PacketWeaver Documentation

Release 0.3

Sébastien Mainand, Florian Maury

Nov 02, 2018

Contents

Get started with PacketWeaver! Users, developers and contributors are all welcome!

CHAPTER 1

Presentation

PacketWeaver (pw) is a Python script organization framework. It provides a nice development workflow for quickly creating re-usable Python scripts.

PacketWeaver is a Python framework, which helps build, organize and reuse your Python scripts. It is the most efficient when used in network script development projects.

It sets you up with a simple boilerplate to help building reusable scripts with a clear self-checked input interface and the ability for scripts to be chained between each others. These scripts can be organized into importable packages. This way they can be grouped by projects or topics and easily shared alongside the framework.

In addition to that, it provides a convenient interactive command line interface (CLI) to search, configure and run your scripts and a CLI to run your scripts from your favorite shell and other tools.

1.1 It's all about abilities

When using PacketWeaver, your script source code is placed in an object inherinting from the `AbilityBase` class. Thus, your scripts are refered to as Abilities during the remaining of this documentation. Your scripts can be arbitrarily complex. They just all starts from the main() function that must be overidden. From there, you can import and run other abilities, build complex pipelines for information that goes through several abilities before rendering a result. That is completely up to you.

Every ability share at least a small common code structure. The first common element is metadata. These metadata uniquely identifies your ability within its package. They also provide additional information such as tags for search purposes, author contact info, and the list of the other abilities that might be invoked/imported from this one.

The second common element is the list of input parameters. These parameters, called options, can be any one from several high-level data types, like an IP address, a boolean, a network interface name or an enumeration.

Using these options offers several benefits. Firstly, you can assign them from the interactive CLI before running an Ability. The value assignment is eased thanks to smart autocompletion of the possible values, whenever possible. Also, values are automatically checked against a set of built-in or customizable constraints before their assignment. This alleviates the need for your script to perform the parsing and the verification of the input values.

For example, we could specify an option that must contain the file system path of a valid executable program:

```
PathOpt("cmd_name", default="/bin/ls", executable=True)
```

Also, some special keyword values can be assigned. These keyword values trigger special handlers that might generated random values for your script to use. Various random value generators are available, including the ability to generate randomly IP address within a network subnet without ever returning twice the same IP address. For instance, the following option might be assigned any integer, but by default, it will return a random integer between 0 and 255 when your script will ask for its value:

```
NumOpt("priority", default="RandByte")
```

Finally, the last common element of all Abilities is the `main` function, whose purpose is similar to that of the C `main()` function. Your script entry point is this function, and your script will end when this function returns or an unhandled exception bubbles up from it.

Within the `main` function and, in fact, anywhere within the object that contains it, you may take advantage of any of the helpers that are provided by PacketWeaver, including:

- access to your input parameters as instance attributes, with `self.cmd_name` or `self.priority`.
- use of the integrated display/log engine to organize and colorize your script outputs, e.g `self._view.success('My log')`
- call other Abilities and orchestrate their collaboration using the pipe notation: `ability1 | ablitiy2`

As such, the boilerplate of your Abilities is as short as:

```
from packetweaver.core.ns import *

class Ability(AbilityBase):
    _info = AbilityInfo(name='Say hello')
    _option_list = []

    def main(self):
        self._view.success('Hello, world')
```

1.2 Developer oriented

PacketWeaver is a framework meant for you to develop new Abilities, build some from existing scripts and reuse them afterwards.

Thus the development workflow has been designed to ease the progressive writing and testing of your Abilities. As such, the PacketWeaver interactive CLI enables you to select any Ability, configure it with its input parameters and test the latest version of your code and parameters with the `run` command. If the source code changes while PacketWeaver is running, the latest version is automatically reloaded upon restart of the Ability, while keeping the Ability configuration across the reload. In fact, you can even edit your Ability source code right from the interactive CLI using the `editor` command.

A typical PacketWeaver could be as follows:

```
pw> list
1 Ping a target -- []
2 Ping a prefix -- []
pw> use 1
pw (Ping a target)> set ip_dst=192.0.2.1
pw (Ping a target)> run
192.0.2.1 is DOWN
```

(continues on next page)

(continued from previous page)

```
pw (Ping a target)> editor
... edit the source code to write DOWN in lowercase ...
pw (Ping a target)> run
192.0.2.1 is down
```

During this session, we listed the available Abilities within the currently loaded packages. We then selected the first Ability listed. We set the IP address to ping to 192.0.2.1. We ran the Ability, and saw that the IP address did not answer to our ping. We launched the default text editor to change the source code of our Ability, and we ran the ability once more, automatically using the updated source code.

1.3 A generic tool

Thanks to its generic design, PacketWeaver is not restricted to a specific use case. Mainly used with Scapy hitherto, the framework features are built to be as generic as possible.

The package mechanism and the use of the Python programming language make it a good option to develop specific tools for a wide variety of use cases. Here are some usage examples where PacketWeaver could fit especially well:

- **In the educational/research field by:**
  - building fill-in-the-blank autonomous exercises;
  - creating standalone demonstrations;
  - demonstrating some network protocol mechanisms;
  - automating network devices testing.
- **During a network security audit by:**
  - building a bank of your favorite packages
  - taking advantage of advanced Abilities, developed in research phases
  - reuse content from one assignment to another, and progressively improve it

1.4 What's next

You can start exploring the different parts of this documentation, and get started with the different aspects of the framework.

We hope PacketWeaver will make your future script development easier and help you enhance your existing Python scripts!

CHAPTER 2

Usage

This first section presents the basic interactions with PacketWeaver:

2.1 Installation

All you need to do is getting the source code using git.

Note: The framework was tested under Ubuntu 18.04 LTS (amd64).

There are no distribution/pip packages of this framework available yet, but this definitely is on our todo-list.

2.1.1 Sources and dependencies

Browse to your favorite git cloning destination folder (e.g. ~/git/) and clone the PacketWeaver directory:

```
git clone https://github.com/ANSSI-FR/packetweaver
cd packetweaver
```

Packetweaver has no strong external dependencies. This means you should be ready to go, if you have just the Python3 interpreter and the Python standard library. If you ever ran Packetweaver that way, however, you would miss quite a bunch of helpers that would be automatically disabled.

To enable these helpers, you might want to install some or all of the following dependencies:

- pyroute2
- gmpy2
- pcapy

The easiest way to install them is to use a combination of your package manager and pip3:

```
sudo apt install python3 python3-dev build-essential libpcap-dev python3-pip libgmp-
↪dev libmpfr-dev libmpc-dev
sudo pip3 install pyroute2 gmpy pcapy pyroute2
```

Packages might require additional dependencies. Please refer to their documentation.

See also:

More dependencies are required if you wish to build this *documentation* offline or run the framework automated tests.

2.1.2 Run it

Once PacketWeaver retrieved from git, you may run it from shell using the `run_pw` script:

```
./run_pw
```

You might need to make some minor adjustments to the configuration file:

```
vim packetweaver/pw.ini
```

See also:

Configuration file documentation section

If PacketWeaver displayed its banner and the `pw>` prompt, you are now good to go.

You can now hit "ctrl+d" or type in the `exit` command to quit PacketWeaver and start browsing the next section of this documentation!

2.2 Shell and interactive command line

PacketWeaver gives you two ways of interacting with your Abilities (see *Introduction* for explanations).

