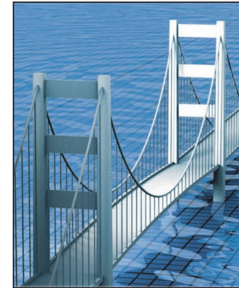
Convenience Over Correctness



Steve Vinoski • *Verivue*

Several of my columns over the years have discussed the remote procedure call (RPC) abstraction. First described in RFC 707,[1] with implementation approaches and details later provided by Andrew Birrell and Bruce Nelson,[2] RPC has influenced distributed systems research and development since the early 1980s. In that decade, distributed systems such as Argus[3] and Emerald[4] explored the possibilities for programming languages themselves to be distributed, thereby building distribution directly into any applications written in those languages. Later in the 1980s and into the 1990s, production RPC systems such as the Apollo Network Computing System (NCS), Sun RPC, and the Open Software Foundation (OSF) Distributed Computing Environment (DCE) provided full RPC capabilities for enterprise developers using general-purpose languages such as C and Pascal. That led to the distributed objects era of the 1990s, in which Corba and Microsoft COM developers primarily used C++. RPC also later influenced Java remote method invocation (RMI), Enterprise Java Beans (EJB), XML-RPC, and SOAP.

Developers have used these technologies and approaches to create countless applications over the years, but the older technologies are all but gone now, and even the newer ones are waning. For example, Corba systems are still around mainly because long-lived telecommunications and systems-management standards build on top of the Corba standard, but most now view it, rightly or wrongly, as legacy technology that's too complicated for new domains and applications.

Despite a highly visible standardization process, significant media coverage, and backing from major vendors such as Microsoft and IBM in the first half of this decade, SOAP seems to have fallen out of favor quite rapidly over the past couple of years. The decline of this, the latest such technology, has left developers who use RPC-oriented systems scrambling to find the next new approach. Some believe they've found it in Facebook's open source Thrift framework, which is billed as a lightweight multilanguage RPC system (see http://developers.facebook.com/thrift/). Others might be awaiting Cisco's open source Etch RPC system, which is slated for initial release in July 2008.

One interesting aspect about the introduction and existence of newer RPC systems is that we've already known for many years that RPC is fundamentally flawed. Distributed systems researchers were well aware of the problems of network partitioning and partial failure by the 1980s. Consider, for example, that Argus included special transaction-oriented features specifically designed to help programmers cope with these issues. In 1994, a small team of developers led by Jim Waldo (now a distinguished engineer at Sun Microsystems Laboratories) published a landmark paper simply entitled, "A Note on Distributed Computing," detailing the fact that local invocations and remote invocations have very different characteristics with respect to latency, memory access, concurrency, and partial failure.[5] That paper remains required reading for any developer who builds distributed systems today.

As if the issues that Waldo and his colleagues described weren't insidious enough, problems with RPC don't stop there. Why, then, do we continue to use RPC-oriented systems when they're fraught with well-known and well-understood problems?

It's Easy

RPC-oriented systems aim to let developers use familiar programming language constructs to invoke remote services, passing requests and data to them and expecting more data in response. On the calling side, developers write ordinary-looking function or method calls,

1089-7801/08/$25.00 © 2008 IEEE

passing instances of data types that the receiver expects. When executed, such a function or method call invokes proxy infrastructure within the calling application that turns the call and its accompanying data into a network message, which it then directs over the network to the intended recipient. On the receiving side, similar infrastructure converts the network message back into a function or method call. Developers write functions or object method implementations that carry out such requests and return the desired output, and they register them with the infrastructure to make them available to remote callers. When an incoming network message arrives, the receiving infrastructure uses an identifier within the message to look up the registered function or object that's supposed to handle the request, invokes it, and then sends its results back in a new network message. The client infrastructure receives this message, converts it back into programming language data type instances, and returns it to the original caller. Because the underlying proxy infrastructure hides all the network operations, the call site within the calling application looks no different than any other ordinary local function or method call.

Unfortunately, this approach is all about developer convenience. It's a classic case of everything looking like a nail because all we have is a hammer. In an object-oriented language such as Java or C++, we represent remote services as objects and call methods or member functions on them. In a procedural language such as C, we represent remote services as functions. We have a general-purpose imperative programming-language hammer, so we treat distributed computing as just another nail to bend to fit the programming models that such languages offer. Despite warnings from Waldo and his colleagues and many others, RPC-oriented systems generally represent remote services using the same abstractions and facilities used to represent local services, thus letting developers stay conveniently within the comfortable confines of their programming languages.

We have a general-purpose imperative programming-language hammer, so we treat distributed computing as just another nail to bend to fit the programming models.

Is developer convenience really more important than all other concerns in this context? Before answering that, let's examine some other problems that RPC brings to the picture.

