# RadSSH Documentation

***Release 1.1.0***

**Paul Kapp**

August 01, 2016

# Contents

RadSSH is a Python package for Python 2.6+/3.3+. It builds upon the Paramiko package, and provides a simple, yet powerful and extensible user “shell” environment, along with some other minor utilities, and a higher level core API for Python programmers.

RadSSH began as a single purpose tool to simplify the operational pattern of constructing ad-hoc ‘for’ loops in shell to invoke the command line ssh client in order to invoke a single command in sequence across dozens or hundreds of server nodes. Since its original implementation, that small utility has evolved into what is now RadSSH.

CHAPTER 1

# Design Principles

**RadSSH strives to always be:**

- Simple: Simple to install, simple to use, simple to understand.
- Powerful: Combining SSH connection with parallel execution.
- Resilient: No operational environment is perfect.
- Flexible: No two operational environments are the same.
- Extensible: As powerful as the core may be, Customization is where the real power lies.

CHAPTER 2

# Installation

## 2.1 Installation Guide

The preferred way to install RadSSH is to use Python's *pip <http://pip.readthedocs.org/en/latest/>* utility. If you do not have administration (root) privileges to install Python packages into the system level directories, pip can install RadSSH to a *virtual environment <https://pypi.python.org/pypi/virtualenv>*.

Pip will handle the appropriate installation of RadSSH for your Python environment, either system-wide or virtual environment, and also utilitze the Python Package Index to install any missing dependencies. RadSSH currently requires *Paramiko <https://pypi.python.org/pypi/paramiko>* and *netaddr <https://pypi.python.org/pypi/netaddr>* packages, and pip will download these (and their dependencies) and install them if they are not already installed.

### 2.1.1 Installing from PyPI

If you are installing to a virtual environment, be sure to activate the environment prior to running pip.

Run the command (with sudo, if needed): `pip install radssh`

This should download and install RadSSH from the internet, along with the dependency packages (as well as their own dependencies, etc.)

### 2.1.2 Installing from GitHub, with pip

Run the command (with sudo, if needed): `pip install git+https://github.com/radssh/radssh`

As with installation from PyPI, this will download and install (from source on GitHub) the latest version of RadSSH, along with its dependencies.

### 2.1.3 Installing from Developer Source

If you have a local source tree, either from a developer checkout or from un-tarred source package, you can install RadSSH in a similar fashion, replacing the URL of the repository with the local directory. Alternatively, you can `cd` into the source directory and run `pip install .`

## 2.2 Verifying the Install

Once installed, you should run `python -m radssh` (or if running Python 2.6, `python -m radssh.__main__`) as a diagnostic test. If successful, it will report the results of loading the RadSSH package and its dependencies, along with some details about the Python runtime environment and current host. It will also run some capacity checks for the system limitations on concurrent open files and execution threads. These upper limits, if listed, are a significant factor in how many concurrent connections RadSSH will be able to handle on your system.

Sample Output:

```
(sample_env)[paul@pkapp2 ~]$ python -m radssh.__main__
RadSSH Main Module
Package RadSSH 1.0.0 [r5239 @ 2014-12-03 16:07:48Z] from (radssh/__init__.py)
  Using Paramiko  1.15.1 from /usr/lib/python2.6/site-packages/paramiko/__init__.pyc
  Using PyCrypto 2.6.1 from /usr/lib64/python2.6/site-packages/Crypto/__init__.pyc
  Using netaddr 0.7.12 from /usr/lib/python2.6/site-packages/netaddr/__init__.pyc

Python 2.6.6 (CPython)
Running on Linux [pkapp2.risk.regn.net]
  Scientific Linux (6.6/Carbon)

Checking runtime limits...
  System is able to open a maximum of 1021 concurrent files
    Attempting to open file #1022 reported (IOError(24, 'Too many open files'))
  File check completed in 0.005508 seconds

  System is able to run 866 concurrent threads
    Attempting to start thread #867 reported (error("can't start new thread",))
  Thread check completed in 0.450732 seconds

End of runtime check
```

CHAPTER 3

# Shell Mode

For non-programmers, RadSSH's main purpose is to provide a rudimentary command execution shell. RadSSH connects and authenticates a user login to a set of hosts, which can be anywhere from a couple, a dozen, a few dozen, up to several hundred. Once connected and authenticated, the user can enter almost any shell command line text, and have it invoked, in parallel, on every remote host. The RadSSH session will remain connected, allowing the user to run multiple commands within a single session.

Users do not need to have any programming experience in order to use the shell, although familiarity with command line utilities is strongly recommended.

Contents:

## 3.1 Introduction to RadSSH Shell

RadSSH provides a user shell-like environment. It is not quite a "real" shell; there are several restrictions and shortcomings that probably warrant calling it a glorified command line dispatcher utility. Unfortunately "shell" is much easier to type and read, and it behaves closely enough to a shell (with restrictions), that it will be called shell.