2.2.1 The pw interactive CLI

The interactive CLI is the main way of interacting with the framework. It is composed of two levels: one that enables you to browse and to select available Abilities, and another one to configure and run a selected Ability.

The interactive CLI offers command history features similar to that of usual shells, such as history search and command recall, using `ctrl+p`, `ctrl+n` and `ctrl+r`.

To cancel a command, you may use `ctrl+c`. `ctrl+d` and `exit` may be used to return to unselect an Ability, or exit the framework. Finally, `tab` can be used to trigger autocompletion of the current command line, whenever available.

Ability browsing and selection

To launch the interactive CLI, you may run the `run_pw` script located at the root of the PacketWeaver git repository:

```
./run_pw
```

This CLI understands a bunch of commands whose list you can obtain by typing the `help` command or `?` for short. Help of any of the listed commands can be obtained by using the `help` command followed by the name of a command:

```
pw> help list
```

For instance, the `list` command displays a list of all the available standalone Abilities that are loaded from the configured *packages*. Each list entry is indexed by a number, so that you can refer to this Ability by its associated number. For instance, to use the `Ping a target` Ability, you would type `list` and then `use 1` to select the Ability for use, because it is the first listed:

```
pw> list
1 Ping a target -- []
2 Ping a prefix -- []
3 DNSProxy -- ['Application_Layer', 'Threaded', 'DNS']
pw> use 1
pw (Ping a target)>
```

While `list` enumerates all loaded standalone Abilities, this might not be convient if you loaded a very large package containing tons of abilities. You may use the `search` command to find any Ability that would be listed by `list`. Search results are indexed, just like listed Abilities are.

`search` matches Ability names by default, using a case-insensitive comparison:

```
pw> search ping
1 ping a target --
2 ping a prefix --
pw> search dns
1 dnsProxy -- Application_Layer, Threaded, DNS
```

You can also search by tags:

```
pw> search app
No matching ability found.
pw> search -a application_layer
1 DNSProxy -- application_layer, Threaded, DNS
```

Searching by tags is eased by autocompletion that will provide you with a list of all the tags that are registered by currently available Abilities.

Searches for tags may use multiple tags simultaneously. When multiple tags using a logical operation between tags (and or or) may be specified with flags. The `-o` flag indicates an OR, while `-a` indicates an AND.

For instance, you could search for all Abilities that are related to DNS and to MITM:

```
pw> search -a dns mitm
No matching ability found.
```

You may also want to list all Abilities that are either related to DNS or to HTTP:

```
pw> search -o dns http
1 DNSProxy -- application_layer, Threaded, DNS
```

Any of these flags can be used if you are filtering with only one tag.

Note: A default index is built when the framework starts. You can quickly re-open you current in-development Ability across framework restart by recalling the last `use` command to access it directly.

After selecting an index, if no errors are displayed, you now are interacting with a CLI specific to the selected Ability. This CLI is described in details in the next chapter.

Configuring, editing and running an Ability

Once more, the `help` command is available to list the commands that are available with this new interactive CLI, and the `help` command may be used to obtain further details about the listed commands.

You may also get more details about the selected Ability, with the `info` command:

```
pw (DNSProxy)> info
------------------------------ [ DNSProxy ] -------------------------------
type: Standalone
description: Replacement for DNSProxy
authors: Florian Maury
tags: Application_Layer, Threaded, DNS
reliability: Incomplete
---------------------------------------------------------------------------
```

Details about the role and the usage of the selected Ability may also be obtained with the `howto` command:

```
pw (DNSProxy)> howto
This DNS proxy intercepts DNS requests at OSI layer 2.
...
the real DNS server is connected to a different card than the victim.
```

The `options` command (or its alias `ls`) lists the Ability parameters that may be set before running that Ability:

```
pw (DNSProxy)> options
------------------------------- [ Options ] -------------------------------
fake_zone = /bin/true
policy_zone = /bin/true
ip_src (Optional) = None
ip_dst (Optional) = None
port_dst (Optional) = 53
interface = None
outerface (Optional) = None
quiet = True
---------------------------------------------------------------------------
```

Options may be set to user values or reset to their default values using the `set` and `clear` commands. For instance, let's set some option value:

```
pw (DNSProxy)> set ip_dst=8.8.8.8
pw (DNSProxy)> set interface=eth0
pw (DNSProxy)> set ip_src=192.0.2.1
pw (DNSProxy)> options
------------------------------- [ Options ] -------------------------------
fake_zone = /bin/true
policy_zone = /bin/true
ip_src (Optional) = 192.0.2.1
ip_dst (Optional) = 8.8.8.8
port_dst (Optional) = 53
interface = eth0
outerface (Optional) = None
quiet = True
---------------------------------------------------------------------------
```

Note: Some input parameters (such as IpOpt) support value generators. You can get a list of them using the autocompletion when trying to set a new value (`set ip_dst=` + `Tab Tab`) to them.

For instance, you may set an IP address to the special value `RandIP4`. A new IP address will be generated each time the Abliity is run:

```
pw (DNSProxy)> set ip_dst=RandIP4
pw (DNSProxy)> options
------------------------------- [ Options ] -------------------------------
fake_zone = /bin/true
policy_zone = /bin/true
ip_src (Optional) = 192.0.2.1
ip_dst (Optional) = RandIP4
port_dst (Optional) = 53
interface = eth0
outerface (Optional) = None
quiet = True
---------------------------------------------------------------------------
```

You may also use this special keyword as a function, to set a random value to the variable. This random value won't change across runs:

```
pw (DNSProxy)> set ip_dst=RandIP4()
pw (DNSProxy)> options
------------------------------- [ Options ] -------------------------------
fake_zone = /bin/true
policy_zone = /bin/true
ip_src (Optional) = 192.0.2.1
ip_dst (Optional) = 198.51.100.42
port_dst (Optional) = 53
interface = eth0
outerface (Optional) = None
quiet = True
---------------------------------------------------------------------------
```

Note: If you try to set a invalid value for an option, you will receive an error message and the stored option value will remain unchanged. Each type of options is designed with a set of constraints, including some customizable ones, to validate the values that may be assigned to it.

Now let's reset the source IP address to its default value:

```
pw (DNSProxy)> clear ip_src
pw (DNSProxy)> options
------------------------------- [ Options ] -------------------------------
fake_zone = /bin/true
policy_zone = /bin/true
ip_src (Optional) = None
ip_dst (Optional) = 8.8.8.8
port_dst (Optional) = 53
interface = eth0
outerface (Optional) = None
quiet = True
---------------------------------------------------------------------------
```

Now let's reset all options to their default value:

```
pw (DNSProxy)> clear
pw (DNSProxy)> options
```

(continues on next page)

(continued from previous page)

```
------------------------------- [ Options ] -------------------------------
fake_zone = /bin/true
policy_zone = /bin/true
ip_src (Optional) = None
ip_dst (Optional) = None
port_dst (Optional) = 53
interface = None
outerface (Optional) = None
quiet = True
---------------------------------------------------------------------------
```

Once you and your options are all set, you may run the Ability with the `run` command:

```
pw (DNSProxy)> run
```

Abilities may run until they are done with their tasks. In that case, they will give back control to the CLI once they terminated. Other Abilities may start some services and are designed to run until interrupted by the SIGINT signal (ctrl+c). Of course, you may interrupt any running Ability, in case it went into an infinite loop of sorts, with the same key sequence.

