New Year's Integration Resolutions

Steve Vinoski • *Verivue*

The arrival of a new year tends to enkindle within many of us a hope that the 12 months to come will be better than the 12 that just ended. Born of this hope are our New Year's resolutions: vows to lose weight, exercise more, stop smoking, or spend less time at the office. Such resolutions are renowned for being broken, of course, but individuals who actually succeed at keeping theirs often end up finding the betterment they seek.

What New Year's resolutions might we make to try to improve our lot in the areas of distributed systems and enterprise integration? Having been involved in these areas for the past two decades, I can assert with certainty that distributed integration projects often encounter tremendous difficulty. Although some of the pain is simply inherent in the problem domain, some is unfortunately the result of architects and developers unwittingly making poor system choices. In this column, I consider some high-level advice that can help keep us on the integration straight and narrow this year.

Know Thy Infrastructure

I hereby resolve to avoid believing that third-party infrastructure can completely solve my integration problems.

From 1991 until early 2007, I worked for two different middleware infrastructure vendors in roles such as architect and chief engineer, and each role involved full-time, hands-on software development. One thing I learned during that time is that, from the vendor perspective, the best customers were always those who behaved more like partners or even fellow developers. They didn't just use the products, they effectively helped build them. They were involved in planning product features, and they helped ensure product quality by proactively participating in alpha and beta product trials. We found that such customers provided what you could describe as *tough love*. Publicly, they'd praise our work and willingly speak to other potential customers about how important our products were to their operations. In private, however, they could be tenacious taskmasters, demanding features or capabilities that were sometimes extremely difficult to design or implement. Interestingly, we could often set our products apart from the competition by fulfilling those difficult demands.

These types of customers benefit greatly from close involvement in the development of the infrastructure products they use. In essence, they get to play a nontrivial part in driving the vendor's in-house developers in a direction likely to yield the infrastructure they need. This lets them keep their own development efforts focused on their own problem domains and yet still be certain that their infrastructure will behave and perform the way they need it to.

Now consider the other extreme: uninvolved customers. Their hope is that by using third-party infrastructure, not only can they save themselves the trouble of developing it, but they can also avoid the need to really understand it, much as many automobile drivers are blissfully unaware of how their vehicles' internals work. Before acquiring the infrastructure software, uninvolved customers might first run a trial application to ensure that the software appears to work as they need it to. Assuming it passes their acceptance test, they decide to use the infrastructure code and expect it to behave as advertised. Aside from defect reports, the trial application is the uninvolved customer's primary and perhaps only interaction with the infrastructure supplier.

The problem with the uninvolved approach is that infrastructure can heavily determine the ease with which your system can integrate with other systems. The infrastructure layer typically involves numerous integration points,

cont. on p. 94

 1089-7801/09/$25.00 © 2009 IEEE

cont. from p. 96
such as application protocols, structures, and formats for marshaled data, event loops, signal handling, configuration files, and services for logging and discovery. Many infrastructures go so far as to provide all these capabilities and more. If you don't understand the infrastructure, then you have no hope of really understanding whether or how you can integrate it with other software.

You could argue that customers who build close relationships with their infrastructure suppliers must be large, well-heeled companies that can afford to invest their own personnel in the relationship. Fortunately, thanks to open source software, this assertion isn't necessarily true. Although there typically isn't a vendor behind an open source system, a community of software users and developers almost certainly exists, and you can contribute to such a community in many ways. For example, to ensure that the system does what you expect it to, at the very least you should try to contribute your acceptance tests to the software's test suite.

Heterogeneity

I hereby resolve to avoid believing my system is homogeneous.

Developers often think they're building a self-contained homogeneous system. They forget or ignore the fact that successful systems tend to live for a long time and that users reuse them outside the context for which the developers had originally designed them. Within those foreign contexts and with the passage of time, programming languages, application protocols, data formats, and distributed services might all be very different than what the supposedly self-contained system expects.

In a self-contained system, developers concern themselves primarily with internal component implementations and how those components interact with each other — they don't consider any factors outside the system. Consequently, developers necessarily consider attributes such as coupling and cohesion with only that internal focus in mind. Thus, coupling between internal components tends to be high because everything is developed together. But when it comes time to integrate this supposedly self-contained system into a larger context involving other independently developed software, all the assumptions of self-containment become readily apparent, and they wind up increasing the difficulty of integrating systems.

Because heterogeneity is inevitable, resign yourself to dealing with it, or better yet, learn how to take advantage of it.

In my experience, systems are rarely homogeneous. The inevitable technology changes stemming from the passage of time are alone more than enough to guarantee heterogeneity. Because heterogeneity is inevitable, resign yourself to dealing with it, or better yet, learn how to take advantage of it.

New Technologies and Approaches

I hereby resolve to keep an open mind about new languages, systems, and approaches that might make my integration problems easier or less expensive to solve.

