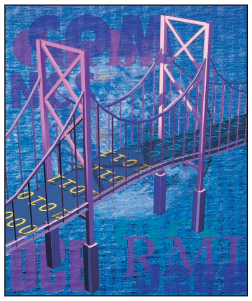

The Performance Presumption

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

Middleware has a variety of qualities, such as size, cost, complexity, flexibility, and performance. For each application, different qualities are more important than others. Choosing the right middleware means first determining the qualities that matter most for your application, and then evaluating different types of middleware to see how well they meet your requirements.

Sounds simple, straightforward, and perhaps even obvious, right? In reality, however, properly evaluating middleware in this fashion can be a nontrivial project that can cost much more time or money than allotted in your budget. It also can require skills and experience that nobody on your staff possesses.

Because of the difficulties involved in thoroughly and accurately evaluating middleware, consumers routinely take shortcuts. Perhaps most common among these is avoiding the issue altogether through supplier loyalty: they simply buy or adopt middleware systems based on recommendations from their current middleware supplier. This approach tends to work reasonably well, because it avoids the switching costs associated with moving to a new supplier.

Obviously, it requires that you select a supplier that can supply viable middleware systems that survive across the life cycles of multiple technologies, rather than going with a "one-trick pony" that is tied to a single technology. It also works only as long as your supplier's middleware systems actually fulfill your requirements. Of course, supplier or brand loyalty is a general market phenomenon — one that's certainly not limited to middleware.

What Can You Measure?

One popular shortcut for technical middleware evaluation is to check only those qualities that are easily measurable. One such quality is performance. Consumers simply evaluate middleware systems by counting the rate at which they can send requests or messages through a given system, and they choose the fastest system.

The most common method for measuring such performance is to write a simple application that sends several thousand messages of a given size through the system, measure the time it takes each message to leave the sender application and arrive at the receiver application, and average the measured times. Taking time measurements at the point of sending or receiving eliminates any application-specific setup or handling code from the samples. The measurement thus reflects only the middleware and everything effectively below it, including the operating system and the network. For round-trip request–response systems, the remote application that's processing the requests is also included in the measurement, as well as the time required for marshaling arguments and return values.

Users Overemphasize Performance

Since the mid 90s, my friend and colleague Douglas C. Schmidt has focused significant effort on researching and analyzing middleware performance issues. (You can find his informative papers on this topic at www.cs.wustl.edu/~schmidt/research.html.) Doug's work — no doubt influenced by others' earlier work on transaction monitor performance — has had a significant positive effect on middleware performance implementations. Some of his initial publications in this area identified the sources of serious performance bottlenecks in the Corba systems of the day, for example, while some of his later research details design patterns for high-performance middleware systems, especially for real-time computing.

An interesting side effect to work like Doug's is that it has unintentionally led many middleware users to presume that "high performance" is the same as "high quality." I've personally observed this phenomenon at work in numerous customer visits and technical conferences.

For certain applications, such as real-time middleware (in which Doug has focused much of his work) or airline reservation systems that experience enormous numbers of transactions per day, performance is indeed critical. Real-time systems, for example, tend to have strict time budgets and deadlines, so minimizing the total time spent in the middleware maximizes the overall time available to the application. For many general middleware applications, however, performance is not at all critical. Why, then, do so many middleware users focus so intently on performance?

For better or worse, one reason middleware performance gets so much attention — regardless of whether the situation warrants it — is simply that it's relatively easy to measure. As I mentioned, it's simple to write applications that measure the time required for message delivery or round-trip request–response time. Other performance aspects, such as marshaling, are equally easy to measure.

Imagine that your department is on the hook to deliver some sort of middleware-based system, and your manager has asked you to evaluate several middleware packages, document your evaluation procedures, and recommend which package to use. As usual, your deadline to deliver all this is tight — perhaps impossibly so. Under such circumstances, it's relatively straightforward to run performance tests for each package, compare the results, and base your recommendation on the fastest system.

Taking this approach is relatively safe (although not as safe as the brand-loyalty approach). It means you can submit a report that cites numerous research papers on middleware performance and includes lots of numbers and smart-looking graphs — all by simply taking the output of a few straightforward test programs. You might even be able to obtain the test programs free from the Internet.

Although this approach might look impressive on paper and keep your salary coming, it could be entirely meaningless, depending on the nature of your middleware application. It's a lot like going out to purchase a new family sports utility vehicle and coming home with a Porsche 911 Turbo: it doesn't have room to actually seat the family, nor is it capable of carrying any cargo or going off-road, but it is the fastest vehicle you can find.

Suppliers Overemphasize Performance

Unfortunately, middleware suppliers get caught up in the performance madness as well. It's a positive feedback cycle: customers demand performance, and suppliers whose products outperform the others win deals. Such suppliers keep pushing to increase performance, but so do their competitors, who try to boost their own performance to obtain more market share. All new customers hear from suppliers is how their product's performance stacks up against the competition, and everything they read talks about performance, so (surprise, surprise) they demand performance.

Some suppliers even resort to performance trickery to fool prospective customers. I've seen cases where one supplier compared its highly tuned shared-memory transport against competitors' TCP/IP-based transports to show a performance advantage, even though the customers' applications were intended to run across multiple machines in a distributed networked application in which their shared memory transports couldn't be used.