If you are satisfied by the results of the Ability that you just run, you may want to save the parameter values that you used. This enables you to reload them, during a future session of PacketWeaver, or to back them up for a future audit report, for instance.

To save the current parameter values, you may use the `save` command:

```
pw (DNSProxy)> save /path/to/file.ini
```

To reload them, you may use the `load` command:

```
pw (DNSProxy)> load /path/to/file.ini
```

At some point, you may feel the need to make some minor code adjustments in the Ability you are about to run (e.g. a bug fix…). You don't need to exit PacketWeaver for this. The `editor` command will open the source code file of the selected Ability and of all other Abilities that take part in the selected Ability operations. Which source code editor is run is configured within PacketWeaver *configuration* file.

Finally, once configured you may ask of PacketWeaver to automatically generate a shell command line that will run this Ability with the current configuration from shell:

```
pw (DNSProxy)> cmd
export PW_OPT_FAKE_ZONE='/bin/true'
export PW_OPT_POLICY_ZONE='/bin/true'
export PW_OPT_IP_SRC='None'
export PW_OPT_IP_DST='None'
export PW_OPT_PORT_DST='53'
export PW_OPT_INTERFACE='None'
export PW_OPT_OUTERFACE='None'
export PW_OPT_QUIET='True'
export PYTHONPATH='/opt/pw/pw/packetweaver:/usr/local/lib/python2.7/dist-packages/
↪python_twitter-1.0-py2.7.egg:/usr/local/lib/python2.7/dist-packages/oauth2-1.5.211-
↪py2.7.egg:/usr/local/lib/python2.7/dist-packages/pympress-0.3-py2.7.egg:/opt/pw/pw:/
↪usr/lib/python2.7:/usr/lib/python2.7/plat-x86_64-linux-gnu:/usr/lib/python2.7/lib-
↪tk:/usr/lib/python2.7/lib-old:/usr/lib/python2.7/lib-dynload:/usr/local/lib/python2.
↪7/dist-packages:/usr/lib/python2.7/dist-packages:/usr/lib/python2.7/dist-packages/
↪PILcompat:/usr/lib/python2.7/dist-packages/gst-0.10:/usr/lib/python2.7/dist-
↪packages/gtk-2.0:/usr/lib/python2.7/dist-packages/ubuntu-sso-client:/usr/lib/
↪python2.7/dist-packages/ubuntuone-client:/usr/lib/python2.7/dist-packages/ubuntuone-
↪couch:/usr/lib/python2.7/dist-packages/ubuntuone-storage-protocol:/opt/pw/scapy'
```

(continues on next page)

(continued from previous page)

```
python /opt/pw/pw/packetweaver/pw.py use -p base -a DNSProxy --fake_zone=${PW_OPT_
↪FAKE_ZONE} --policy_zone=${PW_OPT_POLICY_ZONE} --ip_src=${PW_OPT_IP_SRC} --ip_dst=$
↪{PW_OPT_IP_DST} --port_dst=${PW_OPT_PORT_DST} --interface=${PW_OPT_INTERFACE} --
↪outerface=${PW_OPT_OUTERFACE} --quiet=${PW_OPT_QUIET}
```

This command line can be copy/pasted in a shell console to start the ability with these options.

Warning: This command line generation is experimental and might quickly evolve.

Note: You can also use the `cmd oneline` option to get a bash oneline command that can be directly tested.

2.3 Configuration

PacketWeaver can be configured with an .ini file. The default config file is located in PacketWeaver installation directory. This is the one used when running the `run_pw` shell script. However, you may specify another location using the command line option `--config` or `-c` for short. In that case, the syntax is:

```
$ ./run_pw --config=/path/to/my/pw.ini interactive
```

The configuration file is composed of four sections: * Dependencies * Packages * Tools * Internals

Each of these sections are described herebelow.

Note: Make sure to restart the framework after a modification to take the new parameters into account.

2.3.1 Dependencies Configuration Section

The Dependencies section contains a list of Python module paths that will be added to your PYTHONPATH environment variable during PacketWeaver execution. Its only purpose is to spare you from setting the PYTHONPATH environment variable before each run of PacketWeaver. For this reason, the section only need to contain path of dependencies that are not already in the Python default import paths and in the PYTHONPATH environment variable that you may have set otherwise.

Key names does not matter in this section and only need to be unique.

For example, if you want to use the lastest Scapy version, you may clone it whatever git directory you use (e.g. ~/git) and set your `Dependencies` section as follows:

```
[Dependencies]
scapy=/absolute/path/to/scapy/
```

If you want to use a relative a path, the path must be relative to the directory of your *packetweaver/pw.py* file. If Scapy repository was cloned in the same directory as your PacketWeaver repository, you would set your configuration like this:

```
[Dependencies]
scapy=../../scapy/
```

Finally several paths can be specified, using two different keys:

```
[Dependencies]
scapy=../../scapy/
yourlib=../../yourlib/
```

2.3.2 Packages Configuration Section

PacketWeaver enables you to file your Abilities into seperate logical groups of Abilities. These groups of Abilities are called Packages in the PacketWeaver lingo. You may want to organize your Abilities like this because they all share a common set of dependencies. Or maybe there are all interdependent. Or maybe they share a common goal or purpose. Or maybe you are the type to label everything. No judge.

Packetweaver loads at startup all packages that are declared in this configuration section. The more packages you have, the more results are displayed by the `list` interactive command line command and the more Abilities can be found with the `search` command. The more packages are loaded, the slower Packetweaver is to startup.

Note: In the Packages configuration section, key names are crucial, and renaming them might be expensive. Indeed, there might be some source code references to that name, when declaring interdependencies between Abilities. Thus, it is advised to use virtually unique package names. For instance one could use a prefix to namespace the package names: `mycompanyname_dancing_monkey`.

The entry values of this section are paths to Python modules that are valid PacketWeaver packages. Please, refer to the *Writing a package* section of this documentation for more information about Packages declaration. Paths may be absolute paths on the filesystem, or paths relative to the `pw.py` file, just like for the Dependencies configuration section.

For example, your Packages configuration section might look like this:

```
[Packages]
base=abilities/
mycompany_flying_monkeys=/opt/pw_circus/
mycompany_ducking_ducks=../../pkgs/missing_animals/abilities/
```

2.3.3 Tools Configuration Section

This section only contains one configuration key hitherto: `editor`.

Editor

The Editor option lets you select your favorite text editor. It will be used by the `editor` command available in the interactive CLI, after you selected an Ability. For instance, you would configure this option to be:

- your default system editor (probably graphical):

```
[Tools]
editor=xdg-open
```

- a text mode editor:

```
[Tools]
editor=vim
```

- a text mode editor using options:

```
[Tools]
editor=emacs -nw
```

2.3.4 Internals configuration section

This section only contains one configuration key hitherto: HistFile.

History

The PacketWeaver interactive CLI offers command history. This history is saved to a dedicated file. You can customize its name and location by editing the `HistFile` configuration key. To store it in your home directory, you may specify:

```
[Internals]
HistFile=~/.pwhistory
```

Paths are expanded if need be.

2.4 Packages

Functionally speaking, a PacketWeaver package is a set of Abilities, grouped by purpose, topic, authors, interdependencies, version control access-control or whatever other reasons you might think of. Having this package feature enables you to share/publish only parts of your scripts. It also introduces namespaces, because Abilities must have a unique name within a same package, but a same name can be reused across packages.

