Parallel-SSH Documentation

Release 0+unknown

P Kittenis

Jun 13, 2017

Contents

CHAPTER 1

Parallel-SSH Documentation

Parallel-SSH is a parallel SSH client library. It uses asynchronous SSH connections and is, to date, the only publicly available asynchronous SSH client library for Python, as well as the only asynchronous *parallel* SSH client library available for Python.

Design And Goals

`Parallel-SSH`'s design goals and motivation are to provide a *library* for running *asynchronous* SSH commands in parallel with little to no load induced on the system by doing so with the intended usage being completely programmatic and non-interactive.

To meet these goals, API driven solutions are preferred first and foremost. This frees up the developer to drive the library via any method desired, be that environment variables, CI driven tasks, command line tools, existing OpenSSH or new configuration files, from within an application et al.

Design Principles

Taking a cue from PEP 20, heavy emphasis is in the following areas.

- Readability
- Explicit is better than implicit
- Simple is better than complex
- Beautiful is better than ugly

Contributions are asked to keep these in mind.

Installation

Installation is handled by Python's standard *setuptools* library and *pip*.

Pip Install

```
pip install parallel-ssh
```

If *pip* is not available on your Python platform, see this installation guide.

Old Python Versions

1.1.x and above releases will not be guaranteed to be compatible with Python *2.6*.

If you are running a deprecated Python version such as *2.6* you may need to install an older version of *parallel-ssh* that is compatible with your Python platform.

For example, to install the *1.0.0* version, run the following.

```
pip install parallel-ssh==1.0.0
```

1.0.0 is compatible with all Python versions over or equal to *2.6*, including all of the *3.x* series.

Older versions such as *0.70.x* are compatible with Python *2.5* but not the *3.x* series.

Source Code

Parallel-SSH is hosted on GitHub and the repository can be cloned with the following:

```
git clone git@github.com:ParallelSSH/parallel-ssh.git
```

To install from source run:

```
python setup.py install
```

Or with *pip*'s development mode which will ensure local changes are made available:

```
pip install -e .
```

Contents

Quickstart

First, make sure that `parallel-ssh` is installed.

Note: ParallelSSH uses gevent's monkey patching to enable asynchronous use of the Python standard library's network I/O.

Make sure that ParallelSSH imports come **before** any other imports in your code. Otherwise, patching may not be done before the standard library is loaded which will then cause ParallelSSH to block.

If you are seeing messages like `This operation would block forever`, this is the cause.

Run a command on hosts in parallel

The most basic usage of `parallel-ssh` is, unsurprisingly, to run a command on multiple hosts in parallel.

Examples in this documentation will be using `print` as a function, for which a future import is needed in Python `2.7` and below.

Make a list or other iterable of the hosts to run on:

```
from __future__ import print_function
from pssh.pssh_client import ParallelSSHClient

hosts = ['host1', 'host2', 'host3', 'host4']
```

Where `host1` to `host4` are valid host names. IP addresses may also be used.

Create a client for these hosts:

```
client = ParallelSSHClient(hosts)
```

The client object can, and should, be reused. Existing connections to hosts will remain alive as long as the client object is kept alive. Subsequent commands to the same host(s) will reuse their existing connection and benefit from much faster response times.

Now one or more commands can be run via the client:

```
output = client.run_command('whoami')
```

Once the call to `run_command` returns, the command has started executing in parallel.

Output is keyed by host and contains a host output object. From that, SSH output is available.

Standard Output

Standard output, aka `stdout` for `host1`:

```
for line in output['host1'].stdout:
    print(line)
```

Output

```
<your username here>
```

There is nothing special needed to ensure output is available.

Please note that retrieving all of a command's standard output by definition requires that the command has completed.

Iterating over `stdout` for any host *to completion* will therefor *block* until that host's command has completed unless interrupted.

`stdout` is a generator. Iterating over it will consume the remote standard output stream via the network as it becomes available. To store all of stdout can wrap it with list, per below.

```
stdout = list(output['host1'].stdout)
```

Warning: This will store the entirety of stdout into memory and may exhaust available memory if command output is large enough.

All hosts iteration

Of course, iterating over all hosts can also be done the same way.

```
for host, host_output in output.items():
    for line in host_output.stdout:
        print("Host [%s] - %s" % (host, line))
```

Exit codes

Exit codes are available on the host output object.

First, ensure that all commands have finished and exit codes gathered by joining on the output object, then iterate over all host's output to print their exit codes.

```
client.join(output)
for host, host_output in output.items():
    print("Host %s exit code: %s" % (host, host_output.exit_code))
```

See also:

pssh.output.HostOutput Host output class documentation.

Authentication

By default `parallel-ssh` will use an available SSH agent's credentials to login to hosts via private key authentication.

User/Password authentication

User/password authentication can be used by providing user name and password credentials:

```
client = ParallelSSHClient(hosts, user='my_user', password='my_pass')
```

Programmatic Private Key authentication

It is also possible to programmatically use a private key for authentication.

The helper function *load_private_key* is provided to easily load all possible key types. It takes either a file path or a file-like object.

File path

```
from pssh.pssh_client import ParallelSSHClient
from pssh.utils import load_private_key

pkey = load_private_key('my_pkey.pem')
client = ParallelSSHClient(hosts, pkey=pkey)
```

Host Logger

There is a built in host logger that can be enabled to automatically log output from remote hosts. This requires the `consume_output` flag to be enabled on `run_command`.

