Managing Complexity: Middleware Explained

Andrew T. Campbell, Geoff Coulson, and Michael E. Kounavis

Middleware lets you manage the complexity and heterogeneity of distributed-computing environments. Here's an overview of the middleware world.

Ask a network designer what middleware is, and he'll characterize it as part of the application. Ask an application designer what it is, and she'll say it's part of the OS. Which one is it?

Traditionally, most definitions seeking to characterize middleware suggest that it is the software that facilitates remote database access and systems transactions. More recently, the term has come to be associated—somewhat limitingly—with distributed platforms like the Open Group's Distributed Computing Environment (DCE) and the Object Management Group's Common Object Request Broker Architecture (CORBA). And some have loosely applied it to systems as diverse as workflow support environments and even to the Web itself.

We believe the essential role of middleware is to manage the complexity and heterogeneity of distributed infrastructures and thereby provide a simpler programming environment for distributed application developers. It is therefore most useful to define middleware as any software layer that is placed above the distributed system's infrastructure—the network OS and APIs—and below the application layer.

In the distributed computing model, your enterprise moves applications and data to where they can operate most efficiently: to desktop workstations and to LAN, Web, or remote servers. But dealing with the different protocols and interfaces in a distributed environment can be a nightmare.

For instance, to provide basic communication services, programming languages support sockets, which are end-points in two-way communication links between two programs running on a network. Sockets require client and server to engage in application-level protocols to encode and decode messages. Designing such protocols is cumbersome and can be prone to error.

Middleware is basically an alternative to sockets; it abstracts the communication interface to the level of method invocations. Instead of working directly with sockets, you call a method—rather, you have the *illusion* of calling a method—on a local object. The arguments of the call are in fact packaged up by your middleware and shipped off to the call's remote target. In this case, as in others, middleware provides the isolating layer of software that shields you from the complexities of a heterogeneous environment.

MIDDLEWARE FOR THE MASSES

There are a number of distributed object platforms that have recently become quite popular. These platforms extend earlier distributed systems technologies—like the Open Group's DCE, Sun's Remote Procedure Call (RPC) interface, and Novell's Netware—that weren't object-based. The characteristics of the new platforms include

Inside

1520-9202/99/$10.00 © 1999 IEEE

- masking heterogeneity in the underlying infrastructure by cloaking system specifics;
- permitting heterogeneity at the application level by allowing the various components of the distributed application to be written in any suitable language;
- providing structure for distributed components by adopting object-oriented principles;
- offering invisible behind-the-scenes distribution as well as the ability to know what's happening behind the scenes; and
- providing general-purpose distributed services that aid application development and deployment.

Prime examples of such platforms include OMG's CORBA, Microsoft's Distributed Component Object Model (DCOM), and Sun's Java Remote Method Invocation (RMI) system.

OMG's CORBA

Of the three major middleware technologies, CORBA is the most comprehensive in scope, largely because OMG explicitly addresses both the general- and vertical-market aspects of middleware's potential uses. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who designed them.

The Object Request Broker (ORB) is the CORBA middleware that establishes the client-server relationships. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request. The client does not have to be aware of where the object is located, what language it's written in, or what OS it uses. By intercepting these requests, ORBs can provide interoperability between applications on different machines in heterogeneous distributed environments and can seamlessly connect multiple object systems.

In fielding typical client-server applications, developers use their own design or a recognized standard to define the protocol to be used between devices. How they define the protocol depends on the implementation language, network transport, and a dozen other factors. ORBs simplify this process. With an ORB, the protocol is defined through the application interfaces via a single implementation of a language-independent specification called the Interface Definition Language (IDL). (See the sidebar "A Technical Look at CORBA in Action.")