Impedance Mismatch

RPC systems often employ interface definition languages (IDLs) to define service contracts. Developers detail the functions — or, for distributed objects, the interfaces and their methods — that remote services will offer. Given that methods and functions usually have parameters and return values, a developer will also use the IDL to define specialized data types that serve as parameter types and return types to fully specify a service contract. In all, the effort of defining service contracts is very similar to that of writing functions or objects in actual programming languages, except that IDLs are declarative only. Given RPC's focus on developer convenience, this similarity isn't surprising.

Unfortunately, IDLs usually aren't identical to programming languages. Developers typically use them to write services and clients in several different programming languages, which means their data types are usually abstractions of those found in actual programming languages. Consequently, the IDL types must be mapped to suitable types within each supported programming language. IDL compilers normally perform such mapping by generating the proxy infrastructure code that hides the distributed invocations; the mapping rules are often hard-coded into the IDL compiler. Simple IDL types, such as integers and booleans, often map directly to programming language counterparts, but more complex types — multidimensional arrays, lists, structs, discriminated unions, and object types, for example — are much harder to deal with. Because of the overarching goal of developer convenience, each of these types must be mapped to a programming language type that's as easy and natural as possible for a developer to use within that language. For example, an array might map to a native array type in one language but to a class type in another.

This mapping process is, unfortunately, imperfect. An IDL has to be rich enough to express usable remote services, but if it's too rich, mapping it to different languages becomes problematic. IDL types have to be abstract so that they can apply to multiple languages, but this abstraction means that they often don't map directly or easily into every programming language. For example, Java developers who have used the mapping of Corba IDL to Java tend to view its constructs as non-idiomatic, which isn't too surprising because Corba IDL was initially modeled mostly after C++ and

C. However, a feature added later to Corba IDL — by-value objects — was essentially taken directly from Java and was thus quite difficult to map to other languages. Mappings introduce impedance mismatches between how services are expressed and how services and their clients are realized in actual code. The language mapping is a leaky abstraction; it results in code that looks neither fully natural within the programming language itself, nor exactly like the IDL. Instead, it's a mixture of the two, combining complexity from both, so it's often much less convenient than you'd hope.

Scalability Concerns

The illusion of RPC — the idea that a distributed call can be treated the same as a local call — ignores not only latency and partial failure but also the concerns that spell the difference between a scalable networked system with good performance capabilities and a nonscalable one whose performance characteristics are dictated entirely by the RPC infrastructure. For example, distributed systems typically require intermediaries to perform caching, filtering, monitoring, logging, and handling fan-in and fan-out scenarios. In large-scale systems, these intermediation services are "must haves" that ensure that the system will operate and perform as required. Unfortunately, RPC-oriented calls lack the metadata required to support intermediation because it's simply not a concern for normal local invocations. Some languages try to let developers add that metadata to the system — Java and C# let you attach annotations to classes and methods, for example. Unfortunately, such approaches just add more complexity in trying to maintain the illusion of convenience by, essentially, stepping outside the programming language proper.

Caching is critical to scaling, for example, but nothing about an RPC can indicate whether its results should be cacheable, and if so, the cache validity's duration. Nothing about an RPC lets callers send information along with their requests to let servers return indications that nothing has changed since the last time they called. Such features are not only unnecessary for local calls, they're actually inconvenient. They don't fit the model, so RPC doesn't offer them, even though they're indispensable for large-scale systems.

Representational state transfer (REST), on the other hand, addresses all these concerns and more. It offers clear layering and separation of concerns, and it meets network effects head-on. For example, caching is relatively straightforward with RESTful HTTP because clients can make conditional `GET` requests and servers can specify cache-control headers. HTTP also specifies which of its verbs are idempotent, which helps address partial failure and its resulting indeterminacy issues. RESTful applications are well-equipped to deal with intermediation and loose coupling. Many developers are thus attracted to REST, but unsurprisingly, some try to build programming language frameworks to make it convenient. These frameworks invariably come up short and ignore important REST elements, such as its hypermedia constraint, because those elements don't fit well with typical general-purpose programming language abstractions.

RPC has other problems in the areas of coupling and reuse, but I already covered those in past issues.[6,7]

A Historical Accident?

Does developer convenience really trump correctness, scalability, performance, separation of concerns, extensibility, and accidental complexity? Clearly, the answer is no. We've known about significant problems with RPC for decades; yet, many (including me, until a few years ago) continue to push the RPC abstraction, trying to make it fit distributed applications. What started as a simple developer convenience has evolved into an approach that consists of layer upon layer of leaky abstractions and bandages upon bandages. Simply put, it's time to put the RPC mistake behind us.

