PyQoS Documentation

Release 0.2.0 beta

Anthony Ruhier

Jul 25, 2019

Contents

CHAPTER 1

User's Guide

1.1 Quickstart

PyQoS allows you to setup a simple or a complex group of QoS rules, by allowing to add some intelligences in their relations, but in trying to keep a clear syntax (making it usable by non python developers). You can see it as a wrapper for the tool tc, that is used as a backend for now.

Even if PyQoS helps you to set your rules, it requires you to have some knowledge about how the QoS works (in high level) on Linux, and how to use tc. This documentation will not explain in detail each algorithm, and it will be a lot easier to debug if you understand what PyQoS does in the backend.

Table of Contents

1.1.1 A minimal application

A more split design will certainly be preferred, but to understand the process, this part will be focused on an application written in only one source file.

First you can create a `PyQoS` object, which is used as the application's base. This step is optional, but it will handle all the process of applying each rule and brings a built-in helper so you do not have to rewrite it. Then, bring to this application object a configuration, which will mainly be the network interfaces definition.

Once created, the QoS rules can be defined, by using the models in `pyqos.algorithms`. Then it can be attached to the application:

```
from pyqos import PyQoS
from pyqos.algorithms.htb import RootHTBClass, HTBFilterFQCodel

app = PyQoS()
app.config["INTERFACES"] = {
    "public_if": {  # The key will be used as an alias for your interface
        "name": "eth0",  # real interface name
        "if_speed": 1048576,  # interface speed, in kbits (here, 1Gbps)
        "speed": 5000,  # Upload speed, for trafic to the Internet
    },

    "lan_if": {
        "name": "eth1",
        "if_speed": 1048576,
        "speed": 100000,
    },
}


class GenericRootHTB(RootHTBClass):
    """
    Generic root htb
    """
    default = 1500  # Default mark


class HTBChildExample(HTBFilterFQCodel):
    """
    Example that will match on packets that have the mark 200
    """
    id = 200
    prio = 20
    mark = id
    rate = 20000
    ceil = 100000
    burst = rate * 1.5
    cburst = 1.5 * rate/8 + burst


# Add the 2 rules for each interface defined in the app configuration
for ifname, val in app.config["INTERFACES"].items():
    root_class = RootHTBClass(
        interface=val["name"], rate=val["speed"],
        burst=val["speed"]/8
    )
    root_class.add_child(HTBChildExample())
    app.run_list.append(root_class)

if __name__ == '__main__':
    app.run()  # if you want a built-in argparser
    # app.apply_qos()  # if you want to directly apply the rules linked
```

You can test it by running `python3 qos_rules.py -D start`, which will trigger the dry-run mode, and prints the tc commands that would normally be applied.

For more informations about each algorithm models, you should read the *API documentation*.

1.1.2 Split the configuration in its own file

The configuration can be split in its own source file, for example `config.py` which will be in the root of your application:

```
#!/usr/bin/env python3
# config.py

# INTERFACES
INTERFACES = {
    "public_if": {  # The key will be used as an alias for your interface
        "name": "eth0",  # real interface name
        "if_speed": 1048576,  # interface speed, in kbits (here, 1Gbps)
        "speed": 5000,  # Upload speed, for trafic to the Internet
    },

    "lan_if": {
        "name": "eth1",
        "if_speed": 1048576,
        "speed": 100000,
    },
}
```

Then import it in your app:

```
>>> import config
>>> app.config.from_object(config)
```

or:

```
>>> app.config.from_pyfile("config.py")
```

You can read *the documentation related to the configuration* for more informations.

1.1.3 Debug mode and dry-run

As PyQoS uses tc as backend, enabling the debug mode allows you to see which commands are run. You can also enable the dry run mode, that automatically enable the debug mode, to not apply the commands.

You can trigger these mods by setting your application attribute:

```
>>> app.debug = True
>>> app.dryrun = True
>>> app.run()
```

Or by setting it in your configuration file:

```
DEBUG = True
DRYRUN = True
```

And you can also do it when you are calling your program:

```
$ python3 myapp.py -d -D
```

These three methods are equivalents.

1.1.4 Parental

Classful algorithms can have a parent and children, to construct an entire branch of rules that share some attributes as the network interface, the prefix class id, etc...

