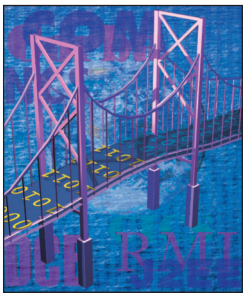
RPC Under Fire



Steve Vinoski • *IONA Technologies*

In 1976, James E. White of the Stanford Research Institute published details in RFC 707 about what would come to be called the remote procedure call (RPC).[1] He described his "procedure call model" as a way of making the networked environment seem completely familiar to application developers, rather than exposing the network directly to them and thus presenting them with a development model so different that they would be scared away from writing distributed programs. Bruce Jay Nelson and Andrew D. Birrell of the Xerox Palo Alto Research Center published their seminal paper on implementing RPC in 1984.[2] Their work provided the cornerstone for the many RPC-based distributed systems that would follow, including Apollo's Network Computing System (NCS), Sun RPC, the Open Software Foundation (OSF) Distributed Computing Environment (DCE), and various object-oriented offspring such as Corba, Microsoft COM, and Java Remote Method Invocation (RMI).

Although many successful distributed systems have been built using RPC, we've known for a while that it's imperfect, even fundamentally flawed, because it ignores the all-too-real possibility of partial failures by attempting to make the network appear to be just another part of the local environment. A partial failure occurs in a distributed system when a remote application or the network itself fails, thereby introducing the need for applications to handle error conditions that simply can't arise with local procedure calls. Lately, however, RPC seems to be taking even more heat than usual, mainly because of continuing advances in Web services and XML-based messaging.

RPC Confusion

First, I want to clear up some confusion that's become a pet peeve of mine: many people incorrectly equate RPC with synchronous request–response messaging. Although RPC requires such messaging capabilities to emulate the call–return semantics of local procedure calls, the inverse is not true; synchronous request–response messaging doesn't automatically imply the use of RPC. After all, an application can perform synchronous messaging by calling low-level general functions such as `send` and `recv`, instead of calling stubs representing specific remote procedures. Request–response is a network-level message exchange pattern, whereas RPC is an application-level abstraction intended, as White explained in RFC 707, to hide the network.

Equating RPC with synchronous messaging means the latter wrongly suffers the same criticisms as RPC. Asynchronous and synchronous messaging each have their places in distributed systems, and neither can completely replace the other. In human communications, for example, we use both asynchronous (email, voice mail, and such) and synchronous messaging (face-to-face conversations and instant messages, for example) every day. If we need information before taking the next step in our daily tasks — performing a Google search, purchasing something online, and so on — we usually choose synchronous messaging. For similar reasons, distributed applications require both synchronous and asynchronous communications as well.

Although RPC is sometimes criticized for its synchronous nature, it generally comes under fire for mixing application-level with message-level issues. This commingling has several knock-on detrimental side effects. What seems like a good, simple idea on the surface — hiding networks and messages behind a more familiar application-development idiom — often causes far more harm than good. Worse still is that it's harm that, even 30 years later, we're still learning about — usually, the hard way.

 1089-7801/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

Critiquing JAX-RPC

In May 2005, Steve Loughran and Edmund Smith of Hewlett-Packard Laboratories published a report that provides a detailed look at the Java API for XML-based RPC (JAX-RPC) and explains why they believe it is fundamentally flawed. The principal problem they describe is the fact that JAX-RPC "relies upon a perfect two-way mapping between XML data and native language objects." They devote much of their report to pointing out why such a mapping is impossible to achieve. In fact, they refer to the mapping as an "object/XML (O/X) mapping" to relate it to the database object/relational (O/R) mapping problem, which is so difficult that decades of work have achieved no highly satisfactory approach. The authors fear that an O/X mapping is comparable in difficulty to the O/R mapping problem, thus implying a concern that our industry could spend years trying to perfect an O/X mapping that simply can't be realized.

One fundamental problem they point out is the sheer difficulty of mapping between XML and Java. XML Schema and Java simply aren't interchangeable, and so not all schema features map cleanly to Java. The authors provide an example of a straightforward schema type used to represent a postal code that, when mapped to Java, becomes a plain Java string, thereby eliminating the characteristics that made it suitable for postal codes in the first place. They also point out that not all XML names can be mapped to Java names.

Another fundamental problem that Loughran and Smith raise is that JAX-RPC's fault-handling approach directly exposes Java faults to other distributed Web services. The problem, of course, is that there's no guarantee that the application on the other end of the wire uses JAX-RPC or is even written in Java. This is an example of a specification feature that was originally intended to make users' lives easier but ends up making them harder in the long run — especially if their applications actually succeed and become legacy applications that are used in large-scale settings involving implementation technologies other than JAX-RPC and Java.

The principal problem they describe is the fact that JAX-RPC 'relies upon a perfect two-way mapping between XML data and native language objects.'

The authors also raise some general RPC concerns, such as coupling and latency. RPC often works well in close-knit systems in which network latencies and the likelihood of network glitches are both low. However, as latencies approach those experienced when sending multimegabyte attachments over the Internet, the RPC model starts to fall apart because of lengthy application blocking times, which can make applications unresponsive while they wait for RPC calls to return. In addition, an application might need to attempt numerous RPC retries because of the increased potential for network failures. With respect to coupling, the authors remind us that one of the primary ideas behind SOAP was to enable loose coupling in distributed systems by avoiding the need for the same heavyweight infrastructure on both ends of the wire. JAX-RPC, they say, simply overlooks this benefit and returns us to requiring the same code in both sender and receiver.

