ARouteServer Documentation

Release latest

Pier Carlo Chiodi

Jul 20, 2017

Contents

A Python tool to automatically build (and test) feature-rich configurations for BGP route servers.

CHAPTER 1

How it works

1. Two YAML files provide *general policies* and *clients configurations* options:

```
cfg:
  rs_as: 999
  router_id: "192.0.2.2"
  add_path: True
  filtering:
    next_hop:
      policy: "same-as"
  blackhole_filtering:
    policy_ipv4: "rewrite-next-hop"
    ...
```

```
clients:
  - asn: 111
    ip:
    - "192.0.2.11"
    - "2001:db8:1:1::11"
    irrdb:
      as_sets:
        - "AS-AS111MAIN"
  ...
```

2. ARouteServer acquires external information to enrich them: i.e. bgpq3 for IRRDb data, PeeringDB for max-prefix limit, ...

3. Jinja2 built-in templates are used to render the final route server's configuration file.

   Currently, **BIRD** (1.6.3) and **OpenBGPD** (OpenBSD 6.0 and 6.1) are supported.

Validation and testing are performed using the built-in **live tests** framework: Docker instances are used to simulate several scenarios, and more custom scenarios can be built on the basis of the user's needs. More details on the Live tests section.

CHAPTER 2

Features

- **Path hiding** mitigation techniques (RFC7947 section 2.3.1).
- Filtering features (most enabled by default):
  - **NEXT_HOP** enforcement (strict / same AS - RFC7948 section 4.8);
  - minimum and maximum IPv4/IPv6 **prefix length**;
  - maximum **AS_PATH length**;
  - reject **invalid AS_PATHs** (containing private/invalid ASNs);
  - reject AS_PATHs containing **transit-free** ASNs;
  - **RPKI**-based filtering (RFC6811);
  - reject **bogons**;
  - prefixes and origin ASNs enforcing via **RPSL/IRRdb AS-SETs** (RFC7948 section 4.6.2);
  - **max-prefix limit** based on global or client-specific values or on **PeeringDB** data.
- **Blackhole filtering** support:
  - optional **NEXT_HOP rewriting**;
  - signalling via BGP Communities (BLACKHOLE and custom communities);
  - client-by-client control over propagation.
- Control and informative communities:
  - prefix/origin ASN present/not present in **IRRDB data**;
  - routes **RPKI** validity state;
  - do (not) announce to any / **peer** / on **RTT basis**;
  - **prepend** to any / **peer** / on **RTT basis**;
  - add **NO_EXPORT / NO_ADVERTISE** to any / **peer**;
  - custom informational BGP communities.

- Optional session features on a client-by-client basis:
  - prepend route server ASN (RFC7947 section 2.2.2.1);
  - active sessions;
  - **GTSM** (Generalized TTL Security Mechanism - RFC5082);
  - **ADD-PATH** capability (RFC7911).
- Automatic building of clients list:
  - integration with **IXP-Manager**;
  - fetch lists from **PeeringDB** records and **Euro-IX member list JSON** files.
- Related tools:
  - Invalid routes reporter, to log or report invalid routes and their reject reason.

A comprehensive list of features can be found within the comments of the distributed configuration file on GitHub or on the documentation web page.

More feature are already planned: see the Future work section for more details.

CHAPTER 3

Contents

Installation

1. Strongly suggested: install `pip` and setup a Virtualenv:

```
# on Debian/Ubuntu:
sudo apt-get install python-virtualenv

# on CentOS:
sudo yum install epel-release
sudo yum install python-pip python-virtualenv

# setup a virtualenv
mkdir -p ~/.virtualenvs/arouteserver
virtualenv ~/.virtualenvs/arouteserver
source ~/.virtualenvs/arouteserver/bin/activate
```

   More: `virtualenv` installation and usage.

2. Install the program.

   - If you plan to run built-in *Live tests* on your own or to contribute to the project, clone the GitHub repository locally and install dependencies:

```
# from within the previously created arouteserver directory
git clone https://github.com/pierky/arouteserver.git ./
export PYTHONPATH="`pwd`"
pip install -r requirements.txt
```

   - If you plan to just use the program to build configurations or to run your own live tests scenarios, you can install it using `pip`:

```
pip install arouteserver
```

3. Setup your system layout (confirmation will be asked before each action):

```
# if you installed from GitHub
export PYTHONPATH="`pwd`"
./scripts/arouteserver setup

# if you used pip
arouteserver setup
```

The program will ask you to create some directories (under `~/arouteserver` by default) and to copy some files there. These paths can be changed by editing the `arouteserver.yml` program configuration file or by using command line arguments. More information in the *configuration section*.

External programs

ARouteServer uses the following external programs:

- (mandatory) bgpq3 is used to gather information from IRRDBs.

  To install it:

```
mkdir /path/to/bgpq3/directory
cd /path/to/bgpq3/directory
git clone https://github.com/snar/bgpq3.git ./
# make and gcc packages required
./configure
make
make install
```

- (optional) Docker is used to perform *live validation* of configurations.

  To install it, please refer to its official guide.

- (optional) KVM is also used to perform *live tests* of OpenBGPD configurations on an OpenBSD virtual machine.

  To install it:

```
apt-get install qemu-kvm virtinst
```

  More details: https://wiki.debian.org/KVM

- (optional) rtrlib and bird-rtrlib-cli; indirectly ARouteServer needs these tools to load RPKI data into BIRD.

  To install them:

```
curl -o rtrlib.zip -L https://github.com/rtrlib/rtrlib/archive/v0.3.6.zip
unzip rtrlib.zip

cd rtrlib-0.3.6 && \
    cmake -D CMAKE_BUILD_TYPE=Release . && \
    make && \
    make install

curl -o bird-rtrlib-cli.zip -L https://github.com/rtrlib/bird-rtrlib-cli/archive/
↪v0.1.1.zip
unzip bird-rtrlib-cli.zip

cd bird-rtrlib-cli-0.1.1 && \
    cmake . && \
    make
```

More details: https://github.com/rtrlib/rtrlib/wiki/Installation

To configure bird-rtrlib-cli please refer to the README.

Upgrading

To upgrade the program, download the new version...

```
# if you cloned the repository from GitHub,
# from within the local repository's directory:
git pull origin master

# if you installed it with pip:
pip install --upgrade arouteserver
```

... then sync the local templates with those distributed in the new version:

```
arouteserver setup-templates
```

If local templates have been edited, make a backup of your files in order to merge your changes in the new ones later. To customize the configuration of the route server with your own options, please consider using *Site-specific custom configuration files* instead of editing the template files.

Usage

The script can be executed via command-line:

```
# if cloned from GitHub, from the repository's root directory:
export PYTHONPATH="`pwd`"
./scripts/arouteserver bird --ip-ver 4 -o /etc/bird/bird4.conf

# if installed using pip:
arouteserver bird --ip-ver 4 -o /etc/bird/bird4.conf
```

It produces the route server configuration for BIRD and saves it on `/etc/bird/bird4.conf`. To build the configuration for OpenBGPD, the `bird` sub-command must be replaced with `openbgpd`.

The `--target-version` argument can be used to set the version of the target BGP daemon for which the configuration is generated: this allows to enable features that are supported only by more recent versions of BGP speakers and that, otherwise, would produce an error.

The script exits with `0` if everything is fine or with an exit code different than zero if something wrong occurs.

It can be scheduled at regular intervals to re-build the configuration (for example to add new clients or to update IRRDB information), test it and finally to deploy it in production:

```
# The following assumes that ARouteServer runs on the
# route server itself, that is a thing that you may want
# to avoid.
arouteserver bird --ip-ver 4 -o /etc/bird/bird4.new && \
    bird -p -c /etc/bird/bird4.new && \
    cp /etc/bird/bird4.new /etc/bird/bird4.conf && \
    birdcl configure
```

Library

ARouteServer can be used as a Python library too: see LIBRARY for more details.

Textual representation

To build an HTML textual representation of route server's options and policies, the `html` command can be used:

```
arouteserver html -o /var/www/html/rs_description.html
```

This command writes an HTML page that contains a brief textual representation of route server's policies. An example can be found here.

Automatic `clients.yml` creation

Create clients.yml file from PeeringDB records

The `clients-from-peeringdb` command can be used to automatically create a `clients.yml` file on the basis of PeeringDB records. Given an IX LAN ID, it collects all the networks which are registered as route server clients on that LAN, then it builds the clients file accordingly.

If the IX LAN ID argument is not given, the script uses the IX-F database to show a list of IXPs and their PeeringDB ID; this can be used to easily search for the IXP PeeringDB ID.

```
$ arouteserver clients-from-peeringdb
Loading IX-F database... OK

Select the IXP for which the clients list must be built
Enter the text to search for (IXP name, country, city): LINX
     ID  IXP description
     18  GB, London, London Internet Exchange LON1 (LINX LON1)
    777  US, Ashburn, LINX NoVA (LINX NoVA)
    321  GB, London, London Internet Exchange LON2 (LINX LON2)

Enter the ID of the IXP you want to use to build the clients list: 18
```

Create clients.yml file from Euro-IX member list JSON file

The Euro-IX member list JSON schema defines a portable output format to export the list of members connected to an Internet Exchange. These files can be used to fetch the list of clients and their attributes (AS-SETs, max-prefix limits) and to use them to automatically build the `clients.yml` file used by ARouteServer to generate route server's configuration.

