

# Charming Python: Distributing computing with RPyC

A native Python option for seamless control of many machines

Skill Level: Intermediate

[David Mertz, Ph.D. \(mertz@gnosis.cx\)](mailto:mertz@gnosis.cx)

Programmer  
Gnosis Software

31 Mar 2009

RPyC is a seamless library for integrating Python processes on many machines/processes. This article looks at the advantages or drawbacks RPyC has over other distributed Python frameworks such as XML-RPC and Pyro. A few simple examples of using RPyC are included to give you a feel for the library.

Back in 2002, I wrote a series of articles on "distributing computing" (see [Resources](#)). At that time, RPyC did not exist, but I covered the Python tool Pyro and the library `xmlrpclib`. Surprisingly little has changed in the last seven years. Pyro is still around and actively maintained; and `xmlrpclib` still exists in Python, as does `SimpleXMLRPCServer` (though these have been refactored as `xmlrpc.client` and `xmlrpc.server` in Python 3).

The space RPyC steps into is very much the one already occupied by Pyro and XML-RPC. RPyC does not really add anything fundamentally new to these longstanding tools, but there are some nice features to the design of RPyC.

RPyC 3.0+, in a nutshell, has two modes to it: a "classic mode," which was already available prior to its version 3, and a "services mode," which was introduced in version 3. The classic mode lacks any real security framework (which isn't always a bad thing) and simply presents remote machines as if they were local resources. The newer, services, mode isolates a collection of published interfaces that a server supports and is secure inasmuch as it prohibits everything not explicitly permitted. The classic mode is essentially identical to Pyro (without Pyro's optional security

framework); the service mode is essentially RPC (for example, XML\_RPC), modulo some details on calling conventions and implementation.

## Background on distributing computing

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 such as printers, scanners, sound systems, modems, game inputs, and so on. Personal computers also have network capabilities, but conventionally, a network card has largely been just another sort of I/O device.

"Distributing computing" is a buzz-phrase that has something to do with providing more diverse relationships between computing resources and actual computers. In the old days, we used to speak of "client/server" and "N-Tier architecture" to describe hierarchical relationships among computers. However, different resources can enter into many different sorts of relationships—some hierarchical, others arranged in lattices, rings, and various other topologies. The emphasis has shifted towards graphs, away from trees. Some of the many possible examples:

- SANs (storage-area networks) centralize persistent disk resources for a large number of computers.
- In the opposite direction, peer-to-peer (P2P) protocols such as 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 such as Linux Beowulf allow many CPUs to share the processing of a complex computation, whereas projects such as 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.
- Ajax is another means, albeit specifically within a Web browser client, of utilizing resources from many sources.

The protocols and programs that distribute what were basically hardware resources of old-fashioned PC applications make up only part of the distributed computing picture. At a more abstract level, much more interesting things can be distributed: data, information, program logic, "objects," and, ultimately, responsibilities. DBMSs are a traditional means of centralizing data and structuring its retrieval. In the other direction NNTP, and later P2P, radically decentralized information storage. Other technologies, such as search engines, restructure and recentralize information

collections. Program logic describes the actual rules of proscribed computation (various types of RMI and RPC distribute this); object brokerage protocols such as DCOM, CORBA, and SOAP recast the notion of logic into an OOP framework. Of course, even old-style DBMSs with triggers, constraints, and normalizations always carried a certain degree of program logic with them. All of these abstract resources are at some point stored to disks and tapes, represented in memory, and sent as bitstreams over networks.

In the end, what is 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 first about particular configurations of hardware, but are almost always about satisfying functional requirements of the recipients.

## Why RPyC is useful

Before outlining what RPyC does, let me suggest three categories of resources/responsibilities that RPyC 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; for example, because it is attached to some sort of automated data collector such as 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 or 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.

RPyC lets you distribute all those resources.

## RPyC classic mode

The RPyC classic mode, in essence, just lets all the capabilities of a remote Python installation run within a local Python system. Security here basically amounts to the same thing as giving a shell account to whoever connects. If security is at issue, the encryption library `tlslite` can encrypt connections and require logins for connections. In other words, you can easily create the equivalent of `ssh` rather than the equivalent of `telnet`. Of course, the advantage here is that these connections can be controlled by Python scripts, and allow more robust interaction between local and remote resources than does a language such as Expect.

To launch a server on a remote machine, simply run `classic_server.py` that comes with RPyC. If you want a secured connection, add the `--vdb` option. To customize a port, use `--port`, or check `--help` for additional options in the server. You can, of course, launch your server as part of general system initialization or in cron jobs, to make sure they are running on a given machine. Once some servers are running, you can connect to them from as many clients as you like. Note, however, that servers are not really *that* special; the same machine and process can utilize many servers and act as server to many clients at the same time (including two machines symmetrically "serving" each other).

