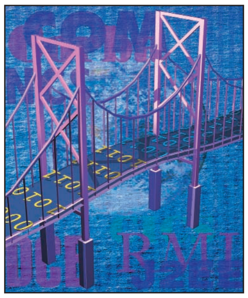
Dark Matter Revisited



Steve Vinoski • *IONA Technologies*

Distributed systems, middleware, and integration are just plain hard. The seemingly intractable difficulties permeating these areas drive our industry to continually seek out easier ways to build such systems. During the past 20 years, we've produced myriad toolkits aimed at simplifying distributed programming and integration, including many proprietary or homegrown-based approaches and many others based on well-known approaches such as Distributed Computing Environment (DCE), Corba, J2EE, and Web services. Regardless of the underlying technology, each approach seems to start out being simple (compared to what went before it); as each matures, however, it seems to wind up just as complicated as the approach it was designed to displace, if not more so.

It's not only distributed computing and integration's inherent difficulties that make these tool-kits complicated. Perhaps ironically, success also makes them more complex. As a toolkit succeeds and grows in popularity, its market grows ever wider, resulting in more and more requirements for it to fulfill. Added requirements typically mean added functionality, which in turn causes the toolkit to grow in size and complexity. Even if the toolkit follows proper architecture and design — so that it grows without turning into a monolithic monstrosity — its sheer size or "surface area" can send potential users scurrying for something simpler.

Dynamic Languages

Naturally, the programming language that a distributed computing toolkit supports has a lot to do with its perceived ease of use. Toolkits written for C++, for example, are often criticized as being too hard to use — mainly due to the perceived difficulty of using the C++ language in general. Those written for Java are deemed somewhat easier to use — though not much, mainly because of the many classes, methods, and packages a developer must typically understand to write any nontrivial application. Interestingly enough, these perceptions appear wholly dependent on the programming languages and are seemingly independent of the particular distributed systems technology underneath the toolkit.

Could it be that these mainstream programming languages are just wrong for many distributed computing and integration applications? Are dynamic programming languages better suited for the job? Having spent quite a bit of my career writing distributed systems in C++, Java, and C, perhaps I ask these questions only because the grass always looks greener on the other side of the fence.

However, I've also spent quite a bit of time writing integration programs in dynamic languages, most often Perl. For example, one of the first things I did when I joined IONA more than seven years ago was to write a Perl script that hid the details of our underlying software configuration management (SCM), allowing new developers to easily build and test the new software we were working on. At the time, I figured the script was just temporary, but we still use it today (and my, how it's grown). Another early task was to develop a Web-based bug-tracking system in Perl, which was integrated with a text-based tracking system stored in our SCM system.

Today, we frequently use WikiWikiWebs[1] for collaboration across our multisite engineering projects, and the implementation we typically use, called MoinMoin (http://moin.sourceforge.net), is written in Python. One of my favorite programs of late, if not of all time, is the SpamBayes spam filter (http://spambayes.sourceforge.net), which is also written in Python and can be integrated into Outlook as a plug-in. The blogging software I use, called MovableType (www.movabletype.org), is written in Perl, as is the spam-prevention application I use with it.

As I noted nearly two years ago in this column, many distributed integration applications, including much of the software running the Web, are based on "middleware dark matter,"[2] which consists of dynamic languages like Python, Perl, and PHP. Not only is this phenomenon still true today but, based on the vast numbers of open-source projects revolving around these languages, it seems to still be rapidly growing.

The power of typical contemporary computers, with multi-GHz CPU speeds and hundreds and thousands of Mbytes of RAM not uncommon, executes applications based on modern dynamic languages quickly and efficiently. I also believe the research and experimentation over the past several years aimed at improving Java virtual machines and bytecode interpretation has drastically improved dynamic languages because, like Java, they tend to be interpreted or bytecode-driven.

Combined with their speed and efficiency, today's dynamic languages' flexibility and ease of use are hard to beat. This appears especially true for distributed systems and integration systems, which seem to undergo changes more frequently than other types of software and thus require rapid application development capabilities. Moreover, dynamic languages' popularity means that you can readily find an already-written open-source dynamic language module or component on the Web to perform almost any task.

In the rest of this column, I survey three dynamic language distributed computing systems whose power and utility might surprise you.

