Distributing Computing

Introduction to remote program logic under Python

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

Within the Python programming language, several approaches to distributed code management exist. The goal of all such approaches is to allow program logic resident on one server to be utilized by processes on other servers or clients. The zero-case of such remote invocation is Python's inherent facility for dynamic reloading of support modules, and this provides a basis for understanding further techniques. Remote, dynamic reloading is discussed in this installment, and 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.

Introduction to Remote Program Logic

Suppose you want to run a process on your local machine, but part of your program logic lives somewhere else. Specifically, let us assume that this program logic is updated from time-to-time, and when you run your process, you would like to use the most current program logic. There are a number of approaches to addressing the requirement just described; this column will walk a reader through several of them. The code samples presented here will be in the Python programming language, but analogues for most of these techniques exist for other languages--future installments might address specifics within other programming langugages.

Consider a data processing application that performs a complex manipulation of a data source (e.g. a data file). This manipulation, by stipulation, is composed of a number of "business rules" that operate either in parallel or in sequence (or in some combination of the two). In particular, the business rules--like most real ones--are ones that can evolve over time, and ones which a variety of different parties have authority to specify. For many typical "real world" processes that meet the above stipulations, a processing application takes a while to run. These are the "batch processes" that mainframes have performed for many decades, and which smaller-scale machines often perform nowadays.

On a local system, running a data processing application is typically performed as a command-line task, with various flags and options used to specify behavior and data sources. For example:

Command-line data processing application

```
% local-application -opt1 -opt2 datafile
```

A line like the example might actually be contained in a batch/shell script and/or launched as a `cron` style job.

In the above usage, we rely on the fact that the most current version of `local-application` is on the local drive and path of the machine it runs on. If the application is updated from time to

time (on another machine), the process is more cumbersome. First the current application version would need to be made remotely accessible for download, such as at a URL. Next, before each run, we would need to check the remote resource, compare version numbers or filedates. Finally we would download the current version, copy it to the right directory, then run the command-line converter.

There are several manual and moderately time consuming steps involved in the above. It ought to be easier, and it can be.

Command-line Web Access

Most people think of the web as a way to browse pages interactively in a GUI environment. Doing that is nice, of course. But there is also a lot of power in a command-line. Systems with the text-mode web-browser `lynx` can largely treat the entire WWW as just another set of files for command-line tools to work with. For example, some commands I find useful are:

Command-line web browsing with lynx

```
Lynx -dump http://gnosis.cx/publish/.
lynx -dump http:///developer.intel.com/ > developer_intel.txt
lynx -dump http://gnosis.cx/TPiP/intro.txt | wc |
           sed "s/\( *[0-9]* *\)\([0-9]*\)\(.*\)/\2 words/"
```

The first command displays my homepage to the console (as ASCII text). The second command saves an ASCII version of IDS's current site to a file. The third example displays the number of words in a text file on my website (don't worry about the specifics, it just shows command-line tools being combined with pipes).

Tools like `lynx` or `wget` provide ways to treat remote data sources in a manner homogeneous with local files. With CGI (Common Gateway Interface), and with similar techniques like ASP, JSP, Cold Fusion, Fast-CGI, mod_perl, etc., actual program logic can equally well reside at the end of a URL. This provides a way to call remotely hosted utilities in much the same one might local utilities.

As a fairly trivial example, I have written a CGI version of the `wc` utility, and hosted it on my website. This utility spends much more effort on the CGI handling (still simple in Python) than it does on the logic in the `wc()` function. But as a template, I can maintain and/or update the logic of the function, while anyone with a way of invoking a URL can utilize it. The example `wc.cgi` program consists of:

http://gnosis.cx/cgi-bin/wc.cgi

```
#!/usr/bin/python

import cgi, sys
from urllib import urlopen
from string import split, join

def wc(s):
    return len(split(s,'\n')), len(split(s)), len(s)

sys.stderr = sys.stdout
```

```
print "Content-type: text/html\n"
form = cgi.FieldStorage()

if form.has_key('source'):
    src = form['source'].value
    s = urlopen(src).read()
    print '<html><head><title>Word count for: %s</title></head>' % src
    print '<body><pre>'
    print '%8i%8i%8i %s' % (wc(s)+(src,))
    print '</pre></body></html>'
else:
    print '<html><head><title>No specified document
    source</title></head>'
    print '<body>No specified document source</body></html>'
```

This program can be invoked in exactly the same manner as above, using `lynx`, generally with the data file indicated as part of the URL:

