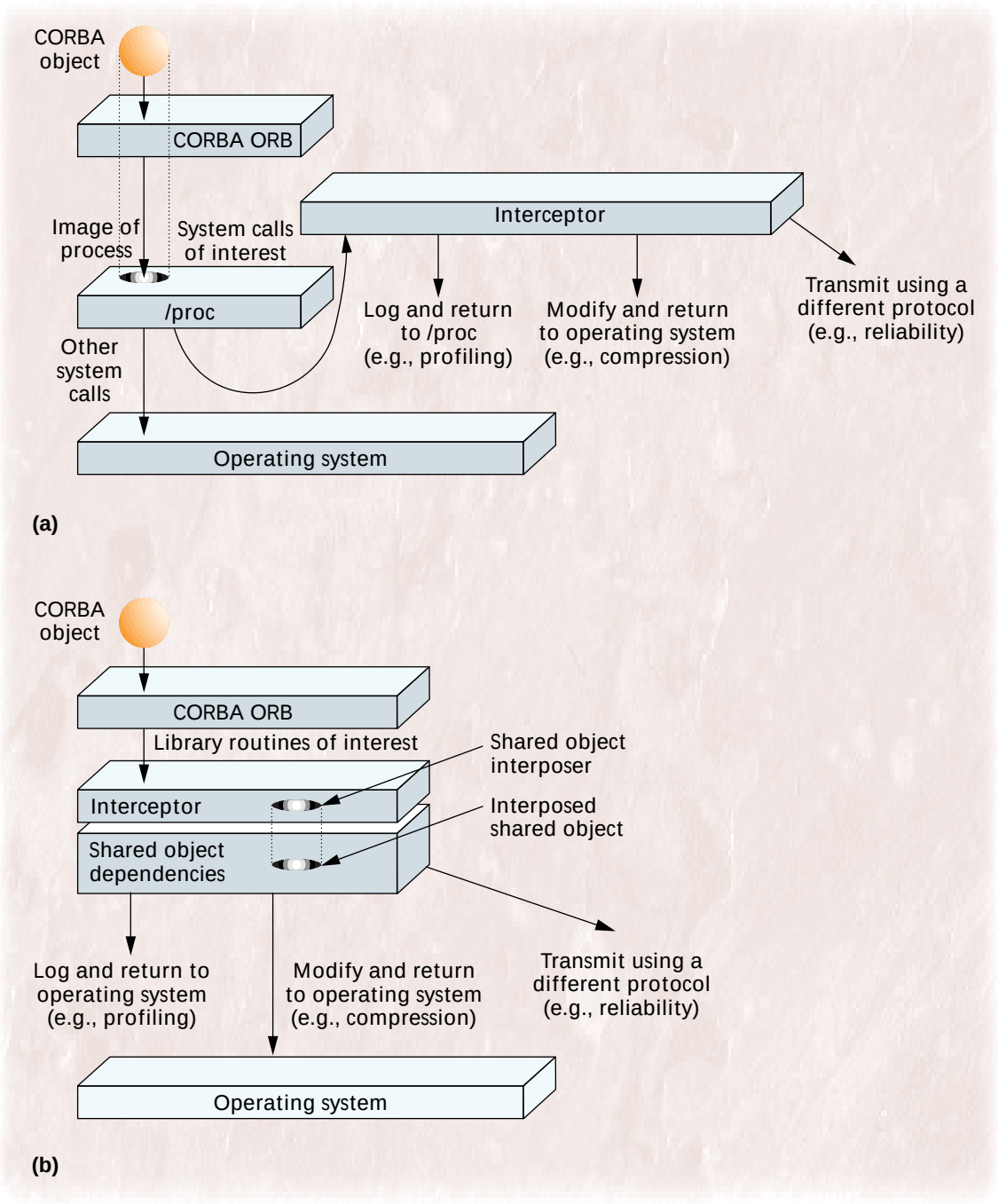

Figure 1. Possible implementations of interceptors using (a) system call interception and (b) library routine interpositioning.

initializes. These shared objects are mapped into the process space after the dynamic executable is launched, and before any of the normal shared object dependencies are loaded into the process space, as Figure 1b shows. To achieve this effect, the runtime linker does not require that the application be modified, relinked, or recompiled.

