ClusterShell Documentation

Release 1.8.2

Stephane Thiell

Aug 15, 2019

Contents

Contents:

CHAPTER 1

Introduction

ClusterShell provides a light, unified and robust command execution Python framework, well-suited to ease daily administrative tasks of nowadays Linux clusters. Some of the most important benefits of using ClusterShell are:

- to provide an efficient, parallel and highly scalable command execution engine in Python,
- to provide an unified node groups syntax and external group access (see the NodeSet class),
- to significantly speed up initial cluster setup and daily administrative tasks when using tools like *clush* and *nodeset*.

Originally created by the HPC Linux system development team at CEA[1] HPC center in France, ClusterShell is designed around medium and long term ideas of sharing cluster administration development time, and this according to two axes:

- sharing administrative applications between main components of the computing center: compute clusters, but also storage clusters and server farms (so they can use the same efficient framework for their administrative applications),
- sharing administration techniques across multiple generations of super-computing clusters (first of all, to avoid that every cluster administration application has to implement its own command execution layer, but also to encourage the adoption of event-based coding model in administration scripts).

Two available coding models make the library well-suited for simple scripts or for complex applications as well. Also, the library is fully cluster-aware and has primarily been made for executing remote shell commands in parallel and gathering output results. But it now also provides the developer a set of extra features for administrative applications, like file copy support or time-based notifications (timers) which are discussed in this documentation.

[1] French Alternative Energies and Atomic Energy Commission, a leading technological research organization in Europe

CHAPTER 2

Release Notes

2.1 Version 1.8

This adaptive release is now compatible with both Python 2 and Python 3.

We hope this release will help you manage your clusters, server farms or cloud farms! Special thanks to the many of you that have sent us feedback on GitHub!

Warning: Support for Python 2.5 and below has been dropped in this version.

2.1.1 Version 1.8.2

This version contains a few minor fixes:

- *clush*: support UTF-8 string encoding with *–diff*
- in some cases, *timers* were too fast due to an issue in `EngineTimer`
- fix issue in the *Slurm group bindings* where job ids were used instead of user names
- performance update for *xCAT group bindings*

For more details, please have a look at GitHub Issues for 1.8.2 milestone.

Python support

Version 1.8.2 adds support for Python 3.7.

Note: This version still supports Python 2.6 and thus also RHEL/CentOS 6, but please note that ClusterShell 1.9 is expected to require at least Python 2.7.

OS support

Version 1.8.2 adds support for RHEL 8/CentOS 8 and Fedora 31+, where only the Python 3 package is provided. The `clustershell` packages will be made available in EPEL-8 as soon as possible.

No packaging changes were made to `clustershell` in RHEL/CentOS 6 or 7.

2.1.2 Version 1.8.1

This update contains a few bug fixes and some performance improvements of the *`NodeSet`* class.

The *tree mode* has been fixed to properly support offline gateways.

We added the following command line options:

- `--conf` to specify alternative clush.conf (clush only)
- `--groupsconf` to specify alternative groups.conf (all CLIs)

In *`EventHandler`*, we reinstated *`EventHandler.ev_error()`*: and *`EventHandler.ev_error()`*: (as deprecated) for compatibility purposes. Please see below for more details about important *`EventHandler`* changes in 1.8.

Finally, *cluset*/*nodeset* have been improved by adding support for:

- litteral new line in `-S`
- multiline shell variables in options

For more details, please have a look at GitHub Issues for 1.8.1 milestone.

2.1.3 Main changes in 1.8

For more details, please have a look at GitHub Issues for 1.8 milestone.

CLI (command line interface)

If you use the *clush* or *cluset*/*nodeset* tools, there are no major changes since 1.7, though a few bug fixes and improvements have been done:

- It is now possible to work with numeric node names with cluset/nodeset:

```
$ nodeset --fold 6704 6705 r931 r930
[6704-6705],r[930-931]

$ squeue -h -o '%i' -u $USER | cluset -f
[680240-680245,680310]
```

  As a reminder, cluset/nodeset has always had an option to switch to numerical cluster ranges (only), using `-R`/`--rangeset`:

```
$ squeue -h -o '%i' -u $USER | cluset -f -R
680240-680245,680310
```

- Node group configuration is now loaded and processed only when required. This is actually an improvement of the *`NodeSet`* class that the tools readily benefit. This should improve both usability and performance.
- YAML group files are now ignored for users that don't have the permission to read them (see *File-based group sources* for more info about group files).

- *clush* now use slightly different colors that are legible on dark backgrounds.
- *Tree execution mode*:
  - Better detection of the Python executable, and, if needed, we added a new environment variable to override it, see *Remote Python executable*.
  - You must use the same major version of Python on the gateways and the root node.

Python library

If you're a developer and use the ClusterShell Python library, please read below.

Python 3 support

Starting in 1.8, the library can also be used with Python 3. The code is compatible with both Python 2 and 3 at the same time. To make it possible, we performed a full code refactoring (without changing the behavior).

Note: When using Python 3, we recommend Python 3.4 or any more recent version.

Improved Event API

We've made some changes to *EventHandler*, a class that defines a simple interface to handle events generated by *Worker*, `EventTimer` and `EventPort` objects.

Please note that all programs already based on *EventHandler* should work with this new version of ClusterShell without any code change (backward API compatibility across 1.x versions is enforced). We use object *introspection*, the ability to determine the type of an object at runtime, to make the Event API evolve smoothly. We do still recommend to change your code as soon as possible as we'll break backward compatibility in the future major release 2.0.

The signatures of the following *EventHandler* methods **changed** in 1.8:

- *EventHandler.ev_pickup()*: new `node` argument
- *EventHandler.ev_read()*: new `node`, `sname` and `msg` arguments
- *EventHandler.ev_hup()*: new `node`, `rc` argument
- *EventHandler.ev_close()*: new `timedout` argument

Both old and new signatures are supported in 1.8. The old signatures will be deprecated in a future 1.x release and **removed** in version 2.0.

The new methods aims to be more convenient to use by avoiding the need of accessing context-specific *Worker* attributes like `worker.current_node` (replaced with the `node` argument in that case).

Also, please note that the following *EventHandler* methods will be removed in 2.0:

- `EventHandler.ev_error()`: its use should be replaced with *EventHandler.ev_read()* by comparing the stream name `sname` with *Worker.SNAME_STDERR*, like in the example below:

```
class MyEventHandler(EventHandler):

    def ev_read(self, worker, node, sname, msg):
        if sname == worker.SNAME_STDERR:
            print('error from %s: %s' % (node, msg))
```

- `EventHandler.ev_timeout()`: its use should be replaced with *`EventHandler.ev_close()`* by checking for the new `timedout` argument, which is set to `True` when a timeout occurred.

We recommend developers to start using the improved *`Event`* API now. Please don't forget to update your packaging requirements to use ClusterShell 1.8 or later.

Task and standard input (stdin)

`Task.shell()` and *`Task.run()`* have a new `stdin` boolean argument which if set to `False` prevents the use of stdin by sending EOF at first read, like if it is connected to /dev/null.

If not specified, its value is managed by the *Library Defaults*. Its default value in *`Defaults`* is set to `True` for backward compatibility, but could change in a future major release.

If your program doesn't plan to listen to stdin, it is recommended to set `stdin=False` when calling these two methods.

Packaging changes

We recommend that package maintainers use separate subpackages for Python 2 and Python 3, to install ClusterShell modules and related command line tools. The Python 2 and Python 3 stacks should be fully installable in parallel.

For the RPM packaging, there is now two subpackages `python2-clustershell` and `python3-clustershell` (or `python34-clustershell` in EPEL), each providing the library and tools for the corresponding version of Python.

The `clustershell` package includes the common configuration files and documentation and requires `python2-clustershell`, mainly because Python 2 is still the default interpreter on most operating systems.

`vim-clustershell` was confusing so we removed it and added the vim extensions to the main `clustershell` subpackage.

Version 1.8 should be readily available as RPMs in the following distributions or RPM repositories:

- EPEL 6 and 7
- Fedora 26 and 27
- openSUSE Factory and Leap

On a supported environment, you can expect a smooth upgrade from version 1.6+.

We also expect the packaging to be updated for Debian.

2.2 Version 1.7

It's just a small version bump from the well-known 1.6 version, but ClusterShell 1.7 comes with some nice new features that we hope you'll enjoy! Most of these features have already been tested on some very large Linux production systems.

Version 1.7 and possible future minor versions 1.7.x are compatible with Python 2.4 up to Python 2.7 (for example: from RedHat EL5 to EL7). Upgrade from version 1.6 to 1.7 should be painless and is fully supported.

2.2.1 Version 1.7.3

This update contains a few bug fixes and some interesting performance improvements. This is also the first release published under the GNU Lesser General Public License, version 2.1 or later (LGPL v2.1+). Previous releases were published under the CeCILL-C V1.

Quite a bit of work has been done on the *fanout* of processes that the library uses to execute commands. We implemenented a basic per-worker *fanout* to fix the broken behaviour in tree mode. Thanks to this, it is now possible to use fanout=1 with gateways. The *documentation* has also been clarified.

An issue that led to broken pipe errors but also affected performance has been fixed in *tree mode* when copying files.

An issue with *clush* -L where nodes weren't always properly sorted has been fixed.

The performance of `MsgTree`, the class used by the library to aggregate identical command outputs, has been improved. We have seen up to 75% speed improvement in some cases.

Finally, a *cluset* command has been added to avoid a conflict with xCAT nodeset command. It is the same command as *nodeset*.

For more details, please have a look at GitHub Issues for 1.7.3 milestone.

ClusterShell 1.7.3 is compatible with Python 2.4 up to Python 2.7 (for example: from RedHat EL5 to EL7). Upgrades from versions 1.6 or 1.7 are supported.

2.2.2 Version 1.7.2

This minor version fixes a defect in *tree mode* that led to broken pipe errors or unwanted backtraces.

The `NodeSet` class now supports the empty string as input. In practice, you may now safely reuse the output of a *nodeset* command as input argument for another *nodeset* command, even if the result is an empty string.

A new option `--pick` is available for *clush* and *nodeset* to pick N node(s) at random from the resulting node set.

For more details, please have a look at GitHub Issues for 1.7.2 milestone.

ClusterShell 1.7.2 is compatible with Python 2.4 up to Python 2.7 (for example: from RedHat EL5 to EL7). Upgrades from versions 1.6 or 1.7 are supported.

2.2.3 Version 1.7.1

This minor version contains a few bug fixes, mostly related to *Node sets handling*.

This version also contains bug fixes and performance improvements in tree propagation mode.

For more details, please have a look at GitHub Issues for 1.7.1 milestone.

ClusterShell 1.7.1 is compatible with Python 2.4 up to Python 2.7 (for example: from RedHat EL5 to EL7). Upgrades from versions 1.6 or 1.7 are supported.

2.2.4 Main changes in 1.7

This new version comes with a refreshed documentation, based on the Sphinx documentation generator, available on http://clustershell.readthedocs.org.

The main new features of version 1.7 are described below.

Multidimensional nodesets

The *NodeSet* class and *nodeset* command-line have been improved to support multidimentional node sets with folding capability. The use of nD naming scheme is sometimes used to map node names to physical location like name-<rack>-<position> or node position within the cluster interconnect network topology.

A first example of 3D nodeset expansion is a good way to start:

```
$ nodeset -e gpu-[1,3]-[4-5]-[0-6/2]
gpu-1-4-0 gpu-1-4-2 gpu-1-4-4 gpu-1-4-6 gpu-1-5-0 gpu-1-5-2 gpu-1-5-4
gpu-1-5-6 gpu-3-4-0 gpu-3-4-2 gpu-3-4-4 gpu-3-4-6 gpu-3-5-0 gpu-3-5-2
gpu-3-5-4 gpu-3-5-6
```

You've probably noticed the /2 notation of the last dimension. It's called a step and behaves as one would expect, and is fully supported with nD nodesets.

All other *nodeset* commands and options are supported with nD nodesets. For example, it's always useful to have a quick way to count the number of nodes in a nodeset:

```
$ nodeset -c gpu-[1,3]-[4-5]-[0-6/2]
16
```

Then to show the most interesting new capability of the underlying *NodeSet* class in version 1.7, a folding example is probably appropriate:

```
$ nodeset -f compute-1-[1-34] compute-2-[1-34]
compute-[1-2]-[1-34]
```

In the above example, nodeset will try to find a very compact nodesets representation whenever possible. ClusterShell is probably the first and only cluster tool capable of doing such complex nodeset folding.

Attention, as not all cluster tools are supporting this kind of complex nodesets, even for nodeset expansion, we added an --axis option to select to fold along some desired dimension:

```
$ nodeset --axis 2 -f compute-[1-2]-[1-34]
compute-1-[1-34],compute-2-[1-34]
```

The last dimension can also be selected using -1:

```
$ nodeset --axis -1 -f compute-[1-2]-[1-34]
compute-1-[1-34],compute-2-[1-34]
```

All set-like operations are also supported with several dimensions, for example *difference* (-x):

```
$ nodeset -f c-[1-10]-[1-44] -x c-[5-10]-[1-34]
c-[1-4]-[1-44],c-[5-10]-[35-44]
```

Hard to follow? Don't worry, ClusterShell does it for you!

File-based node groups

Cluster node groups have been a great success of previous version of ClusterShell and are now widely adopted. So we worked on improving it even more for version 1.7.

For those of you who use the file /etc/clustershell/group to describe node groups, that is still supported in 1.7 and upgrade from your 1.6 setup should work just fine. However, for new 1.7 installations, we have put this file in a different location by default:

```
$ vim /etc/clustershell/groups.d/local.cfg
```

Especially if you're starting a new setup, you have also the choice to switch to a more advanced groups YAML configuration file that can define multiple *sources* in a single file (equivalent to separate namespaces for node groups). The YAML format possibly allows you to edit the file content with YAML tools but it's also a file format convenient to edit just using the vim editor. To enable the example file, you need to rename it first as it needs to have the **.yaml** extension:

```
$ cd /etc/clustershell/groups.d
$ mv cluster.yaml.example cluster.yaml
```

You can make the first dictionary found on this file (named *roles*) to be the **default** source by changing `default: local` to `default: roles` in `/etc/clustershell/groups.conf` (main config file for groups).

For more info about the YAML group files, please see *File-based group sources*.

Please also see *node groups configuration* for node groups configuration in general.

nodeset -L/–list-all option

Additionally, the *nodeset* command also has a new option `-L` or `--list-all` to list groups from all sources (`-l` only lists groups from the **default** source). This can be useful when configuring ClusterShell and/or troubleshooting node group sources:

```
$ nodeset -LL
@adm example0
@all example[2,4-5,32-159]
@compute example[32-159]
@gpu example[156-159]
@io example[2,4-5]
@racks:new example[4-5,156-159]
@racks:old example[0,2,32-159]
@racks:rack1 example[0,2]
@racks:rack2 example[4-5]
@racks:rack3 example[32-159]
@racks:rack4 example[156-159]
@cpu:hsw example[64-159]
@cpu:ivy example[32-63]
```

Special group @*

The special group syntax `@*` (or `@source:*` if using explicit source selection) has been added and can be used in configuration files or with command line tools. This special group is always available for file-based node groups (return the content of the **all** group, or all groups from the source otherwise). For external sources, it is available when either the **all** upcall is defined or both **map** and **list** upcalls are defined. The all special group is also used by `clush -a` and `nodeset -a`. For example, the two following commands are equivalent:

```
$ nodeset -a -f
example[2,4-5,32-159]

$ nodeset -f @*
example[2,4-5,32-159]
```

Exec worker

Version 1.7 introduces a new generic execution worker named *ExecWorker* as the new base class for most exec()-based worker classes. In practice with *clush*, you can now specify the worker in command line using `--worker` or `-R` and use **exec**. It also supports special placeholders for the node (**%h**) or rank (**%n**). For example, the following command will execute *ping* commands in parallel, each with a different host from hosts *cs01*, etc. to *cs05* as argument and then aggregate the results:

```
$ clush -R exec -w cs[01-05] -bL 'ping -c1 %h >/dev/null && echo ok'
cs[01-04]: ok
clush: cs05: exited with exit code 1
```

This feature allows the system administrator to use non cluster-aware tools in a more efficient way. You may also want to explicitly set the fanout (using `-f`) to limit the number of parallel local commands launched.

Please see also *clush worker selection*.

Rsh worker

Version 1.7 adds support for `rsh` or any of its variants like `krsh` or `mrsh`. `rsh` and `ssh` also share a lot of common mechanisms. Worker Rsh was added moving a lot of Worker Ssh code into it.

For `clush`, please see *clush worker selection* to enable `rsh`.

To use `rsh` by default instead of `ssh` at the library level, install the provided example file named `defaults.conf-rsh` to `/etc/clustershell/defaults.conf`.

Tree Propagation Mode

The ClusterShell Tree Mode allows you to send commands to target nodes through a set of predefined gateways (using ssh by default). It can be useful to access servers that are behind some other servers like bastion hosts, or to scale on very large clusters when the flat mode (eg. sliding window of ssh commands) is not enough anymore.

The tree mode is now *documented*, it has been improved and is enabled by default when a `topology.conf` file is found. While it is still a work in progress, the tree mode is known to work pretty well when all gateways are online. We'll continue to improve it and make it more robust in the next versions.

Configuration files

When `$XDG_CONFIG_HOME` is defined, ClusterShell will use it to search for additional configuration files.

PIP user installation support

ClusterShell 1.7 is now fully compatible with PIP and supports user configuration files:

```
$ pip install --user clustershell
```

Please see *Installing ClusterShell as user using PIP*.

CHAPTER 3

Installation

ClusterShell is distributed in several packages. On RedHat-like OS, we recommend to use the RPM package (.rpm) distribution.

As a system software for cluster, ClusterShell is primarily made for system-wide installation to be used by system administrators. However, changes have been made so that it's now easy to install it without root access (see *Installing ClusterShell as user using PIP*).

3.1 Requirements

ClusterShell should work with any Unix[1] operating systems which provides Python 2.6, 2.7 or 3.x and OpenSSH or any compatible Secure Shell clients.

Furthermore, ClusterShell's engine has been optimized when the `poll()` syscall is available or even better, when the `epoll_wait()` syscall is available (Linux only).

For instance, ClusterShell is known to work on the following operating systems:

- GNU/Linux RHEL or CentOS 6 (Python 2.6)
- GNU/Linux RHEL or CentOS 7 (Python 2.7)
- GNU/Linux Fedora 22 to 26 (Python 2.6 or 2.7)
- GNU/Linux Debian wheezy and above (Python 2.7)
- Mac OS X 10.8+ (Python 2.6 or 2.7)

3.2 Distribution

ClusterShell is an open-source project distributed under the GNU Lesser General Public License version or later (LGPL v2.1+), which means that many possibilities are offered to the end user. Also, as a software library, ClusterShell

[1] Unix in the same sense of the *Availability: Unix* notes in the Python documentation

should remain easily available to everyone. Hopefully, packages are currently available for Fedora Linux, RHEL (through EPEL repositories), Debian and Arch Linux.

3.2.1 Fedora

At the time of writing, ClusterShell 1.8.2 is available on Fedora 26 (releases being maintained by the Fedora Project).

Install ClusterShell from *Fedora Updates*

ClusterShell is part of Fedora, so it is really easy to install it with `dnf`, although you have to keep the Fedora *updates* default repository. The following command checks whether the packages are available on a Fedora system:

```
$ dnf list \*clustershell
Available Packages
clustershell.noarch                     1.8-1.fc26                fedora
python2-clustershell.noarch             1.8-1.fc26                fedora
python3-clustershell.noarch             1.8-1.fc26                fedora
```

Then, install ClusterShell's library module and tools using the following command:

```
$ dnf install clustershell
```

Python 2 module and tools are installed by default. If interested in Python 3 development, simply install the additional ClusterShell's Python 3 subpackage using the following command:

```
$ dnf install python3-clustershell
```

Python 3 versions of the tools are installed as *tool-pythonversion*, like `clush-3.6`, `cluset-3.6` or `nodeset-3.6` on Fedora 26.

Install ClusterShell from Fedora Updates Testing

Recent releases of ClusterShell are first available through the Test Updates repository of Fedora, then it is later pushed to the stable *updates* repository. The following `dnf` command will also checks for packages availability in the *updates-testing* repository:

```
$ dnf list \*clustershell --enablerepo=updates-testing
```

To install, also add the `--enablerepo=updates-testing` option, for instance:

```
$ dnf install clustershell --enablerepo=updates-testing
```

3.2.2 Red Hat Enterprise Linux (and CentOS)

ClusterShell packages are maintained on Extra Packages for Enterprise Linux EPEL for Red Hat Enterprise Linux (RHEL) and its compatible spinoffs such as CentOS. At the time of writing, ClusterShell 1.8.2 is available on EPEL 6 and 7.

Install ClusterShell from EPEL

First you have to enable the `yum` EPEL repository. We recommend to download and install the EPEL repository RPM package. On CentOS, this can be easily done using the following command:

```
$ yum --enablerepo=extras install epel-release
```

Then, the ClusterShell installation procedure is quite the same as for *Fedora Updates*, for instance:

```
$ yum install clustershell
```

Python 2 module and tools are installed by default. If interested in Python 3 development, simply install the additional ClusterShell's Python 3 subpackage using the following command:

```
$ yum install python34-clustershell
```

Note: The Python 3 subpackage is named `python34-clustershell` on EPEL 6 and 7, instead of `python3-clustershell`.

Python 3 versions of the tools are installed as *tool-pythonversion*, like `clush-3.4`, `cluset-3.4` or `nodeset-3.4` on EPEL 6 and 7.

3.2.3 openSUSE

ClusterShell is available in openSUSE Tumbleweed (Factory) and Leap since 2017:

```
$ zypper search clustershell
Loading repository data...
Reading installed packages...

S | Name                  | Summary                                                    |␣
↪Type
--+-----------------------+------------------------------------------------------------+----
↪----
  | clustershell          | Python framework for efficient cluster administration |␣
↪package
  | python2-clustershell | ClusterShell module for Python 2                       |␣
↪package
  | python3-clustershell | ClusterShell module for Python 3                       |␣
↪package
```

To install ClusterShell on openSUSE, use:

```
$ zypper install clustershell
```

Python 2 module and tools are installed by default. If interested in Python 3 development, simply install the additional ClusterShell's Python 3 subpackage using the following command:

```
$ zypper install python3-clustershell
```

Python 3 versions of the tools are installed as *tool-pythonversion*, like `clush-3.6`, `cluset-3.6` or `nodeset-3.6`.

3.2.4 Debian

ClusterShell is available in Debian **main** repository (since 2011).