Twisted

Twisted (http://twistedmatrix.com/products/twisted) is an extensive Python-based framework for writing network applications. In days gone by, dynamic-language network-programming toolkits were seldom as full-featured as their mainstream programming language counterparts such as DCE and Corba. In fact, it seemed they were rarely much more than thin layers over sockets. With Twisted, however, this is definitely no longer the case. In fact, even describing Twisted as "extensive" might not do it justice, given the number and variety of features, functions, and applications it provides.

The Twisted framework represents a fairly complete architecture for networked applications. It includes:

Even describing Twisted as "extensive" might not do it justice, given the number and variety of features, functions, and applications it provides.

- a Web server, DNS server, Internet Relay Chat (IRC) server, mail server, and secure shell (SSH);
- enterprise capabilities (including user authentication, a relational database interface, and object-persistence support);
- a distributed-object broker (including communication, serialization, and marshaling support);
- protocol and transport abstractions, with a variety of concrete implementations available underneath (including HTTP, Simple Mail Transfer Protocol (SMTP), IRC, DNS, SSH, Telnet, POP3, TCP, Transport Layer Security (TLS), and UDP;
- event loops that are pluggable and thus replaceable, allowing applications to take advantage of special features available on the underlying platform (such as kqueue[3] on FreeBSD); and
- support for integrating several popular Python GUI toolkits, such as Tkinter (the standard Python GUI; www.python.org/topics/tkinter/) and wxPython (www.wxpython.org/).

Twisted's developers apparently subscribe to the same distributed-services philosophy that I do: services should be strongly separated from the protocols, wire formats, and transports used to communicate with them.[4] This lets applications consume services over multiple protocols, wire formats, and transports, and it also allows services to evolve gracefully and independently of the particular technologies used to communicate with them. Twisted's broad protocol and service coverage makes it relatively simple for developers to provide their services simultaneously over a variety of communication approaches.

The Twisted documentation includes a lengthy tutorial based on a "finger" server (used to find information about other computer users), which shows the server's evolution from something very simple to a full-featured service. The first couple of versions of the finger server essentially do nothing but show the basics of using Twisted's reactor-based event loops. The next versions are augmented to include Twisted's protocol factories and protocols, allowing them to function as actual finger servers. After that, the tutorial shows the use of *deferred objects*, which are used extensively in Twisted's applications to avoid blocking. Given that Twisted is event-based and single-threaded, blocking must be avoided. Deferred objects, which resemble *futures* (objects that encapsulate the retrieval or computation of underlying values), neatly facilitate nonblocking behavior by encapsulating application callbacks that are invoked once data actually becomes available.

The tutorial then proceeds to evolve the finger server into a full-fledged service. It shows multiple versions of the finger server that each obtain their user information from different data sources, such as from local Python data structure, running the `finger` command locally on the same computer and capturing its output, or obtaining the information from a Web page. By encapsulating these implementation details in protocol factories, the developer enables each implementation to reuse the same protocol implementation. The tutorial then extends this notion to show how the finger server can make its information accessible not only over the standard finger protocol on port 79, but also simultaneously via the Web, XML-RPC (remote procedure call; www.xmlrpc.com), and IRC.

To cover all of Twisted's facilities to any depth would require the space of multiple columns, but I hope this brief description is enough to entice you to look into it further.

PEAK

The Python Enterprise Application Kit (http://peak.telecommunity.com), which is currently still under development, is intended to let developers assemble enterprise applications from components. It essentially provides architectural and infrastructure support for Python applications, similar to that which J2EE provides for Java applications. PEAK appears to have descended from Zope (www.zope.org), a popular open-source Web application server written in Python, so it's not unreasonable to expect that it's essentially a next-generation application kit that will incorporate many hard-won lessons from real-world Zope deployments.

PEAK's developers believe their system will be easier, faster, and more scalable than J2EE, which they say is overly large and complex, resulting in resource-intensive implementations; moreover, they state that the Java language isn't particularly suited to rapid application development (and I agree on all counts). The general view of Python, on the other hand, is that it's small, lean, and extraordinarily well-suited to rapid application development. Based on their experiences with Zope and other Python-based enterprise-quality tools, the PEAK developers believe a Python-based approach to component-based applications will result in systems that are simpler, faster, and easier to install, manage, and maintain than anything similar in J2EE.