Here is an example of a root HTB class, linked with an HTB class child:

```
class HTBChildExample(HTBFilterFQCodel):
    """
    Example that will match on packets that have the mark 200
    """
    id = 200
    prio = 20
    mark = id
    rate = 20000
    ceil = 100000
    burst = rate * 1.5
    cburst = 1.5 * rate/8 + burst


class GenericRootHTB(RootHTBClass):
    """
    Generic root htb
    """
    default = 1500  # Default mark

    def __init__(*args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_child(HTBChildExample())
```

In this example, it set the `parent` child's attribute to point on the parent object.

```
>>> root_class = RootHTBClass(default=1500)
>>> child_class = HTBChildExample()
>>> root_class.add_child(child_class)
>>> child_class.parent == root_class
True
>>> child_class in root_class.children
True
```

Important: To add a child, you should not append it manually to the `children` attribute, but always pass by the `add_child` function

---

Classless qdiscs also have a parent attributes, but obviously they do not have a `children` attribute.

1.2 Configuration

PyQoS bring a configuration model very similar to what does Flask (a big part of their code has been copied here), and helps you to share global variables with all your rules.

If you did not, you should read this part before continue: *Split the configuration in its own file*.

For this part, we will use this configuration file, originally named *conf.py*, as an example:

```
#!/usr/bin/env python3
# Author: Anthony Ruhier

# INTERFACES
INTERFACES = {
    "public_if": {  # network card which has the public IP
        "name": "eth0",  # real interface name
        "if_speed": 1048576,  # interface speed, in kbits (here, 1Gbps)
        "speed": 5000,  # Upload speed, for trafic to the Internet
    },

    "lan_if": {  # network card for the LAN subnets
        "name": "eth1",  # real interface name
        "if_speed": 1048576,  # interface speed, in kbits (here, 1Gbps)
        "speed": 100000,  # Download speed, for trafic from the Internet
    },

    "GROUP_EXAMPLE": {  # example of a group of interfaces
        "tap0": {
            "name": "tap0",  # real interface name
            "speed": 10000,  # Speed for trafic from the Internet
        },
        "tap1": {
            "name": "tap1",  # real interface name
        },
    },
}

# If we want to get the speed of tap1 equaled to 40% of the lan_if speed
INTERFACES["GROUP_EXEMPLE"]["tap1"]["speed"] = (
    INTERFACES["public_if"]["speed"] * 0.4)

DEBUG = False
DRYRUN = False
```

1.2.1 Default values

You are free to add any variables you want and need in your configuration, but some are used by the PyQoS application. In case you do not use the built-in application and decide to apply your QoS rules by yourself, you can ignore this part.

A default configuration is already defined in *PyQoS.app*:

```
DEBUG = False
DRYRUN = False
INTERFACES = {}
```

Debug and dry-run

`DEBUG` and `DRYRUN` are detailed *here*.

Interfaces

`INTERFACES` allows you to define different characteristics about the network interface you have. It is used by *PyQoS.app* during the application start, because it will reset any QoS on the defined interfaces before applying the

new ones.

Each item in `INTERFACES` has to be a dictionary, containing at least one key `name` whose the value corresponds to the interface's real name. You can see the keys in `INTERFACES` as aliases for your interfaces, which then target to their real informations.

You can also define a group of interfaces like this:

```
"GROUP_EXAMPLE": {  # example of a group of interfaces
    "eth0": {
        "name": "eth0",  # real interface name
        "speed": 10000,  # Speed for trafic from the Internet
    },
    "eth1": {
        "name": "eth1",  # real interface name
        "speed": 10000,  # Speed for trafic from the Internet
    },
},
```

And if you need to add a bit of intelligence in your configuration, like a speed of a virtual tunnel that depends on an other interface's speed, you can easily do it after the `INTERFACES` definition:

```
INTERFACES["GROUP_EXEMPLE"]["tap1"]["speed"] = (
    INTERFACES["public_if"]["speed"] * 0.4
)
```

1.2.2 Custom variables

Of course you are not limited to the default variables, and are free to add which variable you need. For example, if you would like to standardize the HTTP packets' rate, you can declare in your configuration:

```
HTTP_RATE = (40, 1000,)  # rate is 40% of the parent's one, with a minimum
                         # of 1000kbps
```

And use it in your rules:

```
from pyqos.algorithms.htb import HTBFilterFQCodel
from myrules import app

random_htb_class = HTBFilterFQCodel()
random_htb_class.rate = app.config["HTTP_RATE"]
```

1.3 Tutorial

To propose you a way to write your QoS rules with PyQoS, this tutorial will explain how to come to the example, what you can find in the directory `example` at the root of the repository.