The `clients-from-euroix` command can be used for this purpose.

```
arouteserver clients-from-euroix --url <URL> <ixp_id> -o <output_file>
```

The JSON file may contain information about more than one IXP for every IX. For example, AMS-IX has 'AMS-IX', 'AMS-IX Caribbean', 'AMS-IX Hong Kong' and more. To filter only those clients which are connected to the IXP of interest an identifier (`ixp_id`) is needed. When executed without the `ixp_id` argument, the command prints the list of IXPs and VLANs reported in the JSON file; the ID can be found on this list:

```
$ arouteserver clients-from-euroix --url https://my.ams-ix.net/api/v1/members.json
IXP ID 1, short name 'AMS-IX'
 - VLAN ID 502, name 'GRX', IPv4 prefix 193.105.101.0/25, IPv6 prefix 2001:7f8:86:1::/
↪64
 - VLAN ID 504, name 'MDX', IPv4 prefix 195.60.82.128/26
 - VLAN ID 600, name 'PI'
 - VLAN ID 501, name 'ISP', IPv4 prefix 103.247.139.0/25, IPv6 prefix␣
↪2001:13c7:6004::/64
IXP ID 3, short name 'AMS-IX Caribbean'
 - VLAN ID 600, name 'PI'
 - VLAN ID 501, name 'ISP', IPv4 prefix 103.247.139.0/25, IPv6 prefix␣
↪2001:13c7:6004::/64
IXP ID 2, short name 'AMS-IX Hong Kong'
 - VLAN ID 501, name 'ISP', IPv4 prefix 103.247.139.0/25, IPv6 prefix␣
↪2001:13c7:6004::/64
...
```

Finally, the list of clients and their attributes can be fetched:

```
$ arouteserver clients-from-euroix --url https://my.ams-ix.net/api/v1/members.json 1 -
↪-vlan 502
clients:
- asn: 58453
  description: China Mobile International Limited
  ip: 193.105.101.100
- asn: 33849
  description: Comfone AG
  ip: 193.105.101.30
- asn: 8959
  description: Emirates Telecommunications Corporation (Etisalat) (GRX)
  ip: 193.105.101.22
- asn: 8959
  description: Emirates Telecommunications Corporation (Etisalat) (GRX)
  ip: 193.105.101.62
- asn: 12322
  description: Free SAS
  ip: 193.105.101.28
...
```

An example from the LONAP:

```
$ arouteserver clients-from-euroix --url https://portal.lonap.net/apiv1/member-list/
↪list 1
clients:
- asn: 42
  cfg:
    filtering:
      irrdb:
        as_sets:
        - AS-PCH
      max_prefix:
        limit_ipv4: 100
  description: Packet Clearing House AS42
  ip: 5.57.80.238
- asn: 42
  cfg:
    filtering:
      irrdb:
```

```
        as_sets:
        - AS-PCH
      max_prefix:
        limit_ipv6: 100
  description: Packet Clearing House AS42
  ip: 2001:7f8:17::2a:1
- asn: 714
  cfg:
    filtering:
      irrdb:
        as_sets:
        - AS-APPLE
      max_prefix:
        limit_ipv4: 1000
  description: Apple Europe Ltd
  ip: 5.57.81.57
...
```

To get a list of all the available options, run the `arouteserver clients-from-euroix --help` command.

Integration with IXP-Manager

Since the popular IXP-Manager allows to export the list of members in Euro-IX JSON format, this ARouteServer's command can also be used to integrate the two tools:

```
#!/bin/bash

set -e

# Setup an API key on IXP-Manager and write it below.
# https://github.com/inex/IXP-Manager/wiki/Euro-IX-Member-Data-Export#setting-up-an-
↪api-key
api_key="YOURAPIKEY"

# Adjust the URL and point it to your IXP-Manager application.
url="https://www.example.com/ixp/apiv1/member-list/list/key/$api_key"

# This is the IXP ID you want to export members from.
ixp_id=1

# Path to the clients file.
clients_file=~/ars/clients-from-ixpmanager.yml

# Build the clients file using info from IXP-Manager.
arouteserver clients-from-euroix \
        -o $clients_file \
        --url "$url" $ixp_id

# Build the route server configuration.
arouteserver bird \
        --clients $clients_file \
        --ip-ver 4 \
        -o /etc/bird/bird4.new

# Now test the new configuration and, finally,
# push it to the route server.
```

```
...
```

Live tests, development and customization

Template context data

To dump the list of variables and data that can be used inside a template, the `template-context` command can be used:

```
arouteserver template-context
```

It produces a YAML document that contains the context variables and their values as they are passed to the template engine used to build configurations.

Initialize a custom live test scenario

To setup a new live test scenario:

```
arouteserver init-scenario ~/ars_scenarios/myscenario
```

More details on *How to build custom scenarios*.

Configuration

Program configuration

ARouteServer needs the following files to read its own configuration and to determine the policies to be implemented in the route server:

- `arouteserver.yml`: the main ARouteServer configuration file; it contains options and paths to other files (templates, cache directory, external tools...). By default, ARouteServer looks for this file in `~/arouteserver` and `/etc/arouteserver`. This path can be changed using the `--cfg` command line argument. See its default content on GitHub.

  For details regarding the `rtt_getter_path` option please see RTT_GETTER.

- `general.yml`: this is the most important configuration file, where the route server's options and policies are configured. By default, it is located in the same directory of the main configuration file; its path can be set with the `cfg_general` option in `arouteserver.yml`. See its default content on GitHub.

  An automatically generated *reStructuredText* version of the file with all its options and comments can be found in the GENERAL page.

- `clients.yml`: the list of route server's clients and their options and policies. By default, it is located in the same directory of the main configuration file; its path can be set with the `cfg_clients` option in `arouteserver.yml`. See its default content on GitHub.

- `bogons.yml`: the list of bogon prefixes automatically discarded by the route server. By default, it is located in the same directory of the main configuration file; its path can be set with the `cfg_bogons` option in `arouteserver.yml`. See its default content on GitHub.

The `arouteserver setup` command can be used to setup the environment where ARouteServer is executed and to install the aforementioned files in the proper places.

Route server's configuration

Route server's general configuration and policies are outlined in the `general.yml` file.

Configuration details and options can be found within the distributed general and clients configuration files on GitHub or in the GENERAL page.

Details about some particular topics are reported below.

YAML files inclusion

YAML configuration files can contain a custom directive (`!include <filepath>`) that can be used to include other files. This can be useful, for example, when the same configuration is shared by two route servers that differ only in their router ID:

general-rs1.yml

```
cfg:
  router_id: "192.0.2.1"
  !include general-shared.yml
```

general-rs2.yml

```
cfg:
  router_id: "192.0.2.2"
  !include general-shared.yml
```

general-shared.yml

```
#cfg:
# keep the indentation level of the line where
# the !include statement is placed
  rs_as: 999
  passive: True
  gtsm: True
  filtering:
    [...]
```

Client-level options inheritance

Clients, which are configured in the `clients.yml` file, inherit most of their options from those provided in the `general.yml` file, unless their own configuration sets more specific values.

Options that are inherited by clients and that can be overwritten by their configuration are highlighted in the `general.yml` template file that is distributed with the project.

Example:

general.yml

```
cfg:
  rs_as: 999
  router_id: "192.0.2.2"
  passive: True
  gtsm: True
```

clients.yml

```
clients:
  - asn: 11
    ip: "192.0.2.11"
  - asn: 22
    ip: "192.0.2.22"
    passive: False
  - asn: 33
    ip: "192.0.2.33"
    passive: False
    gtsm: False
```

In this scenario, the route server's configuration will look like this:

- a passive session with GTSM enabled toward AS11 client;
- an active session with GTSM enabled toward AS22 client;
- an active session with GTSM disabled toward AS33 client.

IRRDBs-based filtering

The `filtering.irrdb` section of the configuration files allows to use IRRDBs information to filter or to tag routes entering the route server. Information are acquired using the external program bgpq3: installations details on *Installation* page.

One or more AS-SETs can be used to gather information about authorized origin ASNs and prefixes that a client can announce to the route server. AS-SETs can be set in the `clients.yml` file on a two levels basis:

- within the `asns` section, one or more AS-SETs can be given for each ASN of the clients configured in the rest of the file;
- for each client, one or more AS-SETs can be configured in the `cfg.filtering.irrdb` section.

To gather information from the IRRDBs, at first the script uses the AS-SETs provided in the client-level configuration; if no AS-SETs are provided there, it looks to the ASN configuration. If no AS-SETs are found in both the client and the ASN configuration, only the ASN's autnum object will be used.

Example:

clients.yml

```
asns:
  AS22:
    as_sets:
      - "AS-AS22MAIN"
  AS33:
    as_sets:
      - "AS-AS33GLOBAL"
clients:
  - asn: 11
    ip: "192.0.2.11"
    cfg:
      filtering:
        irrdb:
          as_sets:
            - "AS-AS11NETS"
  - asn: 22
    ip: "192.0.2.22"
  - asn: 33
    ip: "192.0.2.33"
    cfg:
      filtering:
        irrdb:
          as_sets:
            - "AS-AS33CUSTOMERS"
  - asn: 44
    ip: "192.0.2.44"
```

With this configuration, the following values will be used to run the bgpq3 program:

- **AS-AS11NETS** will be used for 192.0.2.11 (it's configured at client-level for that client);
- **AS-AS22MAIN** for the 192.0.2.22 client (it's inherited from the `asns`-level configuration of AS22, client's AS);
- **AS-AS33CUSTOMERS** for the 192.0.2.33 client (the `asns`-level configuration is ignored because a more specific one is given at client-level);
- **AS44** for the 192.0.2.44 client, because no AS-SETs are given at any level.

