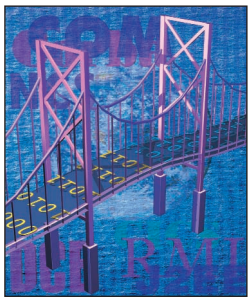

Chain of Responsibility

Steve Vinoski • *IONA Technologies* • *vinoski@ieee.org*

Among the design patterns that recur most often in middleware, such as Wrapper-Façade[1] and Strategy,[2] my favorite is the Chain of Responsibility pattern.[2] In object-oriented terms, the CoR pattern aims to decouple a caller from its target object, and it accomplishes this by interposing a chain of objects between them. This arrangement lets each object in the chain act on a request as it flows from the caller to the target. The pattern evokes images of time-honored approaches such as assembly lines and division of labor.

Researchers have successfully applied the CoR pattern to both operating systems and middleware, enabling flexible and efficient communication bindings between an application and its target. For example, Dennis Ritchie designed his Unix I/O *streams* approach[3] to eliminate unnecessary coupling between orthogonal functions sitting between the user application and the I/O device, and to let developers add new device drivers, especially network drivers, without duplicating protocol code or using a different application model for each new device. Ritchie's design uses separate modules that are linked together to form queues for reading and writing.

In the distributed systems arena, Marc Shapiro based his flexible bindings[4] on creating *stub-scion pair* (SSP)[5] chains to bind a sender to its target. To support bindings that allow uniform transparent access for local, remote, fragmented, or mobile objects, each link in the SSP chain provides a level of indirection allowing for the encapsulation of certain capabilities, such as marshaling, location, or forwarding. Such a chain can be of arbitrary length, but neither its length nor its composition show through to the application.

A common application of the CoR pattern in distributed systems is the chaining of stubs and skeletons between clients and servers in remote procedure call (RPC) systems. In this column, I detail other applications of the CoR pattern that I've seen or used in my own middleware development.

Filters

The first application of the CoR pattern in a middleware product that I recall was in support of the filter feature of IONA Technologies' original Orbix product (I am currently employed by IONA, but wasn't at the time), one of the first successful Corba ORB implementations on the market. Orbix filters let an application insert its own code to intercept a request or reply at certain points along its trip from the caller to the target object and back. For example, a caller could insert a *premarshal filter* to add extra data, such as a caller identifier, into a request before marshaling the identifier for transmission to the target. Alternatively, a caller could insert a *postmarshal filter* into the request path to intercept and encrypt each request before sending it over the network connection to the target. Server objects not only had the same pre- and postmarshal filtering capabilities, but could also employ *thread filters* to intercept requests and dispatch them to different threads depending on the best threading model for the server application.[6]

Filters contributed to Orbix's original success in the marketplace because they provided simple yet useful application flexibility. Filters let applications both effectively extend the ORB's capabilities independent of the vendor and separate infrastructure functionality, such as encryption and multithreaded dispatching, from the business logic functionality implemented in the Corba objects. Keeping such functionality separate facilitates its reuse in other applications.

Pluggable Transports and Protocols

Middleware typically gives applications abstractions for request invocation or message passing. Having these abstractions in place not only makes it easier to isolate applications from the underlying communication protocol details, but it also lets you plug in different protocols and transports

 1089-7801/02/$17.00 ©2002 IEEE

below the application without changing the application itself.

Both Ritchie and Shapiro aimed to provide binding transparency to the application. Ritchie sought to save the application from having to adapt the details of its interactions with the underlying I/O system to its targeted device. Similarly, Shapiro's bindings isolated the application from the details of the target object's location and implementation. Both efforts let applications use alternative communications mechanisms or devices to transparently reach and interact with the target service.

Many middleware systems support pluggable transports and protocols. This support comes in several forms.

- *Compile-time extensibility*. Some extensible networking class frameworks allow different protocol handlers or transports to be compiled into an application. Memory-restricted systems that don't need link-time or runtime flexibility typically use this approach, which normally allows only a single transport or protocol per compiled application.
- *Link-time extensibility*. Some middleware frameworks allow applications to link against multiple transport or protocol libraries, making them available at runtime.
- *Runtime extensibility*. Some middleware systems use dynamic loading to load transport or protocol libraries on demand. This approach is similar to link-time extensibility, except the systems do not load transports and protocols unless and until the application attempts to contact a target that requires them.

Systems that are extensible at link-time and runtime vary in how they support concurrent multiple transports. In some systems, multithreading isolates each transport into its own listening thread. In others, all transports fit into a single shared listening loop, perhaps by adding the file descriptors for their listening ports to a common file descriptor set passed into the `select` system call.

Interceptors

The filters and pluggable transports and protocols described above are instances of *interceptors*. An interceptor is generally one link in a chain of responsibility, processing messages, requests, and replies as they move up and down the chain. Interceptors can examine, modify, and augment each message, and thus influence the message data or destination.