Python-wise, a PacketWeaver package is a Python module, whose `__init__.py` file contains a global variable called `exported_abilities`. This variable contains a list of class objects inheriting (directly or indirectly) from the `AbilityBase` class.

2.4.1 Basic structure

We suggest that you structure your PacketWeaver packages as follows:

```
pw-pkg-test/
├── doc/
└── abilities/
    ├── demo/
    │   ├── demo_app.py
    │   └── __init__.py
    ├── __init__.py
    └── test_app.py
```

The `abilities` folder is a Python module, whose `__init__.py` contains a list of the activated abilities:

```
import test_app
from demo import demo_app

exported_abilities = [
    test_app.Ability,
    demo_app.Ability,
]
```

In that case, `test_app.py` and `demo_app.py` both contain a Python class called `Ability` that herites from `AbilityBase`. Abilities must be declared in this list to be usable by other Abilities or to be listed by the PacketWeaver CLI.

2.4.2 Package usage

To use a package, you must declare it in PacketWeaver configuration file, as described in the *configuration* file section.

2.4.3 Naming convention

Most Abilities are stored in separate Python files, each containing an Ability called `Ability`.

PacketWeaver package directories are generally named pw-pkg-<your_pkg_name>. People will most probably import your package by configuring whatever you defined as `your_pkg_name` as this package key for the PacketWeaver configuration file Package section. Since PacketWeaver package names must be unique in a configuration file, it is advised that you prefix your package name, for instance with the name of your company or group.

CHAPTER 3

Developing Abilities

This section describes the writing of an Ability.

First, we will take a look at the boilerplate, which is common to all Abilities.

Then, we will look at the way to insert your code into this boilerplate, and how to take advantage of the predefined builtin parameter types.

We will also cover how to handle external library imports with regards to PacketWeaver built-in mechanism to detect missing requirements.

Then, we will cover the topic of nested Abilities, and how to send parameters and get results from an Ability.

Finally, we will study how to start parallel Abilities, how they communicate, and how to interact with them.

3.1 Basic ability writing

To write an Ability, you need to use a very short boilerplate:

```
# coding: utf8
from packetweaver.core.ns import *

class Ability(AbilityBase):
    _info = AbilityInfo(
        name='Insert your Ability name here',
    )

    def main(self):
        # Insert your code here
        return 0
```

To get started, you need to:

- copy/paste that boilerplate into a Python file;

- change the name in the AbilityInfo instantiation. This must be a name unique to the package that contains that Ability;
- import this Python file from the `__init__.py` file of the package that contains your Ability and add a reference to your Ability class object to the `exported_abilities` of the package that contains your Ability. This is described in details in the *configuration section* of this documentation;
- stick some code into the `main` mathod.

To check that you got that right, you can simply have the `main` method return a constant and try and run that Ability from the PacketWeaver interactive CLI. For example, let's pretend that you did not change the name of the template Ability, then you can test it like this:

```
./run_pw
pw> list
pw> search insert
1 insert your Ability name here
pw> use 1
pw (Insert your Ability name here)> run
0
pw>
```

The above template will return 0, and this will be displayed by the CLI, upon completion of this Ability.

Tip: Once you have your Ability selected in the interactive CLI, remember that you can type the `editor` command to edit its source code. Every changes will be reloaded automatically, so that you need to type `run` again to see the result of your update.

Warning: Whenever you copy/paste an existing Ability or the template Ability, remember to change the `name` value. Failing to do so will break PacketWeaver import mechanism

3.1.1 Complete the meta-data

The Ability name is not the only information that you may set up.

Some information are just for display purposes, when you enter the `info` command after selecting an Ability in the interactive CLI. This is the case of:

- *description*;
- *authors*;
- *references*;
- *diffusion*;
- *reliability*.

Two other pieces of information have special purposes.

The *tags* are a list of strings, some predefined and standard in PacketWeaver, some custom. Any tag can be searched (and autocompleted) when using the `search` interactive CLI command.

The *type* values either `AbilityType.STANDALONE` or `AbilityType.COMPONENT`. A component Ability is an Ability that can only be used as part of nested Abilities. Conversely, A standalone Ability can be used both as a component of nested Abilities, and can be run directly from the CLI. The rationale is that you may not want to polute your Ability listing in the CLI with all Abilities, including some that are relatively abstract and not meant to be run directly.

Here is a complete example to illustrate them:

```
_info = AbilityInfo(
    name='Ability basics',
    description='Demonstrate a basic ability',
    tags=['myproject', Tag.EXAMPLE],
    authors=['pw-team',],
    references=[['PacketWeaver online documentation', 'packetweaver.readthedocs.io'],
↪],
    diffusion='public',
    reliability=Reliability.RELIABLE,
    type=AbilityType.STANDALONE
)
```

Defined constants

To help building a consistent searchable database of Abilities, some constants were defined. This is the case for the tags or the reliability information.

Note: Using the built-in values is not mandatory (except for the `type`): you can replace them by any string value. Just make sure that it does not make your Ability more difficult to find by adding tags very similar to the default ones. Contributors are encouraged to suggest new built-in tags.

All the built-in values are defined in the *packetweaver/core/models/status.py* source file.

3.1.2 The `howto` method

The `main` method is not the only method that is common to all Abilities and that Ability developers are meant to override. The `howto` method is called by the `howto` command that users may enter after selecting an Ability. While the exact behavior is up to Ability developers, this method is meant to display some kind of message for the user better grasp how to use the Ability they just selected.

Feel free to provide step-by-step descriptions, to add interactions or to provide comprehensive guidance on how to use your ability.

3.1.3 Adding parameters to your Ability

Several types of parameters may be passed to an Ability. Parameters may be set directly from the CLI, or they may be passed by another Ability in case of nested Abilities.

These parameters are strongly-typed: values are automatically checked upon assignment, with an AttributeError exception being raised if the value is inappropriate. These parameters may also contain special values, which triggers value generation at running time.

To add parameters to your Ability, you need to set a class property named `_option_list` containing, as the name implies, a list of options instances.

Here follows a example of such an option list:

```
_option_list = [
    PathOpt('path', default='/bin/true', comment='mon exe', executable=True),
    PathOpt('path', default=None, comment='path to nowhere', optional=True,␣
↪executable=False),
    IpOpt('mon_ip', default='RandIP', comment='mon ip'),
    IpOpt(ns.OptNames.IP_DST, default='127.0.0.1', comment='IP of the target'),
    MacOpt(ns.OptNames.MAC_SRC, default='RandMac', comment='Mac of the sender'),
    StrOpt('data', default='useful string', comment='Some data'),
    NumOpt('number', default=0, comment='A number (like port, counters...'),
    ChoiceOpt('action', ['run', 'stop', 'reboot'], comment='performed on the dstIp '),
]
```

Parameters cover various data types such as IP and MAC addresses, strings, numbers, network cards, IP subnets, booleans, file system paths or choice options.

All parameter types may also contain “None”, which can be assigned to parameters that are optional.

Parameter constructors all receive a name as a first parameter. This name is used to set and get this option value, both from the command line and from the code.

While developing an Ability, you may obtain the current parameter value from any method of the Ability, by accessing it as a attribute from that Ability class instance. For instance, to access the value of a BoolOpt, representing a boolean, called `my_option_name`, you may simply write:

```
self.my_option_name
```

Note: While parameter naming is free of constraints, you might want to use some of the built-in names, that are listed in the `OptNames` class in *packetweaver/core/models/status.py*. Using these names in your Ability creates a sense of consistency that makes the user safe.