**It is like a shell, because:**

- You get a prompt
- You type shell command(s) at the prompt and you press [Enter]
- The commands you typed run (or fail to run), and may produce output
- On completion, you get a prompt

**It is not like a shell, because:**

- No persistent state in between command invocations. You can "run" **cd /tmp**, but on completion you won't be in the `/tmp` directory
- RadSSH does very minimal interpretation of the typed input; it is passed mostly verbatim to the remote hosts
- By default, your connected SSH sessions are not associated with a pty
- Commands that require interactive input are not recommended, and will probably behave badly when run via RadSSH

### 3.1.1 Host Connections

RadSSH can connect to remote hosts either by name (as long as system name resolution is functional) or by IP address (IPv4; not tested with IPv6). Depending on system resource availability, RadSSH is capable of several hundred to over one thousand concurrent connections. RadSSH can also utilize connections through the use of connections through an intermediate host (jumpbox), for use in environments where connection security and/or VLAN isolation may prevent other solutions from working.

You can leverage Bash expansion when invoking RadSSH to provide concise shorthand notation for specifying a range of hosts:

```
# Example for IP expansion
192.168.100.{1..100}
# Example for hostname expansion
webfarm_{00..59}
# Hostname with domain expansion
ldap.{bos,ny,chi,atl,stl}.example.org
```

By default, RadSSH will ensure that all remote host connections are validated against the user's list of accepted host keys, saved in `~/.ssh/known_hosts`. Only hosts whose identities have previously been established (and saved) will be considered trusted.

### 3.1.2 User Authentication

RadSSH supports password authentication as well as SSH Public Key based authentication. Keys can be used from the standard user identity file locations, from a running SSH Agent, or from explicit file locations. Initial authentication is attempted with the current user, but can also be configured to authenticate as a different username for remote login purposes.

### 3.1.3 Resilience

RadSSH does not require 100% success regarding network connectivity or user authentication. A session can be started, attempting to connect to 100 remote host nodes. If 50 connect and authenticate, 30 connect but fail to properly authenticate, and the remaining 20 fail to connect, RadSSH will operate with 50 live connections and 50 inactive connections. At any time during the session, the user can attempt to retry the connection/authentication process to the inactive hosts, via the `*auth` meta-command (See Basic Star Commands in RadSSH). In some environments, having less than 100% success rate in connecting and authenticating winds up being the norm, rather than the exception. RadSSH is designed to cope with these environments as gracefully as possible.

## 3.2 RadSSH Shell - Your First Session

Start here if you want to jump right in!

To test if RadSSH is properly installed with all prerequisites, you can run: `python -m radssh` (If you get an error stating that radssh is a package and can not be directly executed, your Python is likely version 2.6, and you should run `python -m radssh.__main__` instead). You should see a short output summary of RadSSH, along with version information on RadSSH itself, and the Paramiko and other modules that it depends on. If you get errors, there is a problem with the installation, and you should resolve it : see Installation Guide.

If the main module summary is functioning, you should be ready to go!

### 3.2.1 Starting Up A Shell Session

For this example session, I will use huey, dewey, louie, and scrooge as sample hostnames. For your environment, you should use three or four hosts (by name or IP) that you typically access via ssh. You can also, for demonstration purposes, use localhost, 127.0.0.1, and the hostname of your client to mimic 3 separate hosts, even though each connection will wind up being the same destination.

Due to the default behavior, RadSSH will refuse to communicate with connections to unrecognized hosts. Like the Secure Shell (ssh) client, typically only hosts that have their host keys accepted and stored in ~/.ssh/known_hosts file will be able to be properly used. Initially you should only use hosts that you have already established working ssh connections to; configuration options for alternative host key handling behavior are documented in RadSSH Configuration Settings.

Start the session by running `python -m radssh.shell huey dewey louie scrooge`. You should be prompted to enter a password associated with the current user. Once entered, network connections are attempted to the remote (or local) hosts via SSH on the default port (22). Once connections are made, user authentication is attempted with the username and provided password, and at the conclusion, the default **RadSSH $** prompt is displayed:

```
[paul@darkstar ~]$ python -m radssh.shell huey dewey louie scrooge
Please enter a password for (paul) :
Connecting to 4 hosts...
X...
RadSSH $
```

**The progress of connections is indicated by a series of 1-character codes:**

- **.** Connection has been established with successful user authentication
- **O** Connection was established, but user authentication failed
- **X** Network connection failed to be established

### 3.2.2 First Meta-Command: *info

The connection progress line of dots, with maybe a few X's and O's, is intentionally terse. RadSSH does keep more detailed information about connections and authentication status. This information can be printed on demand, using one of the built-in meta-commands, `*info`

```
RadSSH $ *info
*** Cluster Status ***
     dewey : (  0.210s) Authenticated as paul to 10.176.156.25
      huey : (  0.151s) Authenticated as paul to 10.176.156.26
     louie : (  0.313s) Authenticated as paul to 10.173.217.15
   scrooge : (   0.001s) [Errno -2] Name or service not known
Current Quota Settings:
    Idle (not Total) Time: Unlimited
    Output Byte Limit: Unlimited
    Output Byte Limit: Unlimited
Cluster output mode: stream
RadSSH $
```

The names display along with the elapsed time taken to connect and authenticate, as well as the authenticated username and remote IP address of the socket connection. Any problem connections will appear grouped at the end of the list, and include information about what specifically prevented the connection or authentication. In our example, RadSSH failed to establish a connection to `scrooge` because it is not a known host. RadSSH does not treat this as a critical error; commands will not be run on `scrooge` unless it is later reconnected and authenticated. The list entry for `scrooge` is kept as a dormant connection.