Loughran and Smith refer to the JAX-RPC model of defining distributed interfaces as the contract-last model because service contracts are derived from Java classes. They contrast this with the contract-first model in which service interfaces are defined first, typically with WSDL, and are then implemented using Java, C#, Python, or whatever approach the implementer chooses. The latter is clearly the more desirable avenue as it avoids leaking implementation details through service interfaces,[4] yet it's often ignored because the contract-last approach trades off design and maintenance for easier service development. The authors also state their belief that the complexity of WSDL and XML Schema, combined with the fact that both differ significantly from today's popular programming languages, is what drives developers toward contract-last approaches.

The authors provide a brief look at JAX-RPC version 2.0, now renamed JAX for Web services (JAX-WS; www.jcp.org/en/jsr/detail?id=224). It does provide access to raw XML messages, but it still pushes fundamentally flawed Java RPC constructs as the best way to develop XML-based distributed applications. Loughran and Smith conclude with a description of Alpine, the SOAP stack they're building in an attempt to avoid the many problems their paper raises.

I've covered only a small part of the issues that Loughran and Smith explain, so I urge everyone building Java-based Web services to read the paper to learn all the details. Even if you're currently stuck using JAX-RPC and have no immediate plans to explore alternative approaches, you'll almost certainly learn about issues that you should try to avoid to keep your Web services applications from being too brittle.

Radical Alternatives

Loughran and Smith aren't alone in their criticisms of JAX-RPC. For example, Richard Monson-Haefel, senior analyst for the Burton Group and author of several highly regarded Java books, proclaimed in his weblog that "JAX-RPC is bad, bad, bad!" (http://rmh.blogs.com/weblog/2005/06/jaxrpc_is_bad_b.html). He references the Loughran and Smith paper and congratulates them for so clearly explaining many of the problems with JAX-RPC, but he also admonishes them for basing their ideas for Alpine on ideas similar to those embodied in the SOAP with Attachments API for Java (SAAJ) underpinnings of JAX-RPC. Monson-Haefel instead suggests that a more XML-centric approach is required, such as the result of Microsoft's current work on the Cω (pronounced "C-omega") programming language,[5] which is an extension of C#.

One of Cω's main purposes is to avoid the impedance mismatch inherent in the O/X mapping. The language combines ideas from the relational, object, and XML worlds into its type system and aims to ensure that its type system aligns well with XML Schema's. For example, Cω supports streams, which are ordered collections of zero or more items that serve to support other Cω constructs. The language also provides anonymous structs, which have no explicit type names, can have multiple fields with the same name, and can be assigned to other anonymous structs that have the same field types in the same order. Asking for a named field from an anonymous struct containing multiple fields by that name returns all the fields of that name as a stream. The anonymous struct feature lets Cω accurately model certain forms of XML elements that would be impossible to model in a Java class, for example.

Cω also supports query operators — one form using XPath-based operators and the other using SQL-based operators. The latter seem fairly powerful, given that they can be checked at compile time and provide support for projection, joins, filtering, and other useful data manipulations.

It's difficult to tell, however, whether Cω is really a step in the right direction. I haven't actually used the language, but my own research into it leads me to believe that it's too complicated, in part because of its C language heritage — Cω reuses operators such as * and . (dot) in various contexts that seem hard to keep track of. As we know from C++, layering one language over another (in this case, Cω over C#) often results in difficulties where the underlying language shows through, particularly for those who learned the underlying language first. On the other hand, C++ remains immensely popular despite its complexity, so Cω's utility for overcoming the impedance mismatch between XML and programming languages might outweigh its complexity. Cω isn't the only project trying to address this problem, so even if it fails to catch on, perhaps some other novel programming language will show us the way. Time will tell.

Are you stuck with RPC and the O/X impedance mismatch unless you're willing to go out on the bleeding edge and use alternative languages such as Cω? Not entirely. Between those two extremes, we have approaches such as the aforementioned JAX-WS, with its access to raw XML messages, which at least offers developers the choice of bypassing the O/X layer by using `Dispatch` and `Provider` APIs that allow direct access to messages and message payloads. Other work also continues in related areas. For example, Axis2 (http://ws.apache.org/axis2/rest-ws.html) includes new features that let developers use it in a manner promulgated by advocates of the REpresentational State Transfer (REST) architectural style (see http://en.wikipedia.org/wiki/Representational_State_Transfer). One way that REST differs from RPC is that it works with a fixed set of verbs, such as the HTTP verbs `GET` and `POST`, whereas RPC promotes specialized verbs for each interface, such as a special `getBalance` verb for a bank-account interface. Not surprisingly, this difference tends to result in REST applications having much more of a message focus than RPC applications.

Although the REST vs. RPC debate has raged for several years now, it is perhaps telling that Axis2 and other Web services development kits are now explicitly including support for REST in addition to continuing to support RPC. Are the days of RPC, with its code-generated stubs and less-than-perfect programming language mappings, truly numbered?

References

1. J.E. White, *High-level Framework for Network-based Resource Sharing*, RFC 707, Jan. 1976; www.ietf.org/rfc/rfc707.txt.
2. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, vol. 2, no. 1, Feb. 1984, pp. 39–59.
3. S. Loughran and E. Smith, *Rethinking the Java SOAP Stack*, tech. report HPL-2005-83, Hewlett-Packard Bristol Laboratories, May 2005; www.hpl.hp.com/techreports/2005/HPL-2005-83.html.
4. S. Vinoski, "Web Service Interaction Models, Part 1: Current Practice," *IEEE Internet Computing*, May/June 2002, pp. 89–91.
5. D. Obasanjo, "Introducing Comega," XML.com, 12 Jan. 2005; www.xml.com/pub/a/2005/01/12/comega.html.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski is coauthor of *Advanced Corba Programming with C++* (Addison-Wesley Longman, 1999), and he has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.