The helper function `pssh.utils.enable_host_logger` will enable host logging to standard output, for example:

```
from pssh.utils import enable_host_logger
enable_host_logger()
client.join(client.run_command('uname'), consume_output=True)
```

Output

```
[localhost]       Linux
```

Using standard input

Along with standard output and error, input is also available on the host output object. It can be used to send input to the remote host where required, for example password prompts or any other prompt requiring user input.

The `stdin` attribute is a file-like object giving access to the remote stdin channel that can be written to:

```
output = client.run_command('read')
stdin = output['localhost'].stdin
stdin.write("writing to stdin\\n")
stdin.flush()
for line in output['localhost'].stdout:
    print(line)
```

Output

```
writing to stdin
```

Errors and Exceptions

By default, `parallel-ssh` will fail early on any errors connecting to hosts, whether that be connection errors such as DNS resolution failure or unreachable host, SSH authentication failures or any other errors.

Alternatively, the `stop_on_errors` flag is provided to tell the client to go ahead and attempt the command(s) anyway and return output for all hosts, including the exception on any hosts that failed:

```
output = client.run_command('whoami', stop_on_errors=False)
```

With this flag, the `exception` attribute will contain the exception on any failed hosts, or `None`:

```
client.join(output)
for host, host_output in output.items():
    print("Host %s: exit code %s, exception %s" % (
          host, host_output.exit_code, host_output.exception))
```

Output

```
host1: 0, None
host2: None, AuthenticationException <..>
```

Possible exceptions can be found in *pssh.exceptions* module.

Contents

Advanced Usage

There are several more advanced use cases of *ParallelSSH*, such as tunneling (aka proxying) via an intermediate SSH server and per-host configuration and command substitution among others.

Agents and Private Keys

SSH Agent forwarding

SSH agent forwarding, what `ssh -A` does on the command line, is supported and enabled by default. Creating an object as `ParallelSSHClient(hosts, forward_ssh_agent=False)` will disable this behaviour.

Programmatic Private Keys

By default, `ParallelSSH` will use all keys in an available SSH agent and identity keys under the user's SSH directory - `id_rsa` and `id_dsa` in `~/.ssh`.

A private key can also be provided programmatically.

```
from pssh.utils import load_private_key
from pssh import ParallelSSHClient

client = ParallelSSHClient(hosts, pkey=load_private_key('my_key'))
```

Where `my_key` is a private key file in current working directory.

The helper function *`load_private_key`* will attempt to load all available key types and raises *`SSHException`* if it cannot load the key file.

Disabling use of system SSH Agent

Use of an available SSH agent can also be disabled.

```
client = ParallelSSHClient(hosts, pkey=load_private_key('my_key'),
                           allow_agent=False)
```

Warning: For large number of hosts, it is recommended that private keys are provided programmatically and use of SSH agent is disabled via `allow_agent=False` as above.

If the number of hosts is large enough, available connections to the system SSH may be exhausted which will stop the client from working on a subset of hosts.

This is a limitation of the underlying SSH client used by `ParallelSSH`.

Programmatic SSH Agent

It is also possible to programmatically provide an SSH agent for the client to use, instead of a system provided one. This is useful in cases where hosts need different private keys and a system SSH agent is not available.

```
from pssh.agent import SSHAgent
from pssh.utils import load_private_key
from pssh import ParallelSSHClient

agent = SSHAgent()
agent.add_key(load_private_key('my_private_key_filename'))
agent.add_key(load_private_key('my_other_private_key_filename'))
hosts = ['my_host', 'my_other_host']

client = ParallelSSHClient(hosts, agent=agent)
client.run_command(<..>)
```

Note: Supplying an agent programmatically implies that a system SSH agent will *not* be used even if available.

Tunneling

This is used in cases where the client does not have direct access to the target host and has to authenticate via an intermediary, also called a bastion host, commonly used for additional security as only the bastion host needs to have access to the target host.

ParallelSSHClient ——> Proxy host ——> Target host

Proxy host can be configured as follows in the simplest case:

```
hosts = [<..>]
client = ParallelSSHClient(hosts, proxy_host=bastion)
```

Configuration for the proxy host's user name, port, password and private key can also be provided, separate from target host user name.

```
from pssh.utils import load_private_key

hosts = [<..>]
client = ParallelSSHClient(hosts, user='target_host_user',
                           proxy_host=bastion, proxy_user='my_proxy_user',
                           proxy_port=2222,
                           proxy_pkey=load_private_key('proxy.key'))
```

Where `proxy.key` is a filename containing private key to use for proxy host authentication.

In the above example, connection to the target host is made via `my_proxy_user@bastion -> target_host_user@<host>`.

Note: Proxy host connections are asynchronous and use the SSH protocol's native TCP tunneling - aka local port forward. No external commands are used for the proxy connection, unlike the *ProxyCommand* directive in OpenSSH and other utilities.

While the connections initiated by `ParallelSSH` are asynchronous, the connections from proxy host -> target hosts may not be, depending on SSH server implementation. If only one proxy host is used to connect to a large number of target hosts and proxy SSH server connections are *not* asynchronous, this may adversely impact performance on the proxy host.