`*info` is one of many available meta-commands, or StarCommands available when using RadSSH. All command lines that begin with `*` are never invoked on the remote hosts; instead these are handled within RadSSH itself, or a RadSSH plugin. For details, see Basic Star Commands in RadSSH

### 3.2.3 Running Some Basic Commands

You can enter shell command line(s) at the **RadSSH $** prompt. Press [Enter] to submit them for execution on all active remote hosts. The example shows a few basic informational commands `uptime`, `date` and `df -h /` to report on a few details of each server:

```
RadSSH $ uptime
[louie] 13:51  up 512 days,  2:07, 0 users, load averages: 2.15 2.10 2.09
[huey]  13:36:22 up 29 days,  1:41,  8 users,  load average: 0.00, 0.00, 0.00
[dewey]  13:36:23 up 29 days,  2:42,  1 user,  load average: 0.00, 0.01, 0.05

Summary of failures:
None    - ['scrooge']
Average completion time for 3 hosts: 0.050224s
RadSSH $ date
[huey] Tue Jul 22 13:36:31 EDT 2014
[louie] Tue Jul 22 13:51:10 EDT 2014
[dewey] Tue Jul 22 13:36:31 EDT 2014

Summary of failures:
None    - ['scrooge']
Average completion time for 3 hosts: 0.044666s
RadSSH $ df -h /
[louie] Filesystem     Size   Used  Avail Capacity  Mounted on
[louie] /dev/disk0s3   234G   134G    99G    57%    /
[huey] Filesystem                  Size  Used Avail Use% Mounted on
[huey] /dev/mapper/vg-RootFS   24G   16G  7.4G  68% /
[dewey] Filesystem                         Size  Used Avail Use% Mounted on
[dewey] /dev/mapper/vg-LogVol00   20G   15G  3.7G  80% /

Summary of failures:
None    - ['scrooge']
Average completion time for 3 hosts: 0.046714s
RadSSH $
```

Each command resulted in one (or two) lines of output from each host. On terminals with ANSI code support, they should also be presented with color coding, adding distinction between lines of output from different hosts. Because the commands were run on the remote hosts in parallel, the ordering of the output is not guaranteed to be printed in a predetermined order; results are printed (by default) in the order that they arrive on the network. In addition, at the conclusion of the output section is a summary of failures (if any) and a timing summary for the execution of the command line across all of the remote connections.

Try other various commands, and get a taste for the output. The RadSSH shell does retain command line history, so you can use up/down arrows, and <Ctrl-R> for searching history. At this time, Tab-Completion is not available for command and path/filename completion.

### 3.2.4 Ending A RadSSH Session

In most shells, typing `exit` normally exits the shell. In RadSSH, typing exit results in actually running the exit command on all active remote nodes, which produces no output and lands you right back at the **RadSSH $** prompt. You can confirm this not only by running `exit` (which returns a default status code of 0, indicating success), but if you run `exit 100`, you will see in the summary of failures, the return code 100, followed by a list of servers that reported back a process status of 100. In the event that the list of servers is lengthy, then only a count of the number of servers is printed, not the entire list.

In order to cleanly exit from RadSSH, you should indicate EOF on input by typing <Ctrl-D>, or you can use the StarCommand `*exit`. Like `*info`, `*exit` is a RadSSH built-in meta-command that does not get passed on to the remote hosts to run.

### 3.2.5 A Bonus Directory (Default Logging)

When RadSSH exits, and you return back you your normal shell prompt, do a directory listing with `ls -ltr`, and you should see a newly created directory *session_YYYYMMDD_hhmmss* with recent Year+Month+Day and Hour+Minute+Seconds. RadSSH, by default, will log session commands, and host output (both stdout and stderr, if applicable) into individual files in this session directory.

## 3.3 RadSSH Configuration Settings

RadSSH uses a number of configuration settings that may be of interest to customize and tweak the runtime behavior of the shell, its plugins, and/or the API Cluster behavior. While the default settings are intended to be generally appropriate for most environments, they are user-configurable to allow tailoring on a per-user or per-environment need.

The basic default settings are embedded in the RadSSH module itself, but can be amended by up to three additional stages of configuration: System Configuration, User Configuration, and Command Line Options.

To view the predefined default settings, or to produce a template file for a System Configuration file or User Configuration file, you can run `python -m radssh.config`. This utility will output a sample settings file in comment form. If you save the output to a file, you will need to uncomment specific lines (and change values) in order to enable RadSSH configuration changes.

