Choosing Middleware

February 26th, 2009

I have been in the middleware game for fifteen years now and, over these years, have had contact with many potential customers. A potential customer, having made the decision to use middleware, is faced with the question "which one?" In an attempt to exercise due diligence and protect their investment, the customer typically launches a middleware evaluation project to find out which middleware "is best."

Inevitably, one prominent aspect of the evaluation is the well-known product comparison matrix: a spreadsheet with one column for each contender and one row for each feature. The cells contain ticks or crosses depending on whether a particular product does or does not support the feature.

While such tick lists can be useful, frequently, I found them to be irrelevant (if not harmful) for two reasons:

- Features that are not required by the customer are irrelevant to the evaluation and should not even appear on the list.
- The product with the most ticks is useless if it lacks the one critical feature that would save the customer a lot of time and money.

As an example, from my CORBA days, I remember customers who had dutifully collected all the headings from the CORBAservices specification and turned them into rows in their table. Never mind that at least half of all the services had never been implemented or, if they were implemented, provided no useful functionality whatsoever. Yet, the absence of a threading model that was critical to the customer was overlooked, with serious financial consequences. And we see similar feature tick lists today, when a customer decides whether or not to use Ice.

It is fairly obvious that what matters is not the *number of features*, but whether *those features that will shorten development and deployment costs the most* are present. What is less obvious is that more features do not necessarily mean "better." For one, more features imply a larger product in terms of code size, memory requirements, and execution speed, and, as a rule, require a steeper learning curve. (Ice is the notable exception here—it has a lot of features *and* is very careful to not make developers pay for what they do not use.)

Another prominent aspect of middleware evaluation is performance. Being faced with the question "which middleware?", the response often is "the one that goes fastest must be best, right?"

At that point, the customer embarks on lengthy (and expensive) benchmarking projects that, more often than not, measure completely irrelevant things. Often this happens because the actual workload that the application will place on the middleware is completely different, so application performance cannot be extrapolated from the benchmark. Or it happens because the hardware, operating system, network, or compiler used for the benchmarks differ in some detail that happens to have large impact on the results. And ironically, just as often, it happens because middleware performance is irrelevant to the application: if the application does not have high performance requirements, there is no need for high-performance middleware, and the benchmarking amounts to wasted effort.

This point is important because performance is only one of many evaluation criteria; there are dozens of other things that should enter the evaluation too. For example, ease of use of the APIs, quality of the documentation, licensing conditions, support quality and timeliness, and learning curve (among many others) are factors that are often side-lined because they are harder to quantify, even though they may well be far more important to the project than performance alone.

Having said this, of course there *are* projects for which performance is paramount. Typically, these projects don't just require high performance, they also require high scalability. After all, it is rare for a small system to need high performance; more often, the high performance requirement comes from the need to run dozens or hundreds of servers in a system that has tens of thousands of concurrent clients.

And here is where the benchmark fallacy may well strike again: a simple benchmark that shows two different middlewares as having comparable performance with a handful of clients may be useless because, once the system has tens of thousands of clients, the performance characteristics can be completely different. Unless the benchmarks are designed and executed very carefully, they easily lead to invalid conclusions.

With the preceding caveats in mind, here at ZeroC, we decided to go ahead and do some benchmarks regardless. The middlewares involved are Ice, WCF, and RMI. You can find the results in Choosing Middleware: Why Performance and Scalability do (and do not) Matter. We put a lot of effort into these benchmarks. The scalability benchmarks in particular tested our patience and ingenuity: benchmarking a server with a handful of clients is easy—benchmarking a server with 80,000 clients is a lot harder, and a surprising number of things can go wrong.

We learned as we went. For example, we had known for quite some time that use of `select` to monitor incoming connections does not scale well on Windows. (No such problem with Linux, where we use `epoll`.) But it was the benchmark that demonstrated how serious the situation really was: 2,700 clients was the best we could do with Ice for C++ on Windows. (Again, no such problem with Ice for Java and Ice for .NET on Windows, which both scale much better.) Needless to say, we are doing something about this and Ice 3.4 will not have this scalability problem with C++ on Windows.

Another lesson we learned was that Java and C# add scalability issues of their own: as process size and level of activity increase, the cost of garbage collection becomes much more noticeable; this issue does not show up in a small-scale benchmark.

And, of course, we found out about the relative performance of Ice, WCF, and RMI, which you can read about in the paper. While we were at it, we added tips for how you can design and implement an Ice application to get the best performance, and we suggest what factors other than performance and scalability you may want to consider before choosing a particular middleware. So, the paper is more than just raw benchmark numbers.

As to which middleware you end up settling on, we hope that the information in the paper will provide some of the many data points you should consider before you make your final choice. Of course, you may decide that Ice is easy to use, well supported, provides excellent documentation and APIs, *and* is the fastest and most scalable middleware. If you do, please let us know—we appreciate being told when we do something right

Cheers,

Michi.