Library interpositioning exploits the fact that an executable can have symbols (undefined variables and functions) whose definitions remain unresolved until runtime. At runtime, the first shared object that resolves a symbol is accepted as the symbol's definition. If subsequent shared objects in the same process space also provide definitions for the same symbol,

these definitions are ignored by the runtime linker. The first definition is said to *interpose* on all other definitions of the same symbol. A process can thus be made to use custom function definitions provided by an interposed library in place of the original or standard function definitions.

Microsoft Windows NT's Dynamically Linked Libraries (DLLs) provide similar hooks that can be exploited to build interceptors.

USING INTERCEPTORS

Here are the ways you can enhance CORBA for profiling, protocol adaptation, scheduling, and reliability. The interceptor's interface—the system calls that you intercepted in a /proc-based implementation or the library routines that you redefined in a library interpositioning implementation—depends on the information that the interceptor must provide to the added component. The interceptor's interface can cover all (or a subset of) the system calls or library routines used by a CORBA application.

Protocol adaptation components

If you wanted to send the IIOP messages exchanged between CORBA objects using a protocol other than IIOP, you could use an interceptor to add a protocol adaptation component. The protocol adaptation component that the interceptor introduces may simply encapsulate the IIOP messages using a protocol-specific header without ever modifying the IIOP messages. Alternatively, the protocol adaptation component may transform all, or a part of, the contents of the IIOP messages into the equivalent messages of the alternative protocol. In either case, for the protocol adaptation to work, the alternative protocol must preserve the semantics of IIOP communication.

To enable CORBA applications to communicate using an alternative protocol, the interceptor's interface must be targeted not only at the communication of IIOP messages, but also at the establishment of connections, the release of connections, and the queries related to connection status and connection information. The interceptor must prevent the intercepted system calls or library routines from ever reaching TCP/IP or the OS. Instead, these system calls and library routines must be mapped onto the equivalent mechanisms of the alternative protocol. This might require the use of an IIOP parser component.

Using interceptors in this way to construct protocol adapters is particularly useful when CORBA applications are required to communicate with legacy systems that use some proprietary protocol. In this case, an interceptor-based protocol adapter has the advantage of enabling interworking between the legacy system and the CORBA application without requiring you to modify either of them. To meet real-time requirements, some CORBA applications need a more stringent protocol than IIOP. Modification of either the applications or the ORB might be both infeasible and expensive, and interceptor-based protocol adapters can provide an effective solution.

Profiling components

You might want to build execution profiles to determine, for example, a system's resource allocation for a CORBA application or an application's runtime resource usage. The system calls or library routines that comprise the interceptor's interface depend on the application and on the kind of profile to be generated.

Adding interceptors for the purpose of profiling involves recording, but not modifying, the application's behavior. A profile of each application object can be constructed by using an interceptor to add a logging, analysis, or display component to record or interpret the information in the intercepted system calls or library routines. For instance, to build a profile of a CORBA application's network use, the interceptor's interface is targeted to capture the communication of IIOP messages over the network. For a user-friendly presentation of the intercepted information, the interceptor can be chained with a display component that shows the network traffic.

With interceptors you can enhance CORBA applications at runtime with components whose operation is transparent to both the application and the CORBA framework.

Security

An interceptor can be used to enhance CORBA applications with security components that encrypt and decrypt, authenticate, verify, or reject connections on the basis of user-specified or configurable security policies. The encryption and decryption components modify IIOP messages exchanged by the application objects without modifying the application's behavior. Other security components, such as those for authentication, may result in modification of the application's behavior.

For security components that encrypt and decrypt IIOP messages, the interceptor's interface is targeted at the communication of IIOP messages. The interceptor at the sender captures outgoing IIOP messages and passes them to a component that encrypts the data contained in them. The interceptor then passes the IIOP message containing the encrypted data back to the ORB, which sends it over TCP/IP to the target object. At the receiver, the interceptor extracts the IIOP message containing the encrypted data, obtains the original IIOP message using a decryption component, and passes the message to the ORB for delivery to the target object.

Other security components may use protocol adaptation components to send the IIOP messages using a secure protocol instead of TCP/IP. Security components may also use profiling components to monitor CORBA applications. In this case, the profiler watches all connections to and from the application objects, and may reject connections from untrusted computers—or may permit restricted access to specific computers. A security component may also employ a display component to show security-related information in a user-friendly way.

