**[SSH Programming with Paramiko | Completely Different](http://jessenoller.com/blog/2009/02/05/ssh-programming-with-paramiko-completely-different)**

by [jesse](http://jessenoller.com/?author=50b76e12e4b0cca10d7c421a) in [Programming](http://jessenoller.com/?category=Programming), [Python](http://jessenoller.com/?category=Python), [python magazine](http://jessenoller.com/?category=python+magazine)

OpenSSH is the ubiquitous method of remote access for secure remote-machine login and file transfers. Many people -- systems administrators, test automation engineers, web developers and others have to use and interact with it daily. Scripting SSH access and file transfers with Python can be frustrating -- but the Paramiko module solves that in a powerful way.

*This is a reprint of an article I wrote for* [*Python Magazine*](http://static.squarespace.com/static/50b76babe4b05c3cd8bab78a/50b7714be4b0192bc2226c59/50b772fde4b0192bc222822b/1354199805695/?format=original) *as a Completely Different column that was published in the October 2008 issue.* ***I have republished this in its original form, bugs and all***

SSH is everywhere. OS X, Linux, Solaris, and even Windows offer OpenSSH servers for remote access and file transfers. It long ago displaced other methods of remote access like telnet and rlogin. While those other systems may still exist, their widespread usage has faded with the rapid adoption of the OpenSSH suite of tools.

OpenSSH itself is actually a suite of tools based on the ssh2 protocol. The suite provides secure remote login tools (ssh), secure file transfer (scp and sftp), and key management tools.

On most operating systems the client-side tools (ssh, scp, sftp) are already installed for users to leverage. Users can also easily install and configure the server-side utilities on systems they want to remotely access.

Many, many people use OpenSSH daily, and many of them spend a lot of time trying to script its usage. Most of these tools and scripts try to wrap the command line executables (ssh, scp, etc) directly. They use things like Pexpect to provide passwords, and try to rationalize and parse the output of the binaries directly.

Having spent a lot of time scripting around the binaries and trying to manage timeouts, standard out/in/error pipes, authentication, arguments and options all through ''subprocess'', ''popen2'', etc., I'm here to tell you wrapping command line binaries is prone to error, difficult to test, and painful to maintain.

When you're in the business of parsing output from command line utilities, watching for exit codes and juggling timeouts, you're not on a good path. That's where something like Paramiko comes in.

I discovered Paramiko some time ago. It builds on PyCrypto to provide a Python interface to the SSH2 protocol. The module provides all of the faculties you could ask for, including: ssh-key authentication, ssh shell access, and sftp.

Since discovering Paramiko, my entire paradigm and usage of SSH has changed. Instead of the frustrating experience of shelling-out and hacking around the various kinks with that, I can programmatically access all of the protocols and tools I need in a clean, Pythonic way.

**About Paramiko**

Paramiko is a pure-Python module and can be easy\_install'ed as other typical python modules can. However, PyCrypto is written largely in C, so you may need a compiler to install both depending on your platform.

Paramiko itself has extensive API documentation and an active mailing list. As an added bonus, there's a Java port of it as well (don't get me started on controlling SSH within Java) if you need something to achieve the same thing in Java.

Paramiko also offers an implementation of the SSH and SFTP server protocols. It really is feature-rich and complete. I've used it in heavily threaded applications as well as in day-to-day maintenance scripts. There's even an installation and deployment system, named Fabric, that further builds on Paramiko to provide application deployment utilities via SSH.

**Getting started**

The primary class of the Paramiko API is ''paramiko.SSHClient''. It provides the basic interface you are going to want to use to instantiate server connections and file transfers.

Here's a simple example:

import paramiko

ssh = paramiko.SSHClient()

ssh.connect('127.0.0.1', username='jesse',

password='lol')

This creates a new SSHClient object, and then calls ''connect()'' to connect us to the local SSH server. It can't get much easier than that!

**Host Keys**

One of the complicating aspects of SSH authentication is *host keys*. Whenever you make an ssh connection to a remote machine, that host's key is stored automatically in a file in your home directory called ''.ssh/known\_hosts''. If you've ever connected to a new host via SSH and seen a message like this:

The authenticity of host 'localhost (::1)' can't be

established.

RSA key fingerprint is

22:fb:16:3c:24:7f:60:99:4f:f4:57:d6:d1:09:9e:28.

Are you sure you want to continue connecting

(yes/no)?

and typed "yes" -- you've added an entry to the ''known\_hosts'' file. These keys are important because accepting them implies a level of trust of the host. If the key ever changes or is compromised in some way, your client will refuse to connect without notifying you.

Paramiko enforces this same rule. You must accept and authorize the use and storage of these keys on a per-host basis. Luckily, rather then having to be prompted for each one, or manage each one individually, you can set a magic policy.

The default behavior with an SSHClient object is to refuse to connect to a host (''paramiko.RejectPolicy'') who does not have a key stored in your local ''known\_hosts'' file. This can become annoying when working in a lab environment where machines come and go and have the operating system reinstalled constantly.

