Distributing Computing #3

Cross-language remote invocation with XML-RPC

David Mertz, Ph.D. <mertz@gnosis.cx>
Gnosis Software, Inc. <http://gnosis.cx/publish/>
June 2002

The first two installments of this *Distributing Computing* column introduced techniques for remote management of program logic and execution resources within the Python programming language. But it is also possible to implement systems that allow remote invocation in a language-independent fashion. The lightest weight of such technique is XML-RPC. Under XML-RPC, invocation requests are encoded in an XML format, and method call results are returned as XML documents. Libraries for using XML-RPC exist for many programming languages. The design of XML-RPC is discussed and contrasted with heavier-weight protocols like SOAP and CORBA. The article is accompanied by source code examples.

About This Series

In the paradigm of stand-alone personal computing, a user's workstation contains a number of *resources* that are used to run an application: disk storage for programs and data; a CPU; volatile memory; a video display monitor; a keyboard and pointing device; perhaps peripheral I/O devices like printers, scanners, sound systems, modems, game inputs, and so on. Even since the 1980s, it has been common for such personal computers to also have network capabilities, but a network card has largely been just another sort of I/O device in the "traditional" paradigm.

"Distributed computing" is a buzz-phrase that has something to do with providing more diverse relationships between computing resources and actual computers. Different resources can enter into different sorts of relationships—some hierarchical, others arranged in lattices, rings and various other topologies. Some of many possible examples are: SANs (storage-area networks) centralize persistent disk resources for a large number of computers; in the opposite direction, file-sharing protocols like Gnutella and Freenet decentralize data storage and its retrieval; the X Window System and VNC (AT&T's Virtual Network Computing) allow display and input devices to connect to physically remote machines; protocols like Linux Beowulf allow many CPUs to share the processing of a complex computation, while projects like SETI@Home (NASA's Search for Extraterrestrial Intelligence), GIMPS (Great Internet Mersenne Prime Search) and various cryptographic "challenges" do the same with much less need for coordination.

A few years ago—mostly before the explosive growth of the Internet—terms used to describe redistribution of resources were “Client/Server” and “N-Tier Architecture” (both focussing more on hierarchical relations). The Internet, the Web, and the public consciousness of both have changed the words, and have also shifted emphasis towards graphs, away from trees.

The protocols and programs that distribute what were basically hardware resources of PC applications make up only part of the distributed computing picture. At a more abstract level some much more interesting things can be distributed: data; information; program logic; “objects;” and, ultimately, responsibilities. DBMS's are a traditional means of centralizing data and structuring its retrieval. In the other direction "Internet" technologies like Usenet/NNTP radically decentralize information storage; other technologies like search engines restructure and recentralize information collections. Program logic describes the actual rules of proscribed computation (various types of RMI and RPC distribute this); objects brokerage protocols like DCOM, CORBA, and SOAP recast the notion of logic into an OOP framework. Of course, even old-style DBMS's with triggers, constraints, and normalizations always carried a certain degree of program logic with them. Of course, all of these abstract resources are at some point stored to disks and tapes, represented in memory, and sent as bitstreams over network wires.

In the end, however, what is ultimately shared between distributed computers are sets of responsibilities. One computer “promises” another that under certain circumstances it will send some bits that meet certain specifications over a channel. These promises or “contracts” are rarely firstly about particular configurations of hardware, but are almost always about satisfying functional requirements of the recipients. This column will aim to understand the actual requirements and responsibilities of computers, in practical scenarios, and will discuss specific technologies as ways of getting groups of computers to do what is mutually asked of them.

An Update on the Last Installment

Since the last installment was published, Pyro has reached version 3.0. There are a couple changes I would like to note—not necessarily the most important changes to Pyro overall, but those most related to my writeup. The minor bug in (little used) attribute access with static proxies has been fixed. More interesting to me is that Pyro has added an option to use my own `gnosis.xml.pickle` package as a wire protocol for pickled objects. Doing this adds some security advantages, but at the cost of somewhat increased processor requirements and message size. To the point of the current installment, Pyro with `gnosis.xml.pickle` becomes at least superficially similar to XML-RPC in using XML as a means of representing transmitted objects. Perhaps someday someone will even create an interface into Pyro from a programming language other than Python (since most languages can easily parse XML).

Introduction to XML-RPC

XML-RPC is a remote function invocation protocol that has the great virtue of being worse than all its competitors. Compared to Java RMI, or Pyro, or CORBA, or COM, XML-RPC is impoverished in the data it can transmit and obese in message size. XML-RPC abuses the HTTP protocol to circumvent firewalls that exist for good reasons, and as a consequence transmits messages lacking statefulness and incurs a channel bottleneck. Compared to SOAP, XML-RPC lacks both important security mechanisms and a robust object model. Just as a data representation, XML-RPC is slow, cumbersome, and incomplete compared to native programming language mechanisms like Java's ‘serialize’, Python's ‘pickle’, Perl's ‘Data::Dumper’, or similar modules for Ruby, Lisp, PHP, and many other languages.

In other words, XML-RPC is the perfect embodiment Richard Gabriel's motto "worse is better." I can hardly write more glowingly about XML-RPC than the prior paragraph, and I think the protocol is a perfect match for a huge variety of tasks. To understand why, I think it is worth quoting Gabriel characterizing the "worse-is-better philosophy":

- Simplicity: the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.

- Correctness: the design must be correct in all observable aspects. It is slightly better to be simple than correct.

- Consistency: the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.

- Completeness: the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

Writing years before the specific technology existed, Gabriel identifies the virtues of XML-RPC perfectly.

The XML-RPC Protocol

All there really is to XML-RPC is the following few steps.