1.3.1 Part 1: Create the skeleton of your application

Before starting, make sure you have installed the framework first. Then, wherever you want, create a folder that will contain your application. Here, like in the repository, we will name this folder `example`.

Table of Contents

Defining a configuration

Before starting, you should create a configuration for your app. The one in `example` should not match to your setup, but is more here to show the different possibilities it offers.

A more classic configuration might be:

```
INTERFACES = {
    "public_if": {  # network card which has the public IP
        "name": "eth0",  # real interface name
        "if_speed": 1048576,  # interface speed, in kbits (here, 1Gbps)
        "speed": 5000,  # Upload speed, for trafic to the Internet
    },

    "lan_if": {  # network card for the LAN subnets
        "name": "eth1",  # real interface name
        "if_speed": 1048576,  # interface speed, in kbits (here, 1Gbps)
        "speed": 100000,  # Download speed, for trafic from the Internet
    },
}

DEBUG = False
DRYRUN = False
```

`if_speed` is not obligatory, however it can be useful if you want to combine an inter-vlan routing with the shaping for your internet connection.

Change the interfaces name (*eth0* and *eth1*) depending of your setup. Here, `public_if` corresponds to the interface where and from the internet traffic is routed, and `lan_if` corresponds to the network interface where the LAN is. In case you have sub interfaces on `lan_if`, you can let the real interface name so all your subnets will share the same bandwidth (defined in `speed`). However, you are also shaping the intervlan routing, so you have to cheat a bit to avoid this secondary effect.

Save it as `config.py` in your application root directory.

Initialize your application

As we are going to define QoS rules in this app, I originally called `rules` the folder containing them. Maybe you feel yourself more inspired, so you are free to choose its name.

Create this `rules` folder, and create a file `__init__.py` containing:

```
from pyqos import PyQoS
import config

app = PyQoS()
app.config.from_object(config)
```

It just initializes a PyQoS application, and load the configuration file you wrote during the previous step. You can use PyQoS without using the application object and manually apply all your rules, however it embeds some easy tools to really concentrate yourself on your QoS and not on the rest, so you might want to keep it.

In order to use this app, create a file `run.py` in the root of `example`:

```
#!/usr/bin/env python3

from rules import app

if __name__ == '__main__':
    app.run()
```

Then launch `run.py`:

```
$ python3 run.py -h

usage: run.py [-h] [-d] [-D] {start,stop,show} ...

Tool to set, show or delete QoS rules on Linux

positional arguments:
  {start,stop,show}
    start             set QoS rules
    stop              remove all QoS rules
    show              show QoS rules

optional arguments:
  -h, --help          show this help message and exit
  -d, --debug         set the debug level
  -D, --dryrun        dry run
```

Stop resets all qdiscs on every interfaces declared in your configuration. *Start* first calls `stop` to be sure that any other external rule is conflicting, and then recursively calls `apply()` on every rules attached to `app`. *Show* just prints the tc statistics of all your interfaces.

1.3.2 Part 2: Defining root qdisc

This part describes how to attach a qdisc on the interface, depending of the case of QoS type you want (classful or classless).

Table of Contents

Structure

For this example, as the goal is only to shape the internet traffic, I have split the rules in two folders: `download` and `upload`. The `__init__.py` of each folder will define the root QDisc and HTB class for the interface it targets, and then add this root class to the running list of our application.

This structure has worked for me for my setups with HTB, but I do not really recommend it for classless setups. As usual, do not feel restricted by the structure given here, and feel free to adapt it.

This part will introduce the global goal of this tutorial, by showing a graph of the final QoS rules. The different steps to do it will be introduced.

CHAPTER 2

API Reference

If you are looking for the entire documentation about each class and functions.

2.1 API

For more clarity, this section will be split in 2 parts: one about the documentation of the different models directly related to the QoS definition, what concerns directly the final user. The other part, more general, will group all the documentations of the rest of the framework (the backend, the application, etc.), and will be useful if you want to understand how all the thing works and improve it.