Today, PEAK appears to focus mostly on allowing developers to create applications from components. For example, it allows components to be named, discovered by name, configured, and bound together into applications. It uses a novel system called *PyProtocols* (http://peak.telecommunity.com/PyProtocols.html) to adapt one interface to another, which is important for component assembly. It even supports transitive adaptation.

PEAK's developers believe their system will be easier, faster, and more scalable than J2EE, which they say is overly large and complex.

In the future, PEAK will support other areas such as storage, transactions, logging, and other typical application features. Given that the current naming, configuration, and binding capabilities require only 4,000 lines of Python, the broader set of features will likely be equally as compact, thus standing a good chance of meeting the PEAK developers' goals of outdoing J2EE in terms of performance and scalability. Stay tuned.

SlimServer

Unlike Twisted or PEAK, SlimServer is not a general-purpose dynamic-language application framework. Rather, it's a network music server written in Perl that supports the Slim Devices Slimp3 and Squeezebox music players (www.slimdevices.com). I own a Squeezebox, which lets me use my home wireless network to access my music collection (stored in my iTunes library on my iMac), and play it over my home audio system. The Squeezebox is a small hardware device that plugs into my home receiver just like any other audio device. I can control the Squeezebox via a standard remote control or by accessing the SlimServer via my Web browser and having it send commands to the Squeezebox.

I've mentioned the SlimServer here not only because I'm a happy Squeezebox customer, but also because:

- SlimServer inexpensively integrates your home computer with your home audio system. A few years ago, it's likely that few would have even considered using Perl to solve this integration problem, and today, more than a few are probably surprised that a Perl application is fast enough to stream music data over an 802.11b network such that the listener experiences no audible dropouts. I've listened to it for hours at a time without any problems.
- SlimServer can be accessed over multiple protocols. For example, as I mentioned earlier, you can control it with a remote control or via a Web browser. The browser window lets you see and control every aspect of the system, including the audio hardware and the SlimServer itself. SlimServer is also accessible programmatically via a command-line interface and a TCP port. For example, sending the string "`title ?`"

followed by a carriage return to SlimServer, either via its standard input or its TCP port, causes it to return the title of the song that's currently playing. It supports several similar commands for controlling and querying it.
- SlimServer is open-source software that is freely available. As with most successful open-source software, there's a community of developers that work on SlimServer. As a result, SlimServer is remarkably bug-free.

I have only one problem with SlimServer. In looking through the source, it appears that SlimServer is built from scratch, in the sense that it seems to use only its own Perl modules to get its work done. I wonder how much cleaner SlimServer might be if it were instead written in Python and used a network programming framework, such as Twisted. If I had more spare time, I might consider developing such a system myself. Oh well, given the SlimServer's price and the fact that it already works so well, I guess I shouldn't complain.

Conclusion

In the past, the only networking support that dynamic languages provided was low-level access to raw sockets, but these examples show that today's dynamic languages are powering frameworks and applications that provide useful high-level distributed computing abstractions without sacrificing efficiency. For a variety of compelling reasons, including their rich features, flexibility, compact source code, ease of development and maintenance, and proven capabilities, these "dark-matter" languages are leading the next wave of mainstream distributed computing and integration applications.

References

1. B. Leuf and W. Cunningham, *The Wiki Way: Collaboration and Sharing on the Internet*, Addison-Wesley, Apr. 2001.
2. S. Vinoski, "Middleware 'Dark Matter'," *IEEE Internet Computing*, vol. 6, no. 5, Sept./Oct. 2002, pp. 92–95.
3. J. Lemon, *Kqueue: A Generic and Scalable Event Notification Facility*, tech. report, FreeBSD Project, 2000; http://people.freebsd.org/~jlemon/papers/kqueue_freenix.pdf.
4. S. Vinoski, "Integration with Web Services," *IEEE Internet Computing*, vol. 7, no. 6, Nov./Dec. 2003, pp. 75–77.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for 16 years. Vinoski is the coauthor of *Advanced Corba Programming with C++* (Addison Wesley Longman, 1999), and he has helped develop middleware standards for the OMG and W3C. Contact him at vinoski@ieee.org.