Warning: Please keep in mind that if you want to access your option value using the attribute syntax, you need to keep your names within the boundaries of the Python variable naming constraints. If you want to use hyphen, spaces or whatever other invalid characters, you will need to access the parameter value using the following syntax:

```
self.get_opt('my name, containing spaces and punctuation')
```

When reading the value of a parameter containing one of the special expressions that generate data, the latest generated value is cached, so that multiple read yield the same result:

```
a = self.my_option_name
b = self.my_option_name
assert(a == b)
```

You may force the generation of a new value by asking for a cache bypass. For this, there is no direct read of the attribute. Instead, you need to use the `get_opt` method, inherited from `AbilityBase`:

```
a = self.my_option_name
b = self.get_opt('my_option_name', bypass_cache=True)
assert(a != b)  # Most probably different, if the RNG God is nice with us :)
```

Boolean parameters

Booleans are represented by the `BoolOpt` class.

Values that can be successfully assigned to a `BoolOpt` are:

- `True`;
- `False`;
- `"True"`;
- `"False"`;
- `None` or `"None"` if the `BoolOpt` is optional.

You may define a default value using the `default` keyword argument, when declaring the `BoolOpt`:

```
BoolOpt('my_bool', default=False)
```

The default default value is `False`.

String parameters

Strings are represented by the `StrOpt` class.

Any string may be assigned to such a parameter, except `"None"` and `RandString`. The former can be assigned when the `StrOpt` is optional. The latter is a special keyword, which indicates that when reading the parameter value, strings of random length and content must be generated.

As for the others, the `default` keyword argument enables you to define a default value for this parameter. The default default value is `"RandString"`.

Number parameters

Numbers (both integers anf floats) are represented by the `NumOpt` class.

Values that can be successfully assigned to a `NumOpt` are:

- any integer or float in Python int/float format
- any string that can be parsed by Python standard library into an integer or float
- `None` or `"None"` if the `NumOpt` is optional;
- `"RandByte"` to generate a random integer between 0 and 2**8;
- `"RandSByte"` to generate a random integer between -2**7 and 2**7 - 1;
- `"RandShort"` to generate a random integer between 0 and 2**16;
- `"RandSShort"` to generate a random integer between -2**15 and 2**15 - 1;
- `"RandInt"` to generate a random integer between 0 and 2**32;
- `"RandSInt"` to generate a random integer between -2**31 and 2**31 - 1;
- `"RandLong"` to generate a random integer between 0 and 2**64;
- `"RandSLong"` to generate a random integer between -2**63 and 2**63 - 1.

As for the others, the `default` keyword argument enables you to define a default value for this parameter. The default default value is `"RandByte"`.

IP address parameters

IP addresses are represented by the `IpOpt` class. The class may store any IP address, be it in IPv4 or IPv6.

Values that can be successfully assigned to a `IpOpt` are:

- any IPv4 in quad-dotted format;
- any IPv6, compressed or uncompressed;
- `"RandIP4"` to generate a random IPv4 address, which might be anywhere in the address space, including private networks, class D and E, and loopback;
- `"RandIP6"` to generate completely random IPv6 address, with no guarantee that the address will be valid
- `"RandIP_classA"` to generate a random IP within the IPv4 class A
- `"RandIP_classB"` to generate a random IP within the IPv4 class B
- `"RandIP_classC"` to generate a random IP within the IPv4 class C
- `"RandIP_classD"` to generate a random IP within the IPv4 class D
- `"RandIP_classE"` to generate a random IP within the IPv4 class E

Note: A more complete syntax is on our TODO-list to enable you to define random ranges (e.g. 192.168.10-20.*).

As for the others, the `default` keyword argument enables you to define a default value for this parameter. The default default value is `"RandIP4"`.

Warning: `IpOpt` value validation are using either the Python `ipaddress` module or the `netaddr` module. If you do not have any of these, then the validation will not be performed and just about any value will be tolerated.

IP subnet/prefix parameters

IP prefixes are represented by the `PrefixOpt` class. This class may store any IP prefix, be it in IPv4 or IPv6.

This parameter type is meant to enable you to walkthrough the prefix, by generating each and every one IP address of the specified prefix. Generation by either be ordered, from the first address to the last address of the prefix (excluding the network address and the broadcast address).

As for the others, the `default` keyword argument enables you to define a default value for this parameter. The default default value is `"127.0.0.0/8"`.

This parameter constructor also has a `ordered` keyword argument, which values `False` by default. If `True`, the generation of the IP address of the prefix will be from the first address to the last one. If `False` and if prerequisites are met, the IP address will be generated randomly inside the prefix without ever generating the same address twice. This might come in handy when scanning large networks, if you do not want to indirectly harass a middlebox such as a firewall that is on path with many scanned endpoints inside a subnet.

When all IP addresses of the specified prefix are generated, the next cache bypass raises a `StopIteration` exception.

Here follows an exemple of a `PrefixOpt` instantiation:

```
PrefixOpt('MyPrefix', default='192.0.2.0/29', ordered=True)
```

And the usage of such an option could be:

```
try:
    while True:
        print(self.get_opt('MyPrefix', cache_bypass=True))
except StopIteration:
    pass
```

Caution: This parameter does not work with /31 and /32 prefixes, and it will not work either with /127 and /128 prefixes.

Warning: This parameter is heavily based on the Python `ipaddress` or `netaddr` module, so you will need them to get anywhere with this option. Also, you might need the Python `gmpy2` module to have random IP address generation from this parameter.

MAC address parameters

Physical addresses (MAC addresses) are represented by the `MacOpt` class.

Values that can be successfully assigned to a `MacOpt` are:

- any well-formated MAC address, as a string;
- `"Mac00"` as a shorthand for the null MAC address;
- `"MacFF"` as s shorthand for the broadcast MAC address;
- `"RandMac"` to generate a random MAC address;
- `"RandMulticastMac"` to generate a random Multicast MAC address from the IPv4 multicast associated MAC address range;
- `None` or `"None"` if the `MacOpt` is optional.

As for the others, the `default` keyword argument enables you to define a default value for this parameter. The default default value is `"RandMac"`.

Choice parameters

Choice parameters represent alternatives from which you can select one value. The set of available choices is up to Ability developers, who must list them at instantiation of the `ChoiceOpt` class:

```
ChoiceOpt('favorite_food', ['pizza', 'beer', 'greenStuff'])
```

As for the others, the `default` keyword argument enables you to define a default value for this parameter. The default default value is the first choice in the specified list.

If the choice parameter is optional, the special `"None"` value may be assigned too.

Port number parameters

Port numbers are represented by the `PortOpt`.

Values that can be successfully assigned to a `PortOpt` are:

- any port number between 0 and 65535 inclusive as a Python integer or a Python string;

- "RandPort" to generate a random port number between 1 and 65535 inclusive;
- "RandPrivilegedPort" to generate a random port number between 1 and 1023 inclusive.

As for the others, the default keyword argument enables you to define a default value for this parameter. The default default value is "RandPort".

Network card parameters

Network card names are represented by the NicOpt.

Valid values for NicOpt are all network device name on the current computer, be it the name of a network card, a bridge, a virtual Ethernet adapter, or any other types of network devices really.