RPKI-based filtering

RPKI-based validation of routes can be configured using the general `filtering.rpki` section. RFC8097 BGP extended communities are used to mark routes on the basis of their validity state. Depending on the `reject_invalid` configuration, INVALID routes can be rejected before entering the route server or accepted for further processing from external tools or functions provided within *.local files*. INVALID routes are not propagated to clients.

- To acquire RPKI data and load them into BIRD, a couple of external tools from the rtrlib suite are used: rtrlib and bird-rtrlib-cli. One or more trusted local validating caches should be used to get and validate RPKI data before pushing them to BIRD. An overview is provided on the rtrlib GitHub wiki, where also an usage guide can be found.
- RPKI validation is not supported by OpenBGPD.

BGP Communities

BGP communities can be used for many features in the configurations built using ARouteServer: blackhole filtering, AS_PATH prepending, announcement control, various informative purposes (valid origin ASN, valid prefix, ...) and

more. All these communities are referenced by *name* (or *tag*) in the configuration files and their real values are reported only once, in the `communities` section of the `general.yml` file. For each community, values can be set for any of the three *formats*: standard (RFC1997), extended (RFC4360/RFC5668) and large (RFC8092).

Custom BGP Communities

Custom, locally significant BGP communities can also be used for informational purposes, for example to keep track of the geographical origin of a route or the nature of the relation with the announcing route server client.

Custom communities are declared once in the `general.yml` configuration file and then are referenced by clients definitions in the `clients.yml` file.

Example:

general.yml

```
cfg:
  rs_as: 6777
  router_id: "80.249.208.255"
custom_communities:
  colo_digitalrealty_ams01:
    std: "65501:1"
    lrg: "6777:65501:1"
  colo_equinix_am3:
    std: "65501:2"
    lrg: "6777:65501:2"
  colo_evoswitch:
    std: "65501:3"
    lrg: "6777:65501:3"
  member_type_peering:
    std: "65502:1"
    lrg: "6777:65502:1"
  member_type_probono:
    std: "65502:2"
    lrg: "6777:65502:2"
```

clients.yml

```
clients:
  - asn: 112
    ip: "192.0.2.112"
    cfg:
      attach_custom_communities:
      - "colo_digitalrealty_ams01"
      - "member_type_probono"
  - asn: 22
    ip: "192.0.2.22"
    passive: False
    cfg:
      attach_custom_communities:
      - "colo_equinix_am3"
      - "member_type_peering"
  - asn: 33
    ip: "192.0.2.33"
    cfg:
      attach_custom_communities:
      - "colo_evoswitch"
      - "member_type_peering"
```

Site-specific custom configuration files

Local configuration files can be used to load static site-specific snippets of configuration into the BGP daemon, bypassing the dynamic ARouteServer configuration building mechanisms. These files can be used to configure, for example, neighborship with peers which are not route server members or that require custom settings.

Local files inclusion can be enabled by a command line argument, `--use-local-files`: there are some fixed points in the configuration files generated by ARouteServer where local files can be included:

- BIRD:

  `BIRDConfigBuilder.`**`LOCAL_FILES_IDS`** **= ['header', 'header4', 'header6', 'footer', 'footer4', 'footer6', 'client', 'clie**

- OpenBGPD:

  `OpenBGPDConfigBuilder.`**`LOCAL_FILES_IDS`** **= ['header', 'pre-irrdb', 'post-irrdb', 'pre-clients', 'post-clients', 'clie**

One or more of these labels must be used as the argument's value in order to enable the relative inclusion points. For each enabled label, an *include* statement is added to the generated configuration in the point identified by the label itself. To modify the base directory, the `--local-files-dir` command line option can be used.

These files must be present on the host running the route server.

- Example, BIRD, file name "footer4.local" in "/etc/bird" directory:

```
protocol bgp RouteCollector {
  local as 999;
  neighbor 192.0.2.99 as 65535;
  rs client;
  secondary;

  import none;
  export all;
}
```

- Example, OpenBGPD, `header` and `post-clients`:

```
$ arouteserver openbgpd --use-local-files header post-clients
include "/etc/bgpd/header.local"

AS 999
router-id 192.0.2.2

[...]

group "clients" {

        neighbor 192.0.2.11 {
                [...]
        }
}

include "/etc/bgpd/post-clients.local"

[...]
```

  In the example above, the `header` and `post-clients` inclusion points are enabled and allow to insert two `include` statements into the generated configuration: one at the start of the file and one between clients declaration and filters.

- Example, OpenBGPD, `client` and `footer`:

```
$ arouteserver openbgpd --use-local-files client footer --local-files-dir /etc/
AS 999
router-id 192.0.2.2

[...]

group "clients" {

        neighbor 192.0.2.11 {
                include "/etc/client.local"
                [...]
        }

        neighbor 192.0.2.22 {
                include "/etc/client.local"
                [...]
        }
}

[...]

include "/etc/footer.local"
```

The example above uses the `client` label, that is used to add an `include` statement into every neighbor configuration. Also, the base directory is set to `/etc/`.

BIRD hooks

In BIRD, hook functions can also be used to tweak the configuration generated by ARouteServer. Hooks are enabled by the `--use-hooks` command line argument, that accepts one or more of the following hook IDs:

`BIRDConfigBuilder`**.HOOKS = ['pre_receive_from_client', 'post_receive_from_client', 'pre_announce_to_client', 'pos**

Functions with name `hook_<HOOK_ID>` must then be implemented within *.local* configuration files, in turn included using the `--use-local-files` command line argument.

Example:

```
$ arouteserver bird --ip-ver 4 --use-local-files header --use-hooks pre_
↪receive_from_client
router id 192.0.2.2;
define rs_as = 999;

log "/var/log/bird.log" all;
log syslog all;
debug protocols all;

protocol device {};

table master sorted;

include "/etc/bird/header.local";

[...]

filter receive_from_AS3333_1 {
        if !(source = RTS_BGP ) then
```

```
                reject "source != RTS_BGP - REJECTING ", net;

        if !hook_pre_receive_from_client(3333, 192.0.2.11, "AS3333_1") then
                reject "hook_pre_receive_from_client returned false -␣
↪REJECTING ", net;

        scrub_communities_in();

[...]
```

Details about hook functions can be found in the BIRD_HOOKS page.

An example (including functions' prototypes) is provided within the "examples/bird_hooks" directory (also on GitHub).

Reject policy and invalid routes tracking

Invalid routes, that is those routes that failed the validation process, can be simply discarded as they enter the route server (default behaviour) or, optionally, they can be kept for troubleshooting purposes, analysis or statistic reporting.

The `reject_policy` configuration option can be set to `tag` in order to have invalid routes tagged with a user-configurable BGP Community (`reject_reason`) whose purpose is to keep track of the reason for which they are considered to be invalid. These routes are also set with a low local-pref value (`1`) and tagged with a control BGP Community that prevents them from being exported to clients. If configured, the `rejected_route_announced_by` community is used to track the ASN of the client that announced the invalid route to the route server.

The goal of this feature is to allow the deployment of route collectors that can be used to further process invalid routes announced by clients. These route collectors can be configured using *site-specific .local files*. The InvalidRoutesReporter is an example of this kind of route collector.

The reason that brought the server to reject the route is identified using a numeric value in the last part of the BGP Community; the list of reject reasons follow:

ID	Reason
0	Special meaning: the route must be treated as rejected. *
1	Invalid AS_PATH length
2	Prefix is bogon
3	Prefix is in global blacklist
4	Invalid AFI
5	Invalid NEXT_HOP
6	Invalid left-most ASN
7	Invalid ASN in AS_PATH
8	Transit-free ASN in AS_PATH
9	Origin ASN not in IRRDB AS-SETs
10	IPv6 prefix not in global unicast space
11	Prefix is in client blacklist
12	Prefix not in IRRDB AS-SETs
13	Invalid prefix length
14	RPKI INVALID route
65535	Unknown

* This is not really a reject reason code, it only means that the route must be treated as rejected and must not be propagated to clients.

Caveats and limitations

Not all features offered by ARouteServer are supported by both BIRD and OpenBGPD. The following list of limitations is based on the currently supported versions of BIRD (1.6.3) and OpenBGPD (OpenBSD 6.0 and 6.1).

- OpenBGPD
  - Currently, **path hiding** mitigation is not implemented for OpenBGPD configurations. Only single-RIB configurations are generated.
  - **RPKI** validation is not supported by OpenBGPD.
  - **ADD-PATH** is not supported by OpenBGPD.
  - For max-prefix filtering, only the `shutdown` and the `restart` actions are supported by OpenBGPD. Restart is configured with a 15 minutes timer.
  - An issue is preventing next-hop rewriting for **IPv6 blackhole filtering** policies on OpenBGPD/OpenBSD 6.0.
  - **Large communities** are not supported by OpenBGPD 6.0: features that are configured to be offered via large communities only are ignored and not included into the generated OpenBGPD configuration.
  - OpenBGPD does not offer a way to delete **extended communities** using wildcard (`rt xxx:*`): peer-ASN-specific extended communities (such as `prepend_once_to_peer`, `do_not_announce_to_peer`) are not scrubbed from routes that leave OpenBGPD route servers and so they are propagated to the route server clients.