In Corba, both security and distributed transactions rely on interceptors for parts of their functionality. For example, a client-side transaction interceptor adds a transaction identifier to the service context portion of a general inter-ORB protocol (GIOP) request. A matching server-side interceptor uses this identifier to determine the distributed transaction the request belongs to. Similarly, security interceptors can encrypt a message before it's written to the client's network and decrypt it when it arrives at the server.

If a Corba client using an interceptor-based ORB and operating in a transaction context invokes a request requiring encryption, the following processing steps occur:

1. The client invokes the desired request by calling a function on its local proxy object.
2. The proxy turns the function call into a request to be passed down the binding chain.
3. The GIOP interceptor creates a header for the request message.
4. The chain's transaction interceptor marshals the transaction ID into a service context in the request message.
5. The GIOP interceptor finishes marshaling the request header and request body into the request message.
6. The encryption interceptor encrypts the marshaled message buffer.
7. The Internet inter-ORB protocol (IIOP) interceptor sends the encrypted message buffer over a TCP connection to the server.
8. The server's IIOP interceptor, listening on the network, receives the encrypted request message.
9. The encryption interceptor decrypts the message.
10. The GIOP interceptor demarshals the message header.
11. The transaction interceptor demarshals the transaction ID service context in the request header, creating a transaction context in which the invoked function can run.
12. The GIOP interceptor demarshals the remaining request header and request body.
13. The object's skeleton calls the target function on the object's servant, passing all the demarshaled arguments.
14. The reply travels back along the same chain of interceptors, but without a transaction ID.
15. The client's proxy function returns any output arguments and return value to the client application.

Figure 1 (next page) illustrates these steps.

Note the subtle difference between security and transaction interceptors. The encryption interceptor operates on marshaled message buffers, whereas the transaction ID interceptor operates on messages that have either not yet been marshaled (on the client side) or have already been demarshaled (on the server side). This distinction shows that not all interceptors provide the same interface or adhere to exactly the same request-handling semantics. In Corba terminology, request-level interceptors operate on ORB-internal request objects, and message-level interceptors operate directly on marshaled message buffers. You could implement an interceptor that logs requests and replies passing in and out of a server, for example, as a request-level interceptor, but you would implement an interceptor that compresses messages to be sent and decompresses all received messages as a message-level

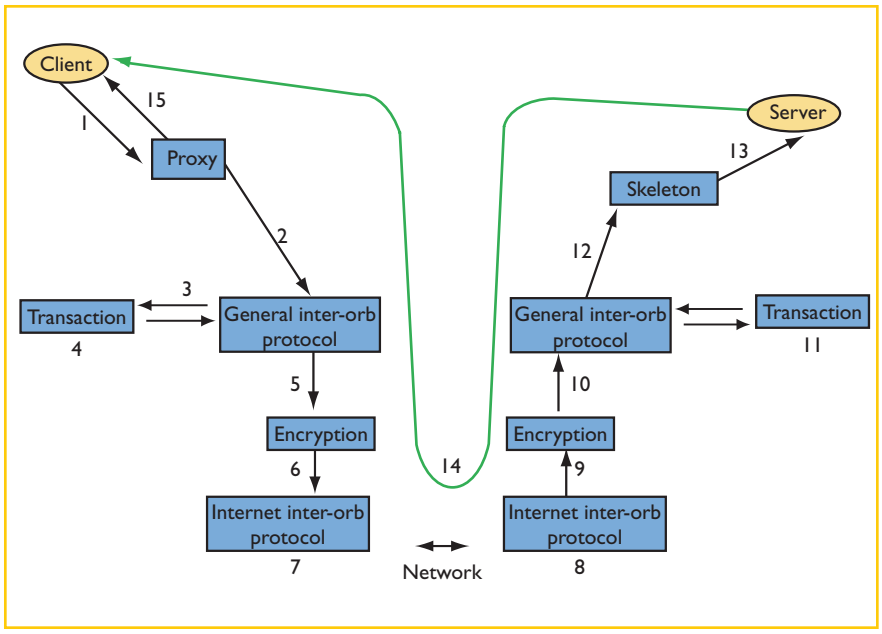

Figure 1. Request flow through an interceptor chain.

interceptor.

Both Corba and J2EE provide application-visible support for interceptors. Using *portable interceptors,* a recent addition to the Corba specification, applications can write portable request-level interceptors. (The portable interceptor specification does not address message-level interceptors, however, due to performance and complexity concerns.) Version 2.3 of the J2EE servlet specification adds filters that let applications intercept and process servlet requests and replies independent of the logic implemented in the servlet itself. Unfortunately, though, interceptors have not made their way into other relevant Java standards, such as the failed Java specification request (JSR) to introduce Orbix-like filters into Java remote method invocation (RMI).