1. A client application creates an XML-formatted request that indicates a remote method (by name), and any number of parameters. The parameters must belong to a fairly constrained set of data types—but these encompass the most common types for most programming languages.

2. The client sends the request to an XML-RPC server URL, using an HTTP POST message.

3. The XML-RPC server processes the formatted request, presumably by calling a method named by the remote request.

4. The XML-RPC server sends an XML-formatted response to the requesting client. If all is well, the body of the response will contain a `<params>` element (containing some `<param>` subelements). If things do not go quite as well a `<fault>` element will be sent in the response instead.

5. The client application does something useful with the response. Generally the first step is to parse out the date in the response, and this action is performed in a simple manner by the underlying XML-RPC library used.

In practice, there is usually even less than these steps involved in writing an XML-RPC client application. For example the following Python shell wraps an entire XML-RPC invocation:

Python shell XML-RPC example

```
>>> import xmlrpclib
>>> betty = xmlrpclib.Server("http://betty.userland.com")
>>> print betty.examples.getStateName(41)
South Dakota
```

An XML-RPC server requires a little bit more code—it needs to talk with, or act as, a web server—but a wrapper library still does most of the work for you. The XML format that is usually hidden when an XML-RPC library is used obeys a very simple DTD. Let us look at an example:

Python shell XML-RPC serialization example

```
>>> import xmlrpclib
>>> class C: pass
...
>>> c = C()
>>> c.bool, c.int, c.tup = (xmlrpclib.True, 37, (11.2, 'spam') )
>>> print xmlrpclib.dumps((c,),'PyObject')
<?xml version='1.0'?>
<methodCall>
<methodName>PyObject</methodName>
<params>
<param>
<value><struct>
<member>
<name>tup</name>
<value><array><data>
<value><double>11.2</double></value>
<value><string>spam</string></value>
</data></array></value>
</member>
<member>
<name>bool</name>
<value><boolean>1</boolean></value>
</member>
<member>
<name>int</name>
<value><int>37</int></value>
</member>
</struct></value>
</param>
</params>
</methodCall>
```

What Does XML-RPC Distribute?

What XML-RPC distributes can be read off its acronym, fairly accurate. At heart, XML-RPC is a way of making “remote procedure calls.” In fact the use of the name “method” in XML-RPC requests is slightly misleading. There need not be anything OOP about an XML-RPC server, even if particular implementations adopt an object-oriented approach.

In other words, essentially all the responsibilities and resource in an XML-RPC transaction fall on the server side. If a particular procedure utilizes memory or CPU resources, those resources must be present on the XML-RPC server; if a data source is utilized in performing a calculation, the database must be accessible to the server, but not necessarily directly to the client. Perhaps most importantly, the program logic of a calculation is under the control of the party who maintains the server, not the party who maintains the client. In most ways, XML-RPC is a form of old-fashioned client-server architecture.

There is another important limitation of XML-RPC: HTTP transactions are stateless. A protocol like Pyro, which the last installment looked at, creates persistent proxies for objects. That is, every XML-RPC call must contain all its requisite data in the call itself, rather than rely on computational results "remembered" by the server. Web application programmers have long been dealing with statelessness, so this constraint is probably familiar to most readers. There are many ways to layer state on a stateless protocol—cookies are perhaps the most common form, but are not available to XML-RPC applications. But roughly equivalent techniques include cookie-like information in URLs and/or hidden form fields. Both these latter approaches can easily be adapted to XML-RPC (there are not *hidden* fields as such, but one could add an extra parameter or two to indicate session ID).

Despite all these enumerated limitations, XML-RPC distributes what is probably the most important thing to distribute: responsibility. The model XML-RPC has in mind is a lot like the model for the WWW and the Internet—many different parties publish resources that they are in a unique position to maintain. For web pages, these resources are words and data that can be read by human eyes; for XML-RPC, these resources are data and information that can be processed by applications. For a huge class of applications, statefulness is really not that important, and information is mostly a one-way flow from experts to inquirers. XML-RPC would be poorly suited to the highly dynamic interactive roles that you might use Pyro for (but one *could* bolt together something based on XML-RPC). But when you want to know the current price of a stock, or the current temperature at a location, or even the current interest formula on a bank loan, there is someone who knows the (parameterized) information, and everyone else can simply ask for the answer.

Moreover, the strength of XML-RPC's simplicity can hardly be overemphasized. Because the protocol itself is so simple, robust implementations are readily available for a great many different programming languages. That in itself is a huge advantage over systems that restrict applications to one or two programming languages. But even beyond the language neutrality, the simplicity makes the creation and debugging of applications far easier and quicker. Sure SOAP or CORBA—or even Pyro—offer better persistence, richer data, and more fine-tuned delegation of responsibilities. But there is no such thing as a *small* project in SOAP or CORBA, and certainly no one-off case that you can cobble together in a day or two; those protocols are simply far too complex to work with without methodologies, code-review, and debugging cycles. With XML-RPC, you write it, it works—end of story!

Resources

Userland's XML-RPC homepage is, naturally, the place to start investigating XML-RPC. Many useful resources can be found there:

http://xmlrpc.com/

While at the XML-RPC homepage, it is particularly worthwhile to investigate the tutorials and articles they provide links for:

http://www.xmlrpc.com/directory/1568/tutorialspress

Richard P. Gabriel's rather famous paper “Lisp: Good News, Bad News, How to Win Big” can be found in full at the below URL. What everyone reads and refers to is the section called “The Rise of 'Worse is Better'”:

Error! Bookmark not defined.

The O'Reilly title _Programming Web Services with XML-RPC_, by Simon St. Laurent, Joe Johnston & Edd Dumbill, is quite excellent. Its spirit matches that of XML-RPC itself.