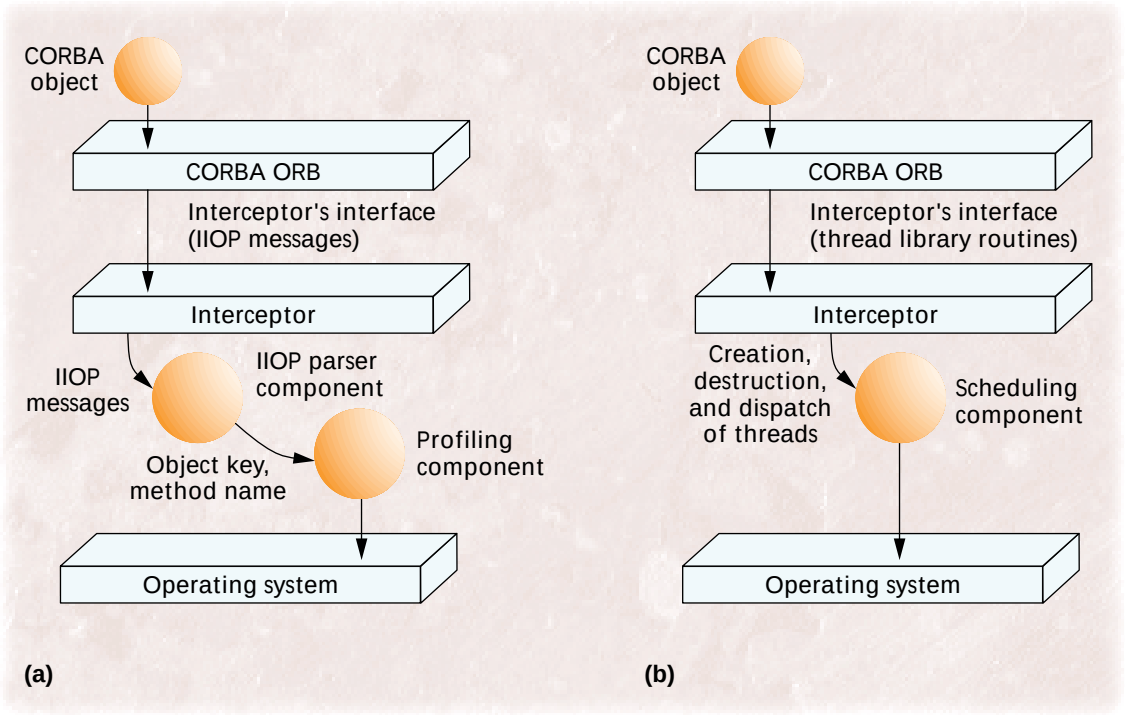

Figure 2. Applications of interceptors to add components for (a) parsing and profiling IIOP messages and (b) scheduling.

To build a profile of a CORBA application's method invocation or object access patterns, an interceptor requires detailed information at the object and method levels. Fortunately, the IIOP messages exchanged between CORBA objects contain information about the identity of the target server object and the method to be invoked on that object (the results and exceptions generated by the invoked method). Unfortunately, CORBA does not currently provide an accessible interface to parse the intercepted IIOP messages and extract this useful information. An interceptor's interface for the capture of IIOP messages, coupled with a custom IIOP parser component, is an effective solution. As Figure 2a shows, the interceptor captures the IIOP messages and passes them to an IIOP parser component, which then extracts the useful information from the message and may use a profiling component to store and display the method invocation and object access statistics.

Scheduling components

You can use interceptors to provide a specialized scheduling strategy for the methods executed by the objects of a CORBA application. While an interceptor doesn't result in the modification of the method that the object executes, it does add a scheduling component to modify the point in time at which the method is executed on the object. An interceptor-based scheduling component can transparently alter the behavior of the application by imposing a specific custom scheduling strategy on method invocations.

Multithreaded ORBs and applications typically employ the standard Posix or Solaris user-level thread libraries for all thread-related functions. The library interpositioning implementation thus lends itself well to the building of thread interceptors. The interceptor interface consists of library routines related to thread creation, thread release, thread management, and the dispatch of method invocations onto threads. For each of these intercepted library routines, the interceptor passes the intercepted information to a scheduling component, as Figure 2b shows. The scheduling component stores this information, enqueues any incoming method invocations, and decides the point in time at which the method invocation is released to the target object and is thus allowed to execute. Thus the scheduling component's multithreading, request handling, and invocation dispatching policies override any that are implemented by the ORB or the CORBA application.

This combination of a thread interceptor and a scheduling component is particularly useful for building a custom threading framework or for introducing a new multithreading policy without requiring modification of the ORB.