I wish the history of distributed computing and programming languages had been different — that the temptation of developer convenience hadn't led us to view distributed computing's necessary complexity as too hard, leaving us to try to replace it with accidental complexity that doesn't really work. For popular imperative languages, using asynchronous messaging, for example, can deeply affect how you write your application — as well it should — but many developers choose to stay away from it because it's inconvenient. Here's what sending an asynchronous message in Erlang looks like:

```
Pid ! Message
```

It says, "send the value of the variable `Message` to the process identified by the variable `Pid`." It neither looks nor acts like a local Erlang function call. Likewise, receiving a message is a separate action that requires calling Erlang's built-in `receive` function, which (among other things) lets developers easily and clearly handle timeouts. Given that these facilities are part of Erlang, they're simple, but unlike RPC, they're not naive.

If history had been different, perhaps those who built the early RPC systems and distributed object systems would have focused on building message queuing systems instead. What if distributed language systems such as Argus and Emerald had focused on making asynchronous messaging available directly to the programmer, like Erlang does? What if, rather than developing RPC

frameworks, Apollo, Sun, and the OSF had instead chosen to ship message queuing frameworks? Perhaps a whole generation of developers would have built their distributed applications such that their code dealt directly with the network and its effects rather than layering their code over leaky RPC abstractions.

Why have there been virtually no freely available language-independent message-queuing systems — perhaps until the recent implementations of the Extensible Messaging and Presence Protocol (XMPP; www.xmpp.org) and the Advanced Message Queuing Protocol (AMQP; www.amqp.org)? For decades, vendors of message-queuing systems have chosen to sell them at high prices, and the market has unfortunately let them do so.

Thankfully, though, things are changing. Many still using RPC in the enterprise are starting to realize they'd be better off with either message queuing or RESTful HTTP, depending on the nature of their applications. The developers of Facebook Thrift and Cisco Etch, as convenient as those systems might be, would have been better off providing an XMPP- or AMQP-based message-queuing system or relying on RESTful HTTP; perhaps both cases are instances of those not knowing history being doomed to repeat it.

It's time for RPC to retire. I won't miss it.

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, 1984, pp. 39–59.
3. B. Liskov, "Distributed Programming with Argus," *Comm. ACM*, vol. 31, no. 3, 1988, pp. 300–312.
4. A. Black et al., "Distribution and Abstract Types in Emerald," *IEEE Trans. Software Eng.*, vol. 13, no. 1, 1987, pp. 65–76.
5. J. Waldo et al., *A Note on Distributed Computing*, tech. report SMLI TR-94-29, Sun Microsystems Laboratories, 1994; www.sunlabs.com/technical-reports/1994/abstract-29.html.
6. S. Vinoski, "Serendipitous Reuse," *IEEE Internet Computing*, Jan./Feb. 2008, pp. 84–87.
7. S. Vinoski, "Demystifying RESTful Data Coupling," *IEEE Internet Computing*, Mar./Apr. 2008, pp. 87–90.

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 his blog at http://steve.vinoski.net/blog/ and contact him at vinoski@ieee.org.

Raise Your Profile
Raise Your Income

A strong majority of managers agree that the CSDP credential:

"validates technical aspects of software development knowledge" **(91%)**

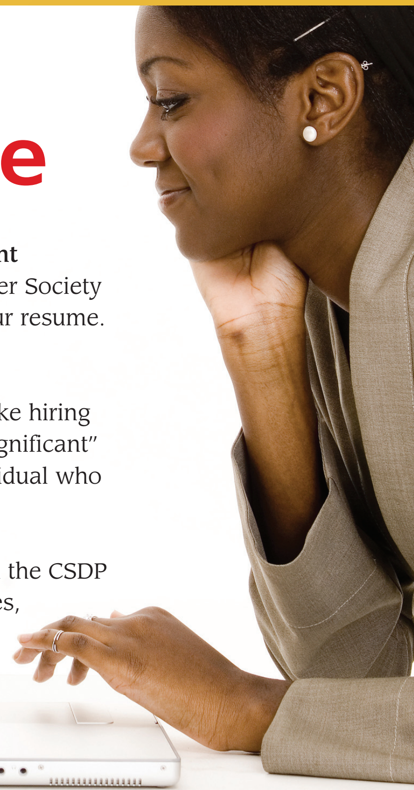
"demonstrates attainment of a professional level of competence by software developers" **(91%)**

"demonstrates a professional commitment" **(96%)**

Obtain the Certified Software Development Professional (CSDP) from the IEEE Computer Society and splash that accomplishment all over your resume. Check out these recent survey results:

72% of hiring officials or those who make hiring recommendations have a "noticeable" or "significant" preference for a CSDP compared to an individual who doesn't possess the credential.

80% of managers value employees with the CSDP credential in ways including job opportunities, interesting work assignments, and salary.



E-mail CSDP@computer.org for information on how the CSDP credential can help boost your career. Make sure to ask about our latest promotions and offerings.