Per-Host Configuration

Sometimes, different hosts require different configuration like user names and passwords, ports and private keys. Capability is provided to supply per host configuration for such cases.

```
from pssh.utils import load_private_key

host_config = {'host1' : {'user': 'user1', 'password': 'pass',
                          'port': 2222,
                          'private_key': load_private_key(
                              'my_key.pem')},
               'host2' : {'user': 'user2', 'password': 'pass',
                          'port': 2223,
                          'private_key': load_private_key(
                              open('my_other_key.pem'))},
               }
hosts = host_config.keys()
```

```
client = ParallelSSHClient(hosts, host_config=host_config)
client.run_command('uname')
<..>
```

In the above example, `host1` will use user name `user1` and private key from `my_key.pem` and `host2` will use user name `user2` and private key from `my_other_key.pem`.

Note: Proxy host cannot be provided via per-host configuration at this time.

Per-Host Command substitution

For cases where different commands should be run each host, or the same command with different arguments, functionality exists to provide per-host command arguments for substitution.

The `host_args` keyword parameter to *`run_command`* can be used to provide arguments to use to format the command string.

Number of `host_args` items should be at least as many as number of hosts.

Any Python string format specification characters may be used in command string.

In the following example, first host in hosts list will use cmd `host1_cmd` second host `host2_cmd` and so on

```
output = client.run_command('%s', host_args=('host1_cmd',
                                             'host2_cmd',
                                             'host3_cmd',))
```

Command can also have multiple arguments to be substituted.

```
output = client.run_command('%s %s',
host_args=(('host1_cmd1', 'host1_cmd2'),
           ('host2_cmd1', 'host2_cmd2'),
           ('host3_cmd1', 'host3_cmd2'),))
```

A list of dictionaries can also be used as `host_args` for named argument substitution.

In the following example, first host in host list will use cmd `host-index-0`, second host `host-index-1` and so on.

```
host_args=[{'cmd': 'host-index-%s' % (i,))
           for i in range(len(client.hosts))]
output = client.run_command('%(cmd)s', host_args=host_args)
```

Hosts filtering and overriding

Iterators and filtering

Any type of iterator may be used as hosts list, including generator and list comprehension expressions.

List comprehension

```
hosts = ['dc1.myhost1', 'dc2.myhost2']
client = ParallelSSHClient([h for h in hosts if h.find('dc1')])
```

Generator

```
hosts = ['dc1.myhost1', 'dc2.myhost2']
client = ParallelSSHClient((h for h in hosts if h.find('dc1')))
```

Filter

```
hosts = ['dc1.myhost1', 'dc2.myhost2']
client = ParallelSSHClient(filter(lambda h: h.find('dc1'), hosts))
client.run_command(<..>)
```

Note: Since generators by design only iterate over a sequence once then stop, `client.hosts` should be re-assigned after each call to `run_command` when using generators as target of *client.hosts*.

Overriding hosts list

Hosts list can be modified in place. A call to `run_command` will create new connections as necessary and output will only contain output for the hosts `run_command` executed on.

```
client.hosts = ['otherhost']
print(client.run_command('exit 0'))
{'otherhost': exit_code=None, <..>}
```

API Documentation

ParallelSSHClient

Package containing ParallelSSHClient class

class pssh.pssh_client.**ParallelSSHClient**(*hosts*, *user=None*, *password=None*, *port=None*, *pkey=None*, *forward_ssh_agent=True*, *num_retries=3*, *timeout=120*, *pool_size=10*, *proxy_host=None*, *proxy_port=22*, *proxy_user=None*, *proxy_password=None*, *proxy_pkey=None*, *agent=None*, *allow_agent=True*, *host_config=None*, *channel_timeout=None*)

Uses *pssh.ssh_client.SSHClient*, performs tasks over SSH on multiple hosts in parallel.

Connections to hosts are established in parallel when `run_command` is called, therefor any connection and/or authentication exceptions will happen on the call to `run_command` and need to be handled there.

Parameters

- **hosts** (*list(str)*) – Hosts to connect to
- **user** (*str*) – (Optional) User to login as. Defaults to logged in user or user from ~/.ssh/config or /etc/ssh/ssh_config if set
- **password** (*str*) – (Optional) Password to use for login. Defaults to no password
- **port** (*int*) – (Optional) Port number to use for SSH connection. Defaults to `None` which uses SSH default

- **pkey** (paramiko.pkey.PKey) – (Optional) Client's private key to be used to connect with
- **num_retries** (*int*) – (Optional) Number of retries for connection attempts before the client gives up. Defaults to 3.
- **timeout** (*int*) – (Optional) Number of seconds to wait before connection and authentication attempt times out. Note that total time before timeout will be `timeout * num_retries + (5 * (num_retries-1))` number of seconds, where (5 * (num_retries-1)) refers to a five (5) second delay between retries.
- **forward_ssh_agent** (*bool*) – (Optional) Turn on/off SSH agent forwarding - equivalent to *ssh -A* from the *ssh* command line utility. Defaults to `True` if not set.
- **pool_size** (*int*) – (Optional) Greenlet pool size. Controls on how many hosts to execute tasks in parallel. Defaults to 10. Overhead in event loop will determine how high this can be set to, see scaling guide lines in project's readme.
- **proxy_host** (*str*) – (Optional) SSH host to tunnel connection through so that SSH clients connect to host via client -> proxy_host -> host
- **proxy_port** (*int*) – (Optional) SSH port to use to login to proxy host if set. Defaults to 22.
- **proxy_user** (*str*) – (Optional) User to login to `proxy_host` as. Defaults to logged in user.
- **proxy_password** (*str*) – (Optional) Password to login to `proxy_host` with. Defaults to no password
- **proxy_pkey** (paramiko.pkey.PKey) – (Optional) Private key to be used for authentication with `proxy_host`. Defaults to available keys from SSHAgent and user's home directory keys
- **agent** (*pssh.agent.SSHAgent*) – (Optional) SSH agent object to programmatically supply an agent to override system SSH agent with
- **host_config** (*dict*) – (Optional) Per-host configuration for cases where not all hosts use the same configuration values.
- **channel_timeout** (*int*) – (Optional) Time in seconds before reading from an SSH channel times out. For example with channel timeout set to one, trying to immediately gather output from a command producing no output for more than one second will timeout.
- **allow_agent** (*bool*) – (Optional) set to False to disable connecting to the system's SSH agent