Other suppliers knowingly compare the performance of their messaging systems (capable of sending only untyped sequences of bytes requiring no data marshaling, for example) against RPC or distributed-object systems that do perform marshaling. Under such circumstances the messaging system obviously appears faster, but the suppliers neglect to mention that because their "fast" middleware doesn't support marshaling, the customers' applications must handle those chores themselves. Buyer beware.

Because of the difficulties involved in thoroughly and accurately evaluating middleware, consumers routinely take shortcuts.

What Else Matters?

The presumption that performance is a measure of middleware quality ignores the fact that performance is not the most pressing issue for many middleware applications. Doug and others who have spent countless hours helping us all by carefully characterizing the performance landscape certainly never intended to pretend that only performance matters. Depending on the application, other qualities, such as scalability, flexibility and adaptability, ease of use, tool support, and standards conformance could very well take precedence over performance. Unfortunately, such qualities are not as easy to measure.

Scalability

All middleware suppliers claim their systems are scalable. They're usually not wrong, as most middleware truly is scalable in one way or another. Typically, scalability tests are difficult to write, for at least three reasons.

- They require a deep understanding of the runtime nature of the application they're intended to mimic. Foremost in this understanding

must be knowledge of exactly what must scale. Is it the number of concurrent user sessions, concurrent requests, concurrent transactions, or something else entirely?
- Scalability tests require a deep understanding of the design patterns and implementation details that the actual application will use. Unless you use the exact same design patterns and implementation approaches that you'll use in the real application, your scalability tests might not yield the desired results. They could, in fact, lead you to choose the wrong middleware.
- Evaluating scalability often requires heavy involvement by the middleware supplier because they usually know how to tune their systems to best suit a given application.

An interesting fact about scalability is that it actually does depend on performance.[1] If you expect a server to be able to handle a high number of concurrent requests, for example, the speed with which it processes each request has an impact on scalability. Naturally, the fewer cycles a server needs to handle a request, the more requests it can handle in a given time period.

Of course, high performance alone does not guarantee high scalability. Some server systems, for example, perform well when run as single-threaded but suffer serious performance degradation as soon as we introduce multiple threads of control. This occurs because all the threads are vying for the same resources and so must block and waste cycles attempting to acquire the locks required to use the resources.

This implies that any serious testing of multithreaded middleware scalability and performance must attempt to stress the system so that as many threads as possible execute concurrently. Such testing can quickly crash code that looks completely correct but contains subtle race conditions, and debugging such crashes can stymie all but the very best developers. Indeed, writing bulletproof multithreaded code remains a bit of an art. In fact, if you want to do everything you can to ensure your application's stability, try to make sure your middleware supplier's developers are better at writing multithreaded code than you are.

Flexibility

Evaluating flexibility is also difficult, again, because you don't completely know the degree of flexibility your application requires until it's been deployed and exercised under normal use. Rather than testing for scalability, middleware evaluators tend to look for "flexibility points" in the system, typically in the form of hooks that let them insert custom code deep in the processes and activities the middleware performs. In a previous column, I discussed a prime example of an approach middleware suppliers can take to provide these kinds of hooks: interceptors and the Chain of Responsibility pattern.[2]

One caveat about flexibility is that it's often at odds with performance. If you adopt a middleware package for a certain project because of its flexibility, you might later find that it simply doesn't do the job on a project that requires high performance. New middleware technology tends to behave in this fashion because the focus tends to be on ease of use and simplicity to attract early adopters.

Other Issues

As if evaluating subjective qualities such as configurability, flexibility, and ease of use wasn't hard enough, middleware users' values vary depending on where they tend to focus on the technology-adoption life-cycle curve.[3] Early adopters and visionaries tend to pay attention to the middleware's "coolness" factor more than practical issues such as performance, for example, whereas conservative customers worry more about performance, enterprise-scale feature completeness, cost, and support. Because a large enterprise is likely to have groups that fall all along the technology-adoption curve, it's important to have a good cross-section of users from various groups participate in any flexibility or ease-of-use evaluation intended to help establish a corporate middleware standard.

Evaluating middleware for conformance to various standards is also far from straightforward. Good standards are always evolving, which makes them moving targets for middleware suppliers, middleware users, and developers of conformance test suites.

Because some standards have no conformance tests, suppliers can claim whatever level of conformance they think they can get away with. Other standards, such as J2EE, come complete with extensive tests that can help with ensuring standardization; unfortunately, they can still contain enough loopholes to let problematic implementations pass the test suite and claim conformance. Given that the sheer size of today's middleware standards makes it difficult for users to be familiar with all corners of the specifications, evaluating a middleware system for strict standards conformance can be a time-consuming and costly exercise.

It's all like the old joke where the patient says, "Doctor, it hurts when I do this," to which the doctor replies, "Well, then don't do that." Evaluating middleware based on qualities other than performance is too hard, so we just don't do it. Unfortunately, we're selling ourselves short in numerous ways when we don't.

References

1. D. Bulka and D. Mayhew, *Efficient C++ Performance Programming Techniques*, Addison Wesley Longman, 2000.
2. S. Vinoski, "Toward Integration: Chain of Responsibility," *IEEE Internet Computing*, vol. 6, no. 6, 2002, pp. 80–83.
3. G.A. Moore, *Crossing the Chasm*, HarperCollins, 1999.

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. Contact him at vinoski@ieee.org.