**In general:**

- Blank lines are ignored
- Lines starting with **#** are ignored (comment lines)
- All other lines should be in the form **keyword=value**, where **keyword** is the option name to set, and **value** is the value to set.

### 3.3.1 System Level Configuration

If the file `/etc/radssh_config` exists, then the settings will be read in from that file. Any option settings found in the System Configuration file will supercede (not add to) the default setting. It is not necessary to specify every configuration option in the System Configuration file; only the settings you have need to change from the default.

**Important** If the System Configuration file sets **user.settings** to a blank string, then the RadSSH shell will not read settings from any User Configuration file, and will also ignore any command line options that would otherwise affect settings. If the **user.settings** option points to a file or pathname, then any settings in that file may override default or system values.

### 3.3.2 User Level Configuration

By default, `~/.radssh_config` is used, but this location may be changed (or removed) by an optional System Level Configuration. Any option settings found in the User Configuration file will supercede the setting values from the default configuration or the System Configuration. The sole exception is the **user.settings** option; altering its value will have no effect if RadSSH is already processing the user settings file.

### 3.3.3 Command Line Options

Settings provided on the command line take the highest precedence. Command line options of the form **keyword=value** override settings that are read from the User Configuration file (or any earlier location).

---

### 3.3.4 General Settings

- **username (default: $SSH_USER or $USER)** Login name for establishing SSH sessions
- **loglevel (default: ERROR)** One of CRITICAL, ERROR, WARNING, INFO, DEBUG. The old **verbose** setting is now deprecated.
- **output_mode (default: stream)** **stream** will output lines of text to the console as they come in. **ordered** will preserve host ordering, which may give the appearance of disrupting parallelism on commands with lengthy output. **off** turns off console output while commands are running, but does not affect file logging of output. Can be changed within the shell via the ***output** command.
- **max_threads (default: 120)** Limit RadSSH processing threads. Independent of the baseline 1 thread per SSH connection overhead.
- **shell.console (default: color)** Set to **mono** if the output color coding is not desired.
- **shell.prompt (default: "RadSSH $")** The RadSSH shell prompt issued before reading each command line.
- **logdir (default: session_%Y%m%d_%H%M%S)** Location of the session logging directory. Defaults to current directory, and supports expansion of datetime elements. Path prefix can be added, including ~ for user home directory. Can be set to blank, to cause logging to go to the console as stderr stream.
- **log_out (default: out.log)** Consolidated (all host) stdout filename. Created in the logdir directory.
- **log_err (default: err.log)** Consolidated (all host) stderr filename. Created in the logdir directory.
- **historyfile (default: ~/.radssh_history)** Save command line history across sessions to this file.
- **socket.timeout (default: 30)** Network connection and read/write timeout (in seconds).
- **keepalive (default: 180)** Send periodic network traffic to prevent connections from being terminated due to being idle.
- **hostkey.verify (default: reject)** Determines how RadSSH handles verification of remote host keys against the ~/.ssh/known_hosts file. **reject** will reject connections if the remote host key is not already validated and accepted in the known hosts file. Other, less secure options include **prompt** which will interactively ask the user to accept unrecognized keys, **accept_new** which will automatically accept new entries, but reject if a previously accepted key no longer matches, and **ignore** which bypasses host key verification completely.
- **hostkey.known_hosts (default: ~/.ssh/known_hosts)** Change to a new file path to keep RadSSH known hosts completely separate from OpenSSH client known hosts. If they share the common default file, keys accepted by one will be treated as accepted by the other.
- **ssh-identity (default: off)** Set to **on** to allow RadSSH to use ~/.ssh/id_rsa, ~/.ssh/id_dsa, and ~/.ssh/identity keys
- **ssh-agent (default: off)** Set to **on** to allow RadSSH to connect to running ssh-agent/keyring service for keys.
- **authfile (default: ~/.radssh_auth)** Optional authentication file, to store collections of passwords and keys to use on a per host basis.