Depending on the features that are enabled in the `general.yml` and `clients.yml` files, compatibility issues may arise; in this case, ARouteServer logs one or more errors, which can be then acknowledged and ignored using the `--ignore-issues` command line option:

```
$ arouteserver openbgpd
ARouteServer 2017-03-23 21:39:45,955 ERROR Compatibility issue ID 'path_hiding'. The
↪'path_hiding'
general configuration parameter is set to True, but the configuration generated by␣
↪ARouteServer for
OpenBGPD does not support path-hiding mitigation techniques.
ARouteServer 2017-03-23 21:39:45,955 ERROR One or more compatibility issues have been␣
↪found.

Please check the errors reported above for more details.
To ignore those errors, use the '--ignore-issues' command line argument and list the␣
↪IDs of the
issues you want to ignore.
$ arouteserver openbgpd --ignore-issues path_hiding
AS 999
router-id 192.0.2.2

fib-update no
log updates
...
```

Examples of configurations

Default

Configurations built using the default `general.yml` and `clients.yml` files distributed with the project.

https://github.com/pierky/arouteserver/blob/master/examples/default

See the textual representation of this configuration.

Feature-rich example

Configurations built using the files provided in the `examples/rich` directory.

- GTSM and ADD-PATH are enabled by default on the route server.
- Next-hop filtering allows clients to set NEXT_HOP of any client in the same AS.
- Local networks are filtered, and also transit-free ASNs, invalid paths and prefixes/origin ASNs which are not authorized by clients' AS-SETs.
- RPKI-based route validation is enabled; INVALID routes are rejected.
- A max-prefix limit is enforced on the basis of PeeringDB information.
- Blackhole filtering is implemented with a rewrite-next-hop policy and can be triggered with BGP communities BLACKHOLE, 65534:0 and 999:666:0.
- Control communities allow selective announcement control and prepending, also on the basis of peers RTT.
- Client timers are configured using the custom, site-specific .local file.
- Informational custom BGP communities are used to tag routes from European or American clients.

https://github.com/pierky/arouteserver/blob/master/examples/rich

See the textual representation of this configuration.

BIRD hooks example

The BIRD configurations provided in this example have been generated enabling BIRD hooks:

```
$ arouteserver bird --ip-ver 4 --use-local-files header --use-hooks pre_receive_from_
↪client post_receive_from_client [...]
```

The above list of hooks passed to the `bird` command has been truncated for the sake of readability; the complete list used in this example is provided below.

The command line argument `--use-local-files` enables the `header` inclusion point, in order to add the `include "/etc/bird/header.local";` configuration statement to the BIRD configuration generated by ARouteServer.

```
define rs_as = 999;

log "/var/log/bird.log" all;
log syslog all;
debug protocols all;

protocol device {};

table master sorted;

include "/etc/bird/header.local";
...
```

This file must be present on the route server where BIRD is executed and must contain the custom functions used to implement the hooks. See the `header.local` file for the functions declaration.

List of hooks used in this example:

- pre_receive_from_client
- post_receive_from_client
- pre_announce_to_client
- post_announce_to_client
- scrub_communities_in
- scrub_communities_out
- apply_blackhole_filtering_policy
- route_can_be_announced_to
- announce_rpki_invalid_to_client

https://github.com/pierky/arouteserver/blob/master/examples/bird_hooks

Clients from Euro-IX member list JSON file

Some clients files automatically built from Euro-IX member list JSON files are reported here.

https://github.com/pierky/arouteserver/blob/master/examples/clients-from-euroix

Tools

Invalid routes reporter

This script is intended to be used as an ExaBGP process to elaborate and report/log invalid routes received by route servers that have been previously configured using the "tag" reject policy option of ARouteServer.

For more information: https://invalidroutesreporter.readthedocs.io

Live tests

Live tests are used to validate configurations built by ARouteServer and to test compliance between expected and real results.

A mix of Python unittest and Docker (and KVM too for OpenBGPD tests) allows to create scenarios where some instances of BGP speakers (the clients) connect to a route server whose configuration has been generated using this tool.

Some built-in tests are included within the project and have been used during the development of the tool; new *custom scenarios* can be easily built by users and IXP managers to test their own policies.

Example: in a configuration where blackhole filtering is enabled, an instance of a route server client (AS1) is used to announce some tagged prefixes (203.0.113.1/32) and the instances representing other clients (AS2, AS3) are queried to ensure they receive those prefixes with the expected blackhole NEXT_HOP (192.0.2.66).

```
def test_071_blackholed_prefixes_as_seen_by_enabled_clients(self):
  for inst in (self.AS2, self.AS3):
    self.receive_route(inst, "203.0.113.1/32", self.rs,
                       next_hop="192.0.2.66",
                       std_comms=["65535:666"], lrg_comms=[])
```

Travis CI log file contains the latest built-in live tests results. Since (AFAIK) OpenBGPD can't be run on Travis CI platform, the full live tests results including those run on OpenBGPD can be found on this file.

Setting up the environment to run live tests

1. To run live tests, Docker must be present on the system. Some info about its installation can be found on the *External programs* installation section.
2. In order to have instances of the route server and its clients to connect each other, a common network must be used. Live tests are expected to be run on a Docker bridge network with name `arouteserver` and subnet `192.0.2.0/24`/`2001:db8:1:1::/64`. The following command can be used to create this network:

```
network create --ipv6 --subnet=192.0.2.0/24 --subnet=2001:db8:1:1::/64␣
↪arouteserver
```

3. Route server client instances used in live tests are based on BIRD 1.6.3, as well as the BIRD-based version of the route server used in built-in live tests; the `pierky/bird:1.6.3` image is expected to be found on the local Docker repository. Build the Docker image (or pull it from Dockerhub):

```
# build the image using the Dockerfile
# from https://github.com/pierky/dockerfiles
mkdir ~/dockerfiles
cd ~/dockerfiles
curl -o Dockerfile.bird -L https://raw.githubusercontent.com/pierky/dockerfiles/
↪master/bird/1.6.3/Dockerfile
docker build -t pierky/bird:1.6.3 -f Dockerfile.bird .

# or pull it from Dockerhub
docker pull pierky/bird:1.6.3
```

If there is no plan to run tests on the OpenBGPD-based version of the route server, no further settings are needed. To run tests on the OpenBGPD-based version too, the following steps must be done as well.

OpenBGPD live-tests environment

1. To run an instance of OpenBGPD, KVM is needed. Some info about its installation can be found on the *External programs* installation section.
2. Setup and install a KVM virtual-machine running OpenBSD 6.0 or 6.1. This VM will be started and stopped many times during tests: don't use a production VM.
   - By default, the VM name must be `arouteserver_openbgpd60` or `arouteserver_openbgpd61`; this can be changed by setting the `VIRSH_DOMAINNAME` environment variable before running the tests.
   - The VM must be connected to the same Docker network created above: the commands `ip link show` and `ifconfig` can be used to determine the local network name needed when creating the VM:

```
$ ifconfig
br-2d2956ce4b64 Link encap:Ethernet  HWaddr 02:42:57:82:bc:91
  inet addr:192.0.2.1  Bcast:0.0.0.0  Mask:255.255.255.0
  inet6 addr: fe80::42:57ff:fe82:bc91/64 Scope:Link
  inet6 addr: 2001:db8:1:1::1/64 Scope:Global
  inet6 addr: fe80::1/64 Scope:Link
  UP BROADCAST MULTICAST  MTU:1500  Metric:1
  ...
```

- In order to run built-in live test scenarios, the VM must be reachable at 192.0.2.2/24 and 2001:db8:1:1::2/64.

On the following example, the virtual disk will be stored in ~/vms, the VM will be reachable by connecting to any IP address of the host via VNC, the installation disk image is expected to be found in the install60.iso file and the network name used is **br-2d2956ce4b64**:

```
sudo virsh pool-define-as --name vms_pool --type dir --target ~/vms
sudo virsh pool-start vms_pool
sudo virt-install \
  -n arouteserver_openbgpd \
  -r 512 \
  --vcpus=1 \
  --os-variant=openbsd4 \
  --accelerate \
  -v -c install60.iso \
  -w bridge:br-2d2956ce4b64 \
  --graphics vnc,listen=0.0.0.0 \
  --disk path=~/vms/arouteserver_openbgpd.qcow2,size=5,format=qcow2
```

Finally, add the current user to the libvirtd group to allow management of the VM:

```
sudo adduser `id -un` libvirtd
```

3. To interact with this VM, the live tests framework will use SSH; by default, the connection will be established using the `root` username and the local key file `~/.ssh/arouteserver`, so the VM must be configured to accept SSH connections using SSH keys:

```
mkdir /root/.ssh
cat << EOF > .ssh/authorized_keys
ssh-rsa [public_key_here] arouteserver
EOF
```

The `StrictHostKeyChecking` option is disabled via command line argument in order to allow to connect to multiple different VMs with the same IP address.

The SSH username and key file path can be changed by setting the `SSH_USERNAME` and `SSH_KEY_PATH` environment variables before running the tests.

Be sure the `bgpd` daemon and the `bgpctl` tool can be executed correctly on the OpenBSD VM: `chmod 0555 /var/www/bin/bgpctl`.

How to run built-in live tests

To run built-in live tests, the full repository must be cloned locally and the environment must be configured as reported above.

To test both the BIRD- and OpenBGPD-based route servers, run the Python unittest using `nose`:

```
# from within the repository's root
nosetests -vs tests/live_tests/
```

How it works

Each directory in `tests/live_tests/scenarios` represents a scenario: the route server configuration is stored in the usual `general.yml` and `clients.yml` files, while other BGP speaker instances (route server clients and their peers) are configured through the `ASxxx.j2` files. These files are Jinja2 templates and are expanded by the Python code at runtime. Containers' configuration files are saved in the local `var` directory and are used to mount the BGP speaker configuration file (currenly, `/etc/bird/bird.conf` for BIRD and `/etc/bgpd.conf` for OpenBGPD). The unittest code sets up a Docker network (with name `arouteserver`) used to attach instances and finally brings instances up. Regular Python unittest tests are then performed and can be used to match expectations to real results.