Reliability components

Perhaps the most striking use of interceptors is in enhancing CORBA with fault tolerance. The Eternal system provides fault tolerance by replicating objects within a CORBA application and transparently distributing the replicas across the system. For replication to be effective, every replica of an object that performs an operation must have the same state so that the replicated object can continue to provide useful service even when one of its replicas fails.

For replica consistency, all of the replicas of an object that execute incoming method invocations must see the same sequence of method invocations in the same order so that they have the same state at the end of each method execution. By the very nature of replication, each replica of a client object that

invokes a method issues the same invocation. Consequently, the target server object may receive duplicate invocations. Such duplicate messages must not be allowed to execute and corrupt the state of the target object.

Multithreading in the ORB and in the application leads to additional difficulties in maintaining replica consistency. The specification of multithreading in CORBA provides no guarantees on the order in which the ORB dispatches incoming method invocations onto the target object's threads. The ORB's handling of multithreading means that the order of method executions in two replicas of the same object might be different. Consequently, their states at the end of a sequence of dispatches of invocations onto threads might be inconsistent.

Maintaining replica consistency while allowing replicas to crash and recover requires difficult programming. Through its novel use of interceptors and chained components, as shown in Figure 3, the Eternal system provides transparent fault tolerance with strong replica consistency for CORBA applications. As shown in Figure 3, for outgoing messages, the profiling component monitors replicas and processor load for resource management, the protocol adaptation component encapsulates IIOP messages for transmission, and the replication management component adds information to enable detection and suppression of duplicate operations.

For incoming messages, the scheduling component schedules and dispatches operations, the replication management component detects and suppresses duplicate operations, the protocol adaptation component converts reliable multicast messages into IIOP messages, and the profiling component behaves the same as with outgoing messages.

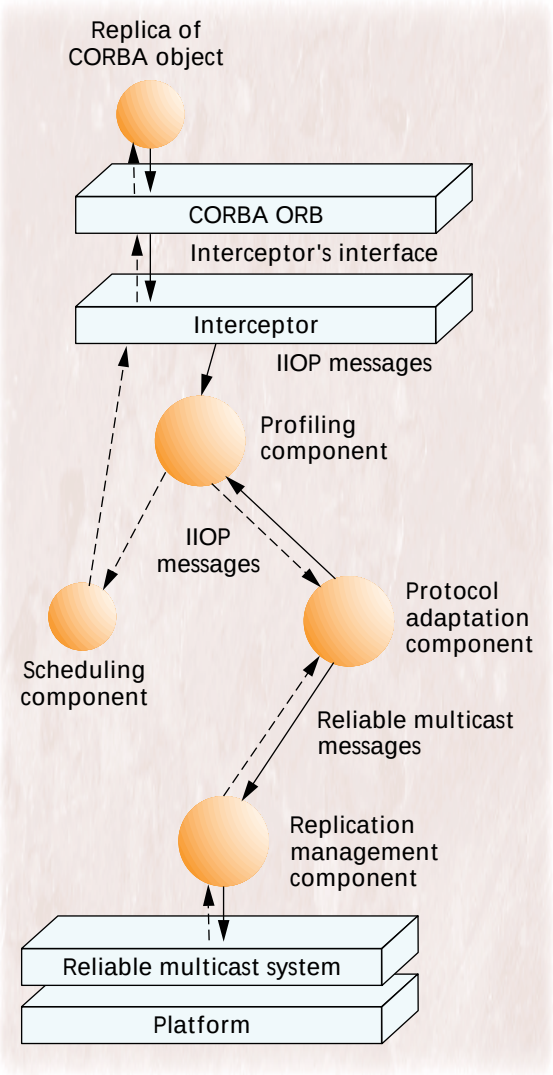

Figure 3. Using an interceptor in the Eternal system for outgoing messages (solid arrows) and incoming messages (dashed arrows).

ETERNAL'S INTERCEPTORS

The interceptor's strength lies in its ability to allow an unmodified precompiled CORBA application to benefit from the enhancements that Eternal provides. In the current implementation of the Eternal system, the interceptor uses the library interpositioning approach to add components for protocol adaptation, scheduling, replication management, and profiling. The ORB-independent nature of the Eternal system's interceptor has made it possible to port these components to multiple commercial ORBs with relative ease.