To install it on Debian, simply use:

```
$ apt-get install clustershell
```

You can get the latest version on:

```
* http://packages.debian.org/sid/clustershell
```

3.2.5 Ubuntu

Like Debian, it is easy to get and install ClusterShell on Ubuntu (also with `apt-get`). To do so, please first enable the **universe** repository. ClusterShell is available since "Natty" release (11.04):

- http://packages.ubuntu.com/clustershell

3.2.6 Installing ClusterShell using PIP

Installing ClusterShell as root using PIP

To install ClusterShell as a standard Python package using PIP[2] as root:

```
$ pip install ClusterShell
```

Or alternatively, using the source tarball:

```
$ pip install ClusterShell-1.x.tar.gz
```

Installing ClusterShell as user using PIP

To install ClusterShell as a standard Python package using PIP as an user:

```
$ pip install --user ClusterShell
```

Or alternatively, using the source tarball:

```
$ pip install --user ClusterShell-1.x.tar.gz
```

Then, you just need to update your `PYTHONPATH` environment variable to be able to import the library and `PATH` to easily use the *Tools*:

```
$ export PYTHONPATH=$PYTHONPATH:~/.local/lib
$ export PATH=$PATH:~/.local/bin
```

Configuration files are installed in `~/.local/etc/clustershell` and are automatically loaded before system-wide ones (for more info about supported user config files, please see the *clush* or *Node groups* config sections).

[2] pip is a tool for installing and managing Python packages, such as those found in the Python Package Index

3.3 Source

Current source is available through Git, use the following command to retrieve the latest development version from the repository:

```
$ git clone git@github.com:cea-hpc/clustershell.git
```

CHAPTER 4

Configuration

4.1 clush

The following configuration file defines system-wide default values for several *clush* tool parameters:

```
/etc/clustershell/clush.conf
```

clush settings might then be overridden per user if one of the following files is found, in priority order:

```
$XDG_CONFIG_HOME/clustershell/clush.conf
$HOME/.config/clustershell/clush.conf (only if $XDG_CONFIG_HOME is not defined)
$HOME/.local/etc/clustershell/clush.conf
$HOME/.clush.conf (deprecated, for 1.6 compatibility only)
```

The following table describes available *clush* config file settings.

Key	Value
fanout	Size of the sliding window of *ssh(1)* connectors.
connect_timeout	Timeout in seconds to allow a connection to establish. This parameter is passed to *ssh(1)*. If set to 0, no timeout occurs.
command_timeout	Timeout in seconds to allow a command to complete since the connection has been established. This parameter is passed to *ssh(1)*. In addition, the ClusterShell library ensures that any commands complete in less than (connect_timeout + command_timeout). If set to 0, no timeout occurs.
color	Whether to use ANSI colors to surround node or nodeset prefix/header with escape sequences to display them in color on the terminal. Valid arguments are *never*, *always* or *auto* (which use color if standard output/error refer to a terminal). Colors are set to `[34m` (blue foreground text) for stdout and `[31m` (red foreground text) for stderr, and cannot be modified.
fd_max	Maximum number of open file descriptors permitted per *clush* process (soft resource limit for open files). This limit can never exceed the system (hard) limit. The *fd_max* (soft) and system (hard) limits should be high enough to run *clush*, although their values depend on your fanout value.
history_size	Set the maximum number of history entries saved in the GNU readline history list. Negative values imply unlimited history file size.
node_count	Should *clush* display additional (node count) information in buffer header? (yes/no)
verbosity	Set the verbosity level: 0 (quiet), 1 (default), 2 (verbose) or more (debug).
ssh_user	Set the *ssh(1)* user to use for remote connection (default is to not specify).
ssh_path	Set the *ssh(1)* binary path to use for remote connection (default is *ssh*).
ssh_options	Set additional (raw) options to pass to the underlying *ssh(1)* command.
scp_path	Set the *scp(1)* binary path to use for remote copy (default is *scp*).
scp_options	Set additional options to pass to the underlying *scp(1)* command. If not specified, *ssh_options* are used instead.
rsh_path	Set the *rsh(1)* binary path to use for remote connection (default is *rsh*). You could easily use *mrsh* or *krsh* by simply changing this value.
rcp_path	Same as *rsh_path* but for rcp command (default is *rcp*).
rsh_options	Set additional options to pass to the underlying rsh/rcp command.

4.2 Node groups

ClusterShell defines a *node group* syntax to represent a collection of nodes. This is a convenient way to manipulate node sets, especially in HPC (High Performance Computing) or with large server farms. This section explains how to configure node group **sources**. Please see also *nodeset node groups* for specific usage examples.

4.2.1 groups.conf

ClusterShell loads *groups.conf* configuration files that define how to obtain node groups configuration, ie. the way the library should access file-based or external node group **sources**.

The following configuration file defines system-wide default values for *groups.conf*:

```
/etc/clustershell/groups.conf
```

groups.conf settings might then be overridden per user if one of the following files is found, in priority order:

```
$XDG_CONFIG_HOME/clustershell/groups.conf
$HOME/.config/clustershell/groups.conf (only if $XDG_CONFIG_HOME is not defined)
$HOME/.local/etc/clustershell/groups.conf
```

This makes possible for an user to have its own *node groups* configuration. If no readable configuration file is found, group support will be disabled but other node set operations will still work.

groups.conf defines configuration sub-directories, but may also define source definitions by itself. These **sources** provide external calls that are detailed in *External group sources*.

The following example shows the content of a *groups.conf* file where node groups are bound to the source named *genders* by default:

```
[Main]
default: genders
confdir: /etc/clustershell/groups.conf.d $CFGDIR/groups.conf.d
autodir: /etc/clustershell/groups.d $CFGDIR/groups.d

[genders]
map: nodeattr -n $GROUP
all: nodeattr -n ALL
list: nodeattr -l

[slurm]
map: sinfo -h -o "%N" -p $GROUP
all: sinfo -h -o "%N"
list: sinfo -h -o "%P"
reverse: sinfo -h -N -o "%P" -n $NODE
```

The *groups.conf* files are parsed with Python's ConfigParser:

- The first section whose name is *Main* accepts the following keywords:
  - *default* defines a **default node group source** (eg. by referencing a valid section header)
  - *confdir* defines an optional list of directory paths where the ClusterShell library should look for **.conf** files which define group sources to use. Each file in these directories with the .conf suffix should contain one or more node group source sections as documented below. These will be merged with the group sources defined in the main *groups.conf* to form the complete set of group sources to use. Duplicate group source sections are not allowed in those files. Configuration files that are not readable by the current user are ignored (except the one that defines the default group source). The variable *$CFGDIR* is replaced by the path of the highest priority configuration directory found (where *groups.conf* resides). The default *confdir* value enables both system-wide and any installed user configuration (thanks to *$CFGDIR*). Duplicate directory paths are ignored.
  - *autodir* defines an optional list of directories where the ClusterShell library should look for **.yaml** files that define in-file group dictionaries. No need to call external commands for these files, they are parsed by the ClusterShell library itself. Multiple group source definitions in the same file is supported. The variable *$CFGDIR* is replaced by the path of the highest priority configuration directory found (where *groups.conf* resides). The default *confdir* value enables both system-wide and any installed user configuration (thanks to *$CFGDIR*). Duplicate directory paths are ignored.
- Each following section (*genders*, *slurm*) defines a group source. The map, all, list and reverse upcalls are explained below in *Group source upcalls*.

4.2.2 File-based group sources

Version 1.7 introduces support for native handling of flat files with different group sources to avoid the use of external upcalls for such static configuration. This can be achieved through the *autodir* feature and YAML files described below.

YAML group files

Cluster node groups can be defined in straightforward YAML files. In such a file, each YAML dictionary defines group to nodes mapping. **Different dictionaries** are handled as **different group sources**.

For compatibility reasons with previous versions of ClusterShell, this is not the default way to define node groups yet. So here are the steps needed to try this out:

Rename the following file:

```
/etc/clustershell/groups.d/cluster.yaml.example
```

to a file having the **.yaml** extension, for example:

```
/etc/clustershell/groups.d/cluster.yaml
```

Ensure that *autodir* is set in *groups.conf*:

```
autodir: /etc/clustershell/groups.d $CFGDIR/groups.d
```

In the following example, we also changed the default group source to **roles** in *groups.conf* (the first dictionary defined in the example), so that *@roles:groupname* can just be shorted *@groupname*.

Here is an example of **/etc/clustershell/groups.d/cluster.yaml**:

```
roles:
    adm: 'mgmt[1-2]'                # define groups @roles:adm and @adm
    login: 'login[1-2]'
    compute: 'node[0001-0288]'
    gpu: 'node[0001-0008]'

    cpu_only: '@compute!@gpu'       # example of inline set operation
                                    # define group @cpu_only with node[0009-0288]

    storage: '@lustre:mds,@lustre:oss' # example of external source reference

    all: '@login,@compute,@storage'  # special group used for clush/nodeset -a
                                    # only needed if not including all groups

lustre:
    mds: 'mds[1-4]'
    oss: 'oss[0-15]'
    rbh: 'rbh[1-2]'
```

Testing the syntax of your group file can be quickly performed through the `-L` or `--list-all` command of *nodeset*:

```
$ nodeset -LL
@adm mgmt[1-2]
@all login[1-2],mds[1-4],node[0001-0288],oss[0-15],rbh[1-2]
@compute node[0001-0288]
@cpu_only node[0009-0288]
@gpu node[0001-0008]
@login login[1-2]
@storage mds[1-4],oss[0-15],rbh[1-2]
@sysgrp sysgrp[1-4]
@lustre:mds mds[1-4]
@lustre:oss oss[0-15]
@lustre:rbh rbh[1-2]
```

4.2.3 External group sources

Group source upcalls

Each node group source is defined by a section name (*source* name) and up to four upcalls:

- **map**: External shell command used to resolve a group name into a node set, list of nodes or list of node sets (separated by space characters or by carriage returns). The variable *$GROUP* is replaced before executing the command.
- **all**: Optional external shell command that should return a node set, list of nodes or list of node sets of all nodes for this group source. If not specified, the library will try to resolve all nodes by using the **list** external command in the same group source followed by **map** for each available group. The notion of *all nodes* is used by `clush -a` and also by the special group name `@*` (or `@source:*`).
- **list**: Optional external shell command that should return the list of all groups for this group source (separated by space characters or by carriage returns). If this upcall is not specified, ClusterShell won't be able to list any available groups (eg. with `nodeset -l`), so it is highly recommended to set it.
- **reverse**: Optional external shell command used to find the group(s) of a single node. The variable *$NODE* is previously replaced. If this external call is not specified, the reverse operation is computed in memory by the library from the **list** and **map** external calls, if available. Also, if the number of nodes to reverse is greater than the number of available groups, the reverse external command is avoided automatically to reduce resolution time.

In addition to context-dependent *$GROUP* and *$NODE* variables described above, the two following variables are always available and also replaced before executing shell commands:

- *$CFGDIR* is replaced by *groups.conf* base directory path
- *$SOURCE* is replaced by current source name (see an usage example just below)

Caching considerations

External command results are cached in memory, for a limited amount of time, to avoid multiple similar calls.

The optional parameter **cache_time**, when specified within a group source section, defines the number of seconds each upcall result is kept in cache, in memory only. Please note that caching is actually only useful for long-running programs (like daemons) that are using node groups, not for one-shot commands like *clush* or *cluset*/*nodeset*.

The default value of **cache_time** is 3600 seconds.

Multiple sources section

Use a comma-separated list of source names in the section header if you want to define multiple group sources with similar upcall commands. The special variable *$SOURCE* is always replaced by the source name before command execution (here *cluster*, *racks* and *cpu*), for example:

```
[cluster,racks,cpu]
map: get_nodes_from_source.sh $SOURCE $GROUP
all: get_all_nodes_from_source.sh $SOURCE
list: list_nodes_from_source.sh $SOURCE
```

is equivalent to:

```
[cluster]
map: get_nodes_from_source.sh cluster $GROUP
all: get_all_nodes_from_source.sh cluster
list: list_nodes_from_source.sh cluster

[racks]
map: get_nodes_from_source.sh racks $GROUP
all: get_all_nodes_from_source.sh racks
list: list_nodes_from_source.sh racks

[cpu]
map: get_nodes_from_source.sh cpu $GROUP
all: get_all_nodes_from_source.sh cpu
list: list_nodes_from_source.sh cpu
```

Return code of external calls

Each external command might return a non-zero return code when the operation is not doable. But if the call return zero, for instance, for a non-existing group, the user will not receive any error when trying to resolve such unknown group. The desired behavior is up to the system administrator.

Slurm group bindings

Enable Slurm node group bindings by renaming the example configuration file usually installed as `/etc/clustershell/groups.conf.d/slurm.conf.example` to `slurm.conf`. Three group sources are defined in this file and are detailed below. Each section comes with a long and short names (for convenience), but actually defines a same group source.

While examples below are based on the *nodeset* tool, all Python tools using ClusterShell and the `NodeSet` class will automatically benefit from these additional node groups.

The first section **slurmpart,sp** defines a group source based on Slurm partitions. Each group is named after the partition name and contains the partition's nodes:

```
[slurmpart,sp]
map: sinfo -h -o "%N" -p $GROUP
all: sinfo -h -o "%N"
list: sinfo -h -o "%R"
reverse: sinfo -h -N -o "%R" -n $NODE
```

Example of use with *nodeset* on a cluster having two Slurm partitions named *kepler* and *pascal*:

```
$ nodeset -s sp -ll
@sp:kepler cluster-[0001-0065]
@sp:pascal cluster-[0066-0068]
```

The second section **slurmstate,st** defines a group source based on Slurm node states. Each group is based on a different state name and contains the nodes currently in that state:

```
[slurmstate,st]
map: sinfo -h -o "%N" -t $GROUP
all: sinfo -h -o "%N"
list: sinfo -h -o "%T" | tr -d '*~#$@+'
reverse: sinfo -h -N -o "%T" -n $NODE | tr -d '*~#$@+'
cache_time: 60
```

Here, *cache_time* is set to 60 seconds instead of the default (3600s) to avoid caching results in memory for too long, in case of state change (this is only useful for long-running processes, not one-shot commands).

Example of use with *nodeset* to get the current nodes that are in the Slurm state *drained*:

```
$ nodeset -f @st:drained
cluster-[0058,0067]
```

The third section **slurmjob,sj** defines a group source based on Slurm jobs. Each group is based on a running job ID and contains the nodes currently allocated for this job:

```
[slurmjob,sj]
map: squeue -h -j $GROUP -o "%N"
list: squeue -h -o "%i" -t R
reverse: squeue -h -w $NODE -o "%i"
cache_time: 60
```

The fourth section **slurmuser,su** defines a group source based on Slurm users. Each group is based on a username and contains the nodes currently allocated for jobs belonging to the username:

```
[slurmuser,su]
map: squeue -h -u $GROUP -o "%N" -t R
list: squeue -h -o "%u" -t R
reverse: squeue -h -w $NODE -o "%i"
cache_time: 60
```

Example of use with *clush* to execute a command on all nodes with running jobs of username:

```
$ clush -bw@su:username 'df -Ph /scratch'
$ clush -bw@su:username 'du -s /scratch/username'
```

cache_time is also set to 60 seconds instead of the default (3600s) to avoid caching results in memory for too long, because this group source is likely very dynamic (this is only useful for long-running processes, not one-shot commands).

You can then easily find nodes associated with a Slurm job ID:

```
$ nodeset -f @sj:686518
cluster-[0003,0005,0010,0012,0015,0017,0021,0055]
```

xCAT group bindings

Enable xCAT node group bindings by renaming the example configuration file usually installed as `/etc/clustershell/groups.conf.d/xcat.conf.example` to `xcat.conf`. A single group source is defined in this file and is detailed below.

> **Warning:** xCAT installs its own nodeset command which usually takes precedence over ClusterShell's *nodeset* command. In that case, simply use *cluset* instead.

While examples below are based on the *cluset* tool, all Python tools using ClusterShell and the *NodeSet* class will automatically benefit from these additional node groups.

The section **xcat** defines a group source based on xCAT static node groups:

```
[xcat]

# list the nodes in the specified node group
map: lsdef -s -t node $GROUP | cut -d' ' -f1

# list all the nodes defined in the xCAT tables
all: lsdef -s -t node | cut -d' ' -f1

# list all groups
list: lsdef -t group | cut -d' ' -f1
```

Example of use with *cluset*:

```
$ lsdef -s -t node dtn
sh-dtn01  (node)
sh-dtn02  (node)

$ cluset -s xcat -f @dtn
sh-dtn[01-02]
```

4.3 Library Defaults

Warning: Modifying library defaults is for advanced users only as that could change the behavior of tools using ClusterShell. Moreover, tools are free to enforce their own defaults, so changing library defaults may not change a global behavior as expected.

Since version 1.7, most defaults of the ClusterShell library may be overridden in *defaults.conf*.

The following configuration file defines ClusterShell system-wide defaults:

```
/etc/clustershell/defaults.conf
```

defaults.conf settings might then be overridden per user if one of the following files is found, in priority order:

```
$XDG_CONFIG_HOME/clustershell/defaults.conf
$HOME/.config/clustershell/defaults.conf (only if $XDG_CONFIG_HOME is not defined)
$HOME/.local/etc/clustershell/defaults.conf
```

4.3.1 Use case: rsh

If your cluster uses a rsh variant like `mrsh` or `krsh`, you may want to change it in the library defaults.

An example file is usually available in `/usr/share/doc/clustershell-*/examples/defaults.conf-rsh` and could be copied to `/etc/clustershell/defaults.conf` or to an alternate path described above. Basically, the change consists in defining an alternate distant worker by Python module name as follow:

```
[task.default]
distant_workername: Rsh
```

4.3.2 Use case: Slurm

If your cluster naming scheme has multiple dimensions, as in `node-93-02`, we recommend that you disengage some nD folding when using Slurm, which is currently unable to detect some multidimensional node indexes when not explicitly enclosed with square brackets.

To do so, define `fold_axis` to -1 in the *Library Defaults* so that nD folding is only computed on the last axis (seems to work best with Slurm):

```
[nodeset]
fold_axis: -1
```

That way, node sets computed by ClusterShell tools can be passed to Slurm without error.

CHAPTER 5

Tools

Three Python scripts using the ClusterShell library are provided with the distribution:

- *cluset* or *nodeset*, both are the same tool to manage cluster node sets and groups,
- *clush*, a powerful parallel command execution tool with output gathering,
- *clubak*, a tool to gather and display results from clush/pdsh-like output (and more).

5.1 nodeset

The *nodeset* command enables easy manipulation of node sets, as well as node groups, at the command line level. As it is very user-friendly and efficient, the *nodeset* command can quickly improve traditional cluster shell scripts. It is also full-featured as it provides most of the `NodeSet` and `RangeSet` class methods (see also *NodeSet class*, and *RangeSet class*).

Most of the examples in this section are using simple indexed node sets, however, *nodeset* supports multidimensional node sets, like *dc[1-2]n[1-99]*, introduced in version 1.7 (see *RangeSetND class* for more info).

This section will guide you through the basics and also more advanced features of *nodeset*.

5.1.1 Usage basics

One exclusive command must be specified to *nodeset*, for example:

```
$ nodeset --expand node[13-15,17-19]
node13 node14 node15 node17 node18 node19

$ nodeset --count node[13-15,17-19]
6

$ nodeset --fold node1-ipmi node2-ipmi node3-ipmi
node[1-3]-ipmi
```

Commands with inputs

Some *nodeset* commands require input (eg. node names, node sets or node groups), and some only give output. The following table shows commands that require some input:

Command	Description
`-c`, `--count`	Count and display the total number of nodes in node sets or/and node groups.
`-e`, `--expand`	Expand node sets or/and node groups as unitary node names separated by current separator string (see `--separator` option described in *Output result formatting*).
`-f`, `--fold`	Fold (compact) node sets or/and node groups into one set of nodes (by previously resolving any groups). The resulting node set is guaranteed to be free from node `--regroup` below if you want to resolve node groups in result). Please note that folding may be time consuming for multidimensional node sets.
`-r`, `--regroup`	Fold (compact) node sets or/and node groups into one set of nodes using node groups whenever possible (by previously resolving any groups). See *Node groups*.

There are three ways to give some input to the *nodeset* command:

- from command line arguments,
- from standard input (enabled when no arguments are found on command line),
- from both command line and standard input, by using the dash special argument "-" meaning you need to use stdin instead.

The following example illustrates the three ways to feed *nodeset*:

```
$ nodeset -f node1 node6 node7
node[1,6-7]

$ echo node1 node6 node7 | nodeset -f
node[1,6-7]

$ echo node1 node6 node7 | nodeset -f node0 -
node[0-1,6-7]
```

Furthermore, *nodeset*'s standard input reader is able to process multiple lines and multiple node sets or groups per line. The following example shows a simple use case:

```
$ mount -t nfs | cut -d':' -f1
nfsserv1
nfsserv2
nfsserv3

$ mount -t nfs | cut -d':' -f1 | nodeset -f
nfsserv[1-3]
```

Other usage examples of *nodeset* below show how it can be useful to provide node sets from standard input (*sinfo* is a SLURM[1] command to view nodes and partitions information and *sacct* is a command to display SLURM accounting data):

```
$ sinfo -p cuda -o '%N' -h
node[156-159]
```

(continues on next page)

[1] SLURM is an open-source resource manager (https://computing.llnl.gov/linux/slurm/)

(continued from previous page)

```
$ sinfo -p cuda -o '%N' -h | nodeset -e
node156 node157 node158 node159

$ for node in $(sinfo -p cuda -o '%N' -h | nodeset -e); do
        sacct -a -N $node > /tmp/cudajobs.$node;
  done
```

Previous rules also apply when working with node groups, for example when using `nodeset -r` reading from standard input (and a matching group is found):

```
$ nodeset -f @gpu
node[156-159]

$ sinfo -p cuda -o '%N' -h | nodeset -r
@gpu
```

Most commands described in this section produce output results that may be formatted using `--output-format` and `--separator` which are described in *Output result formatting*.

Commands with no input

The following table shows all other commands that are supported by *nodeset*. These commands don't support any input (like node sets), but can still recognize options as specified below.

Command w/o input	Description
`-l`, `--list`	List node groups from selected *group source* as specified with `-s` or `--groupsource`. If not specified, node groups from the default *group source* are listed (see *groups configuration* for default *group source* configuration).
`--groupsources`	List all configured *group sources*, one per line, as configured in *groups.conf* (see *groups configuration*). The default *group source* is appended with ‘‘ (default)‘‘, unless the `-q`, `--quiet` option is specified. This command is mainly here to avoid reading any configuration files, or to check if all work fine when configuring *group sources*.