Details about the code behind the live tests can be found in the LIVETESTS_CODEDOC section.

Built-in scenarios

Some notes about the built-in scenarios that are provided with the program follow.

BGP communities

Communities:

fmt	do_not_announce_to_peer	announce_to_peer	do_not_announce_to_any
std	0:peer_as	999:peer_as	0:999
ext	rt:0:peer_as	rt:999:peer_as	rt:0:999
lrg	999:0:peer_as	999:999:peer_as	999:0:999

AS2

- announced prefixes:

Prefix ID	Comms	Prefix	Expected result
AS2_only_to_AS1_s	0:999, 999:1	2.0.1.0/24	only AS1 receives the prefix
AS2_only_to_AS1_e	rt:0:999, rt:999:1	2.0.2.0/24	only AS1 receives the prefix
AS2_only_to_AS1_l	999:0:999, 999:999:1	2.0.3.0/24	only AS1 receives the prefix
AS2_only_to_AS131073_s	0:999, rt:999:131073	2.0.4.0/24	only AS131073 receives the prefix
AS2_only_to_AS131073_l	999:0:999, 999:999:131073	2.0.5.0/24	only AS131073 receives the prefix
AS2_bad_cust_comm	65501:65501	2.0.6.0/24	AS1 and AS131073 receive the prefix without the cust community (scrubbed by the rs)

AS1

- configured to have its routes tagged with cust_comm1 (65501:65501, 999:65501:65501, rt:65501:65501)
- announced prefixes:

Prefix ID	Comms	Prefix	Expected result
AS1_good1		1.0.1.0/24	AS2 and AS131073 receive the prefix, tagged with the custom community

AS131073

Default configuration

A simple scenario to verify that the default `general.yml` and `clients.yml` files distributed with the program lead to a working configuration.

The files used here are links to those provided within the `config.d` directory.

Global scenario

Built to group as many tests as possible in a single scenario.

- **AS1**:

  AS-SETs:

  - AS-AS1 (1.0.0.0/8, 128.0.0.0/7)
  - AS-AS1_CUSTOMERS (101.0.0.0/16)

  clients:

  - AS1_1 (192.0.2.11, RTT 0.1 ms)
    - next-hop-self configured in AS1_1.conf
    - next_hop.policy: strict (inherited from general config)

    Originated prefixes:

Prefix ID	Prefix	AS_PATH	Expected result
AS1_good1	1.0.1.0/24		pass
AS1_good2	1.0.2.0/24		pass
bogon1	10.0.0.0/24		fail prefix_is_bogon
local1	192.0.2.0/24		fail prefix_is_in_global_blacklist
pref_len1	128.0.0.0/7		fail prefix_len_is_valid
peer_as1	128.0.0.0/8	[2, 1]	fail bgp_path.first != peer_as
invalid_asn1	128.0.0.0/9	[1, 65536 1]	fail as_path_contains_invalid_asn
aspath_len1	128.0.0.0/10	[1, 2x6]	fail bgp_path.len > 6

  - AS1_2 (192.0.2.12, RTT 5 ms)
    - NO next-hop-self in AS1_2.conf (next-hop of AS101 used for AS101_good == 101.0.1.0/24)
    - next_hop.policy: same-as (from clients config)
    - not enabled to receive blackhole requests

    Originated prefixes:

Prefix ID	Prefix	Feature	Expected result
AS1_good1	1.0.1.0/24		
AS1_good2	1.0.2.0/24		
AS1_good3	1.0.3.0/24	next_hop=AS1_1	win next_hop_is_valid_for_AS1_2 (same-as)

- **AS2**:

  AS-SETs:

  - AS-AS2 (2.0.0.0/16)
  - AS-AS2_CUSTOMERS (101.0.0.0/16)

  clients:

  - AS2 (192.0.2.21, RTT 17.3 ms)
    * next-hop-self configured in AS2.conf
    * next_hop.policy: authorized_addresses (from clients config)
    * next_hop.authorized_addresses_list: - 192.0.2.21 and 2001:db8:1:1::21, its own IP addresses - 192.0.2.22 and 2001:db8:1:1::22, IP addresses not configured as route server client

    Originated prefixes:

Prefix ID	Prefix	Feature	Expected result
AS2_good1	2.0.1.0/24		
AS2_good2	2.0.2.0/24		
AS2_blackhole1	2.0.3.1/32	announced with BLACKHOLE 65535:666 comm	propagated with only 65535:666 to AS1_1 and AS3 (AS1_2 has "announce_to_client" = False) and next-hop 192.0.2.66; NO_EXPORT also added
AS2_blackhole2	2.0.3.2/32	announced with local 65534:0 comm	as above
AS2_blackhole3	2.0.3.3/32	announced with local 65534:0:0 comm	as above
AS2_nonclient_next_hop1	2.0.4.0/24	announce with an authorized next-hop	received by other clients
AS2_nonclient_next_hop2	2.0.5.0/24	announce with an unknown next-hop	not received by other clients

- **AS3**:

  AS-SETs: none

  clients:

  - AS3 (192.0.2.31, RTT 123.8)
    * no enforcing of origin in AS-SET
    * no enforcing of prefix in AS-SET
    * ADD-PATH enabled
    * passive client-side (no passive on the route server)

    Originated prefixes:

Prefix ID	Prefix	Communities	Expected result
AS3_blacklist1	3.0.1.0/24		fail prefix_is_in_AS3_1_blacklist
AS3_cc_AS1only	3.0.2.0/24	0:999, 65501:1	seen on AS1_1/_2 only
AS3_cc_not_AS1	3.0.3.0/24	0:1	seen on AS2 only
AS3_cc_none	3.0.4.0/24	0:999	not seen
AS3_prepend1any	3.0.5.0/24	999:65501	AS_PATH 3, 3
AS3_prepend2any	3.0.6.0/24	999:65502	AS_PATH 3, 3, 3
AS3_prepend3any	3.0.7.0/24	999:65503	AS_PATH 3, 3, 3, 3
AS3_prepend1_AS1	3.0.8.0/24	65504:1	AS_PATH 3, 3 on AS1 clients
AS3_prepend2_AS2	3.0.9.0/24	65505:2	AS_PATH 3, 3, 3 on AS2 clients
AS3_prep3AS1_1any	3.0.10.0/24	65506:1 999:65501	AS_PATH 3, 3, 3, 3 on AS1 clients, 3, 3 on AS2 clients
AS3_noexport_any	3.0.11.0/24	65507:999	received by all with NO_EXPORT
AS3_noexport_AS1	3.0.12.0/24	65509:1 65506:2	(prepend x3 to AS2) received by AS1 with NO_EXPORT
Default_route	0.0.0.0/0		rejected by rs

- **AS4**:

  AS-SETs: none

  clients:

  - AS4 (192.0.2.41, RTT 600)
    - no enforcing of origin in AS-SET
    - no enforcing of prefix in AS-SET
    - RTT thresholds configured on rs: 5, 10, 15, 20, 30, 50, 100, 200, 500
    - other peers RTTs: - AS1_1: 0.1 - AS1_2: 5 - AS2: 17.3 - AS3: 123.8

  Originated prefixes:

Prefix ID	Prefix	Communities	Goal	Who receives it
AS4_rtt_1	10.1.0.0/24	0:64999 64532:15	Do not announce to any + announce to peers with RTT <= 15 ms	AS1_1, AS1_2
AS4_rtt_2	20.2.0.0/24	0:64999 64532:5	Do not announce to any + announce to peers with RTT <= 5 ms	AS1_1, AS1_2
AS4_rtt_3	30.3.0.0/24	64531:15	Do not announce to peers with RTT > 15 ms	AS1_1, AS1_2
AS4_rtt_4	40.4.0.0/24	64531:5	Do not announce to peers with RTT > 5 ms	AS1_1, AS1_2
AS4_rtt_5	50.5.0.0/24	64531:5 65501:3	Do not announce to peers with RTT > 5 ms but announce to AS3	AS1_1, AS1_2, AS3
AS4_rtt_6	60.6.0.0/24	64530:5 64531:100	Do not announce to peers with RTT <= 5 and Do not announce to peers with RTT > 100	AS2
AS4_rtt_7	70.7.1.1/32	65535:666 64531:20	BLACKHOLE request, do not announce to peers with RTT > 20	AS1_1, AS2 (AS1_2 not enabled to receive blackhole requests)
AS4_rtt_8	80.8.0.0/24	64539:100 64538:10	Prepend 3x to > 100 ms, 2x to > 10 ms	AS1_1, AS1_2, AS2 2x, AS3 3x
AS4_rtt_9	90.9.0.0/24	64536:5 64535:20 999:65501	Prepend 3x to <= 5 ms, 2x to <= 20, 1x to any	AS1_1 & AS1_2 3x, AS2 2x, AS3 1x
AS4_rtt_10	10.10.0.0/24	64537:10 rt:64538:20	Prepend 1x to > 10 ms, 2x to > 20 ms	AS1_1 & AS1_2 no prep, AS2 1x, AS3 2x

- **AS101**:

  clients:

  - Not a route server client, it only peers with AS1_1, AS1_2 and AS2 on 192.0.2.101.
  - RPKI ROAs:

ID	Prefix	Max	ASN
1	101.0.8.0/24		101
2	101.0.9.0/24		102
3	101.0.128.0/20	23	101

  Originated prefixes:

