RPC and REST

Dilemma, Disruption, and Displacement

Steve Vinoski • *Verivue*

In the previous four issues, I've explored problems with the remote procedure call (RPC) abstraction and explained how the Representational State Transfer (REST) architectural style is one alternative that can yield a superior approach to building distributed systems. Because RPC is inherently tied to programming language abstractions, my May/June 2008 column also investigated multilingual programming, in which developers choose languages according to how well they actually fit the problem at hand, rather than the typical approach of choosing a popular general-purpose language and bending it to fit the problem. Choosing the right language and teaming it with a network programming style like REST can obviate the need for problematic techniques like RPC, thus letting developers build distributed systems both conveniently and correctly.

Some readers agree with my conjectures and conclusions in the past several columns, and others vehemently oppose them. Although there's really nothing surprising about that, the forces that lead different readers to agree or disagree are quite interesting. To make sense of these forces, we must try to understand how, when, and why different customers adopt different technologies, based on factors that can extend well outside purely technical characteristics. We must also understand how technologies evolve, why certain approaches win out over others even when they appear to be technically disadvantaged, and how we might be able to analyze and even predict how new technologies will perform in the marketplace. Armed with such knowledge and understanding, each of us can even analyze our own tendencies and preferences when it comes to adopting technologies — perhaps gaining a better understanding of why certain approaches appeal to us more than others.

Innovation

Many are familiar with the popular book *The Innovator's Dilemma*,[1] in which author Clayton Christensen provides important insights about the nature of innovation, technological change, and how technology markets work. He gained these insights by studying companies from several disparate industries, including hard-disk manufacturers, businesses involved in making steel, and firms that create and sell mechanical excavators.

With respect to innovation, Christensen explains that there are two kinds of technologies:

- *Sustaining technologies* are essentially improvements to products or approaches that already satisfy customers within a given market. Christensen states that they "improve the performance of established products, along the dimensions of performance that mainstream customers ... have historically valued."
- *Disruptive technologies* are promising approaches that users of the incumbent sustaining technologies in a given market initially perceive as being less capable. Those that are successful eventually evolve to fulfill the needs of customers within that market at a lower cost than the sustaining technologies can deliver — and often with greater capability as well.

The dilemma to which Christensen's book title refers is that the steps that managers must take to ensure their products' success and growth in the marketplace also make it extremely difficult for them to respond to disruptive technological changes that eventually push their products into obsolescence. Consider how a successful product generally evolves:

 1089-7801/08/$25.00 © 2008 IEEE

1. The product addresses the needs of certain customers within a market. The customers are reasonably happy with the product, but they feel that some added or improved features and capabilities would make it even better.
2. To keep the customers happy, the product's manager ensures that the product is enhanced with the requested additions and improvements.
3. The additions and improvements not only help make existing customers happier but also help attract new customers, for whom the cycle begins all over at step 1.

These steps form a loop that repeats throughout a product's life cycle. Although customers have certainly viewed some product versions and releases as poorer than their predecessors — many feel that Windows Vista falls squarely into this category, for example — a competent product manager would never intentionally choose to release a product version that doesn't, at a minimum, meet existing customers' expectations and requirements. The reason, of course, is that the product can't succeed without those customers. In fact, to achieve the growth rates that firms normally seek, products must gradually move "up-market" to be able to command premium prices from the very best customers.

Overshooting Opens the Door

The dilemma presents itself because existing customers want improvements, not setbacks, but disruptive technologies are initially unable to meet those customers' demands. Product managers have little choice but to avoid disruptive innovations and move forward with sustaining technologies to continually improve their products to meet these customers' demands; by doing so, they're more likely to be able to secure the premium prices they seek. Yet, as Christensen so lucidly explains in his follow-on book, *Seeing What's Next*,[2] catering to higher-end customers can lead to products that overshoot a nontrivial segment of other customers within that market — those who don't want to pay a premium, especially for features and capabilities they don't need.

Managers of successful products generally aren't concerned about this because they view such overshot customers as undesirable compared to their up-market clients. However, this leaves the door open for disruptive products to take root. Overshot customers turn to the less expensive and seemingly less capable disruptive technology because it's "good enough" for them — the initial problems inherent in the newer product simply don't get in their way. This allows the disruptive product to begin the three-step cycle described earlier, and its customers start to drive it to improve. As the disruptive product improves, it appeals to more and more customers, thus driving the incumbent product into smaller market segments in which it can still command the premiums needed to maintain revenue and profit. The manager of the incumbent product is therefore essentially unable to respond to the disruption because doing so would mean lower margins, less profit, and unhappy customers — a dilemma indeed.