- **plugins (default: None)** Set to a comma separated list of directories where RadSSH should look for user plugins. The system level plugin directory inside the RadSSH package is always searched, regardless of this setting; this is for additional plugin directories to be specified.
- **disable_plugins (default: None)** Set to a comma separated list of plugins to bypass loading.
- **quota.time (default: 0)** Avoid runaway command execution by having RadSSH abort commands if host does not produce output for a given duration (in seconds). Setting of 0 = Unlimited.
- **quota.lines (default: 0)** Avoid runaway command execution by having RadSSH abort commands if host produces too many lines of output. Setting of 0 = Unlimited.
- **quota.bytes (default: 0)** Avoid runaway command execution by having RadSSH abort commands if host produces too many bytes of output. Setting of 0 = Unlimited.
- **commands.forbidden (default: telnet,ftp,sftp,vi,vim,ssh)** Prevent use of the comma separated list of programs. Anything that needs interactive keyboard input will not likely behave as anticipated under RadSSH, and should not be run.
- **commands.restricted (default: rm,reboot,shutdown,halt,poweroff,telinit)** Have RadSSH intercept possibly dangerous commands (extremely dangerous if mistakenly run on hundreds of servers simultaneously) and require explicit confirmation that the user intends to do precisely what was typed in.
- **paramiko_log_level** Deprecated setting. Now able to be set via **loglevel** setting.
- **try_auth_none (default: off)** Perform a initial authentication probing request to determine whether the remote host accepts keys or passwords, or both. Setting to **on** may improve connection speeds by bypassing unsupported authentication attempts, but use caution, as some remote SSH implementations, like Cisco switches will abruptly drop connection if auth-none is attempted. OpenSSH on RHEL/CentOS 5 will fail to send a banner unless auth-none is attempted.
- **force_tty=Cisco,force10networks** Set to a comma separated list of SSH host identifiers for connections that do not support SSH exec_command. This triggers a secondary, less reliable command invocation that runs commands through a dedicated tty session. Both Cisco and Force10 switches have been identified as requiring RadSSH operate in this mode; there may be others.
- **force_tty.signon (default: "term length 0")** When a TTY session is required, RadSSH will issue this command after initial signon. For switches, this should avoid accumulating several "–More–" prompts in the output.
- **force_tty.signoff (default: "term length 20")** When a TTY session is required, RadSSH will issue this command prior to a clean termination.

## 3.4 Basic Star Commands in RadSSH

Most of the time, the text that you enter at a RadSSH prompt is sent to execute on remote nodes. Sometimes during a session, it is useful to issue commands that instruct RadSSH to perform something other than its normal command dispatching. Starting any command with an asterisk (`*`) will enable this behavior, and RadSSH will attempt to interpret what special behavior or operation is requested.

This mechanism of intercepting StarCommands operates not only with the colleciton of RadSSH built-in commands, but can be extended by user PlugIns. For details on coding PlugIns to add (or replace) StarCommands, see Developing RadSSH Plugins.

Here is a brief summary of the RadSSH built-in StarCommands:

### 3.4.1 *? or '*'

Prints a summary listing of available *commands. Includes built-in commands as well as any available through loaded plugins.

### 3.4.2 *exit

Exits RadSSH

### 3.4.3 *info

Prints a listing of the host connections handled by the current RadSSH sessions. Authenticated sessions are listed first, and include the time taken to establish the connection, the remote IP address, and authenticated username. Any failed connections or failed authentications are listed at the bottom to make it easier to identify problem hosts.

Also printed are the current session quota settings and output mode.

### 3.4.4 *output

Select one of RadSSH's output modes. Available mode settings:

| Mode | Description |
|---|---|
| stream | Output is printed to the console as it arrives, with limited buffering |
| ordered | Output is collected per host, and printed only when the host has completed execution of the command. Host output is always in order of the listed host connections. |
| off | Console output is disabled. RadSSH still collects and logs output. |

### 3.4.5 *sh

Invoke a local subshell. RadSSH session persists, so when you exit the subshell, you do not need to reconnect to your remote hosts. The subshell will have an environment variable RADSSH_CONNECTIONS set to the current names of the active session hosts.

### 3.4.6 *result

Prints the output of the most recently run command. This does not rerun the command, instead prints the data preserved in the RadSSH buffers. *result can be limited to a single host, multiple hosts, or if no hostnames are listed, all active hosts.

### 3.4.7 *status

Prints a status summary of the most recently run command. This does not rerun the command, instead prints the data preserved inf the RadSSH buffers. Summary status includes command completion, return code, and execution time.

### 3.4.8 *auth

Attempt to reconnect and reauthenticate to any connection that is not already authenticated. *auth allows you to attempt different password sets, and even try alternate usernames to login, without requiring starting a new session. `*auth` will attempt to authenticate as the default user; you can supply an explicit username with `*auth root`, for example.

## 3.5 Additional Star Commands

Info on other StarCommands

### 3.5.1 BuiltIn Star Commands

In addition to the built-in StarCommands listed in the intro (LINK), the following commands are also available:

***enable [host|wildcard|address|subnet] ...** Dynamically reduce (or expand) the active set of connections. RadSSH will only attempt to execute commands against hosts that are enabled. Running ***enable** without any arguments will restore all host connections to the default enabled state. Supplying one or more arguments will explicitly enable only the hosts that match the arguments. Arguments can be hostnames (with or without wildcards), or network addresses (optionally wildcarded or with CIDR subnet notation).

***get </path/to/file>** Retrieve a file from remote hosts. Save contents in a **files** subdirectory in the session log directory.

***quota [time_limit [byte_limit [line_limit]]]** Set (or print) RadSSH quota limits. RadSSH can automatically abandon reading command output when detecting "runaway" commands, based on idle time (no output received) or volume of output, either based on byte count or line count. When run with no arguments, *quota will print the current quota limits.