2.1.1 QoS algorithms

Classless Queuing Disciplines

Cake

class pyqos.algorithms.classless_qdiscs.**Cake**(*bandwidth=None*, *autorate_ingress=False*, *rtt_time=None*, *rtt_preset=None*, *priority_queue_preset=None*, *flow_isolation=None*, *nat=False*, *wash=False*, *split_gso=True*, *ack_filter=False*, *ack_filter_aggressive=False*, *memlimit=None*, *fwmark=None*, *atm_ptm_compensation=None*, *overhead=None*, *mpu=None*, *overhead_preset=None*, *ingress=False*, **args*, ***kwargs*)

Cake (cake) qdisc

Complete documentation about this algorithm can be read here: https://www.bufferbloat.net/projects/codel/wiki/Cake/

parent
Parent object

ack_filter = None
Enable or disable ACK filtering, changing the priority of TCP ACK

ack_filter_aggressive = None
Enable aggressive mode for ACK filtering (useful only if ACK filter is enabled)

autorate_ingress = None
automatic compute of bandwidth. Can be used in conjunction of bandwidth to specify an initial estimate

bandwidth = None
bandwidth, in kbps. For now, does not allow a dynamic rate based on the parent one, as their is not a real purpose without allowing to handle priorities.

flow_isolation = None
flow isolation technique

fwmark = None
Mask applied on the packet firewall mark. If indicated, only packets matching this mask will be accepted by the qdisc.

ingress = None
Is the qdisc ingress. If false, is egress

memlimit = None
Memory limit, Bytes. If None, automatically computed by Cake.

mpu = None
Rounds each packet (including overhead) up to a minimum length, in Bytes

nat = None
Enable or disable NAT lookup

overhead = None
Overhead to apply to the size of each packet, in Bytes. Range is -64 to 256

overhead_preset = None
Overhead preset to apply to the size of each packet. Useless if an overhead size if given. As some preset can be repeated, can be a list of string.

priority_queue_preset = None
preset of priority queue

rtt_preset = None
rtt preset. Useless if an rtt_time is given.

rtt_time = None
rtt time, in ms

split_gso = None
Enable or disable General Segmentation Offload (GSO) splitting

wash = None
Enable or disable extra diffserv “washing”

FQCodel

class pyqos.algorithms.classless_qdiscs.**FQCodel**(*limit=None*, *flows=None*, *target=None*, *interval=None*, *codel_quantum=None*, **args*, ***kwargs*)

FQCodel (fq_codel) qdisc

parent
Parent object

codel_quantum = None
is the number of bytes used as 'deficit' in the fair queuing algorithm

flows = None
is the number of flows into which the incoming packets are classified

interval = None
is used to ensure that the measured minimum delay does not become too stale

limit = None
when this limit is reached, incoming packets are dropped

target = None
is the acceptable minimum standing/persistent queue delay

PFIFO

class pyqos.algorithms.classless_qdiscs.**PFIFO**(*id=None*, *parent=None*, *interface=None*, **args*, ***kwargs*)

PFIFO QDisc

parent
Parent object

SFQ

class pyqos.algorithms.classless_qdiscs.**SFQ**(*perturb=10*, **args*, ***kwargs*)

SFQ QDisc

parent
Parent object

perturb = None
perturb parameter for sfq

Classful Queuing Disciplines

Table of Contents

HTB

HTB is a type of QDisc which allows to set a rate and burst, with priorities between classes. You can get more informations here.

Empty HTB class

class pyqos.algorithms.htb.**EmptyHTBClass**(*id=None*, *rate=None*, *ceil=None*, *burst=None*, *cburst=None*, *quantum=None*, *prio=None*, *children=None*, **args*, ***kwargs*)

HTB that does nothing but can be used as parent for example

Can be useful to simulate, for example, a class already handled by another tool in the system.

add_child(**args*)
Add a class as children

apply(*auto_quantum=True*, *dryrun=False*)
Apply qos with current attributes

The function is recursive, so it will apply the qos of all children too.

branch_id
Id of the current branch

burst = None
Burst can be a callback or a fixed value

If _burst is an integer, its value will be returned directly. Otherwise, if it is a tuple, it will be considered as a callback.