Example Usage

```
from __future__ import print_function
from pprint import pprint

from pssh.pssh_client import ParallelSSHClient
from pssh.exceptions import AuthenticationException, \
    UnknownHostException, ConnectionErrorException

client = ParallelSSHClient(['myhost1', 'myhost2'])
try:
    output = client.run_command('ls -ltrh /tmp/aasdfasdf', sudo=True)
except (AuthenticationException, UnknownHostException,
        ConnectionErrorException):
    pass
```

Commands have started executing at this point. Exit code(s) will not be available immediately.

```
pprint(output)
  {'myhost1':
        host=myhost1
        exit_code=None
        cmd=<Greenlet>
        channel=<channel>
        stdout=<generator>
        stderr=<generator>
        stdin=<channel>
        exception=None
   'myhost2':
        host=myhost2
        exit_code=None
        cmd=<Greenlet>
        channel=<channel>
        stdout=<generator>
        stderr=<generator>
        stdin=<channel>
        exception=None
  }
```

Enabling host logger

There is a host logger in parallel-ssh that can be enabled to show stdout from remote commands on hosts as it comes in.

This allows for stdout to be automatically logged without having to print it serially per host. `pssh.utils.host_logger` is a standard library logger and may be configured to log to anywhere else.

For host logger to log output, `join` must be called with `consume_output=True`

```
import pssh.utils
pssh.utils.enable_host_logger()

output = client.run_command('ls -ltrh')
client.join(output, consume_output=True)
[myhost1]     drwxrwxr-x 6 user group 4.0K Jan 1 HH:MM x
[myhost2]     drwxrwxr-x 6 user group 4.0K Jan 1 HH:MM x
```

Retrieve exit codes after commands have finished as below.

`exit_code` in `output` will be `None` immediately after call to `run_command`.

parallel-ssh starts commands asynchronously to enable starting multiple commands in parallel without blocking.

Because of this, exit codes will not be immediately available even for commands that exit immediately.

Waiting for command completion

At least one of

- Iterating over stdout/stderr to completion
- Calling `client.join(output)`

is necessary to cause `parallel-ssh` to wait for commands to finish and be able to gather exit codes.

An individual command's exit code can be gathered by `get_exit_code(host_output)`

See also:

get_exit_code(), *get_output()*

Note: Joining on client's gevent pool

`client.pool.join()` only blocks *until greenlets have been spawned* which will be immediately as long as pool is not full.

Checking command completion

To check if commands have finished *without blocking* use

```
client.finished(output)
False
```

which returns `True` if and only if all commands in output have finished.

For individual commands the status of channel can be checked

```
output[host].channel.closed
False
```

which returns `True` if command has finished.

Either iterating over stdout/stderr or `client.join(output)` will cause exit codes to become available in output without explicitly calling `get_exit_codes`.

Use `client.join(output)` to block until all commands have finished and gather exit codes at same time.

In versions prior to `1.0.0` only, `client.join` would consume output.

Exit code retrieval

`get_exit_codes` is not a blocking function and will not wait for commands to finish.

`output` parameter is modified in-place.

```
client.get_exit_codes(output)
for host in output:
    print(output[host].exit_code)
0
0
```

Stdout from each host

```
for host in output:
    for line in output[host].stdout:
        print(line)
ls: cannot access /tmp/aasdfasdf: No such file or directory
ls: cannot access /tmp/aasdfasdf: No such file or directory
```

Example with specified private key

```
from pssh.utils import load_private_key
client_key = load_private_key('user.key')
client = ParallelSSHClient(['myhost1', 'myhost2'], pkey=client_key)
```

Multiple commands

```
for cmd in ['uname', 'whoami']:
    client.run_command(cmd)
```

Per-Host configuration

Per host configuration can be provided for any or all of user, password port and private key. Private key value is a `paramiko.pkey.PKey` object as returned by *`pssh.utils.load_private_key()`*.

`pssh.utils.load_private_key()` accepts both file names and file-like objects and will attempt to load all available key types, returning `None` if they all fail.

```
from pssh.utils import load_private_key

host_config = { 'host1' : {'user': 'user1', 'password': 'pass',
                           'port': 2222,
                           'private_key': load_private_key(
                               'my_key.pem')},
                'host2' : {'user': 'user2', 'password': 'pass',
                           'port': 2223,
                           'private_key': load_private_key(
                               open('my_other_key.pem'))},
                }
hosts = host_config.keys()

client = ParallelSSHClient(hosts, host_config=host_config)
client.run_command('uname')
<..>
```

Note: Connection persistence

Connections to hosts will remain established for the duration of the object's life. To close them, just *del* or reuse the object reference

```
client = ParallelSSHClient(['localhost'])
output = client.run_command('ls')
```

netstat `tcp 0 0 127.0.0.1:53054 127.0.0.1:22 ESTABLISHED`

Connection remains active after commands have finished executing. Any additional commands will reuse the same connection.

```
del client
```

Connection is terminated.