Output result formatting

When using the expand command (`-e, --expand`), a separator string is used when displaying results. The option `-S, --separator` allows you to modify it. The specified string is interpreted, so that you can use special characters as separator, like `\n` or `\t`. The default separator is the space character ″ ″. This is an example showing such separator string change:

```
$ nodeset -e --separator='\n' node[0-3]
node0
node1
node2
node3
```

The `-O, --output-format` option can be used to format output results of most *nodeset* commands. The string passed to this option is used as a base format pattern applied to each node or each result (depending on the command and other options requested). The default format string is *"%s"*. Formatting is performed using the Python builtin

string formatting operator, so you must use one format operator of the right type (*%s* is guaranteed to work in all cases). Here is an output formatting example when using the expand command:

```
$ nodeset --output-format='%s-ipmi' -e node[1-2]x[1-2]
node1x1-ipmi node1x2-ipmi node2x1-ipmi node2x2-ipmi
```

Output formatting and separator combined can be useful when using the expand command, as shown here:

```
$ nodeset -O '%s-ipmi' -S '\n' -e node[1-2]x[1-2]
node1x1-ipmi
node1x2-ipmi
node2x1-ipmi
node2x2-ipmi
```

When using the output formatting option along with the folding command, the format is applied to each node but the result is still folded:

```
$ nodeset -O '%s-ipmi' -f mgmt1 mgmt2 login[1-4]
login[1-4]-ipmi,mgmt[1-2]-ipmi
```

5.1.2 Stepping and auto-stepping

The *nodeset* command, as does the *clush* command, is able to recognize by default a factorized notation for range sets of the form *a-b/c*, indicating a list of integers starting from *a*, less than or equal to *b* with the increment (step) *c*.

For example, the *0-6/2* format indicates a range of 0-6 stepped by 2; that is 0,2,4,6:

```
$ nodeset -e node[0-6/2]
node0 node2 node4 node6
```

However, by default, *nodeset* never uses this stepping notation in output results, as other cluster tools seldom if ever support this feature. Thus, to enable such factorized output in *nodeset*, you must specify `--autostep=AUTOSTEP` to set an auto step threshold number when folding nodesets (ie. when using `-f` or `-r`). This threshold number (AUTOSTEP) is the minimum occurrence of equally-spaced integers needed to enable auto-stepping.

For example:

```
$ nodeset -f --autostep=3 node1 node3 node5
node[1-5/2]

$ nodeset -f --autostep=4 node1 node3 node5
node[1,3,5]
```

It is important to note that resulting node sets with enabled auto-stepping never create overlapping ranges, for example:

```
$ nodeset -f --autostep=3 node1 node5 node9 node13
node[1-13/4]

$ nodeset -f --autostep=3 node1 node5 node7 node9 node13
node[1,5-9/2,13]
```

However, any ranges given as input may still overlap (in this case, *nodeset* will automatically spread them out so that they do not overlap), for example:

```
$ nodeset -f --autostep=3 node[1-13/4,7]
node[1,5-9/2,13]
```

A minimum node count threshold **percentage** before autostep is enabled may also be specified as autostep value (or `auto` which is currently 100%). In the two following examples, only the first 4 of the 7 indexes may be represented using the step syntax (57% of them):

```
$ nodeset -f --autostep=50% node[1,3,5,7,34,39,99]
node[1-7/2,34,39,99]

$ nodeset -f --autostep=90% node[1,3,5,7,34,39,99]
node[1,3,5,7,34,39,99]
```

5.1.3 Zero-padding

Sometimes, cluster node names are padded with zeros (eg. *node007*). With *nodeset*, when leading zeros are used, resulting host names or node sets are automatically padded with zeros as well. For example:

```
$ nodeset -e node[08-11]
node08 node09 node10 node11

$ nodeset -f node001 node002 node003 node005
node[001-003,005]
```

Zero-padding and stepping (as seen in *Stepping and auto-stepping*) together are also supported, for example:

```
$ nodeset -e node[000-012/4]
node000 node004 node008 node012
```

Nevertheless, care should be taken when dealing with padding, as a zero-padded node name has priority over a normal one, for example:

```
$ nodeset -f node1 node02
node[01-02]
```

To clarify, *nodeset* will always try to coalesce node names by their numerical index first (without taking care of any zero-padding), and then will use the first zero-padding rule encountered. In the following example, the first zero-padding rule found is *node01*'s one:

```
$ nodeset -f node01 node002
node[01-02]
```

That said, you can see it is not possible to mix *node01* and *node001* in the same node set (not supported by the `NodeSet` class), but that would be a tricky case anyway!

5.1.4 Leading and trailing digits

Version 1.7 introduces improved support for bracket leading and trailing digits. Those digits are automatically included within the range set, allowing all node set operations to be fully supported.

Examples with bracket leading digits:

```
$ nodeset -f node-00[00-99]
node-[0000-0099]

$ nodeset -f node-01[01,09,42]
node-[0101,0109,0142]
```

Examples with bracket trailing digits:

```
$ nodeset -f node-[1-2]0-[0-2]5
node-[10,20]-[05,15,25]
```

Examples with both bracket leading and trailing digits:

```
$ nodeset -f node-00[1-6]0
node-[0010,0020,0030,0040,0050,0060]

$ nodeset --autostep=auto -f node-00[1-6]0
node-[0010-0060/10]
```

Still, using this syntax can be error-prone especially if used with node sets without 0-padding or with the */step* syntax and also requires additional processing by the parser. In general, we recommend writing the whole rangeset inside the brackets.

Warning: Using the step syntax (seen above) within a bracket-delimited range set is not compatible with **trailing** digits. For instance, this is **not** supported: `node-00[1-6/2]0`

5.1.5 Arithmetic operations

As a preamble to this section, keep in mind that all operations can be repeated/mixed within the same *nodeset* command line, they will be processed from left to right.

Union operation

Union is the easiest arithmetic operation supported by *nodeset*: there is no special command line option for that, just provide several node sets and the union operation will be computed, for example:

```
$ nodeset -f node[1-3] node[4-7]
node[1-7]

$ nodeset -f node[1-3] node[2-7] node[5-8]
node[1-8]
```

Other operations

As an extension to the above, other arithmetic operations are available by using the following command-line options (*working set* is the node set currently processed on the command line – always from left to right):

nodeset command option	Operation
`-x NODESET`, `--exclude=NODESET`	compute a new set with elements in *working set* but not in `NODESET`
`-i NODESET`, `--intersection=NODESET`	compute a new set with elements common to *working set* and `NODESET`
`-X NODESET`, `--xor=NODESET`	compute a new set with elements that are in exactly one of the *working set* and `NODESET`

If rangeset mode (`-R`) is turned on, all arithmetic operations are supported by replacing `NODESET` by any `RANGESET`. See *Range sets* for more info about *nodeset*'s rangeset mode.

Arithmetic operations usage examples:

```
$ nodeset -f node[1-9] -x node6
node[1-5,7-9]

$ nodeset -f node[1-9] -i node[6-11]
node[6-9]

$ nodeset -f node[1-9] -X node[6-11]
node[1-5,10-11]

$ nodeset -f node[1-9] -x node6 -i node[6-12]
node[7-9]
```

Extended patterns support

nodeset does also support arithmetic operations through its "extended patterns" (inherited from `NodeSet` extended pattern feature, see *Extended String Pattern*, there is an example of use:

```
$ nodeset -f node[1-4],node[5-9]
node[1-9]

$ nodeset -f node[1-9]\!node6
node[1-5,7-9]

$ nodeset -f node[1-9]\&node[6-12]
node[6-9]

$ nodeset -f node[1-9]^node[6-11]
node[1-5,10-11]
```

5.1.6 Special operations

A few special operations are currently available: node set slicing, splitting on a predefined node count, splitting non-contiguous subsets, choosing fold axis (for multidimensional node sets) and picking N nodes randomly. They are all explained below.

Slicing

Slicing is a way to select elements from a node set by their index (or from a range set when using `-R` toggle option, see *Range sets*. In this case actually, and because *nodeset*'s underlying `NodeSet` class sorts elements as observed after folding (for example), the word *set* may sound like a stretch of language (a *set* isn't usually sorted). Indeed, `NodeSet` further guarantees that its iterator will traverse the set in order, so we should see it as a *ordered set*. The following simple example illustrates this sorting behavior:

```
$ nodeset -f b2 b1 b0 b c a0 a
a,a0,b,b[0-2],c
```

Slicing is performed through the following command-line option:

nodeset command option	Operation
`-I RANGESET`, `--slice=RANGESET`	*slicing*: get sliced off result, selecting elements from provided rangeset's indexes

Some slicing examples are shown below:

```
$ nodeset -f -I 0 node[4-8]
node4

$ nodeset -f --slice=0 bnode[0-9] anode[0-9]
anode0

$ nodeset -f --slice=1,4,7,9,15 bnode[0-9] anode[0-9]
anode[1,4,7,9],bnode5

$ nodeset -f --slice=0-18/2 bnode[0-9] anode[0-9]
anode[0,2,4,6,8],bnode[0,2,4,6,8]
```

Splitting into *n* subsets

Splitting a node set into several parts is often useful to get separate groups of nodes, for instance when you want to check MPI comm between nodes, etc. Based on *NodeSet.split()* method, the *nodeset* command provides the following additional command-line option (since v1.4):

nodeset command option	Operation
`--split=MAXSPLIT`	*splitting*: split result into a number of subsets

`MAXSPLIT` is an integer specifying the number of separate groups of nodes to compute. Input's node set is divided into smaller groups, whenever possible with the same size (only the last ones may be smaller due to rounding). Obviously, if `MAXSPLIT` is higher than or equal to the number N of elements in the set, then the set is split to N single sets.

Some node set splitting examples:

```
$ nodeset -f --split=4 node[0-7]
node[0-1]
node[2-3]
node[4-5]
node[6-7]

$ nodeset -f --split=4 node[0-6]
node[0-1]
node[2-3]
node[4-5]
node6

$ nodeset -f --split=10000 node[0-4]
foo0
foo1
foo2
foo3
foo4

$ nodeset -f --autostep=3 --split=2 node[0-38/2]
node[0-18/2]
node[20-38/2]
```

Splitting off non-contiguous subsets

It can be useful to split a node set into several contiguous subsets (with same pattern name and contiguous range indexes, eg. *node[1-100]* or *dc[1-4]node[1-100]*). The `--contiguous` option allows you to do that. It is based on *NodeSet.contiguous()* method, and should be specified with standard commands (fold, expand, count, regroup). The following example shows how to split off non-contiguous subsets of a specified node set, and to display each resulting contiguous node set in a folded manner to separated lines:

```
$ nodeset -f --contiguous node[1-100,200-300,500]
node[1-100]
node[200-300]
node500
```

Similarly, the following example shows how to display each resulting contiguous node set in an expanded manner to separate lines:

```
$ nodeset -e --contiguous node[1-9,11-19]
node1 node2 node3 node4 node5 node6 node7 node8 node9
node11 node12 node13 node14 node15 node16 node17 node18 node19
```

Choosing fold axis (nD)

The default folding behavior for multidimensional node sets is to fold along all *nD* axis. However, other cluster tools barely support nD nodeset syntax, so it may be useful to fold along one (or a few) axis only. The `--axis` option allows you to specify indexes of dimensions to fold. Using this option, rangesets of unspecified axis there won't be folded. Please note however that the obtained result may be suboptimal, this is because *NodeSet* algorithms are optimized for folding along all axis. `--axis` value is a set of integers from 1 to n representing selected nD axis, in the form of a number or a rangeset. A common case is to restrict folding on a single axis, like in the following simple examples:

```
$ nodeset --axis=1 -f node1-ib0 node2-ib0 node1-ib1 node2-ib1
node[1-2]-ib0,node[1-2]-ib1

$ nodeset --axis=2 -f node1-ib0 node2-ib0 node1-ib1 node2-ib1
node1-ib[0-1],node2-ib[0-1]
```

Because a single nodeset may have several different dimensions, axis indices are silently truncated to fall in the allowed range. Negative indices are useful to fold along the last axis whatever number of dimensions used:

```
$ nodeset --axis=-1 -f comp-[1-2]-[1-36],login-[1-2]
comp-1-[1-36],comp-2-[1-36],login-[1-2]
```

See also the *Use case: Slurm* of Library Defaults for changing it permanently.

Picking N node(s) at random

Use `--pick` with a maximum number of nodes you wish to pick randomly from the resulting node set (or from the resulting range set with `-R`):

```
$ nodeset --pick=1 -f node11 node12 node13
node12
$ nodeset --pick=2 -f node11 node12 node13
node[11,13]
```

5.1.7 Node groups

This section tackles the node groups feature available more particularly through the *nodeset* command-line tool. The ClusterShell library defines a node groups syntax and allow you to bind these group sources to your applications (cf. *node groups configuration*). Having those group sources, group provisioning is easily done through user-defined external shell commands. Thus, node groups might be very dynamic and their nodes might change very often. However, for performance reasons, external call results are still cached in memory to avoid duplicate external calls during *nodeset* execution. For example, a group source can be bound to a resource manager or a custom cluster database.

For further details about using node groups in Python, please see *Node groups*. For advanced usage, you should also be able to define your own group source directly in Python (cf. *Overriding default groups configuration*).

Node group expression rules

The general node group expression is `@source:groupname`. For example, `@slurm:bigmem` represents the group *bigmem* of the group source *slurm*. Moreover, a shortened expression is available when using the default group source (defined by configuration); for instance `@compute` represents the *compute* group of the default group source.

Valid group source names and group names can contain alphanumeric characters, hyphens and underscores (no space allowed). Indeed, same rules apply to node names.

Listing group sources

As already mentioned, the following *nodeset* command is available to list configured group sources and also display the default group source (unless `-q` is provided):

```
$ nodeset --groupsources
local (default)
genders
slurm
```

Listing group names

If the **list** external shell command is configured (see *node groups configuration*), it is possible to list available groups *from the default source* with the following commands:

```
$ nodeset -l
@mgnt
@mds
@oss
@login
@compute
```

Or, to list groups *from a specific group source*, use *-l* in conjunction with *-s* (or *–groupsource*):

```
$ nodeset -l -s slurm
@slurm:parallel
@slurm:cuda
```

Or, to list groups *from all available group sources*, use *-L* (or *–list-all*):

```
$ nodeset -L
@mgnt
@mds
```

(continues on next page)

(continued from previous page)

```
@oss
@login
@compute
@slurm:parallel
@slurm:cuda
```

You can also use `nodeset -ll` or `nodeset -LL` to see each group's associated node sets.

Using node groups in basic commands

The use of node groups with the *nodeset* command is very straightforward. Indeed, any group name, prefixed by @ as mentioned above, can be used in lieu of a node name, where it will be substituted for all nodes in that group.

A first, simple example is a group expansion (using default source) with *nodeset*:

```
$ nodeset -e @oss
node40 node41 node42 node43 node44 node45
```

The *nodeset* count command works as expected:

```
$ nodeset -c @oss
6
```

Also *nodeset* folding command can always resolve node groups:

```
$ nodeset -f @oss
node[40-45]
```

There are usually two ways to use a specific group source (need to be properly configured):

```
$ nodeset -f @slurm:parallel
node[50-81]

$ nodeset -f -s slurm @parallel
node[50-81]
```

Finding node groups

As an extension to the **list** command, you can search node groups that a specified node set belongs to with `nodeset -l[ll]` as follow:

```
$ nodeset -l node40
@all
@oss

$ nodeset -ll node40
@all node[1-159]
@oss node[40-45]
```

This feature is implemented with the help of the *`NodeSet.groups()`* method (see *Finding node groups* for further details).

Resolving node groups

If needed group configuration conditions are met (cf. *node groups configuration*), you can try group lookups thanks to the `-r, --regroup` command. This feature is implemented with the help of the *`NodeSet.regroup()`* method (see *Regrouping node sets* for further details). Only exact matching groups are returned (all containing nodes needed), for example:

```
$ nodeset -r node[40-45]
@oss

$ nodeset -r node[0,40-45]
@mgnt,@oss
```

When resolving node groups, *nodeset* always returns the largest groups first, instead of several smaller matching groups, for instance:

```
$ nodeset -ll
@login node[50-51]
@compute node[52-81]
@intel node[50-81]

$ nodeset -r node[50-81]
@intel
```

If no matching group is found, `nodeset -r` still returns folded result (as does `-f`):

```
$ nodeset -r node40 node42
node[40,42]
```

Indexed node groups

Node groups are themselves some kind of group sets and can be indexable. To use this feature, node groups external shell commands need to return indexed group names (automatically handled by the library as needed). For example, take a look at these indexed node groups:

```
$ nodeset -l
@io1
@io2
@io3

$ nodeset -f @io[1-3]
node[40-45]
```

Arithmetic operations on node groups

Arithmetic and special operations (as explained for node sets in nodeset-arithmetic and nodeset-special are also supported with node groups. Any group name can be used in lieu of a node set, where it will be substituted for all nodes in that group before processing requested operations. Some typical examples are:

```
$ nodeset -f @lustre -x @mds
node[40-45]

$ nodeset -r @lustre -x @mds
@oss
```

(continues on next page)

(continued from previous page)

```
$ nodeset -r -a -x @lustre
@compute,@login,@mgnt
```

More advanced examples, with the use of node group sets, follow:

```
$ nodeset -r @io[1-3] -x @io2
@io[1,3]

$ nodeset -f -I0 @io[1-3]
node40

$ nodeset -f --split=3 @oss
node[40-41]
node[42-43]
node[44-45]

$ nodeset -r --split=3 @oss
@io1
@io2
@io3
```

Extended patterns support with node groups

Even for node groups, the *nodeset* command supports arithmetic operations through its *extended pattern* feature (see *Extended String Pattern*). A first example illustrates node groups intersection, that can be used in practice to get nodes available from two dynamic group sources at a given time:

```
$ nodeset -f @db:prod\&@compute
```

The following fictive example computes a folded node set containing nodes found in node group `@gpu` and `@slurm:bigmem`, but not in both, minus the nodes found in odd `@chassis` groups from 1 to 9 (computed from left to right):

```
$ nodeset -f @gpu^@slurm:bigmem\!@chassis[1-9/2]
```

Also, version 1.7 introduces a notation extension `@*` (or `@SOURCE:*`) that has been added to quickly represent *all nodes* (please refer to *Selecting all nodes* for more details).

Selecting all nodes

The option `-a` (without argument) can be used to select **all** nodes from a group source (see *node groups configuration* for more details on special **all** external shell command upcall). Example of use for the default group source:

```
$ nodeset -a -f
example[4-6,32-159]
```

Use `-s/--groupsource` to select another group source.

If not properly configured, the `-a` option may lead to runtime errors like:

```
$ nodeset -s mybrokensource -a -f
nodeset: External error: Not enough working methods (all or map + list)
    to get all nodes
```

A similar option is available with *clush*, see *selecting all nodes with clush*.

Node wildcards

ClusterShell 1.8 introduces node wildcards: `*` means match zero or more characters of any type; `?` means match exactly one character of any type.

Any wildcard mask found is matched against **all** nodes from the group source (see *Selecting all nodes*).

This can be especially useful for server farms, or when cluster node names differ. Say that your *group configuration* is set to return the following "all nodes":

```
$ nodeset -f -a
bckserv[1-2],dbserv[1-4],wwwserv[1-9]
```

Then, you can use wildcards to select particular nodes, as shown below:

```
$ nodeset -f 'www*'
wwwserv[1-9]

$ nodeset -f 'www*[1-4]'
wwwserv[1-4]

$ nodeset -f '*serv1'
bckserv1,dbserv1,wwwserv1
```

Wildcard masks are resolved prior to *extended patterns*, but each mask is evaluated as a whole node set operand. In the example below, we select all nodes matching `*serv*` before removing all nodes matching `www*`:

```
$ nodeset  -f '*serv*!www*'
bckserv[1-2],dbserv[1-4]
```

5.1.8 Range sets

Working with range sets

By default, the *nodeset* command works with node or group sets and its functionality match most `NodeSet` class methods. Similarly, *nodeset* will match `RangeSet` methods when you make use of the `-R` option switch. In that case, all operations are restricted to numerical ranges. For example, to expand the range "`1-10`", you should use:

```
$ nodeset -e -R 1-10
1 2 3 4 5 6 7 8 9 10
```

Almost all commands and operations available for node sets are also available with range sets. The only restrictions are commands and operations related to node groups. For instance, the following command options are **not** available with `nodeset -R`:

- `-r, --regroup` as this feature is obviously related to node groups,
- `-a / --all` as the **all** external call is also related to node groups.

Using range sets instead of node sets doesn't change the general command usage, like the need of one command option presence (cf. nodeset-commands), or the way to give some input (cf. nodeset-stdin), for example:

```
$ echo 3 2 36 0 4 1 37 | nodeset -fR
0-4,36-37

$ echo 0-8/4 | nodeset -eR -S'\n'
0
4
8
```

Stepping and auto-stepping are supported (cf. *Stepping and auto-stepping*) and also zero-padding (cf. nodeset-zpad), which are both `RangeSet` class features anyway.

The following examples illustrate these last points:

```
$ nodeset -fR 03 05 01 07 11 09
01,03,05,07,09,11

$ nodeset -fR --autostep=3 03 05 01 07 11 09
01-11/2
```

Arithmetic and special operations

All arithmetic operations, as seen for node sets (cf. nodeset-arithmetic), are available for range sets, for example:

```
$ nodeset -fR 1-14 -x 10-20
1-9

$ nodeset -fR 1-14 -i 10-20
10-14

$ nodeset -fR 1-14 -X 10-20
1-9,15-20
```

For now, there is no *extended patterns* syntax for range sets as for node sets (cf. *Extended patterns support*). However, as the union operator `,` is available natively by design, such expressions are still allowed:

```
$ nodeset -fR 4-10,1-2
1-2,4-10
```

Besides arithmetic operations, special operations may be very convenient for range sets also. Below is an example with `-I / --slice` (cf. nodeset-slice):

```
$ nodeset -fR -I 0 100-131
100

$ nodeset -fR -I 0-15 100-131
100-115
```

There is another special operation example with `--split` (cf. nodeset-splitting-n):

```
$ nodeset -fR --split=2 100-131
100-115
116-131
```

Finally, an example of the special operation `--contiguous` (cf. nodeset-splitting-contiguous):

```
$ nodeset -f -R --contiguous 1-9,11,13-19
1-9
11
13-19
```

rangeset alias

When using *nodeset* with range sets intensively (eg. for scripting), it may be convenient to create a local command alias, as shown in the following example (Bourne shell), making it sort of a super seq(1) command:

```
$ alias rangeset='nodeset -R'
$ rangeset -e 0-8/2
0 2 4 6 8
```

5.2 cluset

The *cluset* command is the same as *nodeset* and has been added in ClusterShell 1.7.3 to avoid a conflict with xCAT's nodeset command.

5.3 clush

clush is a program for executing commands in parallel on a cluster and for gathering their results. It can execute commands interactively or can be used within shell scripts and other applications. It is a partial front-end to the `Task` class of the ClusterShell library (cf. *Structure of Task*). *clush* currently makes use of the Ssh worker of ClusterShell that only requires *ssh(1)* (we tested with OpenSSH SSH client).