Prefix ID	Prefix	AS_PATH	Expected result
AS101_good1	101.0.1.0/24		fail next_hop_is_valid_for_AS1_2 (for the prefix announced by AS101 to AS1_2)
AS101_no_rset	101.1.0.0/24		fail prefix_is_in_AS1_1_r_set and prefix_is_in_AS2_1_r_set
AS102_no_asset	102.0.1.0/24	[101 102]	fail origin_as_in_AS1_1_as_set and origin_as_in_AS2_1_as_set
AS101_bad_std_comm	101.0.2.0/24		add 65530:0, scrubbed by rs
AS101_bad_lrg_comm	101.0.3.0/24		add 999:65530:0, scrubbed by rs
AS101_other_s_comm	101.0.4.0/24		add 888:0, NOT scrubbed by rs
AS101_other_l_comm	101.0.5.0/24		add 888:0:0, NOT scrubbed by rs
AS101_bad_good_comms	101.0.6.0/24		add 65530:1,999:65530:1,777:0,777:0:0, 65530 are scrubbed by rs, 777:** are kept
AS101_transitfree_1	101.0.7.0/24	[101 174]	fail as_path_contains_transit_free_asn
AS101_roa_valid1	101.0.8.0/24		roa check ok (roa n. 1), tagged with 64512:1 / 999:64512:1
AS101_roa_invalid1	101.0.9.0/24		roa check fail (roa n. 2, bad origin ASN), rejected
AS101_roa_badlen	101.0.128.0/24		roa check fail (roa n. 3, bad length), rejected
AS101_roa_blackhole	101.0.128.1/32		65535:666, pass because blackhole filtering request
AS101_no_ipv6_gl_unic	8000:1::/32		fail IPv6 global unicast space check

Max-prefix limits

General policy:

- limit: 4
- peering DB: True
- action: block

AS1:

- no peering DB
- no specific limits
- expected limit: 4

AS2:

- peering DB (3)
- no specific limits
- expected limit: 3

AS3:

- specific limit: 2
- expected limit: 2

Path hiding mitigation technique

AS1, AS2, AS3 and AS4 are clients of the route server. AS4 has ADD-PATH rx on.

Only one prefix is used, *AS101_pref_ok1*, announced by AS101 to AS1 and AS2:

- AS101 -> AS1, AS_PATH = [101]

- AS101 -> AS2, AS_PATH = [101 101 101 101]

The route server has the path toward AS1 as the preferred one.

AS1 announces this prefix to the rs after having added the *do not announce to AS3* and *do not announce to AS4* BGP communities.

- When mitigation is on, AS3 and AS4 receive the prefix via the sub-optimal path toward AS2.
- When mitigation is off, AS3 does not receive the prefix at all, AS4 receives it because of ADD-PATH capability.

Rich configuration example

A simple scenario to verify that the `general.yml` and `clients.yml` files distributed within the rich configuration example (`examples/rich`) lead to a working configuration.

The files used here are links to those provided within the `examples/rich` directory.

RPKI INVALID routes tagging

Mostly to test hooks and include files in a scenario where a custom configuration allows to propagate RPKI INVALID routes to some selected clients and to tag them with locally significant BGP communities.

Hooks used:

- `announce_rpki_invalid_to_client`, implemented in the `header[4|6]` include files and used to discriminate which clients should receive INVALIDs;
- `post_announce_to_client`, implemented in the `header` include file and used to convert RFC8097 extended communities into locally significant ones.
- RPKI ROAs:

ID	Prefix	Max	ASN
1	2.0.8.0/24		101
2	2.0.9.0/24		102
3	2.0.128.0/20	23	101
4	3.0.8.0/24		103
5	3.0.9.0/24		102
6	3.0.128.0/20	23	103

ID	Prefix	Max	ASN
1	3002:0:8::/48		101
2	3002:0:9::/48		102
3	3002:0:8000::/33	34	101
4	3003:0:8::/48		103
5	3003:0:9::/48		102
6	3003:0:8000::/33	34	103

- Locally significant communities:

Validity state	BGP community
VALID	64512:1
INVALID	64512:2
UNKNOWN	64512:3

- AS1, receives only

  Configured to receive INVALID routes using the hook `announce_rpki_invalid_to_client`, implemented in the local `header[4|6]` file.

- AS2:

  Configured with `reject_invalid` False.

  Annouced prefixes:

Prefix ID	Prefix	AS_PATH	Expected result and BGP community received by AS1
AS2_valid1	2.0.8.0/24, 3002:0:8::/48	2 101	roa check ok, 64512:1 on AS1 and AS4
AS2_valid2	2.0.128.0/21, 3002:0:8000::/34	2 101	roa check ok, 64512:1 on AS1 and AS4
AS2_invalid1	2.0.9.0/24, 3002:0:9::/48	2	roa check fail (roa n. 2, bad origin ASN), 64512:2 on AS1 only
AS2_badlen	2.0.128.0/24, 3002:0:8000::/35	2 101	roa check fail (roa n. 3, bad length), 64512:2 on AS1 only
AS2_unknown1	2.2.0.0/16 3002:3002::/32	2	roa check unknown, 64512:3 on AS1 and AS4

- AS3:

  Configured with `reject_invalid` True.

  Annouced prefixes:

Prefix ID	Prefix	AS_PATH	Expected result and BGP community received by AS1
AS3_valid1	3.0.8.0/24, 3003:0:8::/48	3 103	roa check ok, 64512:1 on AS1 and AS4
AS3_valid2	3.0.128.0/21, 3003:0:8000::/34	3 103	roa check ok, 64512:1 on AS1 and AS4
AS3_invalid1	3.0.9.0/24, 3003:0:9::/48	3	roa check fail (roa n. 2, bad origin ASN), rejected
AS3_badlen	3.0.128.0/24, 3003:0:8000::/35	3 103	roa check fail (roa n. 3, bad length), rejected
AS3_unknown1	3.2.0.0/16 3003:3003::/32	2	roa check unknown, 64512:3 on AS1 and AS4

- AS4, receives only with no particular configuration.

Tag prefixes/origin ASNs present/not-present in IRRDb

Built to test the `irrdb.tag_as_set` option.

Two sub-scenarios exist for this test:

1. AS-SETs are populated with origin ASNs and prefixes reported below.
2. AS-SETs are empty.

Communities:

OK / Not OK	Comm
prefix OK	64512
prefix NOT OK	64513
origin OK	64514
origin NOT OK	64515

AS2:

- allowed objects:
  - prefix: 2.0.0.0/16
  - origin: [2]

- configuration:
  - enforcing: no
  - tagging: yes

AS2 announces:

prefix	AS_PATH	prefix ok?	origin ok?	expected result 1	expected result 2
2.0.1.0/24	2	yes	yes	64512 64514	64513 64515
2.1.0.0/24	2	no	yes	64513 64514	64513 64515
2.0.2.0/24	2 3	yes	no	64512 64515	64513 64515
3.0.1.0/24	2 3	no	no	64513 64515	64513 64515

AS4:

- allowed objects:
  - prefix: 4.0.0.0/16
  - origin: 4
- configuration:
  - enforcing: origin only
  - tagging: yes

AS4 announces:

prefix	AS_PATH	prefix ok?	origin ok?	expected result 1	expected result 2
4.0.1.0/24	4	yes	yes	64512 64514	rejected
4.1.0.0/24	4	no	yes	64513 64514	rejected
4.0.2.0/24	4 3	yes	no	rejected	rejected
3.0.1.0/24	4 3	no	no	rejected	rejected

AS5:

- allowed objects:
  - prefix: 5.0.0.0/16
  - origin: 5

configuration:

- enforcing: prefix only
- tagging: yes

AS5 announces:

prefix	AS_PATH	prefix ok?	origin ok?	expected result 1	expected results 2
5.0.1.0/24	5	yes	yes	64512 64514	rejected
5.1.0.0/24	5	no	yes	rejected	rejected
5.0.2.0/24	5 3	yes	no	64512 64515	rejected
3.0.1.0/24	5 3	no	no	rejected	rejected

Reject policy: *tag*

This scenario uses the same base layout of the *global* one, with the addition of an *Invalid routes collector* that receives only the routes that have been classified as *invalid* by the route server.

All the test cases used here inherit from the `LiveScenario_TagRejectPolicy` class, that dynamically changes the `general.yml` content to reflect the use of the `tag reject_policy`: the BGP community used to mark the rejected routes and the reject reasons is `65520:x`.

BIRD and OpenBGPD are configured using *.local* files to setup the sessions with the route collector and to properly announce only the invalid routes that have been previously marked with the `reject_reason` BGP community.

How to build custom scenarios

A live test scenario skeleton is provided in the `pierky/arouteserver/tests/live_tests/skeleton` directory.

It seems to be a complex thing but actually most of the work is already done in the underlying Python classes and prepared in the skeleton.

To configure the route server and its clients, please consider that the Docker network used by the framework is on 192.0.2.0/24 and 2001:db8:1:1::/64 subnets.

1. Initialize the new scenario into a new directory:
   - using the `init-scenario` command:

```
arouteserver init-scenario ~/ars_scenarios/myscenario
```

   - manually, by cloning the provided skeleton directory:

```
mkdir -p ~/ars_scenarios/myscenario
cp pierky/arouteserver/tests/live_tests/skeleton/* ~/ars_scenarios/myscenario
```

2. Document the scenario, for example in the `README.rst` file: write down which BGP speakers are involved, how they are configured, which prefixes they announce and what the expected result should be with regards of the route server's configuration and its policies.
3. Put the `general.yml`, `clients.yml` and `bogons.yml` configuration files you want to test in the new directory.
4. Configure your scenario and write your test functions in the `base.py` file.
   - Declare the BGP speakers you want to use in the `_setup_rs_instance()` and `_setup_instances()` methods of the base class.

     classmethod `SkeletonScenario.`**`_setup_instances`**`()`
     Declare the BGP speaker instances that are used in this scenario.