As an exception, this parameter accepts None and "None" even if this parameter is not optionnal. The rationale is that network device names vary between platform and distros and having a non-None default value would break on random platforms. As such, the default default value is None and it is advised not to override it.

Warning: Validation of NicOpt values is performed using the Python pyroute2 library. If this library is missing, any value will be accepted.

Filesystem path parameters

Filesystem paths are represented by PathOpt.

Valid values for this parameter type are all strings. The valid paths may however be constrained even further using a set of keyword arguments at instanciation time. Here follows the list of the various constraints that may be specified:

- must_exist: if True, the value must be the path of an existing file; if False, the file must not exist at the time of check. None means “do not care”;
- readable: if True, the value must be an existing file and the user running PacketWeaver must have read access on that file; if False, the user running PacketWeaver must not have read access on that file. None means “do not care”;
- writable: if True, the value must be an existing file and the user running PacketWeaver must have write access on that file; if False, the user running PacketWeaver must not have write access on that file. None means “do not care”;
- executable: if True, the value must be an existing file and the user running PacketWeaver must have execute access on that file; if False, the user running PacketWeaver must not have execute access on that file. None means “do not care”;
- is_dir: if True, the path must be one of an existing directory. If False, the path may be a directory or not.

Warning: must_exist is subject to race conditions. This constraint is

not for security purposes.

So, basically, if you want to create a PathOpt to write a log file, you might want to be sure that you are not overwriting any existing file:

```
PathOpt('log_file', default='/var/log/mylog.txt', must_exist=False)
```

Caution: `must_exist=False` is incompatible with `readable`, `writable` and `executable`, because the file does not exist, and does not have any ACL (yet).

Paths can be specified in an absolute or relative manner, the latter being interpreted starting from the ability's ability package root path.

3.1.4 Text output

To help you highlight your code output, the `_view` object is available to display colored messages and block structures. Here are some examples you may use:

```
self._view.success('Display in green')
self._view.delimiter('A dashed line with title')  # with a fixed len
self._view.delimiter()  # a dashed line with the same length
self._view.warning('Display in yellow')
self._view.error('Display in red')
self._view.fail('Display in cyan')
self._view.progress('Display in blue')
self._view.debug('Display in purple')
self._view.success('Display in your default terminal color')
```

3.2 Advanced Ability Writing

3.2.1 Using another Ability

About the Ability Types

Simple Abilities might be self-contained, and self-sufficient. However, to improve reusability, one might want to split code into multiple Abilities that may or may not be run independently from the CLI.

There are actually two types of Abilities: standalone ones, and components.

A component is an Ability that can only be called from an other Ability. They are not listed in the CLI. Conversely, a standalone Ability is listed in the CLI and might be used by itself, although it might also be called from an other Ability.

The type of Ability is defined with the `type` parameter of an `AbilityInfo` instance:

```
from packetweaver.core.ns import *

class ComponentAbility(AbilityBase):
    _info = AbilityInfo(name='Example', type=AbilityType.COMPONENT)
...
class StandaloneAbility(AbilityBase):
    _info = AbilityInfo(name='AnotherExemple', type=AbilityType.STANDALONE)
```

Declaring a Dependency to another Ability

An Ability using other Abilities must declare them as dependencies. This is done by defining a class property called `_dependencies`. It contains a list of strings or tuples describing each dependency.

A string may be used when using some of the builtin Abilities from PacketWeaver. For instance, an Ability using the built-in Man-in-the-middle Ability, may declare:

```
class Ability(AbilityBase):
    _dependencies = [ 'mitm' ]
```

In the general case, though, tuples are used. Each tuple is composed of three elements:

- the pet name that you want to use to refer to that dependency within your Ability;
- the name of the package (as defined in the PacketWeaver configuration file) that contains the Ability that your Ability relies on;
- the name of the Ability that your Ability relies on, as declared in that AbilityInfo name attribute.

For instance, if an Ability uses another Ability named *Test your might* stored in the package *TestPackage*, that Ability `_dependencies` declaration could be:

```
_dependencies = [('mytest', 'TestPackage', 'Test your might')]
```

Getting an Instance of another Ability

Once an Ability declared as a dependency, you may obtain an instance of that Ability using the built-in `get_dependency` method:

```
class Ability(AbilityBase):
    _dependencies = [('mytest', 'TestPackage', 'Test your might')]
    def main(self):
        instance = self.get_dependency('mytest')
```

If the dependency has input parameters declared through a `_option_list` class property, you may set them using keyword arguments, during the `get_dependency` call. For instance, if the *Test your might* Ability defined a `NumOpt` called `skill_level`, one could define the argument like this:

```
class Ability(AbilityBase):
    _dependencies = [('mytest', 'TestPackage', 'Test your might')]
    def main(self):
        instance = self.get_dependency('mytest', skill_level=9000)
```

Configuring an Ability Instance

Parameters of Abilities that are not yet started may be set either at instantiation time, with the keyword arguments of the `get_dependency` call or by directly setting them as attributes of the object instance of that Ability. Said otherwise, this line:

```
inst = self.get_dependency('mytest', skill_level=9000)
```

is equivalent to:

```
inst = self.get_dependency('mytest')
inst.skill_level = 9000
```

and this line:

```
inst = self.get_dependency('mytest', skill_level=9000, other_stat=10)
```

is equivalent to:

```
inst = self.get_dependency('mytest')
inst.set_opts(skill_level=9000, other_stat=10)
```

Note: If your parameter name contains characters that are invalid for a Python attribute name, you may set it using `set_opt`:

```
inst.set_opt('skill_level++', 9000)
```

Once started, trying to alter a parameter value leads to an Exception being raised.

Starting the Dependency

Once you have a reference to an object instance representing another Ability, you may run it by calling the `start` method on it:

```
class Ability(AbilityBase):
    _dependencies = [('mytest', 'TestPackage', 'Test your might')]
    def main(self):
        instance = self.get_dependency('mytest')
        instance.start()
```

If arguments are passed to the start invocation, they are passed as is to the `main` method of that Ability. For instance, let's assume that the *Test your might* Ability `main` method is declared as:

```
def main(self, arg1, arg2=True, arg3="Mighty"):
```

One could call that Ability with arguments like this:

```
instance.start("arg1value", arg3="Weak")
```

Whether to use arguments with the `start` method or using PacketWeaver `_option_list` parameters is up to the Ability developer. One case where using the `start` argument is convenient is when one want to pass a data type that is not declared as a PacketWeaver option type, or when the value is an arbitrary mutable Python object reference. In the latter case, a special argument should be passed during `start` invokation, to prevent deepcopy of the parameter value:

```
instance.start({'mutable': 'array'}, deepcopy=False)
```

About Multi-threaded Abilities

All Abilities must inherit directly or indirectly from the `AbilityBase` class.

Abilities inheriting directly from `AbilityBase` are synchronous. That means that when started, they take control over either the CLI or the calling Ability, and they give control back, once they are done with their tasks.

Abilities may however inherit from `ThreadedAbilityBase` instead of `AbilityBase`. In that case, PacketWeaver automatically generates a thread to handle the tasks. That means that when started, these Abilities will execute a separate control flow. An Ability inheriting from `ThreadedAbilityBase` that is run from the CLI executes until the `main` method returns or an unhandled exception bubbles up. It may however call the `_wait` method, to wait for a PacketWeaver stop signal. A stop signal is sent to a threaded Ability when the *ctrl+c* control sequence is entered or when the `stop` method is called from the calling Ability.