ORBs let you choose the most appropriate operating system, execution environment, and even language to use for each component under construction. And they allow you to integrate existing components. In an ORB-based middleware system, developers simply model the legacy component using the same IDL they use for creating new

A Technical Look at CORBA in Action

We can best see how CORBA operates by examining the sequence of actions that take place when a client application invokes a method in a remote object. The sequence of actions begins when the client application somehow obtains an Interoperable Object Reference (IOR)—perhaps from a name server or as the return value of a previous remote invocation. The client binds to this IOR and, as a result, is given access to a *stub*—a small software routine—through which it can invoke the remote object associated with the IOR.

The client application sees the stubs as proxies for remote objects. In other words, they have a language-specific interface that corresponds to the IDL from which they were generated. The function of stubs is to map—the term in CORBA is *marshall*—the arguments and return values of method calls into and out of communications buffers, which can be sent across the network by the Generic InterORB Protocol (GIOP).

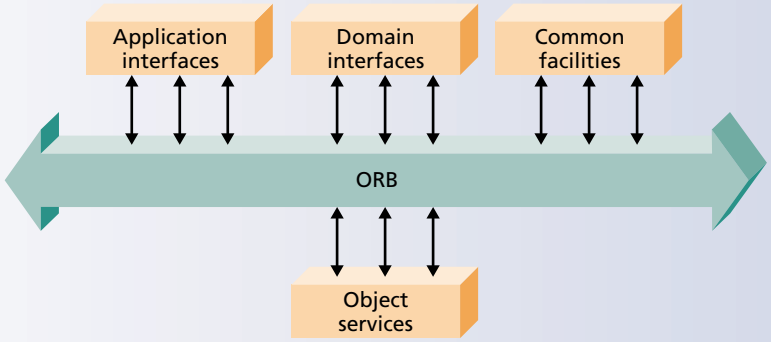
On the server side, the implementer of an IDL provides a language-specific implementation of the interface and registers instances of this implementation with the ORB so that their presence can be advertised. A standard interface—called the Portable Object Adapter (POA)—provides object implementations with a standard environment.

The server implementation is interfaced to the underlying ORB by skeletons in much the same way as clients are interfaced to the ORB by stubs. That is, skeletons are responsible for *unmarshalling* requests, passing the resulting arguments to the target object implementation and marshalling any results into GIOP reply messages.

This description assumes that applications have their stubs and skeletons prelinked with their executables, which is the usual implementation provided by most ORBs. There is, however, an alternative in the shape of dynamic invocation and dynamic skeleton interfaces.

Dynamic interfaces are useful for applications such as browsers and databases in which it would be impossible to link all possible stub and skeleton classes statically. The downside of this feature is that each invocation must be preceded by long and laborious sequences of code that are required to build the appropriate request.

Figure 1. OMG's Object Management Architecture (OMA) reference model, in which CORBA figures prominently as ORB implementations.



Object services are domain-independent interfaces used by distributed object applications. For example, a naming service would allow clients to find objects based on names.

Application interfaces are developed specifically for a given application. Because they are application-specific, and because OMG does not develop applications, these interfaces are not standardized.

Domain interfaces play roles similar to Object Services and Common Facilities but are oriented toward specific application domains. One of the first domain interfaces was for the manufacturing industry.

Common facilities are oriented toward end-user applications. An example of such a facility is one that links a spreadsheet object into a report document.

objects, then write *wrapper* code that translates between the standardized bus and the legacy interfaces.

CORBA itself is merely one component in OMG's Object Management Architecture (OMA). As Figure 1 illustrates, the OMA specifies a range of architectural entities surrounding the core ORB, which is CORBA proper.

Microsoft's DCOM

DCOM is Microsoft's proprietary distributed object technology. DCOM builds on the earlier Component Object Model (COM) architecture that provides a framework for application interoperation within a Windows environment.

A client that needs to communicate with a component in another process cannot call the component directly, but has to use some form of interprocess communication provided by the OS. As Figure 2 illustrates, DCOM provides this communication transparently by intercepting calls from the client and forwarding them to the component in another process.