     The `cls.INSTANCES` attribute is a list of all the instances that are used in this scenario. It is used to render local Jinja2 templates and to transform them into real BGP speaker configuration files.

     The `cls.RS_INSTANCE_CLASS` and `cls.CLIENT_INSTANCE_CLASS` attributes are set by the derived classes (test_XXX.py) and represent the route server class and the other BGP speakers class respectively.
     – The first argument is the instance name.
     – The second argument is the IP address that is used to run the instance. Here, the `cls.DATA` dictionary is used to lookup the real IP address to use, which is configured in the derived classes (test_XXX.py).
     – The third argument is a list of files that are mounted from the local host (where Docker is running) to the container (the BGP speaker). The list is made of pairs in the

form `(local_file, container_file)`. The `cls.build_rs_cfg` and `cls.build_other_cfg` helper functions allow to render Jinja2 templates and to obtain the path of the local output files.

For the route server, the configuration is built using ARouteServer's library on the basis of the options given in the YAML files.

For the other BGP speakers, the configuration must be provided in the Jinja2 files within the scenario directory.

Example:

```
@classmethod
def _setup_instances(cls):
    cls.INSTANCES = [
        cls._setup_rs_instance(),

        cls.CLIENT_INSTANCE_CLASS(
            "AS1",
            cls.DATA["AS1_IPAddress"],
            [
                (
                    cls.build_other_cfg("AS1.j2"),
                    "/etc/bird/bird.conf"
                )
            ]
        ),
        ...
    ]
```

- To ease writing the test functions, set instances names in the `set_instance_variables()` method.

  `SkeletonScenario.`**`set_instance_variables`**`()`
  Simply set local attributes for an easier usage later

  The argument of `self._get_instance_by_name()` must be one of the instance names used in `_setup_instances()`.

  Example:

```
def set_instance_variables(self):
    self.AS1 = self._get_instance_by_name("AS1")
    self.AS2 = self._get_instance_by_name("AS2")
    self.rs = self._get_instance_by_name("rs")
```

- Write test functions to verify that scenario's expectations are met.

  Some helper functions can be used:

  – `LiveScenario.`**`session_is_up`**`(`*inst_a*, *inst_b*`)`
  Test if a BGP session between the two instances is up.

  If a BGP session between the two instances is not up, the `TestCase.fail()` method is called and the test fails.

  Parameters
  * **`inst_a`** – the `BGPSpeakerInstance` instance where the BGP session is looked for.
  * **`inst_b`** – the `BGPSpeakerInstance` instance that *inst_a* is expected to peer with.

  Example:

```
    def test_020_sessions_up(self):
        """{}: sessions are up"""
        self.session_is_up(self.rs, self.AS1)
        self.session_is_up(self.rs, self.AS2)
```

– LiveScenario.**receive_route**(*inst*, *prefix*, *other_inst=None*, *as_path=None*, *next_hop=None*, *std_comms=None*, *lrg_comms=None*, *ext_comms=None*, *local_pref=None*, *filtered=None*, *only_best=None*, *reject_reason=None*)

Test if the BGP speaker receives the expected route(s).

If no routes matching the given criteria are found, the `TestCase.fail()` method is called and the test fails.

Parameters

- **inst** – the `BGPSpeakerInstance` instance where the routes are searched on.
- **prefix** (*str*) – the IPv4/IPv6 prefix of the routes to search for.
- **other_inst** – if given, only routes received from this `BGPSpeakerInstance` instance are considered.
- **as_path** (*str*) – if given, only routes with this AS_PATH are considered.
- **next_hop** – can be a string or a `BGPSpeakerInstance` instance; if given, only routes that have a NEXT_HOP address matching this one are considered.
- **lrg_comms, ext_comms** (*std_comms,*) – if given, only routes that carry these BGP communities are considered. Use an empty list ([]) to consider only routes with no BGP comms.
- **local_pref** (*int*) – if given, only routes with local-pref equal to this value are considered.
- **filtered** (*bool*) – if given, only routes that have been (not) filtered are considered.
- **only_best** (*bool*) – if given, only best routes are considered.
- **reject_reason** (*int*) – valid only if *filtered* is True: if given the route must be reject with this reason code. It can be also a set of codes: in this case, the route must be rejected with one of those codes.

  The list of valid codes is reported in docs/CONFIG.rst or at https://arouteserver.readthedocs.io/en/latest/CONFIG.html#reject-policy

Example:

```
    def test_030_rs_receives_AS2_prefix(self):
        """{}: rs receives AS2 prefix"""
        self.receive_route(self.rs, self.DATA["AS2_prefix1"],
                           other_inst=self.AS2, as_path="2")
```

– LiveScenario.**log_contains**(*inst*, *msg*, *instances={}*)

Test if the BGP speaker's log contains the expected message.

This only works for BGP speaker instances that support message logging: currently only BIRD.

If no log entries are found, the `TestCase.fail()` method is called and the test fails.

Parameters

- **inst** – the `BGPSpeakerInstance` instance where the expected message is searched on.
- **msg** (*str*) – the text that is expected to be found within BGP speaker's log.
- **instances** (*dict*) – a dictionary of pairs "<macro>: <BGPSpeakerInstance>" used to expand macros on the *msg* argument. Macros are expanded using the BGP speaker's specific client ID or protocol name.

Example

Given *self.rs* the instance of the route server, and *self.AS1* the instance of one of its clients, the following code expands the "{AS1}" macro using the BGP speaker specific name for the instance *self.AS1* and then looks for it within the route server's log:

```
self.log_contains(self.rs, "{AS1} bad ASN", {"AS1": self.
AS1})
```

On BIRD, "{AS1}" will be expanded using the "protocol name" that BIRD uses to identify the BGP session with AS1.

Example:

```
    def test_030_rs_rejects_bogon(self):
        """{}: rs rejects bogon prefix"""
        self.log_contains(self.rs,
                          "prefix is bogon - REJECTING {}".format(
                              self.DATA["AS2_bogon1"]))
        self.receive_route(self.rs, self.DATA["AS2_bogon1"],
                           other_inst=self.AS2, as_path="2",
                           filtered=True)
        # AS1 should not receive the bogon prefix from the route server
        with self.assertRaisesRegexp(AssertionError, "Routes not found"):
            self.receive_route(self.AS1, self.DATA["AS2_bogon1"])
```

5. Edit IP version specific and BGP speaker specific classes within the `test_XXX.py` files and set the prefix ID / real IP addresses mapping schema.

class `pierky.arouteserver.tests.live_tests.skeleton.test_bird4.`**SkeletonScenario_BIRDIPv4**

BGP speaker specific and IP version specific derived class.

This class inherits all the test functions from the base class. Here, only IP version specific attributes are set, such as the prefix IDs / real IP prefixes mapping schema.

The prefix IDs reported within the `DATA` dictionary must be used in the parent class' test functions to reference the real IP addresses/prefixes used in the scenario. Also the other BGP speakers' configuration templates must use these IDs. For an example plase see the "AS2.j2" file.

The `SHORT_DESCR` attribute can be set with a brief description of this scenario.

Example:

```
class SkeletonScenario_BIRDIPv4(SkeletonScenario):

    # Leave this to True in order to allow nose to use this class
    # to run tests.
    __test__ = True

    SHORT_DESCR = "Live test, BIRD, skeleton, IPv4"
    CONFIG_BUILDER_CLASS = BIRDConfigBuilder
    RS_INSTANCE_CLASS = BIRDInstanceIPv4
    CLIENT_INSTANCE_CLASS = BIRDInstanceIPv4
    IP_VER = 4

    DATA = {
        "rs_IPAddress":             "99.0.2.2",
        "AS1_IPAddress":            "99.0.2.11",
        "AS2_IPAddress":            "99.0.2.22",

        "AS2_prefix1":              "2.0.1.0/24",
```

```
        "AS2_bogon1":               "192.168.2.0/24"
    }
```

6. Edit (or add) the template files that, once rendered, will produce the configuration files for the other BGP speakers (route server clients) that are involved in the scenario (the skeleton includes two template files, `AS1.j2` and `AS2.j2`).

   Example:

```
router id 192.0.2.22;

# This is the path where Python classes look for
# to search BIRD's log files.
log "/var/log/bird.log" all;
log syslog all;
debug protocols all;

protocol device { }

# Prefixes announced by this BGP speaker to the route server.
#
# The Jinja2 'data' variable refers to the class 'DATA' attribute.
#
# IP prefixes are not configured directly here, only a reference
# to their ID is given in order to maintain a single configuration
# file that can be used for both the IPv4 and the IPv6 versions
# of the scenario.
protocol static own_prefixes {
        route {{ data.AS2_prefix1 }} reject;
        route {{ data.AS2_bogon1 }} reject;
}

protocol bgp the_rs {
        local as 2;
        neighbor {{ data.rs_IPAddress }} as 999;
        import all;
        export all;
        connect delay time 1;
        connect retry time 1;
}
```

7. Run the tests using `nose`:

```
nosetests -vs ~/ars_scenarios/myscenario
```

Details about the code behind the live tests can be found in the LIVETESTS_CODEDOC section.