cburst = None
Cburst can be a callback or a fixed value

If _burst is an integer, its value will be returned directly. Otherwise, if it is a tuple, it will be considered as a callback.

ceil = None
If ceil is an integer, will be used directly. Can also be a tupple to set a relative ceil, equals to a % of the parent class ceil: `(percentage, ceil_min, ceil_max)`. If the parent has no ceil defined, a relative ceil will use the parent's rate instead. The root class cannot have a relative ceil. Will be replaced by a property at init

children = None
children class which will be attached to this class

classid
Return the full_id, corresponding to "branch_id:id"

interface
Get the interface of the current branch

parent = None
parent object

prio = None
priority

quantum
Quantum value

rate = None
If rate is an integer, will be used directly. Can also be a tupple to set a relative rate, equals to a % of the parent class rate: `(percentage, rate_min, rate_max)`. The root class cannot have a relative rate. Will be replaced by a property at init

root
Get the root of the current branch

Basic HTB class

class pyqos.algorithms.htb.**HTBClass**(*id=None*, *rate=None*, *ceil=None*, *burst=None*, *cburst=None*, *quantum=None*, *prio=None*, *children=None*, **args*, ***kwargs*)

Basic HTB class

add_child(**args*)
Add a class as children

apply(*auto_quantum=True*, *dryrun=False*)
Apply qos with current attributes

The function is recursive, so it will apply the qos of all children too.

branch_id
Id of the current branch

classid
Return the full_id, corresponding to "branch_id:id"

interface
Get the interface of the current branch

quantum
Quantum value

root
Get the root of the current branch

Root HTB class

class pyqos.algorithms.htb.**HTBClass**(*id=None*, *rate=None*, *ceil=None*, *burst=None*, *cburst=None*, *quantum=None*, *prio=None*, *children=None*, **args*, ***kwargs*)

Basic HTB class

add_child(**args*)
Add a class as children

apply(*auto_quantum=True*, *dryrun=False*)
Apply qos with current attributes

The function is recursive, so it will apply the qos of all children too.

branch_id
Id of the current branch

classid
Return the full_id, corresponding to "branch_id:id"

interface
Get the interface of the current branch

quantum
Quantum value

root
Get the root of the current branch

HTB filter

class pyqos.algorithms.htb.**HTBFilter**(*mark=None*, *qdisc=None*, *qdisc_kwargs=None*, **args*, ***kwargs*)
Basic class with filtering

add_child(**args*)
Add a class as children

apply(*auto_quantum=True*, *dryrun=False*)
Apply qos with current attributes

The function is recursive, so it will apply the qos of all children too.

branch_id
Id of the current branch

classid
Return the full_id, corresponding to "branch_id:id"

interface
Get the interface of the current branch

mark = None
mark catch by the class

qdisc = None
qdisc associated. Can be a class of an already initialized qdisc.

qdisc_kwargs = {}
dict used during the construction **ONLY**, used as a kwargs to set the qdisc attributes.

quantum
Quantum value

root
Get the root of the current branch

HTB filter with Cake

class pyqos.algorithms.htb.**HTBFilterCake**(*mark=None*, *qdisc=None*, *qdisc_kwargs=None*, **args*, ***kwargs*)

Lazy wrapper to get a HTB class with a filter and a Cake qdisc already set

qdisc

alias of *pyqos.algorithms.classless_qdiscs.Cake*

HTB filter with FQCodel

class pyqos.algorithms.htb.**HTBFilterFQCodel**(*mark=None*, *qdisc=None*, *qdisc_kwargs=None*, **args*, ***kwargs*)

Lazy wrapper to get a HTB class with a filter and a FQCodel qdisc already set

qdisc

alias of *pyqos.algorithms.classless_qdiscs.FQCodel*

HTB filter with PFIFO

class pyqos.algorithms.htb.**HTBFilterPFIFO**(*mark=None*, *qdisc=None*, *qdisc_kwargs=None*, **args*, ***kwargs*)

Lazy wrapper to get a HTB class with a filter and a PFIFO qdisc already set

qdisc

alias of *pyqos.algorithms.classless_qdiscs.PFIFO*

HTB filter with SFQ

class pyqos.algorithms.htb.**HTBFilterSFQ**(*mark=None*, *qdisc=None*, *qdisc_kwargs=None*, **args*, ***kwargs*)

Lazy wrapper to get a HTB class with a filter and a SFQ qdisc already set

qdisc

alias of *pyqos.algorithms.classless_qdiscs.SFQ*

2.1.2 General documentation

App

class pyqos.**PyQoS**(*app_name='pyqos'*, *root_path=None*)

Application to simplify the initialization of the QoS rules. Inspired from the Flask project.