`copy_file`(*local_file*, *remote_file*, *recurse=False*)

Copy local file to remote file in parallel

This function returns a list of greenlets which can be *join*-ed on to wait for completion.

`gevent.joinall()` function may be used to join on all greenlets and will also raise exceptions from them if called with `raise_error=True` - default is *False*.

Alternatively call *.get* on each greenlet to raise any exceptions from it.

Exceptions listed here are raised when either `gevent.joinall(<greenlets>, raise_error=True)` is called or `.get` is called on each greenlet, not this function itself.

Parameters

- **local_file** (*str*) – Local filepath to copy to remote host
- **remote_file** (*str*) – Remote filepath on remote host to copy file to
- **recurse** (*bool*) – Whether or not to descend into directories recursively.

Return type List(gevent.Greenlet) of greenlets for remote copy commands

Raises ValueError when a directory is supplied to local_file and recurse is not set

Raises IOError on I/O errors writing files

Raises OSError on OS errors like permission denied

Note: Remote directories in *remote_file* that do not exist will be created as long as permissions allow.

---

copy_remote_file(*remote_file*, *local_file*, *recurse=False*, *suffix_separator='_'*)

Copy remote file(s) in parallel as <local_file><suffix_separator><host>

With a local_file value of myfile and default separator _ the resulting filename will be myfile_myhost for the file from host myhost.

This function, like *ParallelSSHClient.copy_file()*, returns a list of greenlets which can be *join*-ed on to wait for completion.

gevent.joinall() function may be used to join on all greenlets and will also raise exceptions if called with raise_error=True - default is *False*.

Alternatively call *.get* on each greenlet to raise any exceptions from it.

Exceptions listed here are raised when either gevent.joinall(<greenlets>, raise_error=True) is called or .get is called on each greenlet, not this function itself.

Parameters

- **remote_file** (*str*) – remote filepath to copy to local host
- **local_file** (*str*) – local filepath on local host to copy file to
- **recurse** (*bool*) – whether or not to recurse
- **suffix_separator** (*str*) – (Optional) Separator string between filename and host, defaults to _. For example, for a local_file value of myfile and default separator the resulting filename will be myfile_myhost for the file from host myhost

Return type list(gevent.Greenlet) of greenlets for remote copy commands

Raises ValueError when a directory is supplied to local_file and recurse is not set

Raises IOError on I/O errors writing files

Raises OSError on OS errors like permission denied

Note: Local directories in *local_file* that do not exist will be created as long as permissions allow.

Note: File names will be de-duplicated by appending the hostname to the filepath separated by suffix_separator.

---

finished(*output*)
Check if commands have finished without blocking

Parameters **output** – As returned by *pssh.pssh_client.ParallelSSHClient.get_output()*

Return type bool

get_exit_code(*host_output*)
Get exit code from host output *if available*.

Parameters **host_output** – Per host output as returned by *pssh.pssh_client.ParallelSSHClient.get_output()*

Return type int or None if exit code not ready

get_exit_codes(*output*)
Get exit code for all hosts in output *if available*. Output parameter is modified in-place.

Parameters **output** – As returned by *pssh.pssh_client.ParallelSSHClient.get_output()*

Return type None

get_output(*cmd*, *output*, *encoding='utf-8'*)
Get output from command.

Parameters

- **cmd** (gevent.Greenlet) – Command to get output from
- **output** (*dict*) – Dictionary containing *pssh.output.HostOutput* values to be updated with output from cmd

Return type None

output parameter is modified in-place and has the following structure

```
{'myhost1':
      exit_code=exit code if ready else None
      channel=SSH channel of command
      stdout=<iterable>
      stderr=<iterable>
      cmd=<greenlet>
      exception=<exception object if applicable>
}
```

Stdout and stderr are also logged via the logger named `host_logger` which can be enabled by calling `enable_host_logger`

Example usage:

```
output = client.get_output()
for host in output:
    for line in output[host].stdout:
        print(line)
<stdout>
# Get exit code for a particular host's output after command
# has finished
self.get_exit_code(output[host])
0
```

join(*output*, *consume_output=False*)

Block until all remote commands in output have finished and retrieve exit codes

Parameters

- **output** (dict as returned by *pssh.pssh_client.ParallelSSHClient.get_output()*) – Output of commands to join on
- **consume_output** (*bool*) – Whether or not join should consume output buffers. Output buffers will be empty after `join` if set to `True`. Must be set to `True` to allow host logger to log output on call to `join`.

Enabling host logger

```
from pssh.utils import enable_host_logger
enable_host_logger()
output = client.run_command(<..>)
client.join(output, consume_output=True)

# Output buffers now empty
len(list(output[client.hosts[0]].stdout)) == 0
```

With `consume_output=True`, host logger logs output.

```
[my_host1] <..>
```

With `consume_output=False`, the default, iterating over output is needed for host logger to log anything.

```
output = client.run_command(<..>)
client.join(output, consume_output=False)
for host, host_out in output.items():
    for line in host_out.stdout:
        pass
```

```
[my_host1] <..>
```

run_command(*command*, *sudo=False*, *user=None*, *stop_on_errors=True*, *shell=None*, *use_shell=True*, *use_pty=True*, *host_args=None*, *encoding='utf-8'*)

Run command on all hosts in parallel, honoring self.pool_size, and return output buffers.