Product-adoption rates also figure into the overall equation. As a product matures and its adoption rate increases, its market grows until the product becomes mainstream and then capable of demanding premiums from the best customers. Eventually and inevitably, however, the product's adoption rate starts to decrease, thus beginning a downhill slide that can ultimately end in obsolescence. Graphing the adoption rate reveals a bell curve that's better known as the "technology adoption life cycle" made famous by Geoffrey Moore's book *Crossing the Chasm*.[3] Depending on the market, these curves can span anywhere from just a few years to many decades; consider the long life cycle of the land-line telephone, for example.

The technology adoption lifecycle curve helps categorize customer types. Those on the rising (left) side of the curve are early adopters of technology who are willing to try something new and look past its perceived initial shortcomings in the hope that it will provide a competitive advantage. The opposite customer type is found on the descending (right) side of the bell curve, where well-vetted, mature products live until they become obsolete. These conservative customers want nothing to do with new, unproven, risky, and potentially buggy technologies and products. They want something solid and well-proven, and they typically complain loudly when the odd problem crops up, no matter how trivial. In the middle, we find the average customers whose balanced risk/reward ratio leads them to favor products and approaches that the early adopters have already proven to work reasonably well. The average

Not surprisingly, users who favor RPC approaches view RESTful HTTP with suspicion, just as Christensen's theories and empirical evidence predict they would.

customers seek competitive advantage over more conservative adopters, and — at a minimum — they want the products and approaches they use to help them stay even with other similar competitors without incurring too much risk.

RPC Sustains, REST Disrupts

Applying Christensen's insights about innovation and technological change to the approaches, products, and customers in the enterprise integration space can be illuminating. For example, if we go back over the history of RPC-oriented systems that I covered last time, we see the pattern of sustaining innovations moving systems up-market. Early RPC systems were indeed rudimentary. However, they appealed to overshot customers — developers who didn't have the time, knowledge, or skills required to employ the typical techniques for creating networked applications of the day, which generally involved carefully hand-crafting custom network protocols along with the custom code needed to drive them. Even the earliest, buggiest RPC framework of the time was good enough for the small-scale systems of the day.

Soon, though, customers wanted more, and the march of sustaining innovations began: Sun RPC and Apollo NCS, DCE, Corba, RMI, J2EE, SOAP, and WS-*. These approaches are all relatively similar in form and function, but each was perceived in the market largely as an improvement over what had come before it. Firms that created products based on these technologies moved right along with each change, building their next sustaining products on each as it appeared. Frequently, "new" products were simply adaptation layers for existing products. Customers for these products also tended to follow along with these sustaining innovations. From my own experience, for example, customers using WS-* in this decade were those using Corba in the 1990s, and they refused to even consider using WS-* until it integrated relatively cleanly with their Corba systems, like a good sustaining innovation should.

RESTful HTTP, on the other hand, has all the makings of a disruptive technology to the RPC market. As RPC systems moved up-market and gained capabilities and features over time to continue to satisfy the most demanding customers, they overshot more and more potential users who shunned the complexity and cost of such systems. In RESTful HTTP, which was born in the adjacent market of the World Wide Web and is a sustaining technology there, these overshot users are finding an approach that helps them build solutions that are less expensive, simpler to build, and easier to extend and maintain than what RPC approaches can offer. It's precisely these qualities that make RESTful HTTP a disruptive technology in this context.

Not surprisingly, however, users who favor RPC approaches view RESTful HTTP with suspicion, just as Christensen's theories and empirical evidence predict they would. Such users commonly raise arguments along the lines that RESTful HTTP lacks tooling and interface definition languages, or that it works for human-driven browser-based systems but is unsuitable for application-to-application integration, and it can't adequately support distributed transactions. In short, RESTful HTTP doesn't yet appear to be "good enough" for them.

