Distributing Computing

Introduction to Python Remote Objects (Pyro)

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

The distributed object framework "Pyro" is a wonderful tool for distributing several aspects of computing among multiple networked machines. Pyro is a system specific to applications written in the Python programming language, but in concept it is similar to technologies like Java's RMI (Remote Method Invocation), CORBA (Common Object Resource Broker Architecture), or even XML-RPC (XML Remote Procedure Call). Of course, in keeping with the Python philosophy, Pyro is a lot easier to use than its “competitors.” This installment continues a general overview of what it means to “distribute computing”—with specific source code examples implemented in Pyro.

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.

Introduction to Python Remote Objects (Pyro)

The Pyro framework in many ways formalizes the *ad hoc* techniques for sharing program logic that the first installment of this column introduced. But Pyro also goes considerably farther than does the simple remote code reloading the prior column gave examples of. In fact, Pyro allows users on a network to share nearly every resource and responsibility that might be distributed between different parties.

Before outlining what Pyro *is* exactly, let me provide three categories of resources/ responsibilities that Pyro can be used to distribute:

1. Computational (hardware) resources. Some computers have faster CPUs than others; and some likewise have more free cycles on those CPUs once the process priorities and application loads are considered. Similarly, some computers have more memory than others, or more disk space (important, for example, in certain large-scale scientific calculations). In some cases, specialized peripherals might be attached to one machine rather than another.

2. Informational resources. Some computers may have privileged access to certain data. Such privilege can be of several sorts. On the one hand, a particular machine might be the actual originating source of data; e.g. because it is attached to some sort of automated data collector like a scientific instrument or because it is a terminal into which users enter data (a cash register, a check-in desk, an observation site, etc.). On the other hand, a database might be local to a privileged computer, or at least to a limited group that the machine/account belongs to. Non-privileged computers might nonetheless have reason to have access to certain aggregate or filtered data derived from the database.

3. Business logic expertise. Within any organization—or between organizations—certain parties (individuals, departments, etc.) have the capacity and responsibility to decide the decision rules in certain domains. For example, the Payroll Department might determine (and

sometimes modify) the business logic concerning sick days and bonuses. Or Jane, the Database Administrator, might have the responsibility to determine the most efficient way to extract this datum from complex relational tables.

So just what is Pyro, and how does it let us distribute all those resources?

Pyro lets one application "serve" objects to another application. Usually these applications are on a network, but local loopback on one machine works also. A *remote* objects on a Pyro server becomes attached to a *proxy* object on a Pyro client. Once these two objects are attached, the client application can treat the remote object in a manner entirely homogenous with a true local object. In fact, inasmuch as the connection of proxies to remote objects is often properly done in a support library, an actual application programmer need not even *know* which objects are proxies to remote objects and which are true local objects. It sounds a bit complicated, but once we see the code samples below, it will be clear how amazingly simple the whole thing is.

Pyro sounds, at first brush, like old-fashioned client/server architecture, but it really isn't. One specific object is served by one machine, and utilized by one or more others; but the same machine—and even the same application—which utilizes some remote objects can simultaneously serve other objects. Moreover, a Pyro client application might connect to a large number of different servers, each one providing its own special resources. In fact, a client application usually does not even know or care which remote machine provides a specific object, it just chooses whichever node does what needs doing.

Some of the invisibility of Pyro is achieved by running a special kind of Pyro server called a "Pyro Name Server." This server can run pretty much anywhere, as long as it can be reached via TCP/IP from all of the regular clients and servers (Pyro might support protocols other than TCP/IP in the future, but not currently). On a local LAN, a broadcast mechanism is used to find a Name Server node without specifying any extra information; if firewalls and routers live between the various nodes, you will probably need to use some IP addresses or domain names to find the Name Server. Either way, utilizing a Name Server, while not strictly required for Pyro, automates the registration and location of remote objects by Pyro servers and clients. In principle, you could write your own customized Name Server using the Pyro library, but a simple and perfectly good script already comes with the Pyro package.

A Basic Pyro Client

Looking at some actual code makes understanding Pyro easier. After a few lines of setup code, an application programmer can pretty much forget that Pyro is involved at all. The particular example merely exercises a few typical object behaviors, then exits. But it is easy enough to imagine a real application using the same behaviors.