Functionally and technologically, DCOM is similar to CORBA. For example, both define an IDL and both provide support services such as persistence, security, and transactions. But there are significant differences, the foremost of which is that CORBA is an open specification, while DCOM is not.

Being a closed specification maintained by only one company has one major advantage: DCOM has the potential to evolve at a much faster rate than CORBA because there are no time-consuming politics involved in generating the next version of the specification. But having many interested parties working on CORBA means it can be deployed far more widely than DCOM.

Currently, CORBA runs on most current OS environments, while DCOM is deployed almost exclusively in the Windows environment. Furthermore, porting DCOM to a wider range of environments is problematic precisely due to the fact that there is no real specification other than the Windows implementation itself.

At the present time, it is still difficult to say which of these two significant standards will prevail. CORBA entered the arena first and has had a significant head start. But because it is included with every copy of Windows NT, DCOM is rapidly catching up and—if and when it becomes widely available on a range of platforms—should prove a tough competitor.

Sun's Java RMI

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. Sun's Java RMI offers a solution that enables communication between different program-level objects residing in different address spaces. In such systems, a local surrogate—a *stub*—object manages the invocation on a remote object.

Sun designed RMI to operate in Java. While other RMI systems can be adapted to handle Java objects, these systems fall short of seamless integration because of their interoperability requirements with other languages. For example, CORBA presumes a heterogeneous, multilanguage environment and thus must have a language-neutral object model. In contrast, the Java RMI system assumes the homogeneous environment of the Java virtual machine, which means the system can take advantage

of the Java object model whenever possible, but also means you're largely confined to Java.

As Figure 3 illustrates, RMI consists of three layers:

- the stub/skeleton layer,
- the remote reference layer, and
- the transport layer.

The boundary at each layer is defined by a specific interface and protocol; each layer, therefore, is independent of the next and can be replaced by an alternate implementation without affecting the other layers in the system. For example, the current transport implementation is TCP-based (using Java sockets), but a transport based on UDP could be substituted.

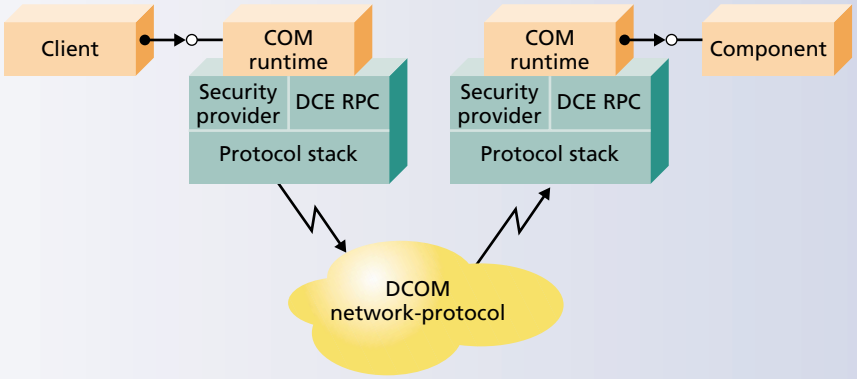
Despite this kind of flexibility, RMI remains a language-specific middleware system. This can be a serious drawback because it requires that all parts of an RMI-based distributed application must be written in a single language, which means that non-Java legacy components cannot be included in an RMI-based application. Nor can real-time components that need to be written in a specialized language. Because of this, it seems unlikely that RMI could ever be a major contender in the middleware wars against CORBA and DCOM.

OPEN IMPLEMENTATIONS

The success of the middleware concept has naturally lead to the desire to deploy the technology in more diverse and demanding application areas than in traditional, networked environments. For example, there are a number of middleware implementations in areas as diverse as multimedia, real-time, and embedded systems, handheld devices, and even mobile networking environments.