Some features of *clush* command line tool are:

- two modes of parallel cluster commands execution:
  - *flat mode*: sliding window of local or remote (eg. *ssh(1)*) commands
  - *tree mode*: commands propagated to the targets through a tree of pre-configured gateways; gateways are then using a sliding window of local or *ssh(1)* commands to reach the targets (if the target count per gateway is greater than the *fanout* value)
- smart display of command results (integrated output gathering, sorting by node, nodeset or node groups)
- standard input redirection to remote nodes
- files copying in parallel
- *pdsh*[1] options backward compatibility

clush can be started non-interactively to run a shell command, or can be invoked as an interactive shell. Both modes are discussed here (clush-oneshot clush-interactive).

5.3.1 Target and filter nodes

clush offers different ways to select or filter target nodes through command line options or files containing a list of hosts.

[1] LLNL parallel remote shell utility (https://computing.llnl.gov/linux/pdsh.html)

Command line options

The `-w` option allows you to specify remote hosts by using ClusterShell `NodeSet` syntax, including the node groups *@group* special syntax (cf. *Node group expression rules*) and the Extended String Patterns syntax (see *Extended String Pattern*) to benefits from `NodeSet` basic arithmetic (like `@Agroup&@Bgroup`). Additionally, the `-x` option allows you to exclude nodes from remote hosts list (the same NodeSet syntax can be used here). Nodes exclusion has priority over nodes addition.

Using node groups

If you have ClusterShell *node groups* configured on your cluster, any node group syntax may be used in place of nodes for `-w` as well as `-x`.

For example:

```
$ clush -w @rhel6 cat /proc/loadavg
node26: 0.02 0.01 0.00 1/202 23042
```

For *pdsh* backward compatibility, *clush* supports two `-g` and `-X` options to respectively select and exclude nodes group(s), but only specified by omitting any `"@"` group prefix (see example below). In general, though, it is advised to use the @-prefixed group syntax as the non-prefixed notation is only recognized by *clush* but not by other tools like *nodeset*.

For example:

```
$ clush -g rhel6 cat /proc/loadavg
node26: 0.02 0.01 0.00 1/202 23033
```

Selecting all nodes

The special option `-a` (without argument) can be used to select **all** nodes, in the sense of ClusterShell node groups (see *node groups configuration* for more details on special **all** external shell command upcall). If not properly configured, the `-a` option may lead to a runtime error like:

```
clush: External error: Not enough working external calls (all, or map +
list) defined to get all node
```

Picking node(s) at random

Use `--pick` with a maximum number of nodes you wish to pick randomly from the targeted node set. **clush** will then run only on selected node(s). The following example will run a script on a single random node picked from the `@compute` group:

```
$ clush -w @compute --pick=1 ./nonreg-single-client-fs-io.sh
```

Host files

The option `--hostfile` (or `--machinefile`) may be used to specify a path to a file containing a list of single hosts, node sets or node groups, separated by spaces and lines. It may be specified multiple times (one per file).

For example:

```
$ clush --hostfile ./host_file -b systemctl is-enabled httpd
```

This option has been added as backward compatibility with other parallel shell tools. Indeed, ClusterShell provides a preferred way to provision node sets from node group sources and flat files to all cluster tools using `NodeSet` (including *clush*). Please see *node groups configuration*.

Note: Use `--debug` or `-d` to see resulting node sets from host files.

5.3.2 Flat execution mode

The default execution mode is to launch commands (local or remote) in parallel, up to a certain limit fixed by the *fanout* value, which is the number of child processes allowed to run at a time. This "sliding window" of active commands is a common technique used on large clusters to conserve resources on the initiating host, while allowing some commands to time out. If used with *ssh(1)*, this does actually limit the number of concurrent ssh connections.

Fanout (sliding window)

The `--fanout` (or `-f`) option of **clush** allows the user to change the default *fanout* value defined in *clush.conf* or in the *library defaults* if not specified.

Indeed, it is sometimes useful to change the fanout value for a specific command, for example to avoid flooding a remote service with concurrent requests generated by that actual command.

The following example will launch up to ten *puppet* commands at a time on the node group named *@compute*:

```
$ clush -w @compute -f 10 puppet agent -t
```

If the fanout value is set to 1, commands are executed sequentially:

```
$ clush -w node[40-42] -f 1 'date +%s; sleep 1'
node40: 1505366138
node41: 1505366139
node42: 1505366140
```

5.3.3 Tree execution mode

ClusterShell's tree execution mode is a major horizontal scalability improvement by providing a hierarchical command propagation scheme.

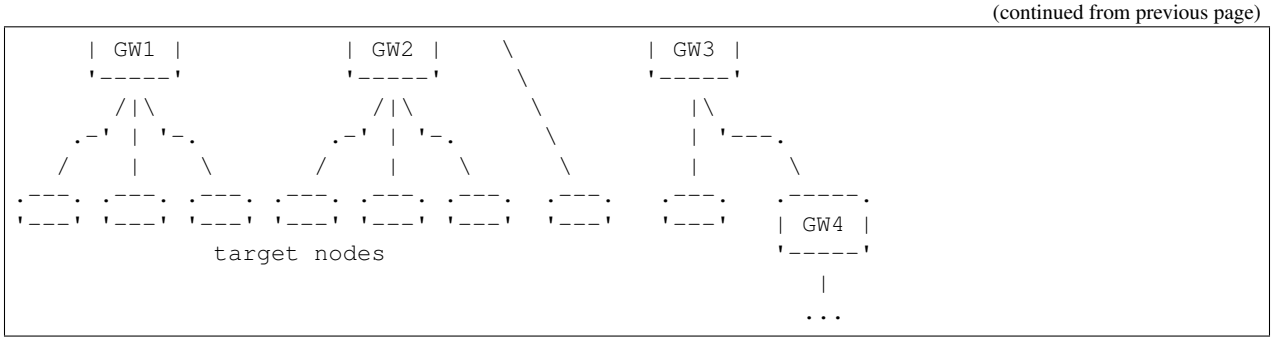
The Tree mode of ClusterShell has been the subject of this paper presented at the Ottawa Linux Symposium Conference in 2012 and at the PyHPC 2013 workshop in Denver, USA.

The diagram below illustrates the hierarchical command propagation principle with a head node, gateways (GW) and target nodes:

```
                    .-----------.
                    | Head node |
                    '-----------'
                         /|\
          .-------------' | '--.------------.
         /                |     \            \
    .-----.            .-----.   \           .-----.
```

(continues on next page)

(continued from previous page)



The Tree mode is implemented at the library level, so that all applications using ClusterShell may benefits from it. However, this section describes how to use the tree mode with the **clush** command only.

Configuration

The system-wide library configuration file **/etc/clustershell/topology.conf** defines the routes of default command propagation tree. It is recommended that all connections between parent and children nodes are carefully pre-configured, for example, to avoid any SSH warnings when connecting (if using the default SSH remote worker, of course).

The content of the topology.conf file should look like this:

```
[routes]
rio0: rio[10-13]
rio[10-11]: rio[100-240]
rio[12-13]: rio[300-440]
```

This file defines the following topology graph:

```
rio0
|- rio[10-11]
|  `- rio[100-240]
`- rio[12-13]
   `- rio[300-440]
```

At runtime, ClusterShell will pick an initial propagation tree from this topology graph definition and the current root node. Multiple admin/root nodes may be defined in the file.

Note: The algorithm used in Tree mode does not rely on gateway system hostnames anymore. In topology.conf, just use the hosts or aliases needed to connect to each node.

Enabling tree mode

Since version 1.7, the tree mode is enabled by default when a configuration file is present. When the configuration file **/etc/clustershell/topology.conf** exists, *clush* will use it by default for target nodes that are defined there. The topology file path can be changed using the `--topology` command line option.

Note: If using `clush -d` (debug option), clush will display an ASCII representation of the initial propagation tree used. This is useful when working on Tree mode configuration.

Enabling tree mode should be as much transparent as possible to the end user. Most **clush** options, including options defined in *clush.conf* or specified using `-O` or `-o` (ssh options) are propagated to the gateways and taken into account there.

Tree mode specific options

The `--remote=yes|no` command line option controls the remote execution behavior:

- Default is **yes**, that will make *clush* establish connections up to the leaf nodes using a *distant worker* like *ssh*.
- Changing it to **no** will make *clush* establish connections up to the leaf parent nodes only, then the commands are executed locally on the gateways (like if it would be with `--worker=exec` on the gateways themselves). This execution mode allows users to schedule remote commands on gateways that take a node as an argument. On large clusters, this is useful to spread the load and resources used of one-shot monitoring, IPMI, or other commands on gateways. A simple example of use is:

```
$ clush -w node[100-199] --remote=no /usr/sbin/ipmipower -h %h-ipmi -s
```

  This command is also valid if you don't have any tree configured, because in that case, `--remote=no` is an alias of `--worker=exec` worker.

The `--grooming` command line option allows users to change the grooming delay (float, in seconds). This feature allows gateways to aggregate responses received within a certain timeframe before transmitting them back to the root node in a batch fashion. This contributes to reducing the load on the root node by delegating the first steps of this CPU intensive task to the gateways.

Fanout considerations

ClusterShell uses a "sliding window" or *fanout* of processes to avoid too many concurrent connections and to conserve resources on the initiating hosts. See *Flat execution mode* for more details about this.

In tree mode, the same *fanout* value is used on the head node and on each gateway. That is, if the *fanout* is **16**, each gateway will initiate up to **16** connections to their target nodes at the same time.

Note: This is likely to **change** in the future, as it makes the *fanout* behaviour different if you are using the tree mode or not. For example, some administrators are using a *fanout* value of 1 to "sequentialize" a command on the cluster. In tree mode, please note that in that case, each gateway will be able to run a command at the same time.

Remote Python executable

You must use the same major version of Python on the gateways and the root node. By default, the same python executable name than the one used on the root node will be used to launch the gateways, that is, *python* or *python3* (using relative path for added flexibility). You may override the selection of the remote Python interpreter by defining the following environment variable:

```
$ export CLUSTERSHELL_GW_PYTHON_EXECUTABLE=/path/to/python3
```

Note: It is highly recommended to have the same Python interpeter installed on all gateways and the root node.

Debugging Tree mode

To debug Tree mode, you can define the following environment variable before running **clush** (or any other applications using ClusterShell):

```
$ export CLUSTERSHELL_GW_LOG_LEVEL=DEBUG  (default value is INFO)
$ export CLUSTERSHELL_GW_LOG_DIR=/tmp     (default value is /tmp)
```

This will generate log files of the form `$HOSTNAME.gw.log` in `CLUSTERSHELL_GW_LOG_DIR`.

5.3.4 Non-interactive (or one-shot) mode

When *clush* is started non-interactively, the command is executed on the specified remote hosts in parallel (given the current *fanout* value and the number of commands to execute (see *fanout* library settings in *Configuring the Task object*).

Output gathering options

If option `-b` or `--dshbak` is specified, *clush* waits for command completion while displaying a *progress indicator* and then displays gathered output results. If standard output is redirected to a file, *clush* detects it and disable any progress indicator.

> **Warning:** *clush* will only consolidate identical command outputs if the command return codes are also the same.

The following is a simple example of *clush* command used to execute `uname -r` on *node40*, *node41* and *node42*, wait for their completion and finally display digested output results:

```
$ clush -b -w node[40-42] uname -r
---------------
node[40-42]
---------------
2.6.35.6-45.fc14.x86_64
```

It is common to cancel such command execution because a node is hang. When using *pdsh* and *dshbak*, due to the pipe, all nodes output will be lost, even if all nodes have successfully run the command. When you hit CTRL-C with *clush*, the task is canceled but received output is not lost:

```
$ clush -b -w node[1-5] uname -r
Warning: Caught keyboard interrupt!
---------------
node[2-4] (3)
---------------
2.6.31.6-145.fc11
---------------
node5
---------------
2.6.18-164.11.1.el5
Keyboard interrupt (node1 did not complete).
```

Performing *diff* of cluster-wide outputs

Since version 1.6, you can use the `--diff` *clush* option to show differences between common outputs. This feature is implemented using Python unified diff. This special option implies `-b` (gather common stdout outputs) but you don't need to specify it. Example:

```
$ clush -w node[40-42] --diff dmidecode -s bios-version
--- node[40,42] (2)
+++ node41
@@ -1,1 +1,1 @@
-1.0.5S56
+1.1c
```

A nodeset is automatically selected as the "reference nodeset" according to these criteria:

1. lowest command return code (to discard failed commands)
2. largest nodeset with the same output result
3. otherwise the first nodeset is taken (ordered (1) by name and (2) lowest range indexes)

Standard input bindings

Unless the option `--nostdin` (or `-n`) is specified, *clush* detects when its standard input is connected to a terminal (as determined by *isatty(3)*). If actually connected to a terminal, *clush* listens to standard input when commands are running, waiting for an Enter key press. Doing so will display the status of current nodes. If standard input is not connected to a terminal, and unless the option `--nostdin` (or `-n`) is specified, *clush* binds the standard input of the remote commands to its own standard input, allowing scripting methods like:

```
$ echo foo | clush -w node[40-42] -b cat
---------------
node[40-42]
---------------
foo
```

Another stdin-bound *clush* usage example:

```
$ ssh node10 'ls /etc/yum.repos.d/*.repo' | clush -w node[11-14] -b xargs ls
---------------
node[11-14] (4)
---------------
/etc/yum.repos.d/cobbler-config.repo
```

Note: Use `--nostdin` (or `-n`) in the same way you would use `ssh -n` to disable standard input. Indeed, if this option is set, EOF is sent at first read, as if stdin were actually connected to /dev/null.

Progress indicator

In *output gathering mode*, *clush* will display a live progress indicator as a simple but convenient way to follow the completion of parallel commands. It can be disabled just by using the `-q` or `--quiet` options. The progress indicator will appear after 1 to 2 seconds and should look like this:

```
clush: <command_completed>/<command_total>
```

If writing is performed to *clush* standard input, like in `command | clush`, the live progress indicator will display the global bandwidth of data written to the target nodes.

Finally, the special option `--progress` can be used to force the display of the live progress indicator. Using this option may interfere with some command outputs, but it can be useful when using stdin while remote commands are silent. As an example, the following command will copy a local file to node[1-3] and display the global write bandwidth to the target nodes:

```
$ dd if=/path/to/local/file | clush -w node[1-3] --progress 'dd of=/path/to/remote/
↪file'
clush: 0/3 write: 212.27 MiB/s
```

5.3.5 Interactive mode

If a command is not specified, *clush* runs interactively. In this mode, *clush* uses the *GNU readline* library to read command lines from the terminal. *Readline* provides commands for searching through the command history for lines containing a specified string. For instance, you can type *Control-R* to search in the history for the next entry matching the search string typed so far.

Single-character interactive commands

clush also recognizes special single-character prefixes that allows the user to see and modify the current nodeset (the nodes where the commands are executed). These single-character interactive commands are detailed below:

Interactive special commands	Comment
`clush> ?`	show current nodeset
`clush> +<NODESET>`	add nodes to current nodeset
`clush> -<NODESET>`	remove nodes from current nodeset
`clush> @<NODESET>`	set current nodeset
`clush> !<COMMAND>`	execute `<COMMAND>` on the local system
`clush> =`	toggle the ouput format (gathered or standard mode)

To leave an interactive session, type `quit` or *Control-D*. As of version 1.6, it is not possible to cancel a command while staying in *clush* interactive session: for instance, *Control-C* is not supported and will abort current *clush* interactive command (see ticket #166).

Example of *clush* interactive session:

```
$ clush -w node[11-14] -b
Enter 'quit' to leave this interactive mode
Working with nodes: node[11-14]
clush> uname
---------------
node[11-14] (4)
---------------
Linux
clush> !pwd
LOCAL: /root
clush> -node[11,13]
Working with nodes: node[12,14]
clush> uname
---------------
node[12,14] (2)
```

(continues on next page)

(continued from previous page)

```
---------------
Linux
clush>
```

The interactive mode and commands described above are subject to change and improvements in future releases. Feel free to open an enhancement ticket if you use the interactive mode and have some suggestions.

5.3.6 File copying mode

When *clush* is started with the `-c` or `--copy` option, it will attempt to copy specified file and/or directory to the provided target cluster nodes. If the `--dest` option is specified, it will put the copied files or directory there.

Here are some examples of file copying with *clush*:

```
$ clush -v -w node[11-12] --copy /tmp/foo
`/tmp/foo' -> node[11-12]:`/tmp'

$ clush -v -w node[11-12] --copy /tmp/foo /tmp/bar
`/tmp/bar' -> aury[11-12]:`/tmp'
`/tmp/foo' -> aury[11-12]:`/tmp'

$ clush -v -w node[11-12] --copy /tmp/foo --dest /var/tmp/
`/tmp/foo' -> node[11-12]:`/var/tmp/'
```

5.3.7 Reverse file copying mode

When *clush* is started with the `--rcopy` option, it will attempt to retrieve specified file and/or directory from provided cluster nodes. If the `--dest` option is specified, it must be a directory path where the files will be stored with their hostname appended. If the destination path is not specified, it will take the first file or dir basename directory as the local destination, example:

```
$ clush -v -w node[11-12] --rcopy /tmp/foo
node[11-12]:`/tmp/foo' -> `/tmp'

$ ls /tmp/foo.*
/tmp/foo.node11  /tmp/foo.node12
```

5.3.8 Other options

Overriding clush.conf settings

clush default settings are found in a configuration described in *clush configuration*. To override any settings, use the `--option` command line option (or `-O` for the shorter version), and repeat as needed. Here is a simple example to disable the use colors in the output nodeset header:

```
$ clush -O color=never -w node[11-12] -b echo ok
---------------
node[11-12] (2)
---------------
ok
```

Worker selection

By default, *clush* is using the default library worker configuration when running commands or copying files. In most cases, this is *ssh* (See *Changing default worker* for default worker selection).

Worker selection can be performed at runtime thanks to `--worker` command line option (or `-R` for the shorter version in order to be compatible with *pdsh* remote command selection option):

```
$ clush -w node[11-12] --worker=rsh echo ok
node11: ok
node12: ok
```

By default, ClusterShell supports the following worker identifiers:

- **exec**: this local worker supports parallel command execution, doesn't rely on any external tool and provides command line placeholders described below:
  - `%h` and `%host` are substitued with each *target hostname*
  - `%hosts` is substitued with the full *target nodeset*
  - `%n` and `%rank` are substitued with the remote *rank* (0 to n-1)

  For example, the following would request the exec worker to locally run multiple *ipmitool* commands across the hosts foo[0-10] and automatically aggregate output results (-b):

```
$ clush -R exec -w foo[0-10] -b ipmitool -H %h-ipmi chassis power status
---------------
foo[0-10] (11)
---------------
Chassis Power is on
```

- **rsh**: remote worker based on *rsh*
- **ssh**: remote worker based on *ssh* (default)
- **pdsh**: remote worker based on *pdsh* that requires *pdsh* to be installed; doesn't provide write support (eg. you cannot `cat file | clush --worker pdsh`); it is primarily an 1-to-n worker example.

5.4 clubak

5.4.1 Overview

clubak is another utility provided with the ClusterShell library that try to gather and sort such dsh-like output:

```
node17: MD5 (cstest.py) = 62e23bcf2e11143d4875c9826ef6183f
node14: MD5 (cstest.py) = 62e23bcf2e11143d4875c9826ef6183f
node16: MD5 (cstest.py) = e88f238673933b08d2b36904e3a207df
node15: MD5 (cstest.py) = 62e23bcf2e11143d4875c9826ef6183f
```

If *file* content is made of such output, you got the following result:

```
$ clubak -b < file
---------------
node[14-15,17] (3)
---------------
 MD5 (cstest.py) = 62e23bcf2e11143d4875c9826ef6183f
```

(continues on next page)

(continued from previous page)

```
---------------
node16
---------------
 MD5 (cstest.py) = e88f238673933b08d2b36904e3a207df
```

Or with `-L` display option to disable header block:

```
$ clubak -bL < file
node[14-15,17]:  MD5 (cstest.py) = 62e23bcf2e11143d4875c9826ef6183f
node16:  MD5 (cstest.py) = e88f238673933b08d2b36904e3a207df
```

Indeed, *clubak* formats text from standard input containing lines of the form *node: output*. It is fully backward compatible with *dshbak(1)* available with *pdsh* but provides additional features. For instance, *clubak* always displays its results sorted by node/nodeset.

But you do not need to execute *clubak* when using *clush* as all output formatting features are already included in *clush* (see *clush -b / -B / -L* examples, *Non-interactive (or one-shot) mode*). There are several advantages of having *clubak* features included in *clush*: for example, it is possible, with *clush*, to still get partial results when interrupted during command execution (eg. with *Control-C*), thing not possible by just piping commands together.

Most *clubak* options are the same as *clush*. For instance, to try to resolve node groups in results, use `-r, --regroup`:

```
$ clubak -br < file
```

Like *clush*, *clubak* uses the *ClusterShell.MsgTree* module of the ClusterShell library.

5.4.2 Tree trace mode (-T)

A special option `-T, --tree`, only available with clubak, can switch on *MsgTree* trace mode (all keys/nodes are kept for each message element of the tree, thus allowing special output display). This mode has been first added to replace *padb*[1] in some cases to display a whole cluster job digested backtrace.

For example:

```
$ cat trace_test
node3: first_func()
node1: first_func()
node2: first_func()
node5: first_func()
node1: second_func()
node4: first_func()
node3: bis_second_func()
node2: second_func()
node5: second_func()
node4: bis_second_func()

$ cat trace_test | clubak -TL
node[1-5]:
 first_func()
node[1-2,5]:
   second_func()
node[3-4]:
   bis_second_func()
```

[1] *padb*, a parallel application debugger (http://padb.pittman.org.uk/)

CHAPTER 6

Programming Guide

This part provides programming information for using ClusterShell in Python applications. It is divided into two sections: node sets handling and cluster task management, in that order, because managing cluster tasks requires some knowledge of how to deal with node sets. Each section also describes the conceptual structures of ClusterShell and provides examples of how to use them.

This part is intended for intermediate and advanced programmers who are familiar with Python programming and basic concepts of high-performance computing (HPC).

6.1 Node sets handling

6.1.1 NodeSet class

NodeSet is a class to represent an ordered set of node names (optionally indexed). It's a convenient way to deal with cluster nodes and ease their administration. *NodeSet* is implemented with the help of two other ClusterShell public classes, *RangeSet* and *RangeSetND*, which implement methods to manage a set of numeric ranges in one or multiple dimensions. *NodeSet*, *RangeSet* and *RangeSetND* APIs match standard Python sets. A command-line interface (*nodeset*) which implements most of *NodeSet* features, is also available.

Other classes of the ClusterShell library makes use of the *NodeSet* class when they come to deal with distant nodes.