Grading on a Curve

The degree to which incumbent RPC users view RESTful HTTP with skepticism depends directly on how far to the right they lie on the technology-adoption life-cycle curve. In fact, many technical arguments and disagreements result not from purely technological differences but from the participants' very different places in the technology-adoption life cycle. With respect to the enterprise integration space, REST proponents tend to inhabit the early adopter side of the curve, whereas RPC supporters hail from the conservative right side. It's no surprise that the RPC vs. REST argument never seems to die down; the participants have completely different risk–reward ratios and value systems, and thus are unable to find common ground. Of course, within any such disagreement, you'll also find the "can't we all just get along" middle-ground folks who point out that both approaches have merits — they, of course, are the pragmatic majority who populate the middle of the bell curve.

Fight or Flight

Another hallmark of a disruptive technology is that as it becomes "good enough" for more users within a market, it gradually displaces the incumbent sustaining technology, thereby invoking "fight or flight" reactions from those still using the sustaining approaches. Such reactions are evident in the consolidation of vendors in the SOA/WS-* market, such as Oracle's acquisition of BEA and Progress Software's purchase of IONA Technologies, and in the fact that some WS-* frameworks and toolkits have incorporated RESTful HTTP into parts of their systems.

For example, WSO2 uses Atom[4] and AtomPub[5] (both built on RESTful HTTP) within its registry product (www.wso2.com/products/registry/), which is part of a set of open source products based on SOA and WS-*. Somewhat ironically, the registry uses a RESTful approach to handle the publication and lookup of metadata for non-RESTful RPC-oriented Web services. Christensen refers to this approach as "cramming," in which firms try to capitalize on disruptive technologies by incorporating them into sustaining products; it's not an approach he recommends

because "it takes an innovation from a circumstance in which its unique features are valuable to a circumstance in which its unique features are a liability."[2] In this case, the benefits of REST are hidden behind an RPC-oriented API for accessing the registry, and those benefits disappear completely as soon as an application uses the registry to find a non-RESTful service and starts to use it. WSO2's strategy might also be risky because it could drive customers away from the company's other non-RESTful products. It's not hard to imagine registry users finding the approach appealing and realizing that they can use similar techniques to gradually rid themselves of their own complicated, expensive, and brittle WS-* implementations in favor of RESTful HTTP Web services.

It's also interesting to think about how new RPC systems such as Facebook's Thrift, Google's Protocol Buffers, and Cisco's Etch fit into the picture. From the enterprise RPC market perspective, these are purely sustaining innovations, and so they're quite unlikely to make inroads with existing customers who view them as inferior to existing products and systems they already use. However, these systems might well take root by targeting non-users of RPC technology in adjacent markets. For example, given Cisco's typical target market, Etch might take root in the embedded networking device space, which is a very conservative market that has started to trust RPC only within the past few years. Similarly, Thrift and Protocol Buffers might find users among developers who build the back ends of Web-based systems. Developers in this space, who tend to worry quite a bit about performance and scalability, are generally loathe to buy into the complexity and runtime overhead of WS-* approaches, but they'll gladly snap up a lightweight framework from the likes of Google and Facebook, who both make it quite clear that they use their respective frameworks themselves with great success.

Whether RESTful HTTP will continue to displace RPC-oriented systems within the enterprise isn't ultimately just a matter of whether one approach is technically "better" than the other. The technology-adoption life cycle clearly indicates that such evaluations are relative. Technology choice is never black-and-white, and in the big picture, the time we spend arguing for one technology over another based on pure technical merit is, frankly, largely wasted. It ultimately comes down to cost — if RESTful HTTP can indeed yield "good enough" integration solutions that cost less to develop and maintain, it will slowly displace heavier, more costly RPC-oriented approaches in more and more enterprise scenarios. As Christensen, Moore, and others have so clearly explained for us, such changes are inevitable, regardless of any technical arguments sustaining technology fans might try to muster to prevent them.

References

1. C.M. Christensen, *The Innovator's Dilemma*, Harvard Business School Press, 1997.
2. C.M. Christensen, S.D. Anthony, and E.A. Roth, *Seeing What's Next*, Harvard Business School Press, 2004.
3. G.A. Moore, *Crossing the Chasm*, HarperCollins, 1999.
4. M. Nottingham and R. Sayre, *The Atom Syndication Format*, IETF RFC 4287, Dec. 2005; www.ietf.org/rfc/rfc4287.txt.
5. J. Gregorio and B. de hOra, *The Atom Publishing Protocol*, IETF RFC 5023, Oct. 2007; www.ietf.org/rfc/rfc5023.txt.

Steve Vinoski is a member of the technical staff at Verivue in Westford, Mass. He is a senior member of the IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog/ and reach him at vinoski@ieee.org.