A shell session (once some servers are launched) shows this:

### Listing 1. System and Python information

```
>>> import sys,os
>>> os.uname()          # Some info about the local machine
('Darwin', 'Mary-Anns-Laptop.local', '10.0.0d1',
'Darwin Kernel Version 10.0.0d1: Tue Jun  3 23:40:01 PDT 2008;
root:xnu-1292.3~1/RELEASE_I386', 'i386')
>>> sys.version_info  # Some info about the local Python version
(2, 6, 1, 'final', 0)
>>> os.getcwd()
'/Users/davidmertz'
```

Now, let's import RPyC and connect to a few servers. Before doing this, I had launched local servers from two different Python versions, and one on a remote machine.

### Listing 2. Importing RPyC and connecting to servers

```
>>> import rpyc
>>> conn26 = rpyc.classic.connect('localhost')
```

```

>>> conn26.modules.os.uname()
('Darwin', 'Mary-Anns-Laptop.local', '10.0.0d1',
'Darwin Kernel Version 10.0.0d1: Tue Jun  3 23:40:01 PDT 2008;
root:xnu-1292.3~1/RELEASE_I386', 'i386')
>>> conn26.modules.sys.version_info
(2, 6, 1, 'final', 0)
>>> conn26.modules.os.getcwd()
'/Users/davidmertz/Downloads/rpyc-3.0.3/rpyc/servers'
>>> conn25 = rpyc.classic.connect('localhost',port=18813)
>>> conn25.modules.os.uname()
('Darwin', 'Mary-Anns-Laptop.local', '10.0.0d1',
'Darwin Kernel Version 10.0.0d1: Tue Jun  3 23:40:01 PDT 2008;
root:xnu-1292.3~1/RELEASE_I386', 'i386')
>>> conn25.modules.sys.version_info
(2, 5, 1, 'final', 0)
>>> conn25.modules.os.getcwd()
'/Users/davidmertz/Downloads/rpyc-3.0.3/rpyc/servers'
>>> connGlarp = rpyc.classic.connect("71.218.122.169")
>>> connGlarp.modules.os.uname()
('FreeBSD', 'antediluvian.glarp.com', '6.1-RELEASE',
'FreeBSD 6.1-RELEASE #0: Fri Jul 18 00:01:34 MDT 2008;
root@antediluvian.glarp.com:/usr/src/sys/i386/compile/ANTEDILUVIAN',
'i386')
>>> connGlarp.modules.sys.version_info
(2, 5, 2, 'final', 0)
>>> connGlarp.modules.os.getcwd()
'/home/dmertz/tmp/rpyc-3.0.3/rpyc/servers'

```

You can see that we have connections to a couple of different machines with different Python versions. The functions and attributes we accessed are arbitrary, but the important point is that we might call *any* functions or classes available on those machines. So, for example, if I know the machine `antediluvian.glarp.com` has a Python module `Payroll` installed on it that has the function `get_salary()` in it, I might call:

### Listing 3. Calling `get_salary()`

```

>>> connGlarp.modules.Payroll.get_salary(last='Mertz',first='David')

```

Over at `antediluvian`, there might be a local database installed, or it might even make its own connections to other resources. What is returned by my function call, however, is simply the same data that would be returned if the function was run locally on `antediluvian`.

### Putting code on the remote machine

Running standard module functions on a remote machine is a cute trick, but what we often want to do more usefully is run our own code remotely. There are several ways of doing this within RPyC's classic mode. The most direct way is perhaps to simply open a Python shell onto that machine via the connection we have established. For example:

### Listing 4. Opening a Python shell on a remote machine

```
>>> conn = rpyc.classic.connect('linux-server.example.com')
>>> rpyc.classic.interact(conn)
Python 2.5.2 (r252:60911, Oct  5 2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
(InteractiveConsole)
>>> #... run any commands on remote machine
```

From such a remote shell, you can define whatever functions or classes you want, import any outside code, or do anything you can do from a local Python shell.

There is also an RPyC function called `deliver()` that seems to send a local object over to a remote server, where presumably it might run in the local context. Unfortunately, I was not able to get this function to act as I expected (I am sure I just have some syntax wrong, but the documentation is vague). As a kludge, you can directly execute (or `eval()`) code in the remote server. For example:

### Listing 5. Executing code on a remote machine

```
>>> # Define a function (or class etc) as actual source code
>>> hello_txt = """
... def hello():
...     import os
...     print "Hello from", os.getcwd()
... """
>>> exec hello_txt          # Run the function definition locally
>>> hello()
Hello from /tmp/clientdir
>>> conn.execute(hello_txt) # Run the function definition remotely
>>> remote_hello = conn.namespace['hello']
>>> remote_hello()         # Displays on remote server
>>> with rpyc.classic.redirected_stdio(conn): # Redirect to local
client
...     conn.namespace['hello']()
...
Hello from /tmp/serverdir
```

What we have done is compile a function into the remote namespace and run it as if it were a local function. However, we need to also grab that remote console output if we want to actually see the output from its `print`. If `hello()` had instead *returned* a value, though, it would still be returned to the local context as in the prior examples.