InsecureAgent_client.py

```
#!/usr/bin/env python
#-- Initialize Pyro client
import Pyro.naming, Pyro.core, sys
Pyro.core.initClient()
```

```
#-- Locate the Name Server
print 'Searching Name Server...'
locator = Pyro.naming.NameServerLocator()
ns = locator.getNS(Pyro.config.PYRO_NS_HOSTNAME)
print 'Name Server found at %s (%s) port %s' % \
      ( ns.URI.address,
        Pyro.protocol.getHostname(ns.URI.address) or '??',
        ns.URI.port )

#-- Locate URI for object in Name Server
print 'binding to object'
try:
      URI = ns.resolve('InsecureAgent')
      print 'URI:',URI
except Pyro.core.PyroError, x:
      print "Couldn't bind object, nameserver says:", x
      raise SystemExit

#-- Create a proxy for the Pyro object, and return that
ProxyAgent = Pyro.core.getAttrProxyForURI(URI)

#-- SampleObjects contains client-maintained logic
import SampleObjects

#-- Perform the actual object manipulation (application-level)
print '---- Informational methods ----'
print 'SYSTEM INFO:  ', ProxyAgent.environ()
print 'AVAILABLE CPU:', ProxyAgent.bogostone()

print '---- Performing long calculation ----'
print 'RESULT:', ProxyAgent.longcalc(1)

print '---- Dispatching custom code ----'
plus = SampleObjects.Plus()
print 'RESULT:', ProxyAgent.runtask(plus,17,22)
print 'LOCAL:', plus.run(17,22)

print '---- Attribute access ----'
print 'object.info:', ProxyAgent.info

print '---- Attribute assignment ----'
ProxyAgent.spam = 'eggs'
print 'object.spam =', ProxyAgent.spam
```

Let us take a look at what is going on in the client, it already suggests several sort of resources that might be shared. The initialization code is brief, and it could be made even shorter if some of the informational messages were not echoed to the client console along the way. The Name Server is located (somewhere on the local network), by a broadcast mechanism. Once found, we ask the Name Server to give us a URI for the object `InsecureAgent`, wherever it might happen to be. We make the assumption here that some server has registered this name; it is also possible to query the Name Server about what names have been registered prior to resolving a URI for a particular name. Moreover, fully hierarchical naming schemes are allowed, using periods as separators (as with Python packages)—e.g. `Gnosis.InsecureAgent.Fury` might be the `InsecureAgent` object that runs on the “Fury” node of Gnosis Software’s LAN.

Once we have an object URI (the below screen copy gives an example of what one looks like), we create a proxy object for it. This is conceptually similar to creating a plain local object, we just use the function `Pyro.core.getAttrProxyForURI()` to generate the object. For our example, the somewhat didactic name `ProxyAgent` is used, but in "normal" code you would probably just call it '`employee`' or '`barometer`' or '`storage`' according to the purpose of the object. Let us emphasize here that the code for `ProxyAgent's` methods simply need not exist on the client system. Maintenance of the business logic of an `InsecureAgent` is assigned to the party responsible for maintaining the server that registers it. Our client has no idea how an `InsecureAgent` is implemented beyond the fact it provides the desired methods and APIs. The responsibility here is quite similar to that suggested in the previous installment regarding the various utility functions maintained by Alice, Bob and Charlie.

A difference between the distributed-responsibility functions of the last installment and a Pyro setup concerns just where the code is running. The prior system downloaded new logical functionality when needed, but ran the code on the local CPU. Under Pyro, `InsecureAgent` utilizes, in main, the CPU/memory/disk resources of the remote system—the local only thinly wraps calls in the `ProxyAgent` local object. The example of `ProxyAgent`'s '`.longcalc()`' method illustrates the difference; this method, by name and outside behavior, takes a while to complete (in the toy example, it mostly just `sleep()`'s). Moreover, prior to calling '`.longcalc()`', the client called '`.bogostone()`' which provides a (crude) estimate of the amount of CPU resources available on the server under current task priority and load. One can imagine a client application that actually polled the '`.bogostone()`' method on several remote objects before deciding which one to use for the long calculation. A more sophisticated client might also dispatch the calculation to a separate thread.

More on Responsibilities

Something very interesting happens in the above Pyro client application. Let us take a quick look at what the output of the client looks like:

InsecureAgent_client output

```
$ python InsecureAgent_client.py
Pyro Client Initialized. Using Pyro V2.7
Searching Name Server...
Name Server found at 192.168.1.103 (CHAOS) port 9090
Binding to object
URI: PYRO://192.168.1.103:7766/ffffffff-3265463b-8b89ecfc-9d11faa1
---- Informational methods ----
SYSTEM INFO:   Windows 98 [Version 4.10.1998]
AVAILABLE CPU: 166
---- Performing long calculation ----
RESULT: 2
---- Dispatching custom code ----
RESULT: 39
LOCAL: 39
---- Attribute access ----
object.info: Windows 98 [Version 4.10.1998]
---- Attribute assignment ----
object.spam = eggs
```

We see a little here about finding a Name Server, and about the URI for '`InsecureAgent`' (in the example, it runs on the same machine as the Name Server, but it need not). But the more interesting aspect is the dispatch of custom code using the '`.runtask()`' method. It's worth looking at some of the source code for the `InsecureAgent` class:

InsecureAgent class

```
class InsecureAgent:

    def bogostone(self):
        "Crude estimate of current available CPU resources"
        [...]
        return self.speed
    def environ(self):
        "Find something to use as description of server config"
        [...]
        return self.info
    def runtask(self, obj, *args, **kw):
        "Call the .run() method on passed object (insecure)"
        self.result = obj.run(*args, **kw)
        return self.result
    def longcalc(self, N=50):
        "Calculate ... "
        return total
```