For instance, let two threaded Abilities *ABC* and *XYZ*, defined as:

```
class Ability(ThreadedAbilityBase):
    _info = AbilityInfo(name='ABC', type=AbilityType.STANDALONE)
    _dependencies = [('xyz', 'MyPkg', 'XYZ')]
    def main(self):
        print "Getting instance of XYZ"
        xyz_instance = self.get_dependency('xyz')

        print "Starting XYZ"
        xyz_instance.start()
        print "Control is immediately given back here, because XYZ is threaded"

        print "Let's now wait for the stop signal from a Ctrl+C"
        self._wait()

        print "Ctrl+C received, let's propagate the stop signal to our dependencies"
        xyz_instance.stop()

        print "Let's now wait for the dependency to terminate"
        xyz_instance.join()

        return 0

class Ability(ThreadedAbilityBase):
    _info = AbilityInfo(name='XYZ', type=AbilityType.COMPONENT)
    def main(self):
        print "Started, let's wait for the stop signal"

        self._wait()

        print "Stop signal received. Let's add some delay"
        time.sleep(10)
        return 0
```

The ABC example Ability gets an instance of the XYZ Ability. It starts it, waits from a stop signal, propagates that stop signal to its instance of XYZ, waits for it to exit, and finally exits itself.

Note: The `_wait` method is implemented using condition variables, so that it puts the thread to sleep without having a busy loop to check for the stop signal.

Note: To emulate `ThreadedAbilityBase` subclasses, classes inheriting from `AbilityBase` also implements a `stop` and a `join`.

Warning: Ability developers should always call `stop` and `join` on Abilities object that they get an instance of. Even though PacketWeaver implements a sort of reaper that cleans up incorrectly handled ThreadedAbilityBase subclasses, one should always clean after themself.

When developing an Ability that subclasses `ThreadedAbilityBase`, the “parent” Ability may send a stop signal at any moment. While it is possible to forcefully terminate a thread in Python, PacketWeaver Abilities should be polite and responsive to stop signals. As such, long-blocking syscalls should be avoided and as well as infinite loops. One should regularly check if the signal stop was sent by calling `self.is_stopped()`, which returns `True` if the current Ability should exit as quickly as possible.

Obtaining Results

Ability `main` method may return a value. When a standalone Ability run from the interactive CLI returns a result, the string representation of this value is printed on console. When a standalone or a component Ability returns a value, the `result` method may be called after the `join` method returns.

Let a component Ability be defined as:

```
import random
from packetweaver.core.ns import *
class Ability(AbilityBase):
    _info = AbilityInfo(name='DoSmth', type=AbilityType.COMPONENT)
    def main(self):
        return random.randint(0, 10)
```

The returned value may be obtained this way:

```
from packetweaver.core.ns import *
class Ability(AbilityBase):
    _info = AbilityInfo(name='main ability')
    _dependencies = [('smth', 'demo', 'DoSmth')]
    def main(self):
        inst = self.get_dependency('smth')
        inst.start()
        inst.stop()
        inst.join()
        # Now that join returned, it is safe to call result()
        self._view.success(inst.result())
```

Starting, Waiting and Stopping Multiple Abilities

A helper method exists if you need to start a bunch of Abilities object, wait for the stop signal, then propagate that stop signal to all those abilities.

This helper, called `_start_wait_and_stop`, is a method of any `AbilityBase` subclass instance. It receives a list of `AbilityBase` subclass instances:

```
inst1 = self.get_dependency('example', port=8080)
inst2 = self.get_dependency('example', port=8081)
self._start_wait_and_stop([inst1, inst2)
```

If more flexibility is needed, a `_start_many` and a `_stop_many` method are also available.

3.2.2 On the use of third-party libraries

Simple Abilities are self-contained and rely on the standard Python library. You may, however, need to write some that import third-party libraries and these third-party libraries may not be installed on every system.

The try and forgive approach of Python means the Python module containing your Ability must try to import the third-party libraries and an exception will be raised if a library is unavailable. While we could live with an ImportError exception bubbling up and killing PacketWeaver, we found that this is suboptimal and not very user-friendly.

The traditional way of handling this situation in PacketWeaver is to try to import the library, and set a boolean to `True` on success and `False` on failure:

```
try:
    import third_party_lib
    HAS_THIRD_PARTY_LIB = True
except ImportError:
    HAS_THIRD_PARTY_LIB = False
```

This boolean may then be used in a special PacketWeaver class method called `check_preconditions`. This class method purpose is to check for the availability of all prerequisites for the current Ability to work. If something is missing, this method must return a list of strings explaining in a user-friendly way, what is broken and what needs fixing. If all preconditions are met, an empty list must be returned. This list is notably used by the interactive CLI to display Abilities that cannot be run in red to indicate that some requirements are unmet.

Here follows an example of such a `check_preconditions` class method:

```
class Ability(...):

   @classmethod
    def check_preconditions(cls, module_factory):
        l = []
        if not HAS_THIRD_PARTY_LIB:
            l.append('Third party library XYZ support missing or broken.')
        l += super(Ability, cls).check_preconditions(module_factory)
        return l
```

As you can see, in this example, the class method does what is needed regarding the current Ability, and then calls the super class method. This super class method will work recursively across all nested Abilities that your Ability may depend on. Thus, if any Ability that your current Ability relies on has a missing dependency, the appropriate error messages will be displayed. It is strongly advised to always perform this super call when you override `check_preconditions`.

3.3 Flow-based Programming and Ability writing

Abilities might be run in parallel, using `ThreadedAbilityBase` subclasses. Yet, they run unbeknownst to each other, save for the parent Ability that know them all.

PacketWeaver offers syntactic sugars for you to write Abilities that communicate with each others, improving on the classic shell pipe syntax.

3.3.1 Doing it all by hand

You may add by hand input and output pipes to any ThreadedAbilityBase subclass instance. These pipes are `multiprocessing.Pipe` instances. As such, adding such pipes may be done like this:

```
inst = self.get_dependency('example')
outputp, inputp = multiprocessing.Pipe()
outputp2, inputp2 = multiprocessing.Pipe()
inst.add_in_pipe(inputp)
inst.add_out_pipe(outputp2)

self._start_wait_and_stop([inst])
```

Once set up like this, the parent Ability may write into `outputp` and read from `inputp2`.

Warning: you cannot add pipes to an already started Ability hitherto.

3.3.2 Reading and writing from an Ability

Reading

For the point of view of the Ability refered to by the pet name *example*, input data may be read from the *standard input* by calling the built-in `_recv()` instance method:

```
def main(self):
   p = self._recv()
```

The `_recv` method consumes the input data as datagrams, not streams. Moreover, you may receive all types of pickable data, which means that in the previous example, `p` might be a full-fledged Python object!

Warning: the `_recv` method is blocking, kernel-wise. This is a problem because well-written Abilities must keep aware of the stop signal. For this reason, the standard way of reading the *standard input* of an Ability is to write a code similar to this one:

```
def main(self):
    try:
        while not self._is_stopped():
            if self._poll(0.1):
                p = self._recv()
                # Do something with p
    except (EOFError, IOError):
        pass
```

The `_poll` method is similar to the Kernel poll syscall. It monitors whether there is a datagram to be read on the standard input, and it times out after a certain delay (100 miliseconds in the previous example). The previous example code thus ensures that this Ability checks at least every 100 ms that a stop signal was received.