Debugging live tests scenarios

To debug custom scenarios some utilities are provided:

- the `REUSE_INSTANCES` environment variable can be set when executing nose to avoid Docker instances to be torn down at the end of a run. When this environment variable is set, BGP speaker instances are started only the first time tests are executed, then are left up and running to allow debugging. When tests are executed again, the BGP speakers' configuration is rebuilt and reloaded. **Be careful**: this mode can be used only when running tests of the same scenario, otherwise Bad Things (tm) may happen.

Example:

```
REUSE_INSTANCES=1 nosetests -vs tests/live_tests/scenarios/global/test_bird4.py
```

- once the BGP speaker instances are up (using the `REUSE_INSTANCES` environment variable seen above), they can be queried using standard Docker commands:

```
$ # list all the running Docker instances
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
↪   STATUS              PORTS               NAMES
142f88379428        pierky/bird:1.6.3   "bird -c /etc/bird..."   18 minutes ago
↪   Up 18 minutes       179/tcp             ars_AS101
26a9ec58dcf1        pierky/bird:1.6.3   "bird -c /etc/bird..."   18 minutes ago
↪   Up 18 minutes       179/tcp             ars_AS2

$ # run 'birdcl show route' on ars_AS101
$ docker exec -it 142f88379428 birdcl show route
```

Some utilities are provided whitin the `/utils` directory to ease these tasks:

```
# execute the 'show route' command on the route server BIRD Docker instance
./utils/birdcl rs show route

# print the log of the route server
./utils/run rs cat /var/log/bird.log
```

The first argument ("rs" in the examples above) is the name of the instance as set in the `_setup_instances()` method.

- the `BUILD_ONLY` environment variable can be set to skip all the tests and only build the involved BGP speakers' configurations. Docker instances are not started in this mode.

  Example:

```
BUILD_ONLY=1 nosetests -vs tests/live_tests/scenarios/global/test_bird4.py
```

Future work

Short term

- RTT-based communities: extend support to add NO_EXPORT / NO_ADVERTISE
- Informative community with the measured RTT of the announcing peer
- New feature: CLI option to build configs based on templates/groups only and avoid client specific settings

Mid term

- New feature: group clients by AFI/ASN (OpenBGPD only)
- Split configuration in multiple files
- Doc: better documentation
- Doc: contributing section

- Doc: schema of data that can be used within J2 templates

Long term

- New feature: path-hiding mitigation technique on OpenBGPD
- New feature: routing policies based on RPSL import-via/export-via
- New feature: other BGP speakers support (GoBGP, ...)
- New feature: balance clients among *n* different configurations (for multiple processes - see Scaling BIRD Route-servers)

Contributing

CODEDOC_enrichers

LIVETESTS_CODEDOC

Change log

Note: Upgrade notes: after upgrading, run the `arouteserver setup-templates` command to sync the local templates with those distributed with the new version. More details on the Upgrading section of the documentation.

v0.9.0

- New feature: RTT-based communities to control propagation of routes on the basis of peers round trip time.
- Improvement: in conjunction with the "tag" reject policy, the `rejected_route_announced_by` BGP community can be used to track the ASN of the client that announced an invalid route to the server.
- Fix: when the "tag" reject policy is used, verify that the `reject_cause` BGP community is also set.

v0.8.1

- Fix: default user configuration path not working.

v0.8.0

- New feature: reject policy configuration option, to control how invalid routes must be treated: immediately discarded or kept for troubleshooting purposes, analysis or statistic reporting.
- New tool: invalid routes reporter.
- Fix: the following networks have been removed from the bogons.yml file: 193.239.116.0/22, 80.249.208.0/21, 164.138.24.80/29.

v0.7.0

- New feature: custom BGP communities can be configured on a client-by-client basis to tag routes entering the route server (for example, for informative purposes).
- Fix: validation of BGP communities configuration for OpenBGPD.

  Error is given if a peer-AS-specific BGP community overlaps with another community, even if the last part of the latter is a private/reserved ASN.
- Improvement: the custom `!include <filepath>` statement can be used now in YAML configuration files to include other files.

  More details here.
- Improvement: IRRDB-based filters can be configured to allow more specific prefixes (`allow_longer_prefixes` option).

v0.6.0

- OpenBGPD 6.1 support: enable large BGP communities support.
- Improvement: the `clients-from-peeringdb` command now uses the IX-F database to show a list of IXP and their PeeringDB ID.
- Improvement: enable NEXT_HOP rewriting for IPv6 blackhole filtering requests on OpenBGPD after OpenBSD 6.1 fixup.

  Related: issue #3.
- Improvement: BIRD, client-level .local file.
- Improvement: next-hop checks, the `authorized_addresses` option allows to authorize IP addresses of non-client routers for NEXT_HOP attribute of routes received from a client.

v0.5.0

- Fix: avoid the use of standard communities in the range 65535:x.
- Improvement: option to set max-prefix restart timer for OpenBGPD.
- Deleted feature: tagging of routes à la RPKI-Light has been removed.
  - The `reject_invalid` flag, that previously was on general scope only, now can be set on a client-by-client basis.
  - The `roa_valid`, `roa_invalid`, and `roa_unknown` communities no longer exist.

  Related: issue #4 on GitHub

  This **breaks backward compatibility**.
- New feature: BIRD hooks to add site-specific custom implementations.
- Improvement: BIRD local files.

  This **breaks backward compatibility**: previously, *.local, *.local4 and *.local6 files that were found in the same directory where the BIRD configuration was stored were automatically included. Now, only the header([4|6]).local and footer([4|6]).local files are included, depending on the values passed to the `--use-local-files` command line argument.

- Improvement: `setup` command and program's configuration file.

  The default path of the cache directory (*cache_dir* option) has changed: it was `/var/lib/arouteserver` and now it is `cache`, that is a directory which is relative to the *cfg_dir* option (by default, the directory where the program's configuration file is stored).

v0.4.0

- OpenBGPD support (some limitations apply).
- Add MD5 password support on clients configuration.
- The `build` command used to generate route server configurations has been removed in favor of BGP-speaker-specific sub-commands: `bird` and `openbgpd`.

v0.3.0

- New `--test-only` flag for builder commands.
- New `--clients-from-euroix` command to build the `clients.yml` file on the basis of records from an Euro-IX member list JSON file.

  This also allows the integration with IXP-Manager.
- New BGP communities: add NO_EXPORT and/or NO_ADVERTISE to any client or to specific peers.
- New option (set by default) to automatically add the NO_EXPORT community to blackhole filtering announcements.

v0.2.0

- `setup-templates` command to just sync local templates with those distributed within a new release.
- Multithreading support for tasks that acquire data from external sources (IRRDB info, PeeringDB max-prefix).

  Can be set using the `threads` option in the `arouteserver.yml` configuration file.
- New `template-context` command, useful to dump the list of context variables and data that can be used inside a template.
- New empty AS-SETs handling: if an AS-SET is empty, no errors are given but only a warning is logged and the configuration building process goes on.

  Any client with IRRDB enforcing enabled and whose AS-SET is empty will have its routes rejected by the route server.

v0.1.2

- Fix local files usage among IPv4/IPv6 processes.

  Before of this release, only *.local* files were included into the route server configuration, for both the IPv4 and IPv6 configurations. After this, *.local* files continue to be used for both the address families but *.local4* and *.local6* files can also be used to include IP version specific options, depending on the IP version used to build the configuration. Details here.

To upgrade:

```
# pull from GitHub master branch or use pip:
pip install --upgrade arouteserver

# install the new template files into local system
arouteserver setup
```

v0.1.1

- Add local static files into the route server's configuration.

v0.1.0

- First beta version.

v0.1.0a11

- The `filtering.rpsl` section of general and clients configuration files has been renamed into `filtering.irrdb`.
- The command line argument `--template-dir` has been renamed into `--templates-dir`.
- New options in the program's configuration file: `bgpq3_host` and `bgpq3_sources`, used to set bgpq3 `-h` and `-S` arguments when gathering info from IRRDBs.

v0.1.0a10

- New command to build textual representations of configurations: `html`.

v0.1.0a9

- New command to initialize a custom live test scenario: `init-scenario`.

v0.1.0a8

- New feature: selective path prepending via BGP communities.
- The `control_communities` general option has been removed: it was redundant.

v0.1.0a7

- Improved communities configuration and handling.
- Fix issue on standard communities matching against 32-bit ASNs.
- Fix issue on IPv6 prefix validation.

v0.1.0a6

- New feature: RPKI-based filtering/tagging.

v0.1.0a5

- New feature: transit-free ASNs filtering.
- Program command line: subcommands + `clients-from-peeringdb`.
- More logging and some warning.

v0.1.0a4

- Fix issue with GTSM default value.
- Add default route to bogons.
- Better as-sets handling and cache handling.
- Config syntax change: clients 'as' -> 'asn'.
- AS-SETs at AS-level.
- Live tests: path hiding mitigation scenario.
- Improvements in templates.

v0.1.0a3

- Fix some cache issues.

v0.1.0a2

- Packaging.
- System setup via `arouteserver --setup`.

v0.1.0a1

First push on GitHub.

CHAPTER 4

Presentations

- RIPE74, 10 May 2017, Connect Working Group: video (9:53), slides (PDF)
- Salottino MIX, 30 May 2017: slides

CHAPTER 5

Status

Beta testing, looking for testers and reviewers.

Anyone who wants to share his/her point of view, to review the output configurations or to test them is **more than welcome**!

CHAPTER 6

Bug? Issues?

But also suggestions? New ideas?

Please create an issue on GitHub or drop me a message.

CHAPTER 7

Author

Pier Carlo Chiodi - https://pierky.com

Blog: https://blog.pierky.com Twitter: @pierky

Index

H

L