Most of the details of `InsecureAgent`'s methods are unimportant. Just assume they do what their docstrings indicate. The method '`.runtask()`' is interesting though—it simply calls the '`.run()`'method of the object that is passed in as an argument (using the arguments also passed in). But the effects of this dispatched invocation are rather profound. Whoever controls the client process, controls the "business logic" embodied in the passed object, but the object actually runs in the environment of the server. The server's environment certainly includes hardware resource, such as CPU cycles—at this level a sort of load-balancing is possible. But the more interesting aspect of the server environment is the informational resources contained at the server. Such resources might include databases and files available on the server, or merely global variables and instance attributes in the server object's scope. Notice that our test client runs the very same object locally that was dispatched to the server via the proxy's method. In our test case, the results are the same (i.e. the sum of the arguments); but in principle the results could be different because of differences in the server and client environments—either context might be useful, depending on exactly what one is trying to accomplish.

There is an important—and probably obvious—point to make here. A server that runs untrusted code passed to it opens a gaping security hole. Pyro includes some capabilities for adding security checks, such as restricting "mobile code" to clients or circumstances meeting whatever limits a programmer implements (e.g. only code from a certain IP address). Moreover, one has to explicitly enable a server to accept such mobile code objects. One might also take advantage of Python restricted execution capabilities to limit security exposure. While enabling mobile code as in the example is only a good idea in a limited range of situations, the fact one can do it at all is remarkably powerful and versatile.

Rounding out the Example

To finish the explanation of Pyro, let us take a look at the example server as well. One of the strengths of Pyro is that it lets one create server objects based on completely generic classes. For example, the `InsecureAgent` class knows absolutely nothing internally about Pyro, and could be used just as well in a local application. In order to create a remote object, several techniques may be used; basically though, you can either use inheritance or delegation to do it. The example server uses inheritance. The Pyro manual contains clear explanations of both approaches, as well as of everything else about using Pyro.

There's a slight wrinkle in the remote object created. Clients may create either "dynamic" or "static" proxies for remote objects. Most Pyro users stick to the dynamic proxies, which require less setup work. The only disadvantage to dynamic proxies is that they require a roundtrip between client and server to discover, for example, that a method doesn't exist in the remote object. Using a static proxy, such a check can be performed locally on the client end, and the network latency avoided altogether. In any case, there is a slight bug in Pyro 2.7 accessing attributes of remote objects using static proxies. The server below fixes the problem—and Pyro itself will have probably addressed this bug by the time you read this:

InsecureAgent_server.py

```
#!/usr/bin/env python

import sys
import Pyro.naming
import Pyro.core
from Pyro.errors import PyroError,NamingError
import Pyro, os

#-- Create remote class by inheritance (fix static proxy)
import InsecureAgent
class RemoteInsecureAgent(Pyro.core.ObjBase,
                          InsecureAgent.InsecureAgent):
    remote_getattr = Pyro.core.ObjBase._r_ga
    remote_setattr = Pyro.core.ObjBase._r_sa

def main():
    #-- Enable module import from client
    Pyro.config.PYRO_MOBILE_CODE = 1
    Pyro.config.PYRO_COMPRESSION = 1

    Pyro.core.initServer()
    PyroDaemon = Pyro.core.Daemon()
    locator = Pyro.naming.NameServerLocator()

    print 'searching for Name Server...'
    ns = locator.getNS(Pyro.config.PYRO_NS_HOSTNAME)

    print 'Name Server found at %s (%s) port %s' % \
          ( ns.URI.address,
            Pyro.protocol.getHostname(ns.URI.address) or '??',
            ns.URI.port )
    PyroDaemon.useNameServer(ns)
```

```
        #-- Connect new object implementation (1st unregister prev one)
        try:                              # name by which our object will
            ns.unregister('InsecureAgent')  # be known to the outside world
        except NamingError:
            pass
        ria = RemoteInsecureAgent()
        print dir(ria)
        PyroDaemon.connect(ria,'InsecureAgent')

        #-- Enter the server loop.
        print 'Server object "InsecureAgent" ready.'
        while 1:
            PyroDaemon.handleRequests(3.0)
            sys.stdout.write('.')
            sys.stdout.flush()

    if __name__=="__main__":
        main()
```

Not much too it, huh?

What Next?

In this installment, we have looked at the distribution of a variety of resources and responsibilities using the very friendly framework “Python Remote Objects” provide us. Still, as easy as Pyro makes things, it is still a Python-specific solution to distributing computing. In the next installment, we will take a look at how one can obtain much of the same distribution using the language-independent XML-RPC API.

Resources

The home page for Pyro is **Error! Bookmark not defined.**.

If you’d like to browse the excellent Pyro manual before downloading, it can be found at **Error! Bookmark not defined.**.