Setting the host key policy takes one method call to the ssh client object (''set\_missing\_host\_key\_policy()''), which sets the way you want to manage inbound host keys. If you're lazy like me, you pass in the ''paramiko.AutoAddPolicy()'' which will auto-accept unknown keys.

import paramiko

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(

paramiko.AutoAddPolicy())

ssh.connect('127.0.0.1', username='jesse',

password='lol')

Of course, don't do this if you're working with machines you don't know or trust! Tools built on Paramiko should make this overly liberal policy a configuration option.

**Running Simple Commands**

So, now that we're connected, we should try running a command and getting some output.

SSH uses the same type of input, output, and error handles you should be familiar with from other Unix-like applications. Errors are sent to standard error, output goes to standard out, and if you want to send data back to the application, you write it to standard in.

So, the response data from client commands are going to come back in a tuple - (stdin, stdout, stderr) - which are file-like objects you can read from (or write to, in the case of stdin). For example:

...

>>> ssh.connect('127.0.0.1', username='jesse',

... password='lol')

>>> stdin, stdout, stderr = \

... ssh.exec\_command("uptime")

>>> type(stdin)

>>> stdout.readlines()

['13:35 up 11 days, 3:13, 4 users, load averages: 0.14 0.18 0.16\n']

Under the covers, Paramiko has opened a new ''paramiko.Channel'' object which represents the secure tunnel to the remote host. The Channel object acts like a normal python socket object. When we call ''exec\_command()'', the Channel to the host is opened, and we are handed back ''paramiko.ChannelFile'' "file-like" objects which represents the data sent to and from the remote host.

One of the documented nits with the ChannelFile objects paramiko passes back to you is that you need to constantly ''read()'' off of the stderr and stdout handles given back to you. If the remote host sends back enough data to fill the buffer, the host will hang waiting for your program to read more. A way around this is to either call ''readlines()'' as we did above, or ''read()''. If you need to internally buffer the data, you can also iterate over the object with ''readline()''.

This is the simplest form of connecting and running a command to get the output back. For many sysadmin tasks, this will be invaluable as you need to parse the output of a returned command to find exactly what you need. With Python's rich string manipulation, this is an easy task. Let's run something with a lot of output, that also requires a password:

ssh.connect('127.0.0.1', username='jesse',

password='lol')

stdin, stdout, stderr = ssh.exec\_command(

"sudo dmesg")

Uh oh. I just called the sudo command. It is going to require me to provide a password interactively with the remote host. No worries:

ssh.connect('127.0.0.1', username='jesse',

password='lol')

stdin, stdout, stderr = ssh.exec\_command(

"sudo dmesg")

stdin.write('lol\n')

stdin.flush()

data = stdout.read.splitlines()

for line in data:

if line.split(':')[0] == 'AirPort':

print line

There! I logged in remotely and found all messages for my Airport card. The key thing to note here is that I wrote my password to the stdin "file" so that sudo allowed me in.

If you're wondering, yes, this provides an easy base to create your own interactive shell. You might want to do something like this to make a little custom admin shell using the Python cmd module to administer machines inside of your lab.

Using Paramiko, this is easy. In Listing 1, I outline a basic way to approach this - we wrap the Paramiko manipulation up in the RunCommand methods, allowing the user to add as many hosts as they want, call connect and then run a command.

Listing 1:

#!/usr/bin/python

import paramiko

import cmd

class RunCommand(cmd.Cmd):

""" Simple shell to run a command on the host """

prompt = 'ssh > '

def \_\_init\_\_(self):

cmd.Cmd.\_\_init\_\_(self)

self.hosts = []

self.connections = []

def do\_add\_host(self, args):

"""add\_host

Add the host to the host list"""

if args:

self.hosts.append(args.split(','))

else:

print "usage: host "

def do\_connect(self, args):

"""Connect to all hosts in the hosts list"""

for host in self.hosts:

client = paramiko.SSHClient()

client.set\_missing\_host\_key\_policy(

paramiko.AutoAddPolicy())

client.connect(host[0],

username=host[1],

password=host[2])

self.connections.append(client)

def do\_run(self, command):

"""run

Execute this command on all hosts in the list"""

if command:

for host, conn in zip(self.hosts, self.connections):

stdin, stdout, stderr = conn.exec\_command(command)

stdin.close()

for line in stdout.read().splitlines():

print 'host: %s: %s' % (host[0], line)

else:

print "usage: run "

def do\_close(self, args):

for conn in self.connections:

conn.close()

if \_\_name\_\_ == '\_\_main\_\_':

RunCommand().cmdloop()

Example output:

ssh > add\_host 127.0.0.1,jesse,lol

ssh > connect

ssh > run uptime

host: 127.0.0.1: 14:49 up 11 days, 4:27, 8 users,

load averages: 0.36 0.25 0.19

ssh > close

This is just designed to be a proof-of concept of a pseudo-interactive shell. There are a few improvements you could make should you use it:

- Better printing for multi-line stdout output. - Handle standard error - Add in a quit method - Thread the command execution/data returned.

Like all shells, the sky is the limit when it comes to data visualization. Tools like pssh, OSH, Fabric, etc., all manage the return data differently, and they all have different ways of aggregating the output from different hosts.