Usually you create a *PyQoS* instance in your main module or in the `__init__.py` file of your package like this:

```
from pyqos import PyQoS
app = PyQoS(application_name)
```

debug

set the main logger in debug mode or not

default_config = {'DEBUG': False, 'DRYRUN': False, 'INTERFACES': {}, 'LOGGER_NAME': No

configuration default values

dryrun
dryrun

init_parser()
Init argparse objects

logger
A `logging.Logger` object for this application. The default configuration is to log to stderr if the application is in debug mode. This logger can be used to (surprise) log messages. Here some examples:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

logger_name
name of the main logger

reset_qos()
Reset QoS for all configured interfaces

run_as_root()
Restart the script as root

run_list = []
list of qos object to apply at run

Backend

TC

pyqos.backend.tc.**filter**(*interface*, *action*, *prio*, *handle*, *flowid*, *parent=None*, *protocol='all'*, *dryrun=False*, **args*, ***kwargs*)
Add/change/replace/delete filter

**kwargs will be used for specific arguments, depending on the algorithm used.

Parameters

- **action** – "add", "replace", "change" or "delete"
- **interface** – target interface
- **prio** – priority
- **handle** – filter id
- **flowid** – target class
- **parent** – parent class/qdisc (default: None)
- **protocol** – protocol to filter. (default: "all")

pyqos.backend.tc.**filter_add**(*interface*, *parent*, *prio*, *handle*, *flowid*, *protocol='all'*, **args*, ***kwargs*)
Add filter

**kwargs will be used for specific arguments, depending on the algorithm used.

Parameters

- **interface** – target interface
- **parent** – parent class/qdisc

- **prio** – priority
- **handle** – filter id
- **flowid** – target class
- **protocol** – protocol to filter (default: "all")

pyqos.backend.tc.**filter_del**(*interface*, *prio*, *handle*, *flowid*, *parent=None*, *protocol='all'*, **args*, ***kwargs*)

Delete filter

**kwargs will be used for specific arguments, depending on the algorithm used.

Parameters

- **interface** – target interface
- **prio** – priority
- **handle** – filter id
- **flowid** – target class
- **parent** – parent class/qdisc (default: None)
- **protocol** – protocol to filter (default: "all")

pyqos.backend.tc.**filter_show**(*interface*, *dryrun=False*)

Show filters

Parameters **interface** – target interface

pyqos.backend.tc.**qdisc**(*interface*, *action*, *algorithm=None*, *handle=None*, *parent=None*, *stderr=None*, *dryrun=False*, *opts_args=None*, ***kwargs*)

Add/change/replace/replace qdisc

**kwargs will be used for specific arguments, depending on the algorithm used.

Parameters

- **action** – "add", "replace", "change" or "delete"
- **interface** – target interface
- **algorithm** – algorithm used for this leaf (htb, pfifo, sfq, ...)
- **handle** – handle parameter for tc (default: None)
- **parent** – if is None, the rule will be added as root. (default: None)
- **stderr** – indicates stderr to use during the tc commands execution
- **opts_args** – list of options without value, to append to the command

pyqos.backend.tc.**qdisc_add**(*interface*, *handle*, *algorithm*, *parent=None*, *opts_args=None*, ***kwargs*)

Add qdisc

**kwargs will be used for specific arguments, depending on the algorithm used.

Parameters

- **interface** – target interface
- **algorithm** – algorithm used for this leaf (htb, pfifo, sfq, ...)
- **handle** – handle parameter for tc
- **parent** – if is None, the rule will be added as root. (default: None)

- **opts_args** – list of options without value, to append to the command

pyqos.backend.tc.**qdisc_del**(*interface*, *algorithm=None*, *handle=None*, *parent=None*, **args*, ***kwargs*)

