Rotonda User Manual

NLnet Labs

Jan 22, 2024

GETTING STARTED

Warning: Rotonda is currently considered to be alpha software, and is actively developed.

Use it to experiment freely (we value your feedback!), but do not use it with data and data-streams that you cannot afford to lose.

You should also be aware that all the APIs, configuration and the `Roto` syntax and grammar are still unstable.

Not all features mentioned in the documentation in this repository are currently implemented. For more information see the ROADMAP.

CHAPTER

ONE

THE COMPOSABLE, PROGRAMMABLE BGP ENGINE

Rotonda enables you to build BGP applications such as route monitors, route collectors, route servers, route reflectors, or any variation or combination thereof. All this without modifying a single line of source code. Rotonda is and always will be free, open-source software.

Below is a brief overview of the key concepts and characteristics of Rotonda. If you want to try it out right away, consider following the *Quick Tour*, or jump directly to the *Installation Instructions*.

For more background, read *why* we made Rotonda and how we envision it can be used for different use cases.

Modular

Rotonda applications are built by combining units into a pipeline through which BGP data will flow. You can filter, modify and store the BGP data along the way, and create signals based on it to send to other applications. Units can be added and removed in a *running* Rotonda application.

Rotonda offers units to create BGP and BMP sessions, filters, Routing Information Bases (RIBs), and more.

Flexible

The behaviour of the units can be modeled by using a small, fun programming language called `Roto`, that we created to combine flexibility and ease-of-use. `Roto` lets you configure a Rotonda application, program units and create queries. `Roto` scripts can be created with your favourite text editor, but they can also be composed from the command line that's included in Rotonda.

Tailored Performance

All data structures come with a trade-off between space and time, and Rotonda is no exception. Rotonda therefore offers units that perform the same task, but with different performance characteristics, so that you can optimize for your needs, be it a high-volume, low latency installation or a small installation in a constraint environment. None of this requires patching the Rotonda source, it's all configurable with a nimble `Roto` script.

Although Rotonda is still in alpha, these performance-critical parts have been battle-tested by, and are indeed being used in, large production environments.

Observable

All Rotonda units have their own finely-grained logging capabilities, and some have built-in queryable JSON API interfaces to give information about their current state and content through Rotonda's built-in HTTPS server. Signals can be sent to other applications. Moreover, Rotonda offers true observability by allowing the user to trace BMP/BGP packets start-to-end through the whole pipeline.

Storage Persistence

By default a Rotonda application stores all the data that you want to collect in memory. It can be configured to persist parts to another storage location, such as files or a database. Whether you put RIBs to files or in a database, you can still query it transparently with `Roto`.

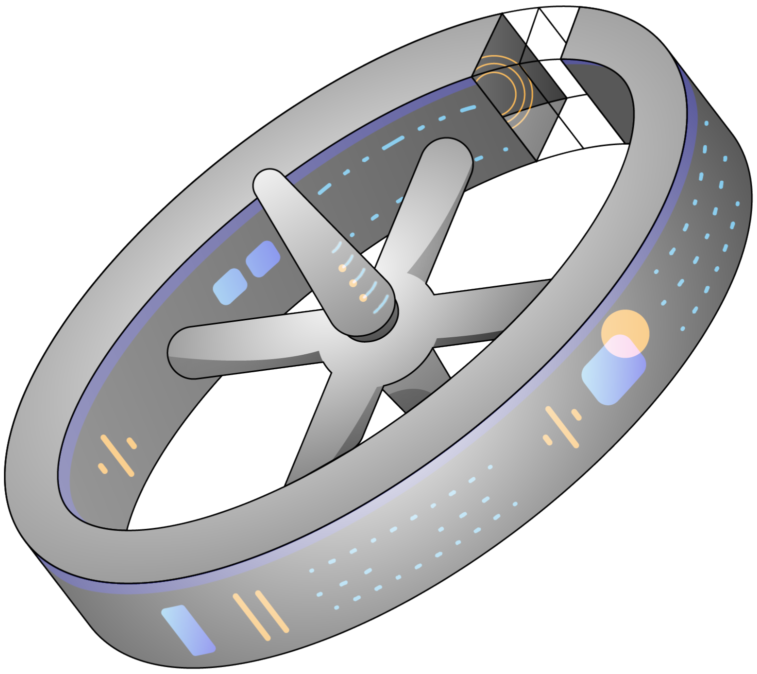
External Data Sources

`Roto` filter units can make decisions based on real-time external data sources, and RIB units can store them as needed. Similarly filter units can make decisions based on data present in multiple RIBs. External data sources can be, among others, files, databases or even a RIB backed by an RTR connection.

Robust & Scalable
Multiple Rotonda instances can synchronize or shard data via our AVRO-based `rotoro` protocol, to create robust redundancy and/or scalability. Again you can still query all the distributed instances with `Roto`.

Secure
Rotonda applications can use data provided by the RPKI through connections with tools like Routinator and Krill. Besides that, Rotonda supports BGPsec out of the box. Again, no patching or recompiling