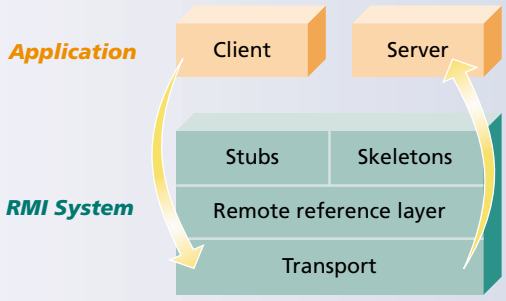
The key to supporting such a wide range of application areas is configurability, which means breaking with the traditional view of middleware as an unalterable black box that has a fixed set of ORB services and a fixed per-platform implementation. Instead, these environments require tailored middleware

Figure 2. Microsoft's DCOM architecture, a proprietary middleware technology similar to CORBA.



In the DCOM model, the Component Object Model (COM) provides object-oriented services to clients and components and uses a distributed computing environment (DCE) remote procedure call (RPC)—and whatever security provider is being used—to generate standard network packets that conform to the DCOM standard.

Figure 3. Sun's RMI system architecture, a Java-based middleware technology.



The application layer sits on top of the RMI system, which consists of the stub/skeleton layer, the remote reference layer, and the transport layer. A remote method invocation from a client to a remote server object travels through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server. A client invoking a method on a remote server object actually uses a stub or proxy for the remote object as a conduit to the remote object. The remote reference layer is responsible for carrying out the semantics of the invocation, and the transport layer is responsible for connection setup, connection management, and keeping track of remote objects residing in the transport's address space.

Middleware Market Will Grow

Growth in the worldwide middleware market is on the fast track. Revenues will increase an astounding 438 percent from $2.2 billion in 1998 to $11.6 billion by 2003. This data comes from International Data Corporation's report "Middleware and Businessware: 1999 Worldwide Markets and Trends" (http://www.idc.com).

According to IDC, throughout 2003, customer demand for middleware technology will shift among market segments. Therefore, the growth rate of the overall market and of each segment are both important indicators of the overall market's potential. IDC believes the fastest-growing segment will be "businessware" systems—middleware specif- ically designed for business transactions—which will earn an annual growth rate of 76.5 percent from 1998 to 2003, which compares with a 40 percent annual growth rate for the entire market.

From 1999 to 2003, IDC expects that event-driven processing and business process automation will increasingly kick in to drive the growth rate. In 1998, middleware and businessware use was highest on the Unix environment, which earned $884.1 million, accounting for 41 percent of the total market's revenues. Unix's numbers are interesting because they indicate the platform is not losing out to NT.

US vendors dominated the overall market, capturing 74 percent of 1998 revenues. The region also consumed almost 49 percent of middleware software. Western Europe was the next largest spender, accounting for more than 32 percent of revenues.

profiles to run on different sorts of platforms for different sorts of applications.

And sometimes the middleware has to be reconfigured while it is running, particularly in the case of mobile environments. In mobile environments, the middleware needs to be highly adaptive internally if it is to shield applications effectively from the effects of unpredictable disconnections and rapid fluctuations in connection quality.

It was problems like this that impelled us to take a look at developing our own open, modular ORB that can be flexibly configured and reconfigured by applications. The ORB we're developing (http://www.comp.lancs.ac.uk) can be augmented with new protocols at runtime to handle new types of multimedia encoding. It can also be used to load a new thread-scheduling algorithm or instantiate a new buffer-management policy.

To ensure that this high degree of configurability can be properly structured and managed, we are employing two key technologies in our CORBA-compatible ORB: component technology and reflective programming. Component technology is already being used to structure applications but has not yet been employed in building middleware itself. Reflective programming—explained in detail at the URL above—complements component technology by allowing better access to the internals of a system. Basically, reflective programming offers metalevel interfaces that are kept strictly separate from the usual base-level interfaces provided by present-day middleware platforms.

NETWORK PROGRAMMING

Middleware can be used in network programming to enable third-party software to run on top of open telecommunications hardware. To program networks, you need to have the ability to access the internal controllers and resources in routers, switches, or base stations.