Delete qdisc

**kwargs will be used for specific arguments, depending on the algorithm used.

Parameters

- **interface** – target interface
- **algorithm** – algorithm used for this leaf (htb, pfifo, sfq, ...)
- **handle** – handle parameter for tc (default: None)
- **parent** – if is None, the rule will be added as root. (default: None)

pyqos.backend.tc.**qdisc_show**(*interface=None*, *show_format=None*, *dryrun=False*)

Show qdiscs

Parameters

- **show_format** – option "FORMAT" for tc. (default: None) "stats" -> -s "details" -> -d "raw" -> -r "pretty" -> -p "iec" -> -i
- **interface** – target interface (default: None)

pyqos.backend.tc.**qos_class**(*interface*, *action*, *parent*, *classid=None*, *algorithm='htb'*, *dryrun=False*, **args*, ***kwargs*)

Add/change/replace/replace class

**kwargs will be used for specific arguments, depending on the algorithm used. Parameters need to be in kbit. If the unit isn't indicated, add it automagically

Parameters

- **action** – "add", "replace", "change" or "delete"
- **interface** – target interface
- **parent** – parent class/qdisc
- **classid** – id for the current class (default: None)
- **algorithm** – algorithm used for this class (default: htb)

pyqos.backend.tc.**qos_class_add**(*interface*, *parent*, *classid*, *algorithm='htb'*, ***kwargs*)

Add class

**kwargs will be used for specific arguments, depending on the algorithm used. Parameters need to be in kbit. If the unit isn't indicated, add it automagically

Parameters

- **interface** – target interface
- **parent** – parent class/qdisc
- **classid** – id for the current class (default: None)
- **algorithm** – algorithm used for this class (default: htb)

pyqos.backend.tc.**qos_class_del**(*interface*, *parent*, *classid=None*, *algorithm='htb'*, ***kwargs*)

Delete class

**kwargs will be used for specific arguments, depending on the algorithm used. Parameters need to be in kbit. If the unit isn't indicated, add it automagically

Parameters

- **interface** – target interface
- **parent** – parent class/qdisc
- **classid** – id for the current class (default: None)
- **algorithm** – algorithm used for this class (default: htb)

pyqos.backend.tc.**qos_class_show**(*interface*, *show_format=None*, *dryrun=False*)

Show classes

Parameters

- **interface** – target interface
- **show_format** – option "FORMAT" for tc. (default: None) "stats" -> -s "details" -> -d "raw" -> -r "pretty" -> -p "iec" -> -i

Config

class pyqos.config.**Config**(*root_path*, *defaults=None*)

Works like a dict but can be filled directly from a python configuration file. Inspired from the Flask Config class (a part of their code has been copied here).

Only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Parameters

- **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application's `root_path`.
- **defaults** – an optional dictionary of default values

clear() → None. Remove all items from D.

copy() → a shallow copy of D

from_object(*obj*)

Updates the values from the given object. An object can be of one of the following two types:

- a string: in this case the object with that name will be imported
- an actual object reference: that object is used directly

Objects are usually either modules or classes. Just the uppercase variables in that object are stored in the config. Example usage:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with *from_pyfile()* and ideally from a location not within the package because the package might be installed system wide.

Parameters **obj** – an import name or object

from_pyfile(*filename*, *silent=False*)

Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the *from_object()* function.

Parameters

- **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class pyqos.config.**ConfigAttribute**(*name*, *get_converter=None*)
Makes an attribute forward to the config

Again, copied from the Flask project

Decorators

pyqos.decorators.**multiple_interfaces**(*f*)
Handle multiple interfaces for tc

If the parameter "interface" is a list of multiple interfaces, it will execute the function f for each interface

Exceptions

exception pyqos.exceptions.**BadAttributeValueException**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyqos.exceptions.**NoParentException**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

Tools

pyqos.tools.**get_child_qdiscid**(*classid*)
Return the id to handle for a child qdisc. By convention, it will take its parent class id

Parameters **classid** – parent class id

pyqos.tools.**get_mtu**(*ifname*)
Use socket ioctl call to get MTU size of an interface

pyqos.tools.**launch_command**(*command*, *stderr=None*, *dryrun=False*)
If the script is launched in debug mode, just prints the command. Otherwise, starts it with subprocess.call()

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

p

Index

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W