***chunk [size [delay]]** Alter the RadSSH command dispatch to limit concurrent command execution to fixed size groupings, or **chunks**, with an optional sleep delay in between each chunk. By default, RadSSH will submit commands for all hosts to a queue, and the commands are executed with maximum parallelism. In some cases, this behavior can be detremental (running a **wget** command in parallel could overload a web server, for example). In these cases, you may want to limit the concurrency to a modest size with ***chunk 10** prior to running **wget**, and the wget will execute in chunks of 10 hosts at a time. Each chunk will be allowed to complete in its entirety before the command is issued to the next chunk of hosts. Running ***chunk 0** will reset back to the default setting of not chunking (maximum concurrency).

***fwd host [port]** **Experimental** Request SSH port forwarding from the connected hosts back through the client for connections to the specified host and optional port (default: 80). In order to reference the tunnel on the remote hosts, command line substitutions are enabled for **%port%** for just the "local" port, or **%tunnel%** for the usable tunnel endpoint (127.0.0.1:%port%)

***vars** **Experimental** Print or set internal variable settings

### 3.5.2 Star Commands From Core Plugins

Many additional StarCommands are loaded by RadSSH in the form of Plugins. These can be copied and customized. Star Commands from user plugins will override the ones loaded from system plugins.

### 3.5.3 Add & Drop Host Connections

To change the cluster connections during an active RadSSH session.

***add host [host] ...** Add host connections to the active cluster.

***drop [host] ...** Drop host connections from the active cluster. If no hosts are listed, then *drop will reference the set of currently enabled hosts (from *enable) and drop the connections to any non-enabled host.

### 3.5.4 Handy Enable Shortcuts

Based on results of the most recently run command, either output or exit status, you can use the following StarCommands to effectively *enable without explicitly listing hosts.

***err [status_code] ...** Enable hosts that had an exit status that matches any listed status code. Omitting any status code will match all non-zero status codes.

***noerr** Enable hosts that returned success (status code 0) from the most recently run command. Equivalent to ***err 0**.

***match text** Enable hosts that contain "text" in the command output (stdout, not stderr) from the most recently run command. As of 1.0.1, stderr content is included in the search.

***nomatch text** The logical inverse of *match. Enables the hosts that DO NOT contain "text" in the output of the most recently run command. As of 1.0.1, stderr content is included in the search.

### 3.5.5 Searching Through Command Output

***grep text** Print matching lines (and line numbers) from the most recently run command. This does not change the enabled set of hosts. Despite the name "grep", this does not match on regular expressions. As of 1.0.1, stderr content is included in the search.

***lines** Print unique lines of output from all hosts from the most recently run command. Prints counts and sorts based on frequency (high to low).

***words** Print unique words (whitespace separated) from the most recently run command. Prints counts and sorts based on frequency (high to low).

### 3.5.6 File Transfer And Script Execution

***sftp source_file [destination_file]** Copy a local file to the remote hosts. File must be on the host where RadSSH is being executed, and will be copied out using the established SSH transport using the sftp subsystem. Remote hosts must have the sftp subsystem enabled for this to work.

***propagate host:/path/to/file** Like *sftp, but file to be copied resides on ONE of the remote hosts, and will be first pulled down to the RadSSH host temporarily, then pushed out to the remainder of the cluster.

***run script_file [arg] ...** Uses *sftp to copy an executable script file from the RadSSH host to a temporary location on the remote hosts, and run with the supplied command line arguments. Equivalent to "*sftp script_file /tmp/script_file; chmod +x /tmp/script_file; /tmp/script_file arg ...".

### 3.5.7 Record And Playback Session Commands

Allow VCR-style recording and playing back commands from a session.

***record filename** Begins recording of commands to local filename. Entering *record without a filename will stop the recording.

***pause** Pauses (or unpauses) the recording of commands. When unpaused, recording resumes to the same exisiting recording file.

***playback filename** Loads and executes session commands that were previously *record'ed

### 3.5.8 Miscellaneous StarCommands

***tty [host] ...** Sequentially invoke TTY sessions on select hosts (or entire cluster, if no hosts listed). Useful for when a fully interactive shell session is required. Since this utilized the established authenticated SSH connections, it avoids the overhead of reestablishing the connections.

***banner** Print the SSH signon banner received from each enabled host. For brevity, this information is not printed during signon, but is made viewable via this command.

# Additional Command Line Utilities

**Supplemental to the RadSSH Shell, the following command line utilities are available.**

- **radssh.config** Print a default configuration template to the console. To assist in creating a customized configuration settings file, run **python -m radssh.config > ~/.radssh_config**, and then edit the `~/.radssh_config` file and uncomment the lines to change the default settings.
- **radssh.plugins** Utility to list available plugins, or install plugin modules from a single source file or all modules from a directory. Installation to the system directory will likely require administrative (root) rights.
- **radssh.pkcs** Utility to use RSA key to encrypt/decrypt. See pkcs for details.
- **radssh.authmgr** Validate a RadSSH AuthFile, and print a summary of its contents. An AuthFile is a convenient way to centrally manage many passwords and/or SSH keys that can be used by RadSSH to authenticate to all of your managed hosts. See authmgr for details.