For example, programmable IP routers need to support open interfaces for managing the router state, including accessing routing tables and controlling forwarding behaviors. In this way, new protocols could be quickly deployed over large, heterogeneous networks consisting of multivendor switching and routing hardware.

Existing network nodes run proprietary OSs that bundle forwarding functions with the algorithms that control and manage networks. Network programmability suggests an alternative approach to open routers, switches, and base stations—abstracting their state, resources, and behavior—so that new network services and architectures can be realized.

So how does network programmability benefit from middleware standards like CORBA? The answer is simple: Middleware standards separate object interfaces from their implementations. Middleware can run on top of routers, switches, and base stations, which allows them to interoperate at the highest possible level. Programmable nodes can expose IDL interfaces, allowing distributed objects to run inside the network. IDL interfaces can be used for binding network algorithms to network resources.

Distributed objects can realize a wide range of diverse network algorithms that include routing in the Internet, connection management in ATM networks, hand-off in wireless networks, and signalling

for resource reservation. Good interfaces at the service level are the macroscopic counterparts of good coding practices at the implementation level.

We have developed open programmable networks over broadband and mobile telecommunications infrastructures (http://www.comet.columbia.edu) and are currently investigating how to automate the deployment and management of network architectures by dynamically dispatching and controlling distributed objects inside the network.

CURRENT DEVELOPMENTS

Middleware is now firmly established as an architectural concept in distributed computing. But the field of distributed computing is far from static, which means that middleware will itself continue to go through many changes.

Perhaps the most significant recent development is components. Component-based middleware evolves beyond object-oriented software: You develop applications by gluing together off-the-shelf components that may be supplied, in binary form, from a range of vendors. This type of development has been strongly influenced by Sun's JavaBeans and by Microsoft's COM and ActiveX technologies.

While component technology appears to be making headway in the middleware community, it is still not altogether easy to apply component concepts to a vendor-neutral and language-independent middleware environment like CORBA. Currently, enabling technologies for components—such as scripting languages and multiple object interfaces—are undergoing development in the CORBA world. CORBA 3.0—the next version of the specification—should make it easier to incorporate component technologies into your middleware environments.

Another issue that will be important to anyone dealing with middleware in the next few years is system integration. For example, Sun is shipping Java 2 with a bundled CORBA implementation, which will permit Java applets running in a Web browser to access network services like legacy databases that have been wrapped as CORBA objects.

Today, perhaps the single most important issue in the middleware world is performance. The current generation of middleware platforms has a deserved reputation for running slowly. However, significant progress is being made in understanding why this is and what we can do about it. We can expect a great deal of performance improvements in the next few years.

Related to performance is of course Quality of Service. Many commercially available ORBs lack interfaces for QoS specification and enforcement. For example, distributed clients cannot explicitly specify priorities characterizing the way their requests are served. And conventional ORB-based systems incur significant throughput and latency overhead. In response to these kinds of limitations, a number of projects—like the Adaptive Communication Environment project (http://www.cs.wustl.edu/~schmidt/TAO.html)—have been initiated to offer high-performance, real-time middleware environments to application developers.

But there are other issues as well. In the near future, watch for improvements in reliability, availability, and scalability. It is likely that such properties will be more effectively supported so that middleware platforms can be profitably deployed in hostile environments involving millions of objects that have to adhere to strict performance requirements. ■

Andrew T. Campbell *is assistant professor in the Electrical Engineering Department at Columbia University. He received a PhD in computer science from Lancaster University. Contact him at campbell@comet.columbia.edu.*

Geof Coulson *is a lecturer in the Computing Department at Lancaster University. He received his PhD in computer science from Lancaster University. Contact him at geoff@comp.lancs.ac.uk.*

Michael E. Kounavis *is a PhD candidate at Columbia University. Contact him at mk@comet.columbia.edu.*