3.3.3 Writing

Writing to the *standard output* of an Ability is relatively simple. You may simply call the `_send` method with any Pickable Python object as argument:

```
def main(self):
    self._send('abc')
```

Caution: `_send` might be blocking at times, which is in violations of the code of conduct of well-written Abilities… This is a known limitation.

3.3.4 Using the Pipe Syntactic Sugar

Whenever an Ability purpose is to orchestrate multiple Abilities, it owns the references to multiple Abilities objects:

```
inst1 = self.get_dependency('example1')
inst2 = self.get_dependency('example2')
inst3 = self.get_dependency('example3')
```

Configuring the pipes between these Abilities by hand might be cumbersome. For this reason, PacketWeaver ships a syntactic sugar similar to shell pipes.

To pipe the *standard output* of the first Ability to the standard input of the second Ability, one can write the following Python expression:

```
inst1 | inst2
```

We use the fluent design pattern, so that one may write:

```
inst1 | inst2 | inst3 | inst1
```

Note: In the previous example, `inst1` is listed twice into the pipeline. This is the same instance of the same Ability. This line means that the *standard output* of `inst1` is piped into `inst2`, and that the *standard output* of `inst3` is piped into the *standard input* of `inst1`. This enables developers to write pipelines of Abilities that are cyclic in a much easier way that it is generally possible in shell (e.g. *netcat* pipelines with named FIFOs)

3.3.5 Multiple Inputs and Outputs

While in shell, it is possible to pipe multiple scripts output into a script standard input, it requires some tricks, for instance using named FIFOs. With Packetweaver, multiple inputs and multiple outputs are seemless:

```
def main(self):
    inst1 = self.get_dependency('example1')
    inst2 = self.get_dependency('example2')
    inst3 = self.get_dependency('example3')
    inst4 = self.get_dependency('example4')
    inst5 = self.get_dependency('example5')

    inst1 | inst2 | inst3
    inst4 | inst2 | inst5

    self._start_wait_and_stop([inst1, inst2, inst3, inst4, inst5])
```

In the previous example, `inst2` is part of two pipeline declarations. However Packetweaver interprets this as: `inst2` standard input is composed of a **round robin read** from `inst1` and `inst4`, and `inst2` standard output is broadcast to `inst3` and `inst5`.

3.3.6 On Detecting Source and Sink Conditions

An Ability may dynamically discover if there are other Abilities that are piped to its standard input by calling `self._is_source()`. If `True`, this Ability has currently no input pipes.

Similarly, an Ability may discover if other Abilities subscribed to its standard output by calling `self._is_sink()`.

3.3.7 Transfering Pipes

Sometimes, the sole purpose of some component Abilities is to instanciate other component Abilities and set up the pipelines. If that orchestrating Ability has standard input and standard output pipes, it would be cumbersome to transfer by hand all input and output messages to the pipeline. For this reason, Packetweaver enables an Ability developer to specify that all input or output pipes are to be transfered from the current Ability to another Ability. This is performed using the `self._transfer_in(otherAbilityInstance)` and `self._transfer_out(otherAbilityInstance)` methods.

The following Ability sets up such a pipeline:

```
def main(self):
    inst1 = self.get_dependency('example1')
    inst2 = self.get_dependency('example2')
    inst3 = self.get_dependency('example3')

    inst1 | inst2 | inst3

    self._transfer_in(inst1)
    self._transfer_out(inst3)

    self._start_wait_and_stop([inst1, inst2, inst3])
```

CHAPTER 4

Contribute

4.1 Documentation

All the documentation is available in the *packetweaver/doc* folder. Written in ReStructuredText, you may build it locally using Sphinx.

If you wish to contribute to the documentation, either by editing the Python *docstring* in the source code or the documentation pages themselves, this a convenient way to visualize what your modifications will look like.

4.1.1 Documentation toolset installation

To install the documentation toolset, you may run the following commands:

```
# pip3 install sphinx pylint
# apt install graphviz
```

Sphinx is the main library that builds the documentation, and *pylint* is used for its library *pyreverse* that draw UML graphs from Python source code. Graphviz is installed to give pyreverse access to the png export format.

If you want to use the ReadTheDocs theme, you may install it using:

```
pip3 install sphinx_rtd_theme
```

It will be automatically used by the Sphinx configuration file *conf.py*.

The different actions to build the documentation are part of the Sphinx Makefile.

To use it, you may browse to the documentation folder (`cd packetweaver/doc/`) and use one of the `make <target>` commands.

Warning: As some Makefile targets use relative paths, please always use it while being in the same directory.

4.1.2 Building the API documentation

Sphinx apidoc is used to build a documentation of the source code files using their docstring content. To generate the corresponding RestructuredText documentation files, you may run:

```
make apidoc
```

It will automatically delete the previous built file and generate new ones from scratch.

4.1.3 Building the complete documentation

To build only the basic documentation content and view it with your default web browser, you may use:

```
make html
make show
```

html is a Sphinx target that specifies an output format. You may use other options such as the single page html version of it with:

```
make singlehtml
xdg-open build/singlehtml/index.html
```

4.1.4 Building UML diagrams

You may build a UML representation of PacketWeaver core module using pyreverse. Similarly, the Makefile target is:

```
make uml-core
xdg-open build/classes_pw_core.png
xdg-open build/packages_pw_core.png
```

4.1.5 Building and cleaning everything

Finally, the *all* target performs all the previous tasks:

```
make clean
make all
make show
```

The *clean* target will clear everything *except* the files generated by the apidoc target, which lives in the *source/* folder. To fully clean the *doc/* folder, you could run:

```
make clean
rm source/apidoc/*
```

CHAPTER 5

Blog and changelog

5.1 Release v0.3

This release is identified by the “v0.3” git tag.

Main changes: * Python3.6+ is the only version supported by the framework * Ubuntu 18.04 is taken as the reference GNU/Linux distribution for testing * Vi is now the default editor in pw.ini * Tox is used to run unit tests and pep8 compliance tests

New features: * A Dockerfile has been added * “ls” can be used as an alias of the “option” command

API evolution: * A basic logging system has been added to track the framework internal activity * A _start_many() and _stop_many() methods has been added to Ability class * A “get_local_net_conf” method is available to get the local machine network configuration

5.2 Release v0.2

This release is identified by the “v0.2” git tag.

New features: * Paths in interactive shell and abilities are now relative to the ability package * Add of generic example abilities * New commands at the ability selection CLI:

- “conf” can be used to edit on place the pw.ini
- “editor” can display source code of abilities, even if they cannot be selected (missing dependency, coding mistakes)

API evolution: * Add possibility to set several parameters at once when configuring nested abilities * Text coloring API can be call by log severity name (‘success’, ‘error’. . .)

Fixes and minor improvements: * Display warnings for invalid configuration when:

- a package does not provide a proper “exported_ablities” module
- if two abilities have the same name

- pw.ini parameters are invalid
- Enhance the interactive CLI “info” command by displaying references/authors information in list
- Display stack trace when an ability crashes due to an non identified error
- Silent exit of the framework

5.3 Conferences

- PacketWeaver will be presented at the 2017 instance of the french SSTIC conference.

CHAPTER 6

Indices and tables

The following links provide you with a set of tools to browse the source code documentation.

- genindex
- modindex
- search