Using NodeSet

If you are used to Python sets, *NodeSet* interface will be easy for you to learn. The main conceptual difference is that *NodeSet* iterators always provide ordered results (and also *NodeSet.__getitem__()* by index or slice is allowed). Furthermore, *NodeSet* provides specific methods like *NodeSet.split()*, *NodeSet.contiguous()* (see below), or *NodeSet.groups()*, *NodeSet.regroup()* (these last two are related to *Node groups*). The following code snippet shows you a basic usage of the *NodeSet* class:

```
>>> from ClusterShell.NodeSet import NodeSet
>>> nodeset = NodeSet()
```

(continues on next page)

(continued from previous page)

```
>>> nodeset.add("node7")
>>> nodeset.add("node6")
>>> print nodeset
node[6-7]
```

NodeSet class provides several object constructors:

```
>>> print NodeSet("node[1-5]")
node[1-5]
>>> print NodeSet.fromlist(["node1", "node2", "node3"])
node[1-3]
>>> print NodeSet.fromlist(["node[1-5]", "node[6-10]"])
node[1-10]
>>> print NodeSet.fromlist(["clu-1-[1-4]", "clu-2-[1-4]"])
clu-[1-2]-[1-4]
```

All corresponding Python sets operations are available, for example:

```
>>> from ClusterShell.NodeSet import NodeSet
>>> ns1 = NodeSet("node[10-42]")
>>> ns2 = NodeSet("node[11-16,18-39]")
>>> print ns1.difference(ns2)
node[10,17,40-42]
>>> print ns1 - ns2
node[10,17,40-42]
>>> ns3 = NodeSet("node[1-14,40-200]")
>>> print ns3.intersection(ns1)
node[10-14,40-42]
```

Unlike Python sets, it is important to notice that *NodeSet* is somewhat not so strict about the type of element used for set operations. Thus when a string object is encountered, it is automatically converted to a NodeSet object for convenience. The following example shows an example of this (set operation is working with either a native nodeset or a string):

```
>>> nodeset = NodeSet("node[1-10]")
>>> nodeset2 = NodeSet("node7")
>>> nodeset.difference_update(nodeset2)
>>> print nodeset
node[1-6,8-10]
>>>
>>> nodeset.difference_update("node8")
>>> print nodeset
node[1-6,9-10]
```

NodeSet ordered content leads to the following being allowed:

```
>>> nodeset = NodeSet("node[10-49]")
>>> print nodeset[0]
node10
>>> print nodeset[-1]
node49
>>> print nodeset[10:]
node[20-49]
>>> print nodeset[:5]
node[10-14]
>>> print nodeset[::4]
node[10,14,18,22,26,30,34,38,42,46]
```

And it works for node names without index, for example:

```
>>> nodeset = NodeSet("lima,oscar,zulu,alpha,delta,foxtrot,tango,x-ray")
>>> print nodeset
alpha,delta,foxtrot,lima,oscar,tango,x-ray,zulu
>>> print nodeset[0]
alpha
>>> print nodeset[-2]
x-ray
```

And also for multidimensional node sets:

```
>>> nodeset = NodeSet("clu1-[1-10]-ib[0-1],clu2-[1-10]-ib[0-1]")
>>> print nodeset
clu[1-2]-[1-10]-ib[0-1]
>>> print nodeset[0]
clu1-1-ib0
>>> print nodeset[-1]
clu2-10-ib1
>>> print nodeset[::2]
clu[1-2]-[1-10]-ib0
```

To split a NodeSet object into *n* subsets, use the *NodeSet.split()* method, for example:

```
>>> for nodeset in NodeSet("node[10-49]").split(2):
...     print nodeset
...
node[10-29]
node[30-49]
```

To split a NodeSet object into contiguous subsets, use the *NodeSet.contiguous()* method, for example:

```
>>> for nodeset in NodeSet("node[10-49,51-53,60-64]").contiguous():
...     print nodeset
...
node[10-49]
node[51-53]
node[60-64]
```

For further details, please use the following command to see full *NodeSet* API documentation.

Multidimensional considerations

Version 1.7 introduces full support of multidimensional NodeSet (eg. *da[2-5]c[1-2]p[0-1]*). The *NodeSet* interface is the same, multidimensional patterns are automatically detected by the parser and processed internally. While expanding a multidimensional NodeSet is easily solved by performing a cartesian product of all dimensions, folding nodes is much more complex and time consuming. To reduce the performance impact of such feature, the *NodeSet* class still relies on *RangeSet* when only one dimension is varying (see *RangeSet class*). Otherwise, it uses a new class named *RangeSetND* for full multidimensional support (see *RangeSetND class*).

Extended String Pattern

NodeSet class parsing engine recognizes an *extended string pattern*, adding support for union (with special character *","*), difference (with special character *"!"*), intersection (with special character *"&"*) and symmetric difference (with special character *"^"*) operations. String patterns are read from left to right, by proceeding any character operators accordingly. The following example shows how you can use this feature:

```
>>> print NodeSet("node[10-42],node46!node10")
node[11-42,46]
```

6.1.2 Node groups

Node groups are very useful and are needed to group similar cluster nodes in terms of configuration, installed software, available resources, etc. A node can be a member of more than one node group.

Using node groups

Node groups are prefixed with @ character. Please see *Node group expression rules* for more details about node group expression/syntax rules.

Please also have a look at *Node groups configuration* to learn how to configure external node group bingings (sources). Once setup (please use the *nodeset* command to check your configuration), the NodeSet parsing engine automatically resolves node groups. For example:

```
>>> print NodeSet("@oss")
example[4-5]
>>> print NodeSet("@compute")
example[32-159]
>>> print NodeSet("@compute,@oss")
example[4-5,32-159]
```

That is, all NodeSet-based applications share the same system-wide node group configuration (unless explicitly disabled — see *Disabling node group resolution*).

When the **all** group upcall is configured (*node groups configuration*), you can also use the following `NodeSet` constructor:

```
>>> print NodeSet.fromall()
example[4-6,32-159]
```

When group upcalls are not properly configured, this constructor will raise a *NodeSetExternalError* exception.

Finding node groups

In order to find node groups a specified node set belongs to, you can use the `NodeSet.groups()` method. This method is used by `nodeset -l <nodeset>` command (see *Finding node groups*). It returns a Python dictionary where keys are groups found and values, provided for convenience, are tuples of the form *(group_nodeset, contained_nodeset)*. For example:

```
>>> for group, (group_nodes, contained_nodes) in NodeSet("@oss").groups().iteritems():
...     print group, group_nodes, contained_nodes
...
@all example[4-6,32-159] example[4-5]
@oss example[4-5] example[4-5]
```

More usage examples follow:

```
>>> print NodeSet("example4").groups().keys()
['@all', '@oss']
>>> print NodeSet("@mds").groups().keys()
```

(continues on next page)

(continued from previous page)

```
['@all', '@mds']
>>> print NodeSet("dummy0").groups().keys()
[]
```

Regrouping node sets

If needed group configuration conditions are met (cf. *node groups configuration*), you can use the *NodeSet.regroup()* method to reduce node sets using matching groups, whenever possible:

```
>>> print NodeSet("example[4-6]").regroup()
@mds,@oss
```

The nodeset command makes use of the *NodeSet.regroup()* method when using the *-r* switch (see *Resolving node groups*).

Overriding default groups configuration

It is possible to override the libary default groups configuration by changing the default *NodeSet resolver* object. Usually, this is done for testing or special purposes. Here is an example of how to override the *resolver* object using *NodeSet.set_std_group_resolver()* in order to use another configuration file:

```
>>> from ClusterShell.NodeSet import NodeSet, set_std_group_resolver
>>> from ClusterShell.NodeUtils import GroupResolverConfig
>>> set_std_group_resolver(GroupResolverConfig("/other/groups.conf"))
>>> print NodeSet("@oss")
other[10-20]
```

It is possible to restore *NodeSet default group resolver* by passing None to the *NodeSet.set_std_group_resolver()* module function, for example:

```
>>> from ClusterShell.NodeSet import set_std_group_resolver
>>> set_std_group_resolver(None)
```

Disabling node group resolution

If for any reason, you want to disable host groups resolution, you can use the special resolver value *RESOLVER_NOGROUP*. In that case, *NodeSet* parsing engine will not recognize @ group characters anymore, for instance:

```
>>> from ClusterShell.NodeSet import NodeSet, RESOLVER_NOGROUP
>>> print NodeSet("@oss")
example[4-5]
>>> print NodeSet("@oss", resolver=RESOLVER_NOGROUP)
@oss
```

Any attempts to use a group-based method (like *NodeSet.groups()* or `NodeSet.regroups()`) on such "no group" NodeSet will raise a *NodeSetExternalError* exception.

6.1.3 NodeSet object serialization

The *NodeSet* class supports object serialization through the standard *pickling*. Group resolution is done before *pickling*.

6.2 Range sets

Cluster node names being typically indexed, common node sets rely heavily on numerical range sets. The *RangeSet* module provides two public classes to deal directly with such range sets, *RangeSet* and *RangeSetND*, presented in the following sections.

6.2.1 RangeSet class

The *RangeSet* class implements a mutable, ordered set of cluster node indexes (one dimension) featuring a fast range-based API. This class is used by the *NodeSet* class (see *NodeSet class*). Since version 1.6, *RangeSet* really derives from standard Python set class (Python sets), and thus provides methods like *RangeSet.union()*, *RangeSet.intersection()*, *RangeSet.difference()*, *RangeSet.symmetric_difference()* and their in-place versions *RangeSet.update()*, *RangeSet.intersection_update()*, *RangeSet.difference_update()* and *RangeSet.symmetric_difference_update()*.

Since v1.6, padding of ranges (eg. *003-009*) can be managed through a public *RangeSet* instance variable named padding. It may be changed at any time. Padding is a simple display feature per RangeSet object, thus current padding value is not taken into account when computing set operations. Also since v1.6, *RangeSet* is itself an iterator over its items as integers (instead of strings). To iterate over string items as before (with optional padding), you can now use the *RangeSet.striter()* method.

6.2.2 RangeSetND class

The *RangeSetND* class builds a N-dimensional RangeSet mutable object and provides the common set methods. This class is public and may be used directly, however we think it is less convenient to manipulate that *NodeSet* and does not necessarily provide the same one-dimension optimization (see *Multidimensional considerations*). Several constructors are available, using RangeSet objects, strings or individual multidimensional tuples, for instance:

```
>>> from ClusterShell.RangeSet import RangeSet, RangeSetND
>>> r1 = RangeSet("1-5/2")
>>> r2 = RangeSet("10-12")
>>> r3 = RangeSet("0-4/2")
>>> r4 = RangeSet("10-12")
>>> print r1, r2, r3, r4
1,3,5 10-12 0,2,4 10-12
>>> rnd = RangeSetND([[r1, r2], [r3, r4]])
>>> print rnd
0-5; 10-12

>>> print list(rnd)
[(0, 10), (0, 11), (0, 12), (1, 10), (1, 11), (1, 12), (2, 10), (2, 11), (2, 12), (3,␣
↪10), (3, 11), (3, 12), (4, 10), (4, 11), (4, 12), (5, 10), (5, 11), (5, 12)]
>>> r1 = RangeSetND([(0, 4), (0, 5), (1, 4), (1, 5)])
>>> len(r1)
4
>>> str(r1)
'0-1; 4-5\n'
```

(continues on next page)

(continued from previous page)

```
>>> r2 = RangeSetND([(1, 4), (1, 5), (1, 6), (2, 5)])
>>> str(r2)
'1; 4-6\n2; 5\n'
>>> r = r1 & r2
>>> str(r)
'1; 4-5\n'
>>> list(r)
[(1, 4), (1, 5)]
```

6.3 Task management

6.3.1 Structure of Task

A ClusterShell *Task* and its underlying *Engine* class are the fundamental infrastructure associated with a thread. An *Engine* implements an event processing loop that you use to schedule work and coordinate the receipt of incoming events. The purpose of this run loop is to keep your thread busy when there is work to do and put your thread to sleep when there is none. When calling the *Task.resume()* or *Task.run()* methods, your thread enters the Task Engine run loop and calls installed event handlers in response to incoming events.

6.3.2 Using Task objects

A *Task* object provides the main interface for adding shell commands, files to copy or timer and then running it. Every thread has a single *Task* object (and underlying *Engine* object) associated with it. The *Task* object is an instance of the *Task* class.

Getting a Task object

To get the *Task* object bound to the **current thread**, you use one of the following:

- Use the *Task.task_self()* function available at the root of the Task module
- or use `task = Task()`; Task objects are only instantiated when needed.

Example of getting the current task object:

```
>>> from ClusterShell.Task import task_self
>>> task = task_self()
```

So for a single-threaded application, a Task is a simple singleton (which instance is also available through *Task.task_self()*).

To get the *Task* object associated to a specific thread identified by the identifier *tid*, you use the following:

```
>>> from ClusterShell.Task import Task
>>> task = Task(thread_id=tid)
```

Configuring the Task object

Each *Task* provides an info dictionary that shares both internal *Task*-specific parameters and user-defined (key, value) parameters. Use the following *Task* class methods to get or set parameters:

- *Task.info()*
- *Task.set_info()*

For example, to configure the task debugging behavior:

```
>>> task.set_info('debug', True)
>>> task.info('debug')
True
```

You can also use the *Task* info dictionary to set your own *Task*-specific key, value pairs. You may use any free keys but only keys starting with *USER_* are guaranteed not to be used by ClusterShell in the future.

Task info keys and their default values:

Info key string	Default value	Comment
debug	False	Enable debugging support (boolean)
print_debug	internal using *print*	Default is to print debug lines to stdout using *print*. To override this behavior, set a function that takes two arguments (the task object and a string) as the value.
fanout	64	Ssh *fanout* window (integer)
connect_timeout	10	Value passed to ssh or pdsh (integer)
command_timeout	0 (no timeout)	Value passed to ssh or pdsh (integer)

Below is an example of *print_debug* override. As you can see, we set the function *print_csdebug(task, s)* as the value. When debugging is enabled, this function will be called for any debug text line. For example, this function searchs for any known patterns and print a modified debug line to stdout when found:

```
def print_csdebug(task, s):
    m = re.search("(\w+): SHINE:\d:(\w+):", s)
    if m:
        print "%s<pickle>" % m.group(0)
    else:
        print s

# Install the new debug printing function
task_self().set_info("print_debug", print_csdebug)
```

Submitting a shell command

You can submit a set of commands for local or distant execution in parallel with *Task.shell()*.

Local usage:

```
task.shell(command [, key=key] [, handler=handler] [, timeout=secs])
```

Distant usage:

```
task.shell(command, nodes=nodeset [, handler=handler] [, timeout=secs])
```

This method makes use of the default local or distant worker. ClusterShell uses a default Worker based on the Python Popen2 standard module to execute local commands, and a Worker based on *ssh* (Secure SHell) for distant commands.

If the Task is not running, the command is scheduled for later execution. If the Task is currently running, the command is executed as soon as possible (depending on the current *fanout*).

To set a per-worker (eg. per-command) timeout value, just use the timeout parameter (in seconds), for example:

```
task.shell("uname -r", nodes=remote_nodes, handler=ehandler, timeout=5)
```

This is the prefered way to specify a command timeout. *EventHandler.ev_timeout()* event is generated before the worker has finished to indicate that some nodes have timed out. You may then retrieve the nodes with *DistantWorker.iter_keys_timeout()*.

Submitting a file copy action

Local file copy to distant nodes is supported. You can submit a copy action with *Task.copy()*:

```
task.copy(source, dest, nodes=nodeset [, handler=handler] [, timeout=secs])
```

This method makes use of the default distant copy worker which is based on scp (Secure CoPy) which comes with OpenSSH.

If the Task is not running, the copy is scheduled for later execution. If the Task is currently running, the copy is started as soon as possible (depending on the current *fanout*).

Starting the Task

Before you run a Task, you must add at least one worker (shell command, file copy) or timer to it. If a Task does not have any worker to execute and monitor, it exits immediately when you try to run it with:

```
task.resume()
```

At this time, all previously submitted commands will start in the associated Task thread. From a library user point of view, the task thread is blocked until the end of the command executions.

Please note that the special method *Task.run()* does a *Task.shell()* and a *Task.resume()* in once.

To set a Task execution timeout, use the optional *timeout* parameter to set the timeout value in seconds. Once this time is elapsed when the Task is still running, the running Task raises `TimeoutError` exception, cleaning by the way all scheduled workers and timers. Using such a timeout ensures that the Task will not exceed a given time for all its scheduled works. You can also configure per-worker timeout that generates an event *EventHandler.ev_timeout()* but will not raise an exception, allowing the Task to continue. Indeed, using a per-worker timeout is the prefered way for most applications.

Getting Task results

After the task is finished (after *Task.resume()* or *Task.run()*) or after a worker is completed when you have previously defined an event handler (at *EventHandler.ev_close()*), you can use Task result getters:

- *Task.iter_buffers()*
- *Task.iter_errors()*
- *Task.node_buffer()*
- *Task.node_error()*
- *Task.max_retcode()*
- *Task.num_timeout()*

- *Task.iter_keys_timeout()*

Note: *buffer* refers to standard output, *error* to standard error.

Please see some examples in *Programming Examples*.

Exiting the Task

If a Task does not have anymore scheduled worker or timer (for example, if you run one shell command and then it closes), it exits automatically from *Task.resume()*. Still, except from a signal handler, you can always call the following method to abort the Task execution:

- *Task.abort()*

For example, it is safe to call this method from an event handler within the task itself. On abort, all scheduled workers (shell command, file copy) and timers are cleaned and *Task.resume()* returns, unblocking the Task thread from a library user point of view. Please note that commands being executed remotely are not necessary stopped (this is due to *ssh(1)* behavior).

Configuring a Timer

A timer is bound to a Task (and its underlying Engine) and fires at a preset time in the future. Timers can fire either only once or repeatedly at fixed time intervals. Repeating timers can also have their next firing time manually adjusted (see *Task.timer()*).

A timer is not a real-time mechanism; it fires when the Task's underlying Engine to which the timer has been added is running and able to check if the timer firing time has passed.

When a timer fires, the method *EventHandler.ev_timer()* of the associated EventHandler is called.

To configure a timer, use the following (secs in seconds with floating point precision):

```
task.timer(self, fire=secs, handler=handler [, interval=secs])
```

Changing default worker

When calling *Task.shell()* or *Task.copy()* the Task object creates a worker instance for each call. When the *nodes* argument is defined, the worker class used for these calls is based on Task default *distant_worker*. Change this value to use another worker class, by example **Rsh**:

```
from ClusterShell.Task import task_self
from ClusterShell.Worker.Rsh import WorkerRsh

task_self().set_default('distant_worker', WorkerRsh)
```

Thread safety and Task objects

ClusterShell is an event-based library and one of its advantage is to avoid the use of threads (and their safety issues), so it's mainly not thread-safe. When possible, avoid the use of threads with ClusterShell. However, it's sometimes not so easy, first because another library you want to use in some event handler is not event-based and may block the current thread (that's enough to break the deal). Also, in some cases, it could be useful for you to run several Tasks at the same time. Since version 1.1, ClusterShell provides support for launching a Task in another thread and some experimental support for multiple Tasks, but:

- you should ensure that a Task is configured and accessed from one thread at a time before it's running (there is no API lock/mutex protection),
- once the Task is running, you should modify it only from the same thread that owns that Task (for example, you cannot call *Task.abort()* from another thread).

The library provides two thread-safe methods and a function for basic Task interactions: *Task.wait()*, *Task.join()* and *Task.task_wait()* (function defined at the root of the Task module). Please refer to the API documentation.

6.3.3 Configuring explicit Shell Worker objects

We have seen in *Submitting a shell command* how to easily submit shell commands to the Task. The *Task.shell()* method returns an already scheduled Worker object. It is possible to instantiate the Worker object explicitly, for example:

```
from ClusterShell.Worker.Ssh import WorkerSsh

worker = WorkerSsh('node3', command="/bin/echo alright")
```

To be used in a Task, add the worker to it with:

```
task.schedule(worker)
```

If you have pdsh installed, you can use it by easily switching to the Pdsh worker, which should behave the same manner as the Ssh worker:

```
from ClusterShell.Worker.Pdsh import WorkerPdsh

worker = WorkerPdsh('node3', command="/bin/echo alright")
```

6.4 Programming Examples

6.4.1 Remote command example (sequential mode)

The following example shows how to send a command on some nodes, how to get a specific buffer and how to get gathered buffers:

```
from ClusterShell.Task import task_self
task = task_self()

task.run("/bin/uname -r", nodes="green[36-39,133]")

print task.node_buffer("green37")

for buf, nodes in task.iter_buffers():
        print nodes, buf

if task.max_retcode() != 0:
    print "An error occurred (max rc = %s)" % task.max_retcode()
```

Result:

```
2.6.32-431.el6.x86_64
['green37', 'green38', 'green36', 'green39'] 2.6.32-431.el6.x86_64
['green133'] 3.10.0-123.20.1.el7.x86_64
Max return code is 0
```

6.4.2 Remote command example with live output (event-based mode)

The following example shows how to use the event-based programmation model by installing an EventHandler and listening for *EventHandler.ev_read()* (we've got a line to read) and *EventHandler.ev_hup()* (one command has just completed) events. The goal here is to print standard outputs of `uname -a` commands during their execution and also to notify the user of any erroneous return codes:

```
from ClusterShell.Task import task_self
from ClusterShell.Event import EventHandler

class MyHandler(EventHandler):

    def ev_read(self, worker, node, sname, msg):
        print "%s: %s" % (node, msg)

    def ev_hup(self, worker, node, rc):
        if rc != 0:
            print "%s: returned with error code %s" % (node, rc)

task = task_self()

# Submit command, install event handler for this command and run task
task.run("/bin/uname -a", nodes="fortoy[32-159]", handler=MyHandler())
```

6.4.3 *check_nodes.py* example script

The following script is available as an example in the source repository and is usually packaged with ClusterShell:

```
#!/usr/bin/python
# check_nodes.py: ClusterShell simple example script.
#
# This script runs a simple command on remote nodes and report node
# availability (basic health check) and also min/max boot dates.
# It shows an example of use of Task, NodeSet and EventHandler objects.
# Feel free to copy and modify it to fit your needs.
#
# Usage example: ./check_nodes.py -n node[1-99]

import optparse
from datetime import date, datetime
import time

from ClusterShell.Event import EventHandler
from ClusterShell.NodeSet import NodeSet
from ClusterShell.Task import task_self


class CheckNodesResult(object):
    """Our result class"""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        """Initialize result class"""
        self.nodes_ok = NodeSet()
        self.nodes_ko = NodeSet()
        self.min_boot_date = None
        self.max_boot_date = None

    def show(self):
        """Display results"""
        if self.nodes_ok:
            print "%s: OK (boot date: min %s, max %s)" % \
                (self.nodes_ok, self.min_boot_date, self.max_boot_date)
        if self.nodes_ko:
            print "%s: FAILED" % self.nodes_ko

class CheckNodesHandler(EventHandler):
    """Our ClusterShell EventHandler"""

    def __init__(self, result):
        """Initialize our event handler with a ref to our result object."""
        EventHandler.__init__(self)
        self.result = result

    def ev_read(self, worker, node, sname, msg):
        """Read event from remote nodes"""
        # this is an example to demonstrate remote result parsing
        bootime = " ".join(msg.strip().split()[2:])
        date_boot = None
        for fmt in ("%Y-%m-%d %H:%M",): # formats with year
            try:
                # datetime.strptime() is Python2.5+, use old method instead
                date_boot = datetime(*(time.strptime(bootime, fmt)[0:6]))
            except ValueError:
                pass
        for fmt in ("%b %d %H:%M",):    # formats without year
            try:
                date_boot = datetime(date.today().year, \
                    *(time.strptime(bootime, fmt)[1:6]))
            except ValueError:
                pass
        if date_boot:
            if not self.result.min_boot_date or \
                self.result.min_boot_date > date_boot:
                self.result.min_boot_date = date_boot
            if not self.result.max_boot_date or \
                self.result.max_boot_date < date_boot:
                self.result.max_boot_date = date_boot
            self.result.nodes_ok.add(node)
        else:
            self.result.nodes_ko.add(node)

    def ev_close(self, worker, timedout):
        """Worker has finished (command done on all nodes)"""
        if timedout:
            nodeset = NodeSet.fromlist(worker.iter_keys_timeout())
            self.result.nodes_ko.add(nodeset)
        self.result.show()
```

(continues on next page)

(continued from previous page)

```
def main():
    """ Main script function """
    # Initialize option parser
    parser = optparse.OptionParser()
    parser.add_option("-d", "--debug", action="store_true", dest="debug",
                      default=False, help="Enable debug mode")
    parser.add_option("-n", "--nodes", action="store", dest="nodes",
                      default="@all", help="Target nodes (default @all group)")
    parser.add_option("-f", "--fanout", action="store", dest="fanout",
                      default="128", help="Fanout window size (default 128)",
                      type=int)
    parser.add_option("-t", "--timeout", action="store", dest="timeout",
                      default="5", help="Timeout in seconds (default 5)",
                      type=float)
    options, _ = parser.parse_args()

    # Get current task (associated to main thread)
    task = task_self()
    nodes_target = NodeSet(options.nodes)
    task.set_info("fanout", options.fanout)
    if options.debug:
        print "nodeset : %s" % nodes_target
        task.set_info("debug", True)

    # Create ClusterShell event handler
    handler = CheckNodesHandler(CheckNodesResult())

    # Schedule remote command and run task (blocking call)
    task.run("who -b", nodes=nodes_target, handler=handler, \
        timeout=options.timeout)


if __name__ == '__main__':
    main()
```

6.4.4 Using NodeSet with Parallel Python Batch script using SLURM

The following example shows how to use the NodeSet class to expand `$SLURM_NODELIST` environment variable in a Parallel Python batch script launched by SLURM. This variable may contain folded node sets. If ClusterShell is not available system-wide on your compute cluster, you need to follow *Installing ClusterShell as user using PIP* first.