**File put and get**

File manipulation within Paramiko is handled via the SFTP implementation, and, like the ssh client command execution, it's easy as pie.

We start by instantiating a new paramiko.SSHClient just as before:

import paramiko

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(

paramiko.AutoAddPolicy())

ssh.connect('127.0.0.1', username='jesse',

password='lol')

This time, we make a call into ''open\_sftp()'' after we perform the connect to the host. ''open\_sftp()'' returns a ''paramiko.SFTPClient'' client object that supports all of the normal sftp operations (stat, put, get, etc.). In this example, we perform a "get" operation to download the file ''remotefile.py'' from the remote system and write it to to the local file, ''localfile.py''.

ftp = ssh.open\_sftp() ftp.get('remotefile.py', 'localfile.py') ftp.close()

Writing a file to the remote host (a "put" operation) works the exact same way. We just transpose the local and remote arguments:

ftp = ssh.open\_sftp()

ftp.get('localfile.py', 'remotefile.py')

ftp.close()

The nice thing about the sftp client implementation that Paramiko provides is that it support things like stat, chmod, chown, etc. Obviously these might act differently depending on the remote server because some servers do not implement all of the protocol, but even so they're incredibly useful.

You could easily write functions like ''glob.glob()'' to transverse a remote directory tree looking for a particular filename pattern. You could also search based on permissions, size, etc.

One thing to note, however, and this bit me a few times: sftp as a protocol is slightly more restrictive than something like normal secure copy (scp). SCP allows you to use Unix wild cards in the file name when grabbing a file from the remote machine. SFTP, on the other hand, expects the full explicit path to the file you want to download. An example of this is:

ftp.get('\*.py', '.')

In most cases, this would mean "download all files with .py" to the local directory on my machine. SFTP is unhappy with this formulation, though (see Listing 2). I learned this the hard way, after I spent several hours pulling apart the sftp client implementation out of frustration.

Listing 2:

>>> ftp.get("./\*.py", '.')

Traceback (most recent call last):

File "", line 1, in

File "/Library/Python/2.5/site-packages/paramiko/sftp\_client.py",

line 567, in get

fr = self.file(remotepath, 'rb')

File "/Library/Python/2.5/site-packages/paramiko/sftp\_client.py",

line 238, in open

t, msg = self.\_request(CMD\_OPEN, filename, imode, attrblock)

File "/Library/Python/2.5/site-packages/paramiko/sftp\_client.py",

line 589, in \_request

return self.\_read\_response(num)

File "/Library/Python/2.5/site-packages/paramiko/sftp\_client.py",

line 636, in \_read\_response

self.\_convert\_status(msg)

File "/Library/Python/2.5/site-packages/paramiko/sftp\_client.py",

line 662, in \_convert\_status

raise IOError(errno.ENOENT, text)

IOError: [Errno 2] No such file

**In Closing**

I hope I've shown you enough to really dig into Paramiko. It's one of the gems from the Python community that helps me on a daily basis. I can do remote administration programmatically, write test plugins that perform remote operations easily, and a lot more, all without needing to install extra daemons on the remote machines.

SSH is everywhere, and sooner or later you're going to need to write a program that interacts with it. Why not save yourself the trouble now and give Paramiko a look?

[Python] paramiko examples

[View **scp\_demo.py**](https://gist.github.com/mlafeldt/841944)

[**scp\_demo.py**](https://gist.github.com/mlafeldt/841944#file-scp_demo-py)

[**Raw**](https://gist.github.com/mlafeldt/841944/raw/704037515d36bee694fdcacae28e8a8fc303de55/scp_demo.py)

File suppressed. Click to show.

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324 | #!/usr/bin/env python    import sys, paramiko    if len(sys.argv) < 5:  print "args missing"  sys.exit(1)    hostname = sys.argv[1]  password = sys.argv[2]  source = sys.argv[3]  dest = sys.argv[4]    username = "root"  port = 22    try:  t = paramiko.Transport((hostname, port))  t.connect(username=username, password=password)  sftp = paramiko.SFTPClient.from\_transport(t)  sftp.get(source, dest)    finally:  t.close() |

[View **scp\_demo.py**](https://gist.github.com/mlafeldt/841944)

[**ssh\_demo.py**](https://gist.github.com/mlafeldt/841944#file-ssh_demo-py)

[**Raw**](https://gist.github.com/mlafeldt/841944/raw/83d23355f2552db075d38d7347feea06a4507a04/ssh_demo.py)

File suppressed. Click to show.

|  |  |
| --- | --- |
| 123456789101112131415161718192021222324252627 | #!/usr/bin/env python    import sys, paramiko    if len(sys.argv) < 4:  print "args missing"  sys.exit(1)    hostname = sys.argv[1]  password = sys.argv[2]  command = sys.argv[3]    username = "admin"  port = 22    try:  client = paramiko.SSHClient()  client.load\_system\_host\_keys()  client.set\_missing\_host\_key\_policy(paramiko.WarningPolicy)    client.connect(hostname, port=port, username=username, password=password)    stdin, stdout, stderr = client.exec\_command(command)  print stdout.read(),    finally:  client.close() |