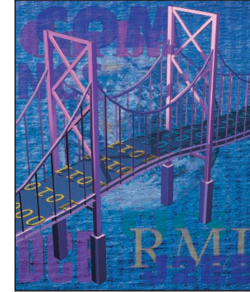
Where is Middleware?



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

This column is all about middleware, and ultimately, middleware is all about integration. Middleware has existed in various forms for many years in systems such as the IBM Customer Information Control System (CICS), numerous message queuing systems such as IBM's MQ Series, the Common Object Request Broker Architecture (Corba), Microsoft's Component Object Model (COM), Java 2 Enterprise Edition (J2EE), and the latest rage, Web services. Virtually every form of application, programming language, operating system, and hardware has been a target of an integration effort involving these middleware systems or their cousins. Middleware is everywhere.

The many reasons we need middleware all boil down to one: As technology continues to evolve at an accelerating rate, nontrivial computing systems will remain diverse and heterogeneous.[1] Computing systems grow over time, which means hardware and applications purchased years ago must work together with those purchased just yesterday. Add factors such as mergers, reorganizations, leadership changes, and e-business into the picture, and the heterogeneity in the overall system rises sharply. As much as we might wish otherwise, the complexity caused by this diversity will not disappear anytime soon, if ever.

We're surrounded by examples of successfully deployed middleware in cost-effective and efficient production computing systems. Nevertheless, it's interesting to note that while middleware eases the diversity and heterogeneity problem, it does not completely solve it. It's ironic that all forms of middleware attempt to reduce complexity by introducing artificial homogeneity into the system, which only delays the inevitable collision between heterogeneous systems. The very ab-stractions and simplifications that allow middleware to address integration issues can also cause problems between middleware systems. After all, middleware systems differ from each other, and system administrators eventually need to integrate two or more systems that use different middleware.

Middleware Classification

We commonly classify (and debate) middleware systems along several dimensions. The following list is not exhaustive, but it still shows that many different types of middleware are possible and necessary to solve all the integration problems we face.

RPC vs. Asynchronous Messaging

At an abstract level, remote procedure calls enable programmers to invoke (possibly remote) services as if they were intra-application procedure calls. Much like function or procedure calls in traditional programming languages, RPCs block the caller's execution while the invoked service carries out the caller's request. In other words, while the called service is busy handling the caller's request, the calling thread stops executing and waits until the request either returns normally or encounters an error such as a timeout condition. Messaging systems, on the other hand, are based on a queuing abstraction in which producers post data to queues for consumers to retrieve and act upon. Messaging systems are typically data- or document-oriented, while RPC systems are procedure- or object-oriented. Middleware applications based on messaging typically have key abstractions and design centers that revolve around information, whereas applications based on RPC center around the objects and functions that provide system services.

Language-Specific vs. Language-Independent

Many middleware systems support the integration of applications written in different programming languages. Corba is probably the best example because it explicitly supports several language mappings for its Interface Definition Language (IDL), which is used for defining contracts for Corba objects. The argument for such middleware is that complex computing systems — perhaps involving

everything from handheld devices to Windows laptops to Unix servers and mainframes — are rarely written in a single language. Nevertheless, many systems, such as J2EE, are based on the simplifying assumption that one programming language is in use. Other single-language distributed systems have also been developed using C++, Modula-3, and Smalltalk.

Proprietary vs. Standards-Based

The argument for middleware standards is that by enabling interoperability and portability between products, they prevent "customer lock-in" and allow users to select middleware based on quality. In the real world, however, this black-and-white standards ideal deteriorates into various shades of gray as some vendors pay lip service to standards while still hooking customers into lock-in.

On the other hand, even those middleware vendors that stay true to standards are typically forced to introduce proprietary features to cover areas the standards do not address. No standard can address all possible problems — not even long-lived standards such as Corba or the collection created under the Java Community Process (www.jcp.org). Furthermore, standardization efforts can be lengthy, bureaucratic, expensive, and political. Proprietary development efforts typically seek to avoid these negative aspects while protecting potentially lucrative secrets from competitors. Very large companies (such as Microsoft) or very small vendors are normally the ones that pursue proprietary efforts. The very large believe their proprietary efforts will eventually become actual or de facto standards due to sheer volume, and the very small often believe their systems will be novel enough to disrupt the market and force others to standardize on their terms.

Embedded vs. Enterprise

Middleware has historically targeted enterprise systems, which typically involve many disparate computing systems, usually including one or more mainframes, across multiple company divisions. Such systems normally seek to improve business process automation. Embedded systems, with their special hardware environments and typically stringent software requirements, were mostly off limits for middleware until recently. Advances in hardware and software have made embedded middleware viable, however, now that developers can create embedded systems using commercial off-the-shelf (COTS) components.[2] Still, embedded middleware faces real-time deadline and predictability constraints that often limit its size and available features.

The most significant challenge...is facilitating Internet-scale application-to-application integration.

Like its embedded counterpart, enterprise middleware also tends to address runtime overhead, but it can typically ignore issues such as memory footprint. Enterprise middleware also tends to be highly dynamically configurable, and it requires runtime management capabilities that allow system operators to monitor for proper operation. Because it needs to integrate a wider array of disparate systems, users often judge enterprise middleware mostly by how easily it allows them to integrate new systems. On the other hand, users typically judge real-time and embedded middleware on memory footprint, performance, and predictability.

Middleware Challenges

Despite the differences among competing systems, all middleware shares some characteristics that make it challenging to build and deploy. Foremost among these is that everyone wants middleware to be as flexible as possible — and to provide high levels of performance. Given that flexibility and performance are often mutually exclusive, achieving acceptable levels of both can be tricky. Power users want hooks into all parts of the middleware so they can take control wherever they deem necessary, but they also want their applications' performance levels to be the same as if they were running directly on the operating system. Too many configuration "knobs," on the other hand, often frighten new users; they want to simply install the middleware and have it work. Finding a suitable balance between these extremes is what keeps middleware architects and designers up at night.