Example of SLURM `pp.sbatch` to submit using `sbatch pp.sbatch`:

```
#!/bin/bash

#SBATCH -N 2
#SBATCH --ntasks-per-node 1

# run the servers
srun ~/.local/bin/ppserver.py -w $SLURM_CPUS_PER_TASK -t 300 &
sleep 10

# launch the parallel processing
python -u ./pp_jobs.py
```

Example of a `pp_jobs.py` script:

```
#!/usr/bin/env python

import os, time
import pp
from ClusterShell.NodeSet import NodeSet

# get the nodelist form Slurm
nodeset = NodeSet(os.environ['SLURM_NODELIST'])

# start the servers (ncpus=0 will make sure that none is started locally)
# casting nodelist to tuple/list will correctly expand $SLURM_NODELIST
job_server = pp.Server(ncpus=0, ppservers=tuple(nodelist))

# make sure the servers have enough time to start
time.sleep(5)

# test function to execute on the remove nodes
def test_func():
    print os.uname()

# start the jobs
job_1 = job_server.submit(test_func,(),(),("os",))
job_2 = job_server.submit(test_func,(),(),("os",))

# retrive the results
print job_1()
print job_2()

# Cleanup
job_server.print_stats()
job_server.destroy()
```

CHAPTER 7

Python API

ClusterShell public API autodoc.

7.1 NodeSet

Cluster node set module.

A module to efficiently deal with node sets and node groups. Instances of NodeSet provide similar operations than the builtin set() type, see http://www.python.org/doc/lib/set-objects.html

7.1.1 Usage example

```
>>> # Import NodeSet class
... from ClusterShell.NodeSet import NodeSet
>>>
>>> # Create a new nodeset from string
... nodeset = NodeSet("cluster[1-30]")
>>> # Add cluster32 to nodeset
... nodeset.update("cluster32")
>>> # Remove from nodeset
... nodeset.difference_update("cluster[2-5,8-31]")
>>> # Print nodeset as a pdsh-like pattern
... print nodeset
cluster[1,6-7,32]
>>> # Iterate over node names in nodeset
... for node in nodeset:
...     print node
cluster1
cluster6
cluster7
cluster32
```

class ClusterShell.NodeSet.**NodeSet**(*nodes=None*, *autostep=None*, *resolver=None*, *fold_axis=None*)

Iterable class of nodes with node ranges support.

NodeSet creation examples:

```
>>> nodeset = NodeSet()                # empty NodeSet
>>> nodeset = NodeSet("cluster3")      # contains only cluster3
>>> nodeset = NodeSet("cluster[5,10-42]")
>>> nodeset = NodeSet("cluster[0-10/2]")
>>> nodeset = NodeSet("cluster[0-10/2],othername[7-9,120-300]")
```

NodeSet provides methods like update(), intersection_update() or difference_update() methods, which conform to the Python Set API. However, unlike RangeSet or standard Set, NodeSet is somewhat not so strict for convenience, and understands NodeSet instance or NodeSet string as argument. Also, there is no strict definition of one element, for example, it IS allowed to do:

```
>>> nodeset = NodeSet("blue[1-50]")
>>> nodeset.remove("blue[36-40]")
>>> print nodeset
blue[1-35,41-50]
```

Additionally, the NodeSet class recognizes the "extended string pattern" which adds support for union (special character ","), difference ("!"), intersection ("&") and symmetric difference ("^") operations. String patterns are read from left to right, by proceeding any character operators accordingly.

Extended string pattern usage examples:

```
>>> nodeset = NodeSet("node[0-10],node[14-16]") # union
>>> nodeset = NodeSet("node[0-10]!node[8-10]")  # difference
>>> nodeset = NodeSet("node[0-10]&node[5-13]")  # intersection
>>> nodeset = NodeSet("node[0-10]^node[5-13]")  # xor
```

__and__(*other*)
Implements the & operator. So `s & t` returns a new nodeset with elements common to s and t.

__contains__(*other*)
Is node contained in NodeSet ?

__copy__()
Return a shallow copy of a NodeSet.

__delattr__
x.__delattr__('name') <==> del x.name

__eq__(*other*)
NodeSet equality comparison.

__format__()
default object formatter

__ge__(*other*)
Report whether this nodeset contains another nodeset.

__getattribute__
x.__getattribute__('name') <==> x.name

__getitem__(*index*)
Return the node at specified index or a subnodeset when a slice is specified.

__getstate__()
Called when pickling: remove references to group resolver.

__gt__(*other*)
x.__gt__(y) <==> x>y

__hash__

__iand__(*other*)
Implements the &= operator. So `s &= t` returns nodeset s keeping only elements also found in t. (Python version 2.5+ required)

__init__(*nodes=None*, *autostep=None*, *resolver=None*, *fold_axis=None*)
Initialize a NodeSet object.

The *nodes* argument may be a valid nodeset string or a NodeSet object. If no nodes are specified, an empty NodeSet is created.

The optional *autostep* argument is passed to underlying *RangeSet.RangeSet* objects and aims to enable and make use of the range/step syntax (eg. `node[1-9/2]`) when converting NodeSet to string (using folding). To enable this feature, autostep must be set there to the min number of indexes that are found at equal distance of each other inside a range before NodeSet starts to use this syntax. For example, *autostep=3* (or less) will pack `n[2,4,6]` into `n[2-6/2]`. Default autostep value is None which means "inherit whenever possible", ie. do not enable it unless set in NodeSet objects passed as *nodes* here or during arithmetic operations. You may however use the special `AUTOSTEP_DISABLED` constant to force turning off autostep feature.

The optional *resolver* argument may be used to override the group resolving behavior for this NodeSet object. It can either be set to a *NodeUtils.GroupResolver* object, to the `RESOLVER_NOGROUP` constant to disable any group resolution, or to None (default) to use standard NodeSet group resolver (see *set_std_group_resolver()* at the module level to change it if needed).

nD nodeset only: the optional *fold_axis* parameter, if specified, set the public instance member *fold_axis* to an iterable over nD 0-indexed axis integers. This parameter may be used to disengage some nD folding. That may be useful as all cluster tools don't support folded-nD nodeset syntax. Pass `[0]`, for example, to only fold along first axis (that is, to fold first dimension using `[a-b]` rangeset syntax whenever possible). Using *fold_axis* ensures that rangeset won't be folded on unspecified axis, but please note however, that using *fold_axis* may lead to suboptimal folding, this is because NodeSet algorithms are optimized for folding along all axis (default behavior).

__ior__(*other*)
Implements the |= operator. So `s |= t` returns nodeset s with elements added from t. (Python version 2.5+ required)

__isub__(*other*)
Implement the -= operator. So `s -= t` returns nodeset s after removing elements found in t. (Python version 2.5+ required)

__iter__()
Iterator on single nodes as string.

__ixor__(*other*)
Implement the ^= operator. So `s ^= t` returns nodeset s after keeping all nodes that are in exactly one of the nodesets. (Python version 2.5+ required)

__le__(*other*)
Report whether another nodeset contains this nodeset.

__len__()
Get the number of nodes in NodeSet.

__lt__(*other*)
x.__lt__(y) <==> x<y

__new__(*S*, ...) → a new object with type S, a subtype of T

__or__(*other*)
Implements the | operator. So s | t returns a new nodeset with elements from both s and t.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__setstate__(*dic*)
Called when unpickling: restore parser using non group resolver.

__sizeof__() → int
size of object in memory, in bytes

__str__()
Get ranges-based pattern of node list.

__sub__(*other*)
Implement the - operator. So `s - t` returns a new nodeset with elements in s but not in t.

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

__xor__(*other*)
Implement the ^ operator. So `s ^ t` returns a new NodeSet with nodes that are in exactly one of the nodesets.

add(*other*)
Add node to NodeSet.

autostep
Get autostep value (property)

clear()
Remove all nodes from this nodeset.

contiguous()
Object-based NodeSet iterator on contiguous node sets.

Contiguous node set contains nodes with same pattern name and a contiguous range of indexes, like foobar[1-100].

copy()
Return a shallow copy of a NodeSet.

difference(*other*)
`s.difference(t)` returns a new NodeSet with elements in s but not in t.

difference_update(*other*, *strict=False*)
s.difference_update(t) removes from s all the elements found in t. If strict is True, raise KeyError if an element in t cannot be removed from s.

classmethod fromall(*groupsource=None*, *autostep=None*, *resolver=None*)
Class method that returns a new NodeSet with all nodes from optional groupsource.

classmethod fromlist(*nodelist*, *autostep=None*, *resolver=None*)
Class method that returns a new NodeSet with nodes from provided list.

get_autostep()
Get autostep value (property)

groups(*groupsource=None*, *noprefix=False*)
Find node groups this nodeset belongs to.

Return a dictionary of the form: group_name => (group_nodeset, contained_nodeset)

Group names are always prefixed with "@". If groupsource is provided, they are prefixed with "@groupsource:", unless noprefix is True.

intersection(*other*)
s.intersection(t) returns a new set with elements common to s and t.

intersection_update(*other*)
s.intersection_update(t) returns nodeset s keeping only elements also found in t.

issubset(*other*)
Report whether another nodeset contains this nodeset.

issuperset(*other*)
Report whether this nodeset contains another nodeset.

nsiter()
Object-based NodeSet iterator on single nodes.

regroup(*groupsource=None*, *autostep=None*, *overlap=False*, *noprefix=False*)
Regroup nodeset using node groups.

Try to find fully matching node groups (within specified groupsource) and return a string that represents this node set (containing these potential node groups). When no matching node groups are found, this method returns the same result as str().

remove(*elem*)
Remove element elem from the nodeset. Raise KeyError if elem is not contained in the nodeset.

Raises **KeyError** – elem is not contained in the nodeset

set_autostep(*val*)
Set autostep value (property)

split(*nbr*)
Split the nodeset into nbr sub-nodesets (at most). Each sub-nodeset will have the same number of elements more or less 1. Current nodeset remains unmodified.

```
>>> for nodeset in NodeSet("foo[1-5]").split(3):
...     print nodeset
foo[1-2]
foo[3-4]
foo5
```

striter()
Iterator on single nodes as string.

symmetric_difference(*other*)
`s.symmetric_difference(t)` returns the symmetric difference of two nodesets as a new NodeSet.

(ie. all nodes that are in exactly one of the nodesets.)

symmetric_difference_update(*other*)
s.symmetric_difference_update(t) returns nodeset s keeping all nodes that are in exactly one of the nodesets.

union(*other*)
s.union(t) returns a new set with elements from both s and t.

update(*other*)
s.update(t) returns nodeset s with elements added from t.

updaten(*others*)
s.updaten(list) returns nodeset s with elements added from given list.

ClusterShell.NodeSet.**expand**(*pat*)
Commodity function that expands a nodeset pattern into a list of nodes.

ClusterShell.NodeSet.**fold**(*pat*)
Commodity function that clean dups and fold provided pattern with ranges and "/step" support.

ClusterShell.NodeSet.**grouplist**(*namespace=None*, *resolver=None*)
Commodity function that retrieves the list of raw groups for a specified group namespace (or use default namespace). Group names are not prefixed with "@".

ClusterShell.NodeSet.**std_group_resolver**()
Get the current resolver used for standard "@" group resolution.

ClusterShell.NodeSet.**set_std_group_resolver**(*new_resolver*)
Override the resolver used for standard "@" group resolution. The new resolver should be either an instance of NodeUtils.GroupResolver or None. In the latter case, the group resolver is restored to the default one.

7.2 NodeUtils

Cluster nodes utility module

The NodeUtils module is a ClusterShell helper module that provides supplementary services to manage nodes in a cluster. It is primarily designed to enhance the NodeSet module providing some binding support to external node groups sources in separate namespaces (example of group sources are: files, jobs scheduler, custom scripts, etc.).

class ClusterShell.NodeUtils.**GroupSource**(*name*, *groups=None*, *allgroups=None*)
ClusterShell Group Source class.

A Group Source object defines resolv_map, resolv_list, resolv_all and optional resolv_reverse methods for node group resolution. It is constituting a group resolution namespace.

__init__(*name*, *groups=None*, *allgroups=None*)
Initialize GroupSource

Parameters

- **name** – group source name
- **groups** – group to nodes dict
- **allgroups** – optional _all **groups_** result (string)

__weakref__
list of weak references to the object (if defined)

resolv_all()
Return the content of all groups as defined by this GroupSource

resolv_list()
Return a list of all group names for this group source

resolv_map(*group*)
Get nodes from group *group*

resolv_reverse(*node*)
Return the group name matching the provided node.

class ClusterShell.NodeUtils.**GroupResolver**(*default_source=None*, *illegal_chars=None*)
Base class GroupResolver that aims to provide node/group resolution from multiple GroupSources.

A GroupResolver object might be initialized with a default GroupSource object, that is later used when group resolution is requested with no source information. As of version 1.7, a set of illegal group characters may also be provided for sanity check (raising GroupResolverIllegalCharError when found).

__init__(*default_source=None*, *illegal_chars=None*)
Lazy initialization of a new GroupResolver object.

__weakref__
list of weak references to the object (if defined)

add_source(**args*)
Add a GroupSource to this resolver.

all_nodes(*namespace=None*)
Find all nodes. You may specify an optional namespace.

default_source_name
Get default source name of resolver.

group_nodes(*group*, *namespace=None*)
Find nodes for specified group name and optional namespace.

grouplist(*namespace=None*)
Get full group list. You may specify an optional namespace.

has_node_groups(*namespace=None*)
Return whether finding group list for a specified node is supported by the resolver (in optional namespace).

node_groups(*node*, *namespace=None*)
Find group list for specified node and optional namespace.

set_verbosity(**args*)
Set debugging verbosity value (DEPRECATED: use logging.DEBUG).

sources(**args*)
Get the list of all resolver source names.

class ClusterShell.NodeUtils.**GroupResolverConfig**(*filenames*, *illegal_chars=None*)
GroupResolver class that is able to automatically setup its GroupSource's from a configuration file. This is the default resolver for NodeSet.

__init__(*filenames*, *illegal_chars=None*)
Lazy init GroupResolverConfig object from filenames.

7.3 RangeSet

Cluster range set module.

Instances of RangeSet provide similar operations than the builtin set type, extended to support cluster ranges-like format and stepping support ("0-8/2").

class ClusterShell.RangeSet.**RangeSet**(*pattern=None*, *autostep=None*)

Mutable set of cluster node indexes featuring a fast range-based API.

This class aims to ease the management of potentially large cluster range sets and is used by the *NodeSet* class.

RangeSet basic constructors:

```
>>> rset = RangeSet()             # empty RangeSet
>>> rset = RangeSet("5,10-42")    # contains 5, 10 to 42
>>> rset = RangeSet("0-10/2")     # contains 0, 2, 4, 6, 8, 10
```

Also any iterable of integers can be specified as first argument:

```
>>> RangeSet([3, 6, 8, 7, 1])
1,3,6-8
>>> rset2 = RangeSet(rset)
```

Padding of ranges (eg. "003-009") can be managed through a public RangeSet instance variable named padding. It may be changed at any time. Padding is a simple display feature per RangeSet object, thus current padding value is not taken into account when computing set operations. RangeSet is itself an iterator over its items as integers (instead of strings). To iterate over string items with optional padding, you can use the *RangeSet.striter()*: method.

RangeSet provides methods like *RangeSet.union()*, *RangeSet.intersection()*, *RangeSet.difference()*, *RangeSet.symmetric_difference()* and their in-place versions *RangeSet.update()*, *RangeSet.intersection_update()*, *RangeSet.difference_update()*, *RangeSet.symmetric_difference_update()* which conform to the Python Set API.

__and__(*other*)

Return the intersection of two RangeSets as a new RangeSet.

(I.e. all elements that are in both sets.)

__contains__(*element*)

Report whether an element is a member of a RangeSet. Element can be either another RangeSet object, a string or an integer.

Called in response to the expression `element in self`.

__copy__()

Return a shallow copy of a RangeSet.

__eq__(*other*)

RangeSet equality comparison.

__ge__(*other*)

Report whether this RangeSet contains another set.

__getitem__(*index*)

Return the element at index or a subrange when a slice is specified.

__gt__(*other*)

x.__gt__(y) <==> x>y

__iand__(*other*)
Update a RangeSet with the intersection of itself and another.

__init__(*pattern=None*, *autostep=None*)
Initialize RangeSet object.

Parameters

- **pattern** – optional string pattern
- **autostep** – optional autostep threshold

__ior__(*other*)
Update a RangeSet with the union of itself and another.

__isub__(*other*)
Remove all elements of another set from this RangeSet.

__iter__()
Iterate over each element in RangeSet.

__ixor__(*other*)
Update a RangeSet with the symmetric difference of itself and another.

__le__(*other*)
Report whether another set contains this RangeSet.

__lt__(*other*)
x.__lt__(y) <==> x<y

__or__(*other*)
Return the union of two RangeSets as a new RangeSet.

(I.e. all elements that are in either set.)

__reduce__()
Return state information for pickling.

__repr__()
Get comma-separated range-based string (x-y/step format).

__setstate__(*dic*)
called upon unpickling

__str__()
Get comma-separated range-based string (x-y/step format).

__sub__(*other*)
Return the difference of two RangeSets as a new RangeSet.

(I.e. all elements that are in this set and not in the other.)

__xor__(*other*)
Return the symmetric difference of two RangeSets as a new RangeSet.

(I.e. all elements that are in exactly one of the sets.)

add(*element*, *pad=0*)
Add an element to a RangeSet. This has no effect if the element is already present.

add_range(*start*, *stop*, *step=1*, *pad=0*)
Add a range (start, stop, step and padding length) to RangeSet. Like the Python built-in function *range()*, the last element is the largest start + i * step less than stop.

autostep
autostep threshold public instance attribute

clear()
Remove all elements from this RangeSet.

contiguous()
Object-based iterator over contiguous range sets.

copy()
Return a shallow copy of a RangeSet.

difference(*other*)
Return the difference of two RangeSets as a new RangeSet.

(I.e. all elements that are in this set and not in the other.)

difference_update(*other*, *strict=False*)
Remove all elements of another set from this RangeSet.

If strict is True, raise KeyError if an element cannot be removed. (strict is a RangeSet addition)

dim()
Get the number of dimensions of this RangeSet object. Common method with RangeSetND. Here, it will always return 1 unless the object is empty, in that case it will return 0.

discard(*element*)
Remove element from the RangeSet if it is a member.

If the element is not a member, do nothing.

classmethod fromlist(*rnglist*, *autostep=None*)
Class method that returns a new RangeSet with ranges from provided list.

classmethod fromone(*index*, *pad=0*, *autostep=None*)
Class method that returns a new RangeSet of one single item or a single range (from integer or slice object).

get_autostep()
Get autostep value (property)

intersection(*other*)
Return the intersection of two RangeSets as a new RangeSet.

(I.e. all elements that are in both sets.)

intersection_update(*other*)
Update a RangeSet with the intersection of itself and another.

issubset(*other*)
Report whether another set contains this RangeSet.

issuperset(*other*)
Report whether this RangeSet contains another set.

remove(*element*)
Remove an element from a RangeSet; it must be a member.

Parameters **element** – the element to remove

Raises

- **KeyError** – element is not contained in RangeSet
- **ValueError** – element is not castable to integer

set_autostep(*val*)
Set autostep value (property)

slices()
Iterate over RangeSet ranges as Python slice objects.

split(*nbr*)
Split the rangeset into nbr sub-rangesets (at most). Each sub-rangeset will have the same number of elements more or less 1. Current rangeset remains unmodified. Returns an iterator.

```
>>> RangeSet("1-5").split(3)
RangeSet("1-2")
RangeSet("3-4")
RangeSet("foo5")
```

striter()
Iterate over each (optionally padded) string element in RangeSet.

symmetric_difference(*other*)
Return the symmetric difference of two RangeSets as a new RangeSet.

(ie. all elements that are in exactly one of the sets.)

symmetric_difference_update(*other*)
Update a RangeSet with the symmetric difference of itself and another.

union(*other*)
Return the union of two RangeSets as a new RangeSet.