CHAPTER 5

# Programmer Mode (API)

For users with Python programming knowledge, the RadSSH modules can be used in other Python scripts to automate processes that can make use of SSH connectivity and parallel execution.

In addition, the RadSSH shell can make use of modular extensions in the form of plugins, which can be loaded by the shell to provide additional functionality.

## 5.1 Developing RadSSH Plugins

As an alternative to writing Python programs that use the RadSSH API, the RadSSH Shell supports dynamic runtime extension through a loadable plugin architecture. By relying on the RadSSH shell to manage the cluster creation and management, the code required to implement a plugin module is very small in relation to the powerful functionality that can be added. While a basic understanding of the RadSSH API is beneficial to a plugin developer, it is not necessary to achieve expert knowledge about the API in order to start writing some very powerful extensions to the RadSSH shell.

RadSSH plugins are written in Python, so it is expected that plugin developers know the Python language. Familiarity with the RadSSH Shell, from a user's perspective, is also needed.

There are two basic types of RadSSH plugins: Command plugins, which define (or redefine) RadSSH StarCommands, and Lookup plugins, which can handle custom translation of a symbolic name (or names) into a collection of hosts. A plugin module can be both a Command plugin as well as a Lookup plugin at the same time. Future releases of RadSSH may introduce additional plugin types.

By default, the RadSSH shell will load plugins from the installation directory's `plugins` directory only. Additional plugin directories can be specified with the `plugins` configuration setting, either listed in the RadSSH configuration file, or on the command line with the form `plugins=~/radssh_plugins`

### 5.1.1 Anatomy of a RadSSH Plugin Module

A RadSSH Plugin module is an importable Python source file. It must end with a **.py** filename suffix, and reside either in the system plugin directory, or in a directory that is listed in the configuration setting parameter named "plugins", which can be defined either in the `.radssh_config` file or on the RadSSH Shell command line. Running the RadSSH Shell with `verbose=on` will print out plugin names as they are loaded by the shell, and include errors and traceback information when a plugin fails to load. It is recommended to have `verbose=on` when developing and testing plugins.

A plugin module may contain whatever Python code is needed for the plugin to operate, including `import` statements, class and function definitions, and variables. Executable statements are also permitted, and will be executed when the module is imported by the RadSSH shell. In order for the plugin to be able to be integrated into the RadSSH shell itself, Python objects with specific names need to be defined within the module. It is permissible to define any combination of these special object names, or even define none of them (resulting in a very limited plugin).

**The plugin "special" names are:**

- **init - A Python function that will be called after the module is successfully imported by the RadSSH Shell. This gives**

  **Keyword arguments currently passed from the RadSSH Shell (subject to change):**

  - **defaults** - Python dict object of the user configuration setting defaults
  - **auth** - The AuthManager object used by the shell
  - **plugins** - Python dict object of loaded plugin modules, indexed by module name
  - **star_commands** - Python dict object of StarCommands
  - **shell** - The RadSSH shell invocation function itself

  All but the shell object are mutable; use extreme caution if you attempt to alter them. When in doubt, treat them as read-only.

  If your plugin has no need to read (or update) these objects, and does not require any runtime initialization steps, you do not need to define an **init** function.

- **lookup** - A Python function that takes a single argument (name), and returns an iterator object or None. If the symbolic name passed in can be translated to a set of zero or more hosts, the iterator object will be used by the shell to fetch the connection info. If the name can not be translated by the lookup function, the function should return None.

  When returning an iterator, the iterator should produce 3-tuples or the form (label, hostname, socket). The label should be a string, which is used for labeling the console output, as well as the logfile(s) associated with the host. It is not required for the label to be a resolvable hostname or IP address. The hostname element of the tuple should be an actual hostname or IP address that can be used as a destination for connecting a socket. It may include a port specifier, if the connection destination is not the default SSH port (22); use the form "hostname:port" if needed. The socket element of the tuple, under normal circumstances should be None, in which case the shell will establish the socket connection using the hostname (and possibly port) passed back in the previous tuple element. If the host can not be connected to by a plain socket connection, perhaps needing some firewall/proxy/port-knock manipulation, then it is the responsibility of the plugin to perform these steps and open the socket connection, passing the open socket object back via the iterator

- **command_listener** - A Python function that takes a single argument (command_line). The plugin is notified of each command line content from the shell, just prior to execution. Informative only.

- **star_commands** - A Python dict object, with strings of the form `*command` as keys, and values of StarCommand class objects (plain functions will be converted to default StarCommand instances). This dict handles the mapping of `*command` to the underlying callable function that will be invoked by the shell. A plugin module can define and implement multiple `*commands` in the dict object.

### 5.1.2 StarCommand Class Objects

The `radssh.plugins.StarCommand` class object provides the standardized interface for defining custom `*command` functions. It extends the basic call interface to include independent function synopsis and help_text strings (previously were combined as the function docstring), and under what conditions the shell will print help text on demand or if an improper number of arguments are supplied on the command line.