Remote-logic invocation with lynx

```
lynx -dump "http://gnosis.cx/cgi-
bin/wc.cgi?source=http://gnosis.cx/TPiP/intro.txt"
```

Readers familiar with HTML forms will see that a call to my `wc.cgi` program could easily be incorporated into an interactive web page also. Of course if using a remote application is a repeated process, the call can be wrapped with a local application, such as:

remote_wc.py command-line utility

```
#!/usr/bin/python

import sys, string
from urllib import urlopen, urlencode
from htmllib import HTMLParser
from formatter import AbstractFormatter, DumbWriter

if len(sys.argv) > 1:
    cgi = 'http://gnosis.cx/cgi-bin/wc.cgi'
    opts = urlencode({'source':sys.argv[1]})
    fhin = urlopen(cgi, opts)
    parser = HTMLParser(AbstractFormatter(DumbWriter()))
    parser.feed(fhin.read())
else:
    print "No specified URL for remote word count"
```

To run this script, execute something like:

```
% remote_wc.py Error! Bookmark not defined.
```

Should I decide to develop a new and better word count algorithm in the future, users of `remote_wc.py` can simply continue to use their local utility unchanged, confident the most up-to-date algorithm will be invoked.

Dynamic Initialization

Using `remote_wc.py` assures that the latest program logic is always used in conversions. Another thing this approach does, however, is move the processor (and memory) requirements onto the gnosis.cx webserver. The load imposed by this particular process is slight, but it is easy to imagine other types of processes where processing on the client is more efficient and desirable.

The way many programs are organized--is with a couple core flow-control functions assisted by a variety of utility functions. In particular, the utility functions are the ones whose exact specification and maintenance might be an outside responsibility. We might helpfully update the utility functions at each program run. Under this arrangement, a local machine assumes responsibility for processor/memory requirements, but a remote machine assumes responsibility for (part of) the program logic.

As an example, I have created a few text analysis functions, and made them available at my website:

> http://gnosis.cx/download/text_utils.py

```
from string import split

def wc(s):
    return len(split(s,'\n')), len(split(s)), len(s)

def histogram(s):
    hist = {}
    for word in split(s):
        hist[word] = hist.get(word, 0)+1
    return hist

def top10(hist):
    entries = []
    for word, cnt in hist.items():
        entries.append((cnt, word))
    entries.sort()
    entries.reverse()
    return entries[:10]
```

A user who wishes to incorporate the latest-and-greatest version of my provided utilities with each call to her local application can simply attempt to obtain them as part of program initialization. For example:

> dyn_hist.py command-line utility

```
#!/usr/bin/python

import sys, string
from urllib import urlopen

# Check for updated functions (fail gracefully if not fetchable)
try:
    url = urlopen('http://gnosis.cx/download/text_utils.py')
    updates = url.read()
    fh = open('wc_utils.py', 'w')
```

```
        fh.write(updates)
        fh.close()
    except:
        sys.stderr.write('Cannot currently download text_utils updates')

    # Import the updated functions (if available)
    try:
        from text_utils import *
    except:
        sys.stderr.write('Cannot import the updated text_utils functions')

    if len(sys.argv) > 1:
        s = urlopen(sys.argv[1]).read()
        print "Top ten words:"
        for cnt, word in top10(histogram(s)):
            print word, '\t', cnt
    else:
        print "No specified URL"
```

A certain flexibility is provided in the example inasmuch it will fall back to the last downloaded version of the utilities if the current set cannot be obtained at runtime. Python's *`urllib`* is sufficiently versatile to allow opening either remote or local resources, so this same utility can be used without any Internet connection, e.g.:

```
% dyn_hist.py file:///book/intro.txt
Top ten words:
the     88
of      66
a       63
is      48
and     47
to      43
in      37
that    27
Python  27
will    25

% dyn_hist.py http://gnosis.cx/TPiP/intro.txt
[...]
```

One minor matter is that different systems handle writes to STDERR differently. Under Unix-like systems, you can redirect STDERR when you run the script; however, under my current OS/2 shell, and under Windows/DOS, the STDERR messages will be appended to the console output. You might want to write the above errors/warning to a log file instead.

In a simple case like that above, a single responsible party could maintain a set of utility functions. But more generally, the program logic could be distributed more widely. You might have Alice, Bob and Charlie be responsible for modules *`Funcs_A`*, *`Funcs_B`* and *`Funcs_C`*, respectively. Each of them make periodic (and independent) changes to the functions under their control, and upload the current versions to their own website (such as http://alice.com/Funcs_A.py). A script similar to `dyn_hist.py` can straightforwardly be extended to try importing *`Funcs_A`*, *`Funcs_B`* and *`Funcs_C`* all at startup (and fallback to last-available versions if these resources cannot be obtained).