Containers and Object Adapters

The various types of interceptors I've described add flexibility to the middleware infrastructure, but only to a degree. What they do not address is flexible application programming models. Regardless, this is a shortcoming of most middleware designs and implementations, rather than a problem inherent in the CoR pattern.

Containers and object adapters also participate in binding chains, and thus are targets for the CoR pattern. Generally, containers and object adapters host entities that implement business logic, supplying common services in areas such as activation, security, and transactions to those entities. When a request arrives, a container or object adapter must locate the target entity and pass it the request for processing. Containers (the J2EE Enterprise JavaBeans [EJB] container and servlet container, for example) and object adapters (such as the Corba Portable Object Adapter [POA]), fit naturally into chains of interceptors, converting internal request objects into programming language function calls on an artifact (usually an object) used to implement the target entity.

The focus of my work over the past six years has been IONA's Adaptive Runtime Technology, a flexible and efficient framework that supports our various middleware products. ART uses the CoR pattern heavily, applying it not only to transports, protocols, and general interceptors, but also to containers and object adapters. You can configure ART to support different transports, protocols, and interceptors, as well as different containers, including a Corba POA, J2EE EJB and servlet containers, and a Web services container, even concurrently. Given that we originally designed ART to support only Corba, it's surprising and gratifying to know that through the CoR pattern it can also support other programming model alternatives.

Performance

Applying interceptors and the CoR pattern lets you separate orthogonal functions into reusable units and avoid creating fragile and inflexible monolithic subsystems. Rather than building a logging function directly into an IIOP implementation, for example, you could easily build a Corba portable interceptor to handle logging and apply it equally well in a binding that uses an alternative, shared-memory based transport.

Unfortunately, as most of us know, we usually gain flexibility at the cost of reduced performance. My own experience with ART and its heavy reliance on interceptors and the CoR pattern shows, however, that such designs need not be any slower than their monolithic counterparts. A well-designed interceptor chain adds only the overhead of the few function calls needed to pass requests, replies, and messages up and down the chain. As with all distributed middleware, the real key to performance is to avoid expensive activities such as buffer copying, heap memory allocation, operating system kernel calls, and thread context switching in the request path. Nothing inherent in interceptors or the CoR pattern requires these expensive items.

One last benefit of the CoR pattern is that its avoidance of monolithic subsystems makes for easier software maintenance. For example, in a system supporting runtime loading and unloading of pluggable interceptors, you could fix a bug in an interceptor in a binary-compatible fashion such that a running application could unload the old buggy version and load the new fixed version. Generally, bugs

are easier to locate and fix when you avoid monolithic software, and even the inevitable "feature creep" becomes easier to deal with because individual features can be isolated to the appropriate interceptor. Aspect-oriented programming (AOP)[7] takes a novel approach to these types of development and maintenance problems by using interceptors to solve common problems that cut across multiple software functions, such as logging and error handling, without requiring code duplication.

An interesting result of applying the CoR pattern is that when your code invokes a request or operation, the presence of the interceptor chain means that you're not precisely sure of everything your code is invoking, or what it's communicating to what it invokes. If you're new to CoR, this may seem troubling at first, but it need not be an issue as long as the contract, semantics, and quality you expect from the service you're invoking are fulfilled. Such service abstractions are obviously at the heart of service-oriented architectures (SOAs), which are critically important for middleware-based systems.

References

1. D. C. Schmidt et al., *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects*, vol. 2, John Wiley & Sons, New York, 2000.
2. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*., Addison-Wesley, Reading, Mass., 1995.
3. D.M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical J.*, vol. 63, no. 8, Oct. 1984; available at http://cm.bell-labs.com/cm/cs/who/dmr/st.html.
4. M. Shapiro, "Flexible Bindings for Fine-Grain, Distributed Objects," Rapport de Recherche INRIA 2007, August 1993, available at www-sor.inria.fr/publi/FLEX_rr2007.html.
5. M. Shapiro, P. Dickman, and D. Plainfosse. "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection," tech. report 1799, INRIA, Rocquencourt, France, Nov. 1992.
6. D.C. Schmidt and S. Vinoski. "Comparing Alternative Programming Techniques for Multi-threaded Servers," *SIGS C++ Report*, vol. 8, no. 2, Feb. 1996, available at www.iona.com/hyplan/vinoski/col5.pdf.
7. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming* (ECOOP 97), *Lecture Notes in Computer Science* 1241, Springer-Verlag, Berlin, 1997, pp. 220–242; also available at www.parc.xerox.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf.

Steve Vinoski is vice president of platform technologies and chief architect for IONA Technologies. He is coauthor of *Advanced CORBA Programming with C++* (Addison Wesley Longman, 1999). Vinoski serves as IONA's alternate representative to the W3C's Web Services Architecture working group.