This function will block until all commands have been *sent* to remote servers and then return immediately

More explicitly, function will return after connection and authentication establishment and after commands have been sent to successfully established SSH channels.

Any connection and/or authentication exceptions will be raised here and need catching *unless* `run_command` is called with `stop_on_errors=False` in which case exceptions are added to host output instead.

Parameters

- **command** (*str*) – Command to run
- **sudo** (*bool*) – (Optional) Run with sudo. Defaults to False
- **user** (*str*) – (Optional) User to run command as. Requires sudo access for that user from the logged in user account.
- **stop_on_errors** (*bool*) – (Optional) Raise exception on errors running command. Defaults to True. With stop_on_errors set to False, exceptions are instead added to output of *run_command*. See example usage below.

- **shell** (*str*) – (Optional) Override shell to use to run command with. Defaults to login user's defined shell. Use the shell's command syntax, eg *shell='bash -c'* or *shell='zsh -c'*.
- **use_shell** (*bool*) – (Optional) Run command with or without shell. Defaults to True - use shell defined in user login to run command string
- **use_pty** (*bool*) – (Optional) Enable/Disable use of pseudo terminal emulation. Disabling it will prohibit capturing standard input/output. This is required in majority of cases, exceptions being where a shell is not used and/or input/output is not required. In particular when running a command which deliberately closes input/output pipes, such as a daemon process, you may want to disable `use_pty`. Defaults to `True`
- **host_args** (*tuple or list*) – (Optional) Format command string with per-host arguments in `host_args`. `host_args` length must equal length of host list - *pssh.exceptions.HostArgumentException* is raised otherwise
- **encoding** (*str*) – Encoding to use for output. Must be valid Python codec

Return type Dictionary with host as key and *pssh.output.HostOutput* as value as per *pssh.pssh_client.ParallelSSHClient.get_output()*

Raises *pssh.exceptions.AuthenticationException* on authentication error

Raises *pssh.exceptions.UnknownHostException* on DNS resolution error

Raises *pssh.exceptions.ConnectionErrorException* on error connecting

Raises *pssh.exceptions.SSHException* on other undefined SSH errors

Raises *pssh.exceptions.HostArgumentException* on number of host arguments not equal to number of hosts

Raises `TypeError` on not enough host arguments for cmd string format

Raises `KeyError` on no host argument key in arguments dict for cmd string format

Example Usage

Simple run command

```
output = client.run_command('ls -ltrh')
```

Print stdout for each command

```
from __future__ import print_function

for host in output:
    for line in output[host].stdout:
        print(line)
```

Get exit codes after command has finished

```
from __future__ import print_function

client.get_exit_codes(output)
for host in output:
    print(output[host].exit_code)
0
0
```

Wait for completion, print exit codes

```
client.join(output)
print(output[host].exit_code)
0
for line in output[host].stdout:
    print(line)
```

Run with sudo

```
output = client.run_command('ls -ltrh', sudo=True)
```

Capture stdout

Warning: This will store the entirety of stdout into memory and may exhaust available memory if command output is large enough.

Iterating over stdout/stderr to completion by definition implies blocking until command has finished. To only log output as it comes in without blocking the host logger can be enabled - see *Enabling Host Logger* above.

```
from __future__ import print_function

for host in output:
    stdout = list(output[host].stdout)
    print("Complete stdout for host %s is %s" % (host, stdout,))
```

Command with per-host arguments

`host_args` keyword parameter can be used to provide arguments to use to format the command string.

Number of `host_args` should be at least as many as number of hosts.

Any string format specification characters may be used in command string.

Examples

```
# Tuple
#
# First host in hosts list will use cmd 'host1_cmd',
# second host 'host2_cmd' and so on
output = client.run_command('%s', host_args=('host1_cmd',
                                             'host2_cmd',
                                             'host3_cmd',))

# Multiple arguments
#
output = client.run_command('%s %s',
                            host_args=(('host1_cmd1', 'host1_cmd2'),
                                       ('host2_cmd1', 'host2_cmd2'),
                                       ('host3_cmd1', 'host3_cmd2'),))

# List of dict
#
```

```
# First host in host list will use cmd 'host-index-0',
# second host 'host-index-1' and so on
output = client.run_command(
  '%(cmd)s', host_args=[{'cmd': 'host-index-%s' % (i,))
                        for i in range(len(client.hosts))])
```

Expression as host list

Any type of iterator may be used as host list, including generator and list comprehension expressions.

```
hosts = ['dc1.myhost1', 'dc2.myhost2']
# List comprehension
client = ParallelSSHClient([h for h in hosts if h.find('dc1')])
# Generator
client = ParallelSSHClient((h for h in hosts if h.find('dc1')))
# Filter
client = ParallelSSHClient(filter(lambda h: h.find('dc1'), hosts))
client.run_command(<..>)
```

Note: Since generators by design only iterate over a sequence once then stop, `client.hosts` should be re-assigned after each call to `run_command` when using generators as target of *client.hosts*.