A Long-running Dynamic Process

The tools we have looked at so far get their dynamic program logic by downloading updated resources at initialization. This makes a lot of sense for command-line or batch processes. But what about long-running applications? Such long-running applications are likely to be server processes that respond to client requests continuously. For this article, however, I present a simple interactive version of the previous command-line utilities.

An interactive application can potentially be left running in the background all the time, and we would like it to be able to utilize up-to-date program logic when we switch to its session. For this specific simple example, it admittedly would not be difficult to close and relaunch the application, and no particular disadvantages would be incurred. But it is easy to imagine other processes that genuinely do depend on being left running all the time, perhaps ones that are stateful as to action performed in a session. Below is an interactive "text analysis" shell:

textutil_shell.py interactive application

```
from string import upper, split
from urllib import urlopen
import sys
import text_utils   # Updateable functions

def update():
    try:    # Check for updated functions (fail gracefully)
        url = urlopen('http://gnosis.cx/download/text_utils.py')
        updates = url.read()
        fh = open('wc_utils.py', 'w')
        fh.write(updates)
        fh.close()
        print 'Download of text_utils updates successful'
    except:
        print 'Cannot currently download text_utils updates'
    try:    # Import the updated functions (if available)
        reload(text_utils)
        print 'Reload of text_utils successful'
    except:
        print 'Cannot import the updated text_utils functions'
        sys.exit()

if __name__=='__main__':
    while 1:
        print '-'*70, "\nCOMMAND:",
        print "(U)pdate / (W)ordcount <URL> / (H)istogram <URL> /
(Q)uit"
        print '-'*70
        command = raw_input('>>> ')
        action = upper(command[:1])
        if   action == 'Q': break
        elif action == 'U': update()
        elif action == 'W':
            _, url = split(command)
            s = urlopen(url).read()
            print '%8i%8i%8i %s' % (text_utils.wc(s)+(url,))
        elif action == 'H':
            _, url = split(command)
```

```
            s = urlopen(url).read()
            print "Top ten words:"
            for cnt, word in text_utils.top10(text_utils.histogram(s)):
                print word, '\t', cnt
        else:
            print "Unrecognized command"
```

The main body of this application is a line-input loop that performs one of a few actions depending on command specified. The `update()` function is basically the same as what we saw in the `dyn_hist.py` application, except that it calls Python's `reload()` function instead of an `import` command. Just performing a brand new `import text_utils` will **not** overwrite the functions previously imported. Watch out for this! A lot of Python beginners assume that reimporting a module will update the version in memory. It won't. Instead, the way to update the in-memory image of the functions in a module is to `reload()` the module.

Let us take a look at the application in action:

```
% python textutil_shell.py
----------------------------------------------------------------------
COMMAND:  (U)pdate / (W)ordcount <URL> / (H)istogram <URL> / (Q)uit
----------------------------------------------------------------------
>>> wc http://gnosis.cx/TPiP/intro.txt
     303    2219   14717 http://gnosis.cx/TPiP/intro.txt
----------------------------------------------------------------------
COMMAND:  (U)pdate / (W)ordcount <URL> / (H)istogram <URL> / (Q)uit
----------------------------------------------------------------------
>>> update
Download of text_utils updates successful
Reload of text_utils successful
----------------------------------------------------------------------
COMMAND:  (U)pdate / (W)ordcount <URL> / (H)istogram <URL> / (Q)uit
----------------------------------------------------------------------

>>> quit
```

This particular application only updates the remote program logic at user request, but it would be simple to create an application that attempted reloading on other triggers--after a certain amount of time, if certain functions were being performed, etc.

What Next?

In this installment, we have looked at ways of distributing program logic using only standard Python modules and functions. With just these basics, you can arrive at just about any desired system for delegating responsibilities surrounding algorithms. However, we will see in later columns that it is possible to wrap much of the delegation in standard API's, as well as to encapsulate responsibilities even more neatly than these basic approaches do. After looking at a few more options in Python, we will look at other programming languages, and ways of cooperating between languages. Eventually, this column will consider other resources that might be shared besides program logic.

Resources

The recently released SOAP-based Google Web API is an interesting example of a programmatic interface to what had traditionally been an interactive web-based service. Information is at:

Error! Bookmark not defined.

A Python interface for the Google Web API can be found at:

Error! Bookmark not defined.