Reliable multicast

To facilitate consistent replication, the interceptor introduces a protocol adaptation component to convey the application's IIOP messages over a reliable totally ordered multicast protocol, such as Totem,[9] instead of TCP/IP. The protocol adapter merely encapsulates, but does not transform, each intercepted IIOP message with a header that contains sufficient information to enable the message's transmission over the reliable multicast protocol. The information in the reliable multicast protocol header is derived in part from the information (for example, the identity of the target object) in the IIOP message.

Consistent multithreading

To preserve replica consistency for multithreaded objects, or for processes containing multiple single-threaded objects that share data, the interceptor introduces a scheduling component. The scheduling component ensures a consistent order for the dispatch of threads and method invocations within the appli-

cation, regardless of the ORB's or the application's multithreading policies.

Replication management

To enable the detection and suppression of duplicate invocations and duplicate responses, the interceptor adds a replication management component. This component inserts information into each outgoing message before transmission to enable duplicate detection at the receiving end. An additional component is used to enable the logging of messages, and the transfer of state to new and recovering replicas, in order to bring their states up-to-date.

Resource management

To support the practical deployment of replicated objects, resource management is essential. The interceptor adds a profiling component that monitors the loads and the network traffic on the computers that host the replicas, and provides information that enables the migration of replicas for the purpose of load balancing. In addition, the profiler provides information about the resource usage of each computer in the system, which is used in the allocation of new replicas to computers.

Although we have described interceptors here in the context of CORBA, the interception approach is generic and can be applied to other distributed object frameworks. The value of interceptors lies in their ability to enhance an existing framework—and supported applications—transparently. ❖

Acknowledgments

This work has been supported by the US Defense Advanced Research Projects Agency in conjunction with the Office of Naval Research and the Air Force Research Laboratory, Rome, N.Y., under contracts N00174-95-K-0083 and F30602-97-1-0248.

References

1. *The Common Object Request Broker: Architecture and Specification*, Rev. 2.2, Object Management Group, Framingham, Mass., 1998; ftp://ftp.omg.org/pub/docs/formal/98-07-01.pdf.
2. *Portable Interceptors: Request for Proposals*, Object Management Group, Framingham, Mass., 1998; ftp://ftp.omg.org/pub/docs/orbos/98-09-11.pdf.
3. Y.M. Wang and W.J. Lee, "COMERA: COM Extensible Remoting Architecture," *Proc. Fourth Usenix Conf. Object-Oriented Technologies and Systems*, Usenix, Berkeley, Calif., June 1998, pp. 79-88.
4. L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan, "Consistent Object Replication in the Eternal System," *Theory and Practice of Object Systems*, Apr. 1998, pp. 81-92.
5. P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, "Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance," *Proc. Third Usenix Conf. Object-Oriented Technologies and Systems*, Usenix, Berkeley, Calif., June 1997, pp. 81-90.
6. P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, "Replica Consistency of CORBA Objects in Partitionable Distributed Systems," *Distributed Systems Eng.*, Sept. 1997, pp. 139-150.
7. A.D. Alexandrov et al., "Ufo: A Personal Global File System Based on User-level Extensions to the Operating System," *ACM Trans. on Computer Systems*, Aug. 1998, pp. 207-233.
8. T. Curry, "Profiling and Tracing Dynamic Library Usage via Interposition," *Proc. Summer 1994 Usenix Conf.*, Usenix, Berkeley, Calif., June 1994, pp. 267-278.
9. L.E. Moser et al., "Totem: A Fault-Tolerant Multicast Group Communication System," *Comm. ACM*, Apr. 1996, pp. 54-63.

Priya Narasimhan is a PhD candidate in electrical and computer engineering at the University of California, Santa Barbara. Her research interests include distributed object systems, components, frameworks, CORBA, fault tolerance, and evolution. She received an MS in electrical and computer engineering from the University of California, Santa Barbara.

Louise E. Moser is a professor of electrical and computer engineering at the University of California, Santa Barbara. She has served as an area editor for Computer *magazine in the area of networks and is currently an associate editor for* IEEE Transactions on Computers. *Her research interests span the areas of distributed systems, computer networks, and software engineering. She received a PhD in mathematics from the University of Wisconsin, Madison.*

P.M. Melliar-Smith is a professor of electrical and computer engineering at the University of California, Santa Barbara. His research interests include fault-tolerant distributed systems and high-speed communication networks and protocols. He received a PhD in computer science from the University of Cambridge, England.

Contact the authors at the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106; priya@alpha.ece.ucsb.edu; moser@ece.ucsb.edu; pmms@ece.ucsb.edu.