Overriding host list

Host list can be modified in place. Call to *run_command* will create new connections as necessary and output will only contain output for the hosts `run_command` executed on.

```
client.hosts = ['otherhost']
print(client.run_command('exit 0'))
{'otherhost': exit_code=None, <..>}
```

Run multiple commands in parallel

This short example demonstrates running multiple long running commands in parallel on the same host, how long it takes for all commands to start, blocking until they complete and how long it takes for all commands to complete.

See examples directory for complete script.

```
output = []
host = 'localhost'

# Run 10 five second sleeps
cmds = ['sleep 5' for _ in xrange(10)]
start = datetime.datetime.now()
for cmd in cmds:
    output.append(client.run_command(cmd, stop_on_errors=False))
end = datetime.datetime.now()
print("Started %s commands in %s" % (len(cmds), end-start,))
start = datetime.datetime.now()
for _output in output:
    for line in _output[host].stdout:
        print(line)
```

```
end = datetime.datetime.now()
print("All commands finished in %s" % (end-start,))
```

Output

```
Started 10 commands in 0:00:00.428629
All commands finished in 0:00:05.014757
```

Output format

```
{'myhost1':
      host=myhost1
      exit_code=exit code if ready else None
      channel=SSH channel of command
      stdout=<iterable>
      stderr=<iterable>
      stdin=<file-like writable channel>
      cmd=<greenlet>
      exception=None}
```

Do not stop on errors, return per-host exceptions in output

```
output = client.run_command('ls -ltrh', stop_on_errors=False)
client.join(output)
print(output)
```

```
{'myhost1':
      host=myhost1
      exit_code=None
      channel=None
      stdout=None
      stderr=None
      cmd=None
      exception=ConnectionErrorException(
                  "Error connecting to host '%s:%s' - %s - "
                  "retry %s/%s",
                   host, port, 'Connection refused', 3, 3)}
```

Using stdin

```
output = client.run_command('read')
stdin = output['localhost'].stdin
stdin.write("writing to stdin\n")
stdin.flush()
for line in output['localhost'].stdout:
    print(line)

writing to stdin
```

SSHClient

SSHClient is a single host client and is suitable for talking to a single SSH server asynchronously. All SSH functionality is implemented in SSHClient and it is used by ParallelSSHClient. Package containing SSHClient class.

class pssh.ssh_client.**SSHClient**(*host*, *user=None*, *password=None*, *port=None*, *pkey=None*, *forward_ssh_agent=True*, *num_retries=3*, *agent=None*, *allow_agent=True*, *timeout=10*, *proxy_host=None*, *proxy_port=22*, *proxy_user=None*, *proxy_password=None*, *proxy_pkey=None*, *channel_timeout=None*, *_openssh_config_file=None*)

Wrapper class over paramiko.SSHClient with sane defaults Honours `~/.ssh/config` and `/etc/ssh/ssh_config` host entries for host username overrides

Parameters

- **host** (*str*) – Hostname to connect to
- **user** (*str*) – (Optional) User to login as. Defaults to logged in user or user from ~/.ssh/config if set
- **password** (*str*) – (Optional) Password to use for login. Defaults to no password
- **port** (*int*) – (Optional) Port number to use for SSH connection. Defaults to `None` which uses SSH default
- **pkey** (`paramiko.pkey.PKey`) – (Optional) Client's private key to be used to connect with
- **num_retries** (*int*) – (Optional) Number of retries for connection attempts before the client gives up. Defaults to 3.
- **timeout** (*int*) – (Optional) Number of seconds to timeout connection attempts before the client gives up
- **forward_ssh_agent** (*bool*) – (Optional) Turn on SSH agent forwarding - equivalent to *ssh -A* from the *ssh* command line utility. Defaults to True if not set.
- **agent** (`paramiko.agent.Agent`) – (Optional) Override SSH agent object with the provided. This allows for overriding of the default paramiko behaviour of connecting to local SSH agent to lookup keys with our own SSH agent object.
- **forward_ssh_agent** – (Optional) Turn on SSH agent forwarding - equivalent to *ssh -A* from the *ssh* command line utility. Defaults to True if not set.
- **proxy_host** (*str*) – (Optional) SSH host to tunnel connection through so that SSH clients connects to self.host via client -> proxy_host -> host
- **proxy_port** (*int*) – (Optional) SSH port to use to login to proxy host if set. Defaults to 22.
- **channel_timeout** (*int*) – (Optional) Time in seconds before an SSH operation times out.
- **allow_agent** (*bool*) – (Optional) set to False to disable connecting to the SSH agent

copy_file(*local_file*, *remote_file*, *recurse=False*, *sftp=None*)

Copy local file to host via SFTP/SCP

Copy is done natively using SFTP/SCP version 2 protocol, no scp command is used or required.

Parameters

- **local_file** (*str*) – Local filepath to copy to remote host
- **remote_file** (*str*) – Remote filepath on remote host to copy file to
- **recurse** (*bool*) – Whether or not to descend into directories recursively.

Raises `ValueError` when a directory is supplied to `local_file` and `recurse` is not set

Raises `IOError` on I/O errors writing files

Raises `OSError` on OS errors like permission denied

copy_remote_file(*remote_file*, *local_file*, *recurse=False*, *sftp=None*)
Copy remote file to local host via SFTP/SCP

Copy is done natively using SFTP/SCP version 2, no scp command is used or required.