(I.e. all elements that are in either set.)

union_update(*other*)
Update a RangeSet with the union of itself and another.

update(*iterable*)
Add all integers from an iterable (such as a list).

updaten(*rangesets*)
Update a rangeset with the union of itself and several others.

7.4 RangeSetND

class ClusterShell.RangeSet.**RangeSetND**(*args=None*, *pads=None*, *autostep=None*, *copy_rangeset=True*)
Build a N-dimensional RangeSet object.

> **Warning:** You don't usually need to use this class directly, use *NodeSet* instead that has ND support.

Empty constructor:

```
RangeSetND()
```

Build from a list of list of *RangeSet* objects:

```
RangeSetND([[rs1, rs2, rs3, ...], ...])
```

Strings are also supported:

```
RangeSetND([["0-3", "4-10", ...], ...])
```

Integers are also supported:

```
RangeSetND([(0, 4), (0, 5), (1, 4), (1, 5), ...]
```

__and__(*other*)
Implements the & operator. So `s & t` returns a new object with elements common to s and t.

__contains__(***kwargs*)
Report whether an element is a member of a RangeSetND. Element can be either another RangeSetND object, a string or an integer.

Called in response to the expression `element in self`.

__copy__(***kwargs*)
Return a new, mutable shallow copy of a RangeSetND.

__eq__(*other*)
RangeSetND equality comparison.

__ge__(***kwargs*)
Report whether this RangeSetND contains another RangeSetND.

__getitem__(***kwargs*)
Return the element at index or a subrange when a slice is specified.

__gt__(*other*)
x.__gt__(y) <==> x>y

__iand__(*other*)
Implements the &= operator. So `s &= t` returns object s keeping only elements also found in t (Python 2.5+ required).

__init__(*args=None*, *pads=None*, *autostep=None*, *copy_rangeset=True*)
RangeSetND initializer

All parameters are optional.

Parameters

- **args** – generic "list of list" input argument (default is None)
- **pads** – list of 0-padding length (default is to not pad any dimensions)
- **autostep** – autostep threshold (use range/step notation if more than #autostep items meet the condition) - default is off (None)
- **copy_rangeset** – (advanced) if set to False, do not copy RangeSet objects from args (transfer ownership), which is faster. In that case, you should not modify these objects afterwards (default is True).

__ior__(*other*)
Update a RangeSetND with the union of itself and another.

__isub__(*other*)
Remove all elements of another set from this RangeSetND.

__ixor__(*other*)
Implement the ^= operator. So `s ^= t` returns object s after keeping all items that are in exactly one of the RangeSetND (Python 2.5+ required).

__le__(*other*)
Report whether another set contains this RangeSetND.

__len__()
Count unique elements in N-dimensional rangeset.

__lt__(*other*)
x.__lt__(y) <==> x<y

__or__(*other*)
Return the union of two RangeSetNDs as a new RangeSetND.

(I.e. all elements that are in either set.)

__str__(***kwargs*)
String representation of N-dimensional RangeSet.

__sub__(*other*)
Return the difference of two RangeSetNDs as a new RangeSetND.

(I.e. all elements that are in this set and not in the other.)

__weakref__
list of weak references to the object (if defined)

__xor__(*other*)
Implement the ^ operator. So `s ^ t` returns a new RangeSetND with nodes that are in exactly one of the RangeSetND.

autostep
autostep threshold public instance attribute

contiguous(***kwargs*)
Object-based iterator over contiguous range sets.

copy(***kwargs*)
Return a new, mutable shallow copy of a RangeSetND.

difference(*other*)
`s.difference(t)` returns a new object with elements in s but not in t.

difference_update(*other*, *strict=False*)
Remove all elements of another set from this RangeSetND.

If strict is True, raise KeyError if an element cannot be removed (strict is a RangeSet addition)

dim()
Get the current number of dimensions of this RangeSetND object. Return 0 when object is empty.

fold(***kwargs*)
Explicit folding call. Please note that folding of RangeSetND nD vectors are automatically managed, so you should not have to call this method. It may be still useful in some extreme cases where the RangeSetND is heavily modified.

get_autostep()
Get autostep value (property)

intersection(*other*)
`s.intersection(t)` returns a new object with elements common to s and t.

intersection_update(*other*)
`s.intersection_update(t)` returns nodeset s keeping only elements also found in t.

issubset(*other*)
Report whether another set contains this RangeSetND.

issuperset(***kwargs*)
Report whether this RangeSetND contains another RangeSetND.

iter_padding(***kwargs*)
Iterate through individual items as tuples with padding info.

pads()
Get a tuple of padding length info for each dimension.

class precond_fold
Decorator to ease internal folding management

__call__(*...*) <==> *x(...)*

__weakref__
list of weak references to the object (if defined)

set_autostep(*val*)
Set autostep value (property)

symmetric_difference(*other*)
`s.symmetric_difference(t)` returns the symmetric difference of two objects as a new RangeSetND.

(ie. all items that are in exactly one of the RangeSetND.)

symmetric_difference_update(*other*)
`s.symmetric_difference_update(t)` returns RangeSetND s keeping all nodes that are in exactly one of the objects.

union(*other*)
Return the union of two RangeSetNDs as a new RangeSetND.

(I.e. all elements that are in either set.)

union_update(*other*)
Add all RangeSetND elements to this RangeSetND.

update(*other*)
Add all RangeSetND elements to this RangeSetND.

veclist
Get folded veclist

vectors()
Get underlying *RangeSet* vectors

7.5 MsgTree

MsgTree

ClusterShell message tree module. The purpose of MsgTree is to provide a shared message tree for storing message lines received from ClusterShell Workers (for example, from remote cluster commands). It should be efficient, in term of algorithm and memory consumption, especially when remote messages are the same.

class ClusterShell.MsgTree.**MsgTree**(*mode=0*)
MsgTree maps key objects to multi-lines messages.

MsgTree is a mutable object. Keys are almost arbitrary values (must be hashable). Message lines are organized as a tree internally. MsgTree provides low memory consumption especially on a cluster when all nodes return similar messages. Also, the gathering of messages is done automatically.

__getitem__(*key*)
Return the message of MsgTree with specified key. Raises a KeyError if key is not in the MsgTree.

__init__(*mode=0*)
MsgTree initializer

The 'mode' parameter should be set to one of the following constant:

MODE_DEFER: all messages are processed immediately, saving memory from duplicate message lines, but keys are associated to tree elements usually later when tree is first "walked", saving useless state updates and CPU time. Once the tree is "walked" for the first time, its mode changes to MODE_SHIFT to keep track of further tree updates. This is the default mode.

MODE_SHIFT: all keys and messages are processed immediately, it is more CPU time consuming as MsgTree full state is updated at each add() call.

MODE_TRACE: all keys and messages and processed immediately, and keys are kept for each message element of the tree. The special method walk_trace() is then available to walk all elements of the tree.

__iter__()
Return an iterator over MsgTree's keys.

__len__()
Return the number of keys contained in the MsgTree.

__weakref__
list of weak references to the object (if defined)

add(*key*, *msgline*)
Add a message line (in bytes) associated with the given key to the MsgTree.

clear()
Remove all items from the MsgTree.

get(*key*, *default=None*)
Return the message for key if key is in the MsgTree, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

items(*match=None*, *mapper=None*)
Return (key, message) for each key of the MsgTree.

keys()
Return an iterator over MsgTree's keys.

messages(*match=None*)
Return an iterator over MsgTree's messages.

remove(*match=None*)
Modify the tree by removing any matching key references from the messages tree.

Example of use:

```
>>> msgtree.remove(lambda k: k > 3)
```

walk(*match=None*, *mapper=None*)
Walk the tree. Optionally filter keys on match parameter, and optionally map resulting keys with mapper function. Return an iterator over (message, keys) tuples for each different message in the tree.

walk_trace(*match=None*, *mapper=None*)
Walk the tree in trace mode. Optionally filter keys on match parameter, and optionally map resulting keys with mapper function. Return an iterator over 4-length tuples (msgline, keys, depth, num_children).

7.6 Task

ClusterShell Task module.

Simple example of use:

```
>>> from ClusterShell.Task import task_self, NodeSet
>>>
>>> # get task associated with calling thread
... task = task_self()
>>>
>>> # add a command to execute on distant nodes
... task.shell("/bin/uname -r", nodes="tiger[1-30,35]")
<ClusterShell.Worker.Ssh.WorkerSsh object at 0x7f41da71b890>
>>>
>>> # run task in calling thread
... task.run()
>>>
>>> # get results
... for output, nodelist in task.iter_buffers():
...     print '%s: %s' % (NodeSet.fromlist(nodelist), output)
...
```

class ClusterShell.Task.**Task**(*thread=None*, *defaults=None*)
The Task class defines an essential ClusterShell object which aims to execute commands in parallel and easily get their results.

More precisely, a Task object manages a coordinated (ie. with respect of its current parameters) collection of independent parallel Worker objects. See ClusterShell.Worker.Worker for further details on ClusterShell Workers.

Always bound to a specific thread, a Task object acts like a "thread singleton". So most of the time, and even more for single-threaded applications, you can get the current task object with the following top-level Task module function:

```
>>> task = task_self()
```

However, if you want to create a task in a new thread, use:

```
>>> task = Task()
```

To create or get the instance of the task associated with the thread object thr (threading.Thread):

```
>>> task = Task(thread=thr)
```

To submit a command to execute locally within task, use:

```
>>> task.shell("/bin/hostname")
```

To submit a command to execute to some distant nodes in parallel, use:

```
>>> task.shell("/bin/hostname", nodes="tiger[1-20]")
```

The previous examples submit commands to execute but do not allow result interaction during their execution. For your program to interact during command execution, it has to define event handlers that will listen for local or remote events. These handlers are based on the EventHandler class, defined in ClusterShell.Event. The following example shows how to submit a command on a cluster with a registered event handler:

```
>>> task.shell("uname -r", nodes="node[1-9]", handler=MyEventHandler())
```

Run task in its associated thread (will block only if the calling thread is the task associated thread):

```
>>> task.resume()
```

or:

```
>>> task.run()
```

You can also pass arguments to task.run() to schedule a command exactly like in task.shell(), and run it:

```
>>> task.run("hostname", nodes="tiger[1-20]", handler=MyEventHandler())
```

A common need is to set a maximum delay for command execution, especially when the command time is not known. Doing this with ClusterShell Task is very straighforward. To limit the execution time on each node, use the timeout parameter of shell() or run() methods to set a delay in seconds, like:

```
>>> task.run("check_network.sh", nodes="tiger[1-20]", timeout=30)
```

You can then either use Task's iter_keys_timeout() method after execution to see on what nodes the command has timed out, or listen for ev_close() events in your event handler and check the timedout boolean.

To get command result, you can either use Task's iter_buffers() method for standard output, iter_errors() for standard error after command execution (common output contents are automatically gathered), or you can listen for ev_read() events in your event handler and get live command output.

To get command return codes, you can either use Task's iter_retcodes(), node_retcode() and max_retcode() methods after command execution, or listen for ev_hup() events in your event handler.

__init__(*thread=None*, *defaults=None*)
Initialize a Task, creating a new non-daemonic thread if needed.

static __new__(*cls*, *thread=None*, *defaults=None*)
For task bound to a specific thread, this class acts like a "thread singleton", so new style class is used and new object are only instantiated if needed.

__weakref__
list of weak references to the object (if defined)

abort(*kill=False*)
Abort a task. Aborting a task removes (and stops when needed) all workers. If optional parameter kill is True, the task object is unbound from the current thread, so calling task_self() creates a new Task object.

copy(*source*, *dest*, *nodes*, ***kwargs*)
Copy local file to distant nodes.

default(*default_key*, *def_val=None*)
Return per-task value for key from the "default" dictionary. See set_default() for a list of reserved task default_keys.

default_excepthook(*exc_type*, *exc_value*, *tb*)
Default excepthook for a newly Task. When an exception is raised and uncaught on Task thread, excepthook is called, which is default_excepthook by default. Once excepthook overridden, you can still call default_excepthook if needed.

flush_buffers()
Flush all task messages (from all task workers).

flush_errors()
Flush all task error messages (from all task workers).

info(*info_key*, *def_val=None*)
Return per-task information. See set_info() for a list of reserved task info_keys.

iter_buffers(*match_keys=None*)
Iterate over buffers, returns a tuple (buffer, keys). For remote workers (Ssh), keys are list of nodes. In that case, you should use NodeSet.fromlist(keys) to get a NodeSet instance (which is more convenient and efficient):

Optional parameter match_keys add filtering on these keys.

Usage example:

```
>>> for buffer, nodelist in task.iter_buffers():
...     print NodeSet.fromlist(nodelist)
...     print buffer
```

iter_errors(*match_keys=None*)
Iterate over error buffers, returns a tuple (buffer, keys).

See iter_buffers().

iter_keys_timeout()
Iterate over timed out keys (ie. nodes).

iter_retcodes(*match_keys=None*)
Iterate over return codes of command-based workers, returns a tuple (rc, keys).

Optional parameter match_keys add filtering on these keys.

> If the process exits normally, the return code is its exit status. If the process is terminated by a signal, the return code is 128 + signal number.

join()
Suspend execution of the calling thread until the target task terminates, unless the target task has already terminated.

key_buffer(*key*)
Get buffer for a specific key. When the key is associated to multiple workers, the resulting buffer will contain all workers content that may overlap. This method returns an empty buffer if key is not found in any workers.

key_error(*key*)
Get error buffer for a specific key. When the key is associated to multiple workers, the resulting buffer will contain all workers content that may overlap. This method returns an empty error buffer if key is not found in any workers.

key_retcode(*key*)
Return return code for a specific key. When the key is associated to multiple workers, return the max return code from these workers. Raises a KeyError if key is not found in any finished workers.

load_topology(*topology_file*)
Load propagation topology from provided file.

On success, task.topology is set to a corresponding TopologyTree instance.

On failure, task.topology is left untouched and a TopologyError exception is raised.

max_retcode()

Get max return code encountered during last run

or None in the following cases:

- all commands timed out,
- no command-based worker was executed.

If the process exits normally, the return code is its exit status. If the process is terminated by a signal, the return code is 128 + signal number.

node_buffer(*key*)
Get buffer for a specific key. When the key is associated to multiple workers, the resulting buffer will contain all workers content that may overlap. This method returns an empty buffer if key is not found in any workers.

node_error(*key*)
Get error buffer for a specific key. When the key is associated to multiple workers, the resulting buffer will contain all workers content that may overlap. This method returns an empty error buffer if key is not found in any workers.

node_retcode(*key*)
Return return code for a specific key. When the key is associated to multiple workers, return the max return code from these workers. Raises a KeyError if key is not found in any finished workers.

num_timeout()
Return the number of timed out "keys" (ie. nodes).

port(*handler=None*, *autoclose=False*)
Create a new task port. A task port is an abstraction object to deliver messages reliably between tasks.

Basic rules:

- A task can send messages to another task port (thread safe).
- A task can receive messages from an acquired port either by setting up a notification mechanism or using a polling mechanism that may block the task waiting for a message sent on the port.
- A port can be acquired by one task only.

If handler is set to a valid EventHandler object, the port is a send-once port, ie. a message sent to this port generates an ev_msg event notification issued the port's task. If handler is not set, the task can only receive messages on the port by calling port.msg_recv().

rcopy(*source*, *dest*, *nodes*, ***kwargs*)
Copy distant file or directory to local node.

remove_port(***kwargs*)
Close and remove a port from task previously created with port().

resume(*timeout=None*)
Resume task. If task is task_self(), workers are executed in the calling thread so this method will block until all (non-autoclosing) workers have finished. This is always the case for a single-threaded application (eg. which doesn't create other Task() instance than task_self()). Otherwise, the current thread doesn't block. In that case, you may then want to call task_wait() to wait for completion.

Warning: the timeout parameter can be used to set an hard limit of task execution time (in seconds). In that case, a TimeoutError exception is raised if this delay is reached. Its value is 0 by default, which means no task time limit (TimeoutError is never raised). In order to set a maximum delay for individual command execution, you should use Task.shell()'s timeout parameter instead.

run(*command=None*, ***kwargs*)
With arguments, it will schedule a command exactly like a Task.shell() would have done it and run it. This is the easiest way to simply run a command.

```
>>> task.run("hostname", nodes="foo")
```

Without argument, it starts all outstanding actions. It behaves like Task.resume().

```
>>> task.shell("hostname", nodes="foo")
>>> task.shell("hostname", nodes="bar")
>>> task.run()
```

When used with a command, you can set a maximum delay of individual command execution with the help of the timeout parameter (see Task.shell's parameters). You can then listen for ev_close() events and check the timedout boolean in your Worker event handlers, or use num_timeout() or iter_keys_timeout() afterwards. But, when used as an alias to Task.resume(), the timeout parameter sets an hard limit of task execution time. In that case, a TimeoutError exception is raised if this delay is reached.

running()
Return True if the task is running.

schedule(***kwargs*)
Schedule a worker for execution, ie. add worker in task running loop. Worker will start processing immediately if the task is running (eg. called from an event handler) or as soon as the task is started otherwise. Only useful for manually instantiated workers, for example:

```
>>> task = task_self()
>>> worker = WorkerSsh("node[2-3]", None, 10, command="/bin/ls")
>>> task.schedule(worker)
>>> task.resume()
```

set_default(*default_key*, *value*)
Set task value for specified key in the dictionary "default". Users may store their own task-specific key, value pairs using this method and retrieve them with default().

Task default_keys are:

- "stderr": Boolean value indicating whether to enable stdout/stderr separation when using task.shell(), if not specified explicitly (default: False).
- "stdin": Boolean value indicating whether to enable stdin when using task.shell(), if not explicitly specified (default: True)
- "stdout_msgtree": Whether to instantiate standard output MsgTree for automatic internal gathering of result messages coming from Workers (default: True).
- "stderr_msgtree": Same for stderr (default: True).
- "engine": Used to specify an underlying Engine explicitly (default: "auto").
- "port_qlimit": Size of port messages queue (default: 32).
- "worker": Worker-based class used when spawning workers through shell()/run().

Unlike set_info(), when called from the task's thread or not, set_default() immediately updates the underlying dictionary in a thread-safe manner. This method doesn't wake up the engine when called.

set_info(***kwargs*)
Set task value for a specific key information. Key, value pairs can be passed to the engine and/or workers.

Users may store their own task-specific info key, value pairs using this method and retrieve them with info().

The following example changes the fanout value to 128:

```
>>> task.set_info('fanout', 128)
```

The following example enables debug messages:

```
>>> task.set_info('debug', True)
```

Task info_keys are:

- "debug": Boolean value indicating whether to enable library debugging messages (default: False).
- "print_debug": Debug messages processing function. This function takes 2 arguments: the task instance and the message string (default: an internal function doing standard print).
- "fanout": Max number of registered clients in Engine at a time (default: 64).
- "grooming_delay": Message maximum end-to-end delay requirement used for traffic grooming, in seconds as float (default: 0.5).
- "connect_timeout": Time in seconds to wait for connecting to remote host before aborting (default: 10).
- "command_timeout": Time in seconds to wait for a command to complete before aborting (default: 0, which means unlimited).

Unlike set_default(), the underlying info dictionary is only modified from the task's thread. So calling set_info() from another thread leads to queueing the request for late apply (at run time) using the task dispatch port. When received, the request wakes up the engine when the task is running and the info dictionary is then updated.

shell(*command*, ***kwargs*)

Schedule a shell command for local or distant parallel execution. This essential method creates a local or remote Worker (depending on the presence of the nodes parameter) and immediately schedules it for execution in task's runloop. So, if the task is already running (ie. called from an event handler), the command is started immediately, assuming current execution constraints are met (eg. fanout value). If the task is not running, the command is not started but scheduled for late execution. See resume() to start task runloop.

The following optional parameters are passed to the underlying local or remote Worker constructor:

- handler: EventHandler instance to notify (on event) – default is no handler (None)
- timeout: command timeout delay expressed in second using a floating point value – default is unlimited (None)
- autoclose: if set to True, the underlying Worker is automatically aborted as soon as all other non-autoclosing task objects (workers, ports, timers) have finished – default is False
- stderr: separate stdout/stderr if set to True – default is False.
- stdin: enable stdin if set to True or prevent its use otherwise – default is True.

Local usage::

task.shell(command [, key=key] [, handler=handler] [, timeout=secs] [, autoclose=enable_autoclose] [, stderr=enable_stderr][, stdin=enable_stdin]))

Distant usage::

task.shell(command, nodes=nodeset [, handler=handler] [, timeout=secs], [, autoclose=enable_autoclose] [, tree=None|False|True] [, remote=False|True] [, stderr=enable_stderr][, stdin=enable_stdin]))

Example:

```
>>> task = task_self()
>>> task.shell("/bin/date", nodes="node[1-2345]")
>>> task.resume()
```

suspend()
Suspend task execution. This method may be called from another task (thread-safe). The function returns False if the task cannot be suspended (eg. it's not running), or returns True if the task has been successfully suspended. To resume a suspended task, use task.resume().

class tasksyncmethod
Class encapsulating a function that checks if the calling task is running or is the current task, and allowing it to be used as a decorator making the wrapped task method thread-safe.

__call__(*...*) <==> *x*(*...*)

__weakref__
list of weak references to the object (if defined)

timer(*fire*, *handler*, *interval=-1.0*, *autoclose=False*)
Create a timer bound to this task that fires at a preset time in the future by invoking the ev_timer() method of 'handler' (provided EventHandler object). Timers can fire either only once or repeatedly at fixed time intervals. Repeating timers can also have their next firing time manually adjusted.

The mandatory parameter 'fire' sets the firing delay in seconds.

The optional parameter 'interval' sets the firing interval of the timer. If not specified, the timer fires once and then is automatically invalidated.

Time values are expressed in second using floating point values. Precision is implementation (and system) dependent.

The optional parameter 'autoclose', if set to True, creates an "autoclosing" timer: it will be automatically invalidated as soon as all other non-autoclosing task's objects (workers, ports, timers) have finished. Default value is False, which means the timer will retain task's runloop until it is invalidated.

Return a new EngineTimer instance.

See ClusterShell.Engine.Engine.EngineTimer for more details.

classmethod wait(*from_thread*)
Class method that blocks calling thread until all tasks have finished (from a ClusterShell point of view, for instance, their task.resume() return). It doesn't necessarily mean that associated threads have finished.

ClusterShell.Task.**task_self**(*defaults=None*)
Return the current Task object, corresponding to the caller's thread of control (a Task object is always bound to a specific thread). This function provided as a convenience is available in the top-level ClusterShell.Task package namespace.

ClusterShell.Task.**task_wait**()
Suspend execution of the calling thread until all tasks terminate, unless all tasks have already terminated. This function is provided as a convenience and is available in the top-level ClusterShell.Task package namespace.

ClusterShell.Task.**task_terminate**()
Destroy the Task instance bound to the current thread. A next call to task_self() will create a new Task object.