As hardware performance has increased, so has the capacity to tune middleware through configuration, rather than through programming. By separating development from deployment issues, this approach lowers maintenance costs for middleware applications. Our goal is to be able to affect the application's behavior without going back to modify, recompile, retest, and finally redeploy its source code. The deployment descriptor design of Enterprise Java Beans (EJB) is a prime example of this approach. An unfortunate side effect of increasing the configurability of applications, however, is that configuration has become nearly as complicated as programming.

Reducing system complexity is indeed a significant challenge for middleware suppliers. However, contemporary middleware systems address such a wide variety of problems that they themselves have become complicated, in some cases overly so. Clearly, middleware designers need to make sure that their systems are flexible, but they need to establish reasonable default settings that make reconfiguration unnecessary for the most common cases.

Perhaps the most significant challenge facing middleware today is facilitating Internet-scale application-to-application integration. The World Wide Web has shown us how the Internet can be used to support successful consumer-to-business interactions. However, its browser-Web site architecture not only resembles the classic two-tier application model, it

also shares some of the same limitations. While traditional "back office" integration projects, such as encapsulating databases behind application servers, are never simple nor easy, contemporary middleware has made them relatively straightforward. The next goal is to build on that success and extend application integration from the intranet to the Internet.

Web Services

Web services currently present the most promising way to facilitate application-to-application integration on the Internet.[3] Unfortunately, the tremendous amount of hype surrounding Web services makes it difficult to keep their fundamental aspects clear. Part of the appeal is that there is nothing really new about Web services; they simply use the ubiquitous Internet infrastructure to apply proven approaches from mature middleware. Web services are based on the convergence of four technology streams.[4] To use them well, we need not relearn what we've already figured out the hard way.

- *Ubiquitous infrastructure*. Web services operate over the ubiquitous infrastructure of the Web, or more accurately, the Internet. They normally communicate with other applications via Internet protocols such as HTTP or SMTP.
- *Proven approaches*. Web services incorporate fundamental aspects of proven middleware. They encourage the creation of service-oriented architectures, as systems such as Corba have done for the past decade. Unlike most middleware, though, they support both RPC-oriented and message-oriented systems equally well, which makes them extremely flexible.
- *XML*. Web services contracts are defined in XML and communications occur via XML-based messages. XML's flexibility and ubiquity help set Web services apart from previous middleware technologies. Developers can use XML to represent any structured information, and they can create and manipulate it with domain-independent tools. This means that Web services do not require specialized IDLs or specialized compilers or code generators for such languages. This alone is an enormous leap forward.
- *Business standards*. To facilitate integration between trading partners, all cooperating parties must fully understand Web services semantics. Electronic Data Interchange, the e-business standard that's been supporting automated interactions between trading partners for about 20 years, includes standardized business processes and documents. EDI is the forerunner of today's XML-based e-business standards such as ebXML (www.ebxml.org), RosettaNet (www.rosettanet.org), and UCCNet (www.uccnet.org). If Web services are to succeed on an Internet scale, they must incorporate standard business documents and processes to enable correct interactions.

An important strength of Web services is that they intentionally accommodate diversity and heterogeneity, not only in applications, operating systems, and hardware platforms, but also in other middleware systems. One way to think of Web services is as "middleware for middleware." Given that mature systems have been hurt by their inability to incorporate useful features and approaches from each other, it is tremendously powerful that Web services are "middleware agnostic." That means rather than replacing existing middleware solutions, you can just integrate and expand their capabilities via Web services.

Those with a flair for the dramatic like to create artificial technology wars, such as "COM vs. Corba" or "RPC vs. messaging." They also like to make ceremonious declarations like "application servers are dead," but such actions serve only to hurt vendors and users alike. In real life, successful technologies never die; technologies that some view as competitive, such as RPC and messaging, often must be applied together to solve real-world problems. After all, there is no one-size-fits-all solution to the problems middleware addresses.

Only the Beginning

This column is about middleware and how it enables integration. This being my inaugural column in *IEEE Internet Computing*, I've supplied only a brief high-level overview of the issues and challenges we face as middleware continues to mature and evolve. I find the configurability of modern middleware interesting, and I intend to explore it further in future columns. I also dug into Web services a bit because I believe they hold promise as the next generation of successful middleware, and I'll cover them in greater depth as well. If you have issues you'd like me to address in future columns, or comments you'd like to share with me on anything I've written here, please e-mail me.

References

1. S. Vinoski, "Corba: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Comm.*, vol. 35, no. 2, pp. 46-55.
2. R.E. Schantz and D.C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," *Encyclopedia of Software Eng.*, Wiley & Sons, New York, 2001; also available at http://www.cs.wustl.edu/~schmidt/PDF/middleware-chapter.pdf.
3. F. Curbera et. al., "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, March/April 2002, pp. 86-93.
4. S. Vinoski, "The Chief Architect's View: Web services," *IONAsphere*, IONA Technologies, May 2001; available at http://www.iona.com/devcenter/articles/stevev/0501sv.htm.

Steve Vinoski is vice president of platform technologies and chief architect for IONA Technologies. Vinoski helped develop several aspects of the OMG Corba standard, including its C++ Language Mapping and Portable Object Adapter. He is coauthor of *Advanced CORBA Programming with C++* (Addison Wesley Longman, 1999) and has written the "Object Interconnections" column for the *C++ Report* and the *C/C++ Users Journal* with Douglas C. Schmidt since 1995. Vinoski currently serves as IONA's representative to the W3C's Web Services Architecture working group.