Parameters

- **remote_file** (*str*) – Remote filepath to copy from
- **local_file** (*str*) – Local filepath where file(s) will be copied to
- **recurse** (*bool*) – Whether or not to recursively copy directories

Raises `ValueError` when a directory is supplied to `local_file` and `recurse` is not set

Raises `IOError` on I/O errors creating directories or file

Raises `OSError` on OS errors like permission denied

exec_command(*command*, *sudo=False*, *user=None*, *shell=None*, *use_shell=True*, *use_pty=True*)
Wrapper to `paramiko.SSHClient.exec_command()`

Opens a new SSH session with a new pty and runs command before yielding the main gevent loop to allow other greenlets to execute.

Parameters

- **command** (*str*) – Command to execute
- **sudo** (*bool*) – (Optional) Run with sudo. Defaults to False
- **user** (*str*) – (Optional) User to switch to via sudo to run command as. Defaults to user running the python process
- **shell** – (Optional) Shell override to use instead of user login configured shell. For example `shell='bash -c'`
- **use_shell** (*bool*) – (Optional) Force use of shell on/off. Defaults to *True* for on
- **use_pty** (*bool*) – (Optional) Enable/Disable use of pseudo terminal emulation. This is required in vast majority of cases, exception being where a shell is not used and/or stdout/stderr/stdin buffers are not required. Defaults to `True`
- **kwargs** (*dict*) – (Optional) Keyword arguments to be passed to remote command

Return type Tuple of *(channel, hostname, stdout, stderr, stdin)*. Channel is the remote SSH channel, needed to ensure all of stdout has been got, hostname is remote hostname the copy is to, stdout and stderr are buffers containing command output and stdin is standard input channel

mkdir(*sftp*, *directory*)
Make directory via SFTP channel.

Parent paths in the directory are created if they do not exist.

Parameters

- **sftp** (`paramiko.sftp_client.SFTPClient`) – SFTP client object
- **directory** (*str*) – Remote directory to create

Catches and logs at error level remote IOErrors on creating directory.

read_output_buffer(*output_buffer*, *prefix=''*, *callback=None*, *callback_args=None*, *encoding='utf-8'*)
Read from output buffers and log to host_logger

Parameters

- **output_buffer** (*iterator*) – Iterator containing buffer
- **prefix** (*str*) – String to prefix log output to `host_logger` with
- **callback** (*function*) – Function to call back once buffer is depleted:
- **callback_args** (*tuple*) – Arguments for call back function

Host Output

Output module of ParallelSSH

class `pssh.output.`**HostOutput**(*host*, *cmd*, *channel*, *stdout*, *stderr*, *stdin*, *exit_code=None*, *exception=None*)
Class to hold host output

Parameters

- **host** (*str*) – Host name output is for
- **cmd** (`gevent.Greenlet`) – Command execution object
- **channel** (`socket.socket` compatible object) – SSH channel used for command execution
- **stdout** (*generator*) – Standard output buffer
- **stderr** (*generator*) – Standard error buffer
- **stdin** (`file()`-like object) – Standard input buffer
- **exit_code** (*int or None*) – Exit code of command
- **exception** (`Exception` or `None`) – Exception from host if any

update(*update_dict*)
Override of dict update function for backwards compatibility

SSH Agent

SSH agent module of ParallelSSH

class `pssh.agent.`**SSHAgent**
`paramiko.agent.Agent` compatible class for programmatically supplying an SSH agent

Example Usage

```
from pssh.agent import SSHAgent
from pssh.utils import load_private_key
from pssh import ParallelSSHClient

agent = SSHAgent()
agent.add_key(load_private_key('my_private_key_filename'))
agent.add_key(load_private_key('my_other_private_key_filename'))
hosts = ['my_host', 'my_other_host']
```

```
client = ParallelSSHClient(hosts, agent=agent)
client.run_command('uname')
```

add_key(*key*)
Add key to agent.

Parameters **key** (paramiko.pkey.PKey) – Key to add

Utility functions

Package containing static utility functions for parallel-ssh module.

pssh.utils.**enable_host_logger**()
Enable host logger for logging stdout from remote commands as it becomes available.

pssh.utils.**enable_logger**(*_logger*, *level=20*)
Enables logging to stdout for given logger

pssh.utils.**load_private_key**(*_pkey*)
Load private key from pkey file object or filename

Parameters **pkey** (*file/str*) – File object or file name containing private key

pssh.utils.**read_openssh_config**(*_host*, *config_file=None*)
Parses user's OpenSSH config for per hostname configuration for hostname, user, port and private key values

Parameters **_host** – Hostname to lookup in config

Exceptions

Exceptions raised by parallel-ssh classes.

exception pssh.exceptions.**AuthenticationException**
Raised on authentication error (user/password/ssh key error)

exception pssh.exceptions.**ConnectionErrorException**
Raised on error connecting (connection refused/timed out)

exception pssh.exceptions.**HostArgumentException**
Raised on errors with per-host command arguments

exception pssh.exceptions.**SSHException**
Raised on SSHException error - error authenticating with SSH server

exception pssh.exceptions.**UnknownHostException**
Raised when a host is unknown (dns failure)

CHAPTER 2

In a nutshell

```
from pssh.pssh_client import ParallelSSHClient

client = ParallelSSHClient(['localhost'])
output = client.run_command('whoami')
for line in output['localhost'].stdout:
    print(line)
```

Output

```
<your username here>
```

Indices and tables

- genindex

Python Module Index

p

Index