Not to be called from a signal handler. This function provided as a convenience is available in the top-level ClusterShell.Task package namespace.

ClusterShell.Task.**task_cleanup**()
Cleanup routine to destroy all created tasks. This function provided as a convenience is available in the top-level ClusterShell.Task package namespace. This is mainly used for testing purposes and should be avoided otherwise. task_cleanup() may be called from any threads but not from a signal handler.

7.7 Defaults

ClusterShell Defaults module.

Manage library defaults.

class ClusterShell.Defaults.**Defaults**(*filenames*)
Class used to manipulate ClusterShell defaults.

The following attributes may be read at any time and also changed programmatically, for most of them **before** ClusterShell objects (Task or NodeSet) are initialized.

NodeSet defaults:

- fold_axis (tuple of axis integers; default is empty tuple `()`)

Task defaults:

- stderr (boolean; default is `False`)
- stdin (boolean; default is `True`)
- stdout_msgtree (boolean; default is `True`)
- stderr_msgtree (boolean; default is `True`)
- engine (string; default is `'auto'`)
- port_qlimit (integer; default is `100`)
- local_workername (string; default is `'exec'`)
- distant_workername (string; default is `'ssh'`)
- debug (boolean; default is `False`)
- print_debug (function; default is internal)
- fanout (integer; default is `64`)
- grooming_delay (float; default is `0.25`)
- connect_timeout (float; default is `10`)
- command_timeout (float; default is `0`)

Example of use:

```
>>> from ClusterShell.Defaults import DEFAULTS
>>> from ClusterShell.Task import task_self
>>> # Change default distant worker to rsh (WorkerRsh)
... DEFAULTS.distant_workername = 'rsh'
>>> task = task_self()
>>> task.run("uname -r", nodes="cs[01-03]")
<ClusterShell.Worker.Rsh.WorkerRsh object at 0x1f4a410>
```

(continues on next page)

(continued from previous page)

```
>>> list((str(msg), nodes) for msg, nodes in task.iter_buffers())
[('3.10.0-229.7.2.el7.x86_64', ['cs02', 'cs01', 'cs03'])]
```

The library default values of all of the above attributes may be changed using the defaults.conf configuration file, except for *print_debug* (cf. *Library Defaults*). An example defaults.conf file should be included with ClusterShell. Remember that this could affect all applications using ClusterShell.

`ClusterShell.Defaults.`**`DEFAULTS`**

Globally accessible *Defaults* object.

7.8 Event

ClusterShell Event handling.

This module contains the base class *EventHandler* which defines a simple interface to handle events generated by *Worker*, `EventTimer` and `EventPort` objects.

class `ClusterShell.Event.`**`EventHandler`**

ClusterShell EventHandler interface.

Derived class should implement any of the following methods to listen for *Worker*, `EventTimer` or `EventPort` selected events. If not implemented, the default behavior is to do nothing.

ev_start(*worker*)

Called to indicate that a worker has just started.

Parameters **worker** – *Worker* derived object

ev_pickup(*worker*, *node*)

Called for each node to indicate that a worker command for a specific node (or key) has just started.

> **Warning:** The signature of *EventHandler.ev_pickup()* changed in ClusterShell 1.8, please update your *EventHandler* derived classes and add the node argument.

New in version 1.7.

Parameters

- **worker** – *Worker* derived object
- **node** – node (or key)

ev_read(*worker*, *node*, *sname*, *msg*)

Called to indicate that a worker has data to read from a specific node (or key).

> **Warning:** The signature of *EventHandler.ev_read()* changed in ClusterShell 1.8, please update your *EventHandler* derived classes and add the node, sname and msg arguments.

Parameters

- **worker** – *Worker* derived object
- **node** – node (or key)
- **sname** – stream name

- **msg** – message

ev_error(*worker*)
Called to indicate that a worker has error to read on stderr from a specific node (or key).

[DEPRECATED] use ev_read instead and test if sname is 'stderr'

Parameters **worker** – *Worker* object

Available worker attributes: * *Worker.current_node* - node (or key) * *Worker.current_errmsg* - read error message

ev_written(*worker*, *node*, *sname*, *size*)
Called to indicate that some writing has been done by the worker to a node on a given stream. This event is only generated when `write()` is previously called on the worker.

This handler may be called very often depending on the number of target nodes, the amount of data to write and the block size used by the worker.

New in version 1.7.

Parameters

- **worker** – *Worker* derived object
- **node** – node (or) key
- **sname** – stream name
- **size** – amount of bytes that has just been written to node/stream associated with this event

ev_hup(*worker*, *node*, *rc*)
Called for each node to indicate that a worker command for a specific node has just finished.

> **Warning:** The signature of *EventHandler.ev_hup()* changed in ClusterShell 1.8, please update your *EventHandler* derived classes to add the node and rc arguments.

Parameters

- **worker** – *Worker* derived object
- **node** – node (or key)
- **rc** – command return code (or None if the worker doesn't support command return codes)

ev_timeout(*worker*)
Called to indicate that a worker has timed out (worker timeout only).

[DEPRECATED] use ev_close instead and check if timedout is True

Parameters **worker** – *Worker* object

ev_close(*worker*, *timedout*)
Called to indicate that a worker has just finished.

> **Warning:** The signature of *EventHandler.ev_close()* changed in ClusterShell 1.8, please update your *EventHandler* derived classes to add the timedout argument. Please use this argument instead of the method `ev_timeout`.

Parameters

- **worker** – *Worker* derived object
- **timedout** – boolean set to True if the worker has timed out

ev_msg(*port*, *msg*)
Called to indicate that a message has been received on an EnginePort.

Used to deliver messages reliably between tasks.

Parameters

- **port** – EnginePort object on which a message has been received
- **msg** – the message object received

ev_timer(*timer*)
Called to indicate that a timer is firing.

Parameters **timer** – *EngineTimer* object that is firing

7.9 EngineTimer

class ClusterShell.Engine.Engine.**EngineTimer**(*fire_delay*, *interval*, *autoclose*, *handler*)
Concrete class EngineTimer

An EngineTimer object represents a timer bound to an engine that fires at a preset time in the future. Timers can fire either only once or repeatedly at fixed time intervals. Repeating timers can also have their next firing time manually adjusted.

A timer is not a real-time mechanism; it fires when the task's underlying engine to which the timer has been added is running and able to check if the timer's firing time has passed.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__init__(*fire_delay*, *interval*, *autoclose*, *handler*)
Create a base timer.

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

invalidate()
Invalidates a timer object, stopping it from ever firing again.

is_valid()
Returns a boolean value that indicates whether an EngineTimer object is valid and able to fire.

set_nextfire(*fire_delay*, *interval=-1*)
Set the next firing delay in seconds for an EngineTimer object.

The optional parameter 'interval' sets the firing interval of the timer. If not specified, the timer fires once and then is automatically invalidated.

Time values are expressed in second using floating point values. Precision is implementation (and system) dependent.

It is safe to call this method from the task owning this timer object, in any event handlers, anywhere.

However, resetting a timer's next firing time may be a relatively expensive operation. It is more efficient to let timers autorepeat or to use this method from the timer's own event handler callback (ie. from its ev_timer).

7.10 Workers

ClusterShell public Workers API autodoc.

Notes:

- Workers named *NameWorker* are new-style workers.
- Workers named *WorkerName* are old-style workers.

Contents:

7.10.1 Worker

ClusterShell worker interface.

A worker is a generic object which provides "grouped" work in a specific task.

class ClusterShell.Worker.Worker.**Worker**(*handler*)
Worker is an essential base class for the ClusterShell library. The goal of a worker object is to execute a common work on a single or several targets (abstract notion) in parallel. Concrete targets and also the notion of local or distant targets are managed by Worker's subclasses (for example, see the DistantWorker base class).

A configured Worker object is associated to a specific ClusterShell Task, which can be seen as a single-threaded Worker supervisor. Indeed, the work to be done is executed in parallel depending on other Workers and Task's current parameters, like current fanout value.

ClusterShell is designed to write event-driven applications, and the Worker class is key here as Worker objects are passed as parameter of most event handlers (see the ClusterShell.Event.EventHandler class).

Example of use:

```
>>> from ClusterShell.Event import EventHandler
>>> class MyOutputHandler(EventHandler):
...     def ev_read(self, worker, node, sname, msg):
...             print "%s: %s" % (node, line)
...
```

SNAME_STDERR = 'stderr'
stream name usually used for stderr

SNAME_STDIN = 'stdin'
stream name usually used for stdin

SNAME_STDOUT = 'stdout'
stream name usually used for stdout

__init__(*handler*)
Initializer. Should be called from derived classes.

__weakref__
list of weak references to the object (if defined)

abort()
Abort processing any action by this worker.

Safe to call on an already closing or aborting worker.

current_errmsg = None
set to stderr message in event handler

current_msg = None
set to stdout message in event handler

current_node = None
set to node in event handler

current_rc = None
set to return code in event handler

current_sname = None
set to stream name in event handler

did_timeout()
Return whether this worker has aborted due to timeout.

eh = None
associated *EventHandler*

flush_buffers()
Flush any messages associated to this worker.

flush_errors()
Flush any error messages associated to this worker.

last_error()
Get last error message from event handler. [DEPRECATED] use current_errmsg

last_read()
Get last read message from event handler. [DEPRECATED] use current_msg

read(*node=None*, *sname='stdout'*)
Read worker stream buffer.

Return stream read buffer of current worker.

Arguments:

node – node name; can also be set to None for simple worker having worker.key defined (default is None)

sname – stream name (default is 'stdout')

started = None
set to True when worker has started

task = None
worker's task when scheduled or None

class ClusterShell.Worker.Worker.**DistantWorker**(*handler*)
Base class DistantWorker.

DistantWorker provides a useful set of setters/getters to use with distant workers like ssh or pdsh.

iter_buffers(*match_keys=None*)
Returns an iterator over available buffers and associated NodeSet. If the optional parameter match_keys is defined, only keys found in match_keys are returned.

iter_errors(*match_keys=None*)
Returns an iterator over available error buffers and associated NodeSet. If the optional parameter match_keys is defined, only keys found in match_keys are returned.

iter_keys_timeout()
Iterate over timed out keys (ie. nodes) for a specific worker.

iter_node_buffers(*match_keys=None*)
Returns an iterator over each node and associated buffer.

iter_node_errors(*match_keys=None*)
Returns an iterator over each node and associated error buffer.

iter_node_retcodes()
Returns an iterator over each node and associated return code.

iter_retcodes(*match_keys=None*)
Returns an iterator over return codes and associated NodeSet. If the optional parameter match_keys is defined, only keys found in match_keys are returned.

last_error()
Get last (node, error_buffer), useful in an EventHandler.ev_error() [DEPRECATED] use (current_node, current_errmsg)

last_node()
Get last node, useful to get the node in an EventHandler callback like ev_read(). [DEPRECATED] use current_node

last_read()
Get last (node, buffer), useful in an EventHandler.ev_read() [DEPRECATED] use (current_node, current_msg)

last_retcode()
Get last (node, rc), useful in an EventHandler.ev_hup() [DEPRECATED] use (current_node, current_rc)

node_buffer(*node*)
Get specific node buffer.

node_error(*node*)
Get specific node error buffer.

node_error_buffer(*node*)
Get specific node error buffer.

node_rc(*node*)
Get specific node return code.

Raises **KeyError** – command on node has not yet finished (no return code available), or this node is not known by this worker

node_retcode(*node*)
Get specific node return code.

Raises **KeyError** – command on node has not yet finished (no return code available), or this node is not known by this worker

num_timeout()
Return the number of timed out "keys" (ie. nodes) for this worker.

7.10.2 ExecWorker

class ClusterShell.Worker.Exec.**ExecWorker**(*nodes*, *handler*, *timeout=None*, ***kwargs*)
ClusterShell simple execution worker Class.

It runs commands locally. If a node list is provided, one command will be launched for each node and specific keywords will be replaced based on node name and rank.

Local shell usage example:

```
>>> worker = ExecWorker(nodeset, handler=MyEventHandler(),
...                     timeout=30, command="/bin/uptime")
>>> task.schedule(worker)    # schedule worker for execution
>>> task.run()               # run
```

Local copy usage example:

```
>>> worker = ExecWorker(nodeset, handler=MyEventHandler(),
...                     source="/etc/my.cnf",
...                     dest="/etc/my.cnf.bak")
>>> task.schedule(worker)      # schedule worker for execution
>>> task.run()                 # run
```

connect_timeout option is ignored by this worker.

COPY_CLASS
alias of CopyClient

SHELL_CLASS
alias of *ExecClient*

__init__(*nodes*, *handler*, *timeout=None*, ***kwargs*)
Create an ExecWorker and its engine client instances.

abort()
Abort processing any action by this worker.

set_write_eof(*sname=None*)
Tell worker to close its writer file descriptors once flushed. Do not perform writes after this call.

write(*buf*, *sname=None*)
Write to worker clients.

class ClusterShell.Worker.Exec.**ExecClient**(*node*, *command*, *worker*, *stderr*, *timeout*, *autoclose=False*, *rank=None*)
Run a simple local command.

Useful as a superclass for other more specific workers.

__init__(*node*, *command*, *worker*, *stderr*, *timeout*, *autoclose=False*, *rank=None*)
Create an EngineClient-type instance to locally run ‘command’.

Parameters **node** – will be used as key.

7.10.3 StreamWorker

class ClusterShell.Worker.Worker.**StreamWorker**(*handler*, *key=None*, *stderr=False*, *timeout=-1*, *autoclose=False*, *client_class=<class 'ClusterShell.Worker.Worker.StreamClient'>*)
StreamWorker base class [v1.7+]

The StreamWorker class implements a base (but concrete) Worker that can read and write to multiple streams. Unlike most other Workers, it does not execute any external commands by itself. Rather, it should be pre-bound to "streams", ie. file(s) or file descriptor(s), using the two following methods:

```
>>> worker.set_reader('stream1', fd1)
>>> worker.set_writer('stream2', fd2)
```

Like other Workers, the StreamWorker instance should be associated with a Task using task.schedule(worker). When the task engine is ready to process the StreamWorker, all of its streams are being processed together. For that reason, it is not possible to add new readers or writers to a running StreamWorker (ie. task is running and worker is already scheduled).

Configured readers will generate ev_read() events when data is available for reading, with the stream name passed as one of its argument.

Configured writers will allow the use of the method write(), eg. worker.write(data, ’stream2’), to write to the stream.

__init__(*handler*, *key=None*, *stderr=False*, *timeout=-1*, *autoclose=False*, *client_class=<class 'ClusterShell.Worker.Worker.StreamClient'>*)
Initializer. Should be called from derived classes.

abort()
Abort processing any action by this worker.

Safe to call on an already closing or aborting worker.

read(*node=None*, *sname='stdout'*)
Read worker stream buffer.

Return stream read buffer of current worker.

Arguments:

node – node name; can also be set to None for simple worker having worker.key defined (default is None)

sname – stream name (default is ’stdout’)

set_key(*key*)
Source key for this worker is free for use.

Use this method to set the custom source key for this worker.

set_reader(*sname*, *sfile*, *retain=True*, *closefd=True*)
Add a readable stream to StreamWorker.

Arguments: sname – the name of the stream (string) sfile – the stream file or file descriptor retain – whether the stream retains engine client

(default is True)

closefd – whether to close fd when the stream is closed (default is True)

set_write_eof(*sname=None*)
Tell worker to close its writer file descriptor once flushed.

Do not perform writes after this call. Like write(), sname can be optionally specified to target a specific writable stream, otherwise all writable streams are marked as EOF.

set_writer(*sname*, *sfile*, *retain=True*, *closefd=True*)
Set a writable stream to StreamWorker.

Arguments: sname – the name of the stream (string) sfile – the stream file or file descriptor retain – whether the stream retains engine client

(default is True)

closefd – whether to close fd when the stream is closed (default is True)

write(*buf*, *sname=None*)
Write to worker.

If sname is specified, write to the associated stream, otherwise write to all writable streams.

class ClusterShell.Worker.Worker.**StreamClient**(*worker*, *key*, *stderr*, *timeout*, *autoclose*)
StreamWorker's default EngineClient.

StreamClient is the EngineClient subclass used by default by StreamWorker. It handles some generic methods to pass data to the StreamWorker.

set_write_eof(*sname=None*)
Set EOF flag to writable stream(s).

write(*buf*, *sname=None*)
Write to writable stream(s).

7.10.4 WorkerRsh

class ClusterShell.Worker.Rsh.**WorkerRsh**(*nodes*, *handler*, *timeout=None*, ***kwargs*)
ClusterShell rsh-based worker Class.

Remote Shell (rsh) usage example:

```
>>> worker = WorkerRsh(nodeset, handler=MyEventHandler(),
...                    timeout=30, command="/bin/hostname")
>>> task.schedule(worker)       # schedule worker for execution
>>> task.resume()               # run
```

Remote Copy (rcp) usage example:

```
>>> worker = WorkerRsh(nodeset, handler=MyEventHandler(),
...                    source="/etc/my.conf",
...                    dest="/etc/my.conf")
>>> task.schedule(worker)      # schedule worker for execution
>>> task.resume()              # run
```

connect_timeout option is ignored by this worker.

COPY_CLASS
alias of *RcpClient*

SHELL_CLASS
alias of *RshClient*

class ClusterShell.Worker.Rsh.**RshClient**(*node*, *command*, *worker*, *stderr*, *timeout*, *autoclose=False*, *rank=None*)

Rsh EngineClient.

class ClusterShell.Worker.Rsh.**RcpClient**(*node*, *source*, *dest*, *worker*, *stderr*, *timeout*, *autoclose*, *preserve*, *reverse*, *rank=None*)

Rcp EngineClient.

7.10.5 WorkerPdsh

class ClusterShell.Worker.Pdsh.**WorkerPdsh**(*nodes*, *handler*, *timeout=None*, ***kwargs*)

ClusterShell pdsh-based worker Class.

Remote Shell (pdsh) usage example:

```
>>> worker = WorkerPdsh(nodeset, handler=MyEventHandler(),
...                    timeout=30, command="/bin/hostname")
>>> task.schedule(worker)      # schedule worker for execution
>>> task.resume()              # run
```

Remote Copy (pdcp) usage example:

```
>>> worker = WorkerPdsh(nodeset, handler=MyEventHandler(),
...                    timeout=30, source="/etc/my.conf",
...                    dest="/etc/my.conf")
>>> task.schedule(worker)      # schedule worker for execution
>>> task.resume()              # run
```

Known limitations:

- write() is not supported by WorkerPdsh
- return codes == 0 are not garanteed when a timeout is used (rc > 0 are fine)

COPY_CLASS
alias of *PdcpClient*

SHELL_CLASS
alias of *PdshClient*

set_write_eof()
Tell worker to close its writer file descriptor once flushed. Do not perform writes after this call.

Not supported by PDSH Worker.

write(*buf*)
Write data to process. Not supported with Pdsh worker.

class ClusterShell.Worker.Pdsh.**PdshClient**(*node*, *command*, *worker*, *stderr*, *timeout*, *autoclose=False*, *rank=None*)

EngineClient which run 'pdsh'

__init__(*node*, *command*, *worker*, *stderr*, *timeout*, *autoclose=False*, *rank=None*)

Create an EngineClient-type instance to locally run 'command'.

Parameters **node** – will be used as key.

class ClusterShell.Worker.Pdsh.**PdcpClient**(*node*, *source*, *dest*, *worker*, *stderr*, *timeout*, *autoclose*, *preserve*, *reverse*, *rank=None*)

EngineClient when pdsh is run to copy file, using pdcp.

__init__(*node*, *source*, *dest*, *worker*, *stderr*, *timeout*, *autoclose*, *preserve*, *reverse*, *rank=None*)

Create an EngineClient-type instance to locally run 'cp'.

7.10.6 WorkerPopen

class ClusterShell.Worker.Popen.**WorkerPopen**(*command*, *key=None*, *handler=None*, *stderr=False*, *timeout=-1*, *autoclose=False*)

Implements the Popen Worker.

__init__(*command*, *key=None*, *handler=None*, *stderr=False*, *timeout=-1*, *autoclose=False*)

Initialize Popen worker.

retcode()

Return return code or None if command is still in progress.

class ClusterShell.Worker.Popen.**PopenClient**(*worker*, *key*, *stderr*, *timeout*, *autoclose*)

__init__(*worker*, *key*, *stderr*, *timeout*, *autoclose*)

EngineClient initializer.

Should be called from derived classes.

Arguments: worker – parent worker instance key – client key used by MsgTree (eg. node name) stderr – boolean set if stderr is on a separate stream timeout – client execution timeout value (float) autoclose – boolean set to indicate whether this engine

client should be aborted as soon as all other non-autoclosing clients have finished.

7.10.7 WorkerSsh

class ClusterShell.Worker.Ssh.**WorkerSsh**(*nodes*, *handler*, *timeout=None*, ***kwargs*)

ClusterShell ssh-based worker Class.

Remote Shell (ssh) usage example:

```
>>> worker = WorkerSsh(nodeset, handler=MyEventHandler(),
...                    timeout=30, command="/bin/hostname")
>>> task.schedule(worker)       # schedule worker for execution
>>> task.resume()               # run
```

Remote Copy (scp) usage example:

```
>>> worker = WorkerSsh(nodeset, handler=MyEventHandler(),
...                    timeout=30, source="/etc/my.conf",
...                    dest="/etc/my.conf")
```

(continues on next page)

(continued from previous page)

```
>>> task.schedule(worker)       # schedule worker for execution
>>> task.resume()               # run
```

COPY_CLASS
alias of *ScpClient*

SHELL_CLASS
alias of *SshClient*

class ClusterShell.Worker.Ssh.**SshClient**(*node*, *command*, *worker*, *stderr*, *timeout*, *autoclose=False*, *rank=None*)

Ssh EngineClient.

class ClusterShell.Worker.Ssh.**ScpClient**(*node*, *source*, *dest*, *worker*, *stderr*, *timeout*, *autoclose*, *preserve*, *reverse*, *rank=None*)

Scp EngineClient.

CHAPTER 8

Going further

8.1 Running the test suite

Get the latest *Source* code first.

Note: "The intent of regression testing is to assure that in the process of fixing a defect no existing functionality has been broken. Non-regression testing is performed to test that an intentional change has had the desired effect." (from Wikipedia)

The *tests* directory of the source archive (not the RPM) contains all regression and non-regression tests. To run all tests with Python 2, use the following commands:

```
$ cd tests
$ nosetests -sv --all-modules .
```

Or run all tests with Python 3 by using the following command instead:

```
$ nosetests-3 -sv --all-modules .
```

Some tests assume that *ssh(1)* to localhost is allowed for the current user. Some tests use *bc(1)*. And some tests need *pdsh(1)* installed.

8.2 Bug reports

We use Github Issues as issue tracking system for the ClusterShell development project. There, you can report bugs or suggestions after logged in with your Github account.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W