**The `*command` handler function itself will be passed in the following arguments:**

- **cluster** - The RadSSH Cluster object - Can be used to read the results of the previous command, or to execute command lines from under control of the `*command` itself. Refer to the API documentation `radssh.ssh.Cluster`
- **logdir** - the path to the RadSSH Shell session specific logging directory.
- **command_line** - the text input as the command line (with spaces preserved)
- ***args** - The (space delimited) split of command line arguments

## 5.2 RadSSH Programmer's Guide

This is the documentation for the RadSSH API for Programmers.

If you are developing in Python, and want to make use of RadSSH, start with this overview on how to get started. Familiarity of the behavior of the RadSSH Shell is beneficial, but not completely necessary. Basic understanding of SSH command line utilitie, as well as Python programming knowledge is expected.

The primary Python class provided by RadSSH is the Cluster object. A single Cluster object incorporates a number of SSH connections, a thread pool to handle parallel execution, a flexible mechanism to stream output from many hosts to the console, and a common logging facility. Typically, a single Cluster object is created and used for the duration of a session, however some applications can make use of multiple different Cluster objects at the same time. RadSSH does not impose specific limits on the size or number of Clusters that can be created, the upper limit is dependent on operating system resources (typically threads or maximum number of open files/sockets).

Prior to creating a RadSSH Cluster object, you should familiarize yourself with some of the more fundamental building blocks that are part of RadSSH.

### 5.2.1 AuthManager

The RadSSH AuthManager object is used to attempt SSH user authentication. It provides a common mechanism to identify available ways to try to authenticate a user for SSH login to various remote hosts. AuthManager supports password-based authentication as well as public key-based authentication. It also permits association of a password or a key with a specific host or matching range of hosts.

An AuthManager is associated with a single user login name. If you need to support multiple usernames, you will need to create unique AuthManager objects for each. Most SSH servers do not permit authentication attempts to switch user names during the process anyway, so a disconnect/reconnect will be necessary when switching to the new AuthManager attempts.

For basic password authentication, you can get by with only a username when creating a AuthManager object. If there are no available authentication options, then the AuthManager constructor will interactively prompt the user for the password. If you supply the password in the constructor call, then it will be registered without prompting the user. Alternatively, you can explicitly pass in either `‘include_userkeys=True‘` or `‘include_agent=True‘` to have the AuthManager load user identity keys from their home directory or access keys from a locally running SSH Agent. Either or both options can be requested. If you also supply a default password, then all available keys will be attempted first, and the password will be used if none of the keys granted access.

For more elaborate control of limiting passwords and/or keys to specific hosts, you can construct an AuthFile, and have the AuthManager load the collection of passwords and keys from the AuthFile.

If you do not pass an AuthManager object to the Cluster constructor, one will be created automatically for the currently logged in user, and the user will be prompted for a password.

### 5.2.2 Console

The RadSSH Console object uses a Python Queue object to coordinate the output from multiple concurrent SSH session streams to a single console output stream. With support for ANSI escape sequences, the output on screen can be color coded to visually identify output lines from the same source. ANSI sequnces are also used to change the window title bar to present status messages that are not prone to scrolling off the screen.

A single background thread is started when the Console object is created. It loops constantly reading messages off the queue, and formatting them and printing them to the user.

As with AuthManager objects discussed above, if you do not pass a Console object when creating a Cluster, a default one will be created for you.

### 5.2.3 Configuration Settings

Most of the settings applicable to the RadSSH Shell have direct influence on the behavior of the RadSSH Cluster object itself. If you want to provide the same custom configuration options as the RadSSH shell, you should call radssh.config.load_settings() to get a Python dictionary that is the accumulation of the embedded default settings, the (optional) system settings read from '`/etc/radssh_config`', and the (optional) user settings read from '`~/.radssh_config`'. You may also include any additional override settings, similar to RadSSH Shell command line options, passing in a list of strings of the form **–keyword=value**.

### 5.2.4 Creating The Cluster Object

The only required parameter to creating the Cluster object is a list of hosts, specifically a list of host/connection pairs. The first element in the pair is a label, typically a hostname or IP address, but it can be any arbitrary string. The second element should be either a connected network socket object, or **None** if the label should be used as a hostname for creating socket connections.

Connections will be established, if needed, and RadSSH handles the SSH protocol negotiation, host key verification (if enabled), and user authentication. Note that any error(s) that are encountered at any stage of the process will not raise an exception or error back to the caller. The Cluster object returned can be considered a valid Cluster, even if some (or all) of the connections are not able to be connected or authenticated; attempts to execute a command on any such host will merely be skipped. You can get the status of all host connections from the Cluster.status() call.

Once you do have a Cluster object, you can execute commands via the run_command() call, transfer files to the remote system with the sftp() call, get results from the Cluster.last_result object.

When done using a cluster, you should call Cluster.close_connections() to cleanly log out of all established sessions.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search