### Monkey patching

What we did with the `with` context above was basically a kind of "monkey patching." That is, we temporarily used a local STDIO in place of a remote STDIO. RPyC lets you do this in general with system resources. Code might run (that is, use CPU and memory) on a local machine but still use some key resource on the remote machine. That might be a computationally expensive function call if the server has more CPU resources, but often it is something such as a socket where the remote machine has

different access domains. For example:

### Listing 6. Monkey patching a socket connection

```
import myserver
c = rpyc.classic.connect('machine-outside-firewall')
myserver.socket = c.modules.socket
# ...run myserver, which will now open sockets outside firewall
```

### Asynchronous resources

If utilizing a resource on a remote machine is time consuming, the local program using that resource might seem to stall while the remote action is completing. However, this need not be so if you make these calls *asynchronous*. A remote object can report whether it has a result ready, and you can do other local actions in the meantime (or perhaps actions utilizing other remote servers).

### Listing 7. An asynchronous call

```
>>> conn.modules.time.sleep(15)      # This will take a while
>>> # Let the server do the waiting
>>> asleep = rpyc.async(conn.modules.time.sleep)
>>> asleep
async(<built-in function sleep>)
>>> resource = asleep(15)
>>> resource.ready
False
>>> # Do some other stuff for a while
>>> resource.ready
True
>>> print resource.value
None
>>> resource
<AsyncResult object (ready) at 0x0102e960>
```

In our example `resource.value` is uninteresting. However, if the remote method that we made asynchronous returned a value, that value would be available once `resource.ready` became `True`.

### RPyC service mode

I will write relatively little about the newer RPyC service mode. The classic mode is really the more general system, even though it is itself built as a new-style service in just a few lines of code. A service under RPyC is really little different from XML-RPC (or whatever-RPC). Classic mode is just a service that exposes *everything* on the remote system, but you can build services that expose just a few things in a small number of lines of code. For example:

### Listing 8. Using service mode to expose a few methods

```
import rpyc
class DoStuffService(rpyc.Service):
    def on_connect(self):
        "Do some things when a connection is made"
    def on_disconnect(self):
        "Do some things AFTER a connection is dropped"
    def exposed_func1(self, *args, **kws):
        "Do something useful and maybe return a value"
    def exposed_func2(self, *args, **kws):
        "Like func1, but do something different"

if __name__ == '__main__':
    rpyc.utils.server.ThreadedServer(DoStuffService).start()
```

From a client, this service is just like the classic mode server, except all it exposes is the methods that are prefixed by `exposed_` (minus the prefix). Trying to access other methods (such as builtin modules) will fail. So a client might look like this:

### Listing 9. Calling service-exposed methods

```
>>> import rpyc
>>> conn = rpyc.connect('dostuff.example.com')
>>> myval = conn.root.func1() # Special 'root' of connection
>>> local_computation(myval)
```

## Conclusion

There is more in RPyC than I have mentioned. For example, like Pyro, RPyC provides a "Registry" that lets you name services and access them by name rather than by domain name or IP address. This is straightforward, and the RPyC documentation explains it.

As I have indicated in this article—and as RPyC's own documentation says explicitly—RPyC is in many ways "Yet another RPC package." For example, the service we briefly constructed above is nearly identical to the same code we would write using `SimpleXMLRPCServer`. The only difference is the wire protocol used for requests and results. Nonetheless, despite some minor glitches I encountered, RPyC is well constructed and very simple to get running. You can easily construct a robust distribution of resources and responsibilities using just a few lines of RPyC code.

# Resources

## Learn

- Read David's series of "Distributing Computing" articles:
  - ["Introduction to remote program logic under Python"](#) (Mertz, April 2002)
  - ["Introduction to Python Remote Objects \(Pyro\)"](#) (Mertz, April 2002)
  - ["Cross-language remote invocation with XML-RPC"](#) (Mertz, June 2002)
  - ["Cooperative computing with mobile agents"](#) (Rempt and Mertz, July 2002)
- In the [developerWorks Linux zone](#), find more resources for Linux developers, and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

## Get products and technologies

- The [RPyC home page](#) contains good documentation, as well as the downloads and links to screencasts and discussion areas.
- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

## Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and spaces.

## About the author

David Mertz, Ph.D.

David Mertz thinks that flat address spaces are better than nested. David may be reached at [mertz@gnosis.cx](mailto:mertz@gnosis.cx); his life pored over at <http://gnosis.cx/publish/>. Check out David's book *Text Processing in Python* (<http://gnosis.cx/TPiP/>).

## Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.