It's easy to get stuck in a rut. As a developer, you find a technology or approach that seems to work well for one problem, so you stick with it. The better you learn it, the more powerful and capable you feel, so you try to use it for every problem that comes along. Eventually, you wind up doing such a fine job convincing your management of your new favorite approach's effectiveness that they decide to make it the standard way — in some cases, sadly, the only way — to do things at your company. In other words, you've just painted yourself into a corner.

Rather than fighting technological change, which is pointless, make it work for you. Like anything else, vetting new technologies and approaches takes practice — the more you do it, the easier it becomes and the more agile you'll be. If you're not regularly reading and experimenting to keep current with the changes and advances in the realms of distributed systems and integration, then you're likely to find yourself unable to change direction when the situation demands it.

For example, try to make a list of your systems' trouble spots and, as you hear or read about new technologies or approaches, objectively consider whether you can use them to improve those problem areas. If any solution shows promise, use it to build some experiments or prototypes to see if it delivers useful improvements. Keep an open mind, and don't grow personally attached to the technologies you currently employ — doing so will only prevent you from switching to something better when you really need to.

Communicate Effectively

I hereby resolve to use the right communication patterns for the problem at hand.

Unfortunately, some developers do get very attached to certain distributed communication styles and overuse them as a result. For example, I recall a project in which someone decided to allow only asyn-

chronous calls because synchronous calls block the client and tie up a server thread for the call's duration.

One problem with such a rule is that it fails to recognize that certain communications are naturally either synchronous or asynchronous. Service discovery calls, for example, are normally synchronous because the caller needs to know how to reach the service it's searching for before it can proceed. Because the client can't proceed until it receives the service–discovery information, forcing the client to make the call asynchronously will gain nothing. In fact, doing so can make the client more complicated than it needs to be by requiring it to have a server-style architecture for registering message handlers, receiving incoming messages, and dispatching them to the right handlers. It also additionally complicates both client and server by forcing them to be able to properly correlate asynchronous requests with their replies. Additional complexity always negatively impacts integration efforts.

Events and notifications, on the other hand, are naturally asynchronous. For this case, synchronous client polling unnecessarily ties up the client and uses server resources, even without pending events, whereas asynchronous notifications let servers notify clients only when they have events to send. Distributed logging messages are also naturally asynchronous.

Another problem with an all-calls-must-be-asynchronous rule is that it represents a failure to understand appropriate system layering. The underlying distribution infrastructure doesn't need to handle synchronously what appears synchronous to a given application's thread. Don't overcomplicate your application by infusing it with what should be infrastructure-level details.

Use Existing Agreements

I hereby resolve to use an existing agreement wherever possible.

This resolution paraphrases a line from an October 2008 blog entry by Mark Baker of Coactus Consulting (see www.markbaker.ca/blog/2008/10/rim-doesnt-get-the-web). Here, the term *agreement* refers to anything two or more integrated components or applications must agree on in order to successfully interact with each other, such as interfaces, data exchange formats, and application and network protocols.

When you reuse an existing agreement, you increase the chances of being able to integrate with other systems that already understand that agreement. Conversely, inventing a new agreement means that existing systems and components can't participate without modification. If such changes are necessary, then the more distributed a system is, the less likely it is that all the participants can incorporate the required modifications in a synchronized fashion.

Existing agreements often represent proven solutions. In the real world, there's a good chance that someone else has already solved whatever it is you're trying to solve. Some developers have no problem with this idea. In fact, their development efforts consist largely of performing Web searches for existing solutions and copying them. At the other extreme are developers who think their requirements are so unique that a solution couldn't possibly already exist. Both approaches are dangerous. The first is like trying to learn how to use a calculator without learning the underlying mathematical principles — you wind up being able to solve only those problems that exactly fit the sequence of the calculator button presses you've memorized. The second approach is flawed because it means you not only waste time solving problems that someone else has already solved, but there's a good chance your solution might not be as informed or optimized as what's already available.

Understanding an existing agreement's constraints is key to being able to use it. Knowing the constraints not only tells you where you must conform to the agreement but also implies what areas are left unrestricted. The unconstrained areas provide open spaces in which your application or component can reside and perform its operations, whereas the constraints generally dictate how you connect it to the rest of the system.

My favorite example that illustrates the utility of using existing agreements is the Unix command pipeline, where the shell connects the output of one command into the input of another. This simple yet highly effective agreement constrains the file descriptors a command application must use for pipeline I/O, but it doesn't restrict the type of data the application produces or consumes or the application's use of other file descriptors. It also doesn't restrict how you implement the application — it can be a shell script; an interpreted Ruby, Perl, or Python program; or a compiled Java, C++, or C application, for example. Users benefit when applications employ the existing pipeline agreement because it lets them combine applications in unforeseen but helpful ways. Application developers also benefit because they can avoid the complexity and overhead of augmenting their applications with capabilities that other Unix commands already provide.

Like real-life New Year's resolutions, our integration resolutions are easy to make and, unfortunately, just as easy to break. Keep them, and you'll surely save yourself some integration headaches.

Steve Vinoski is a member of the technical staff at Verivue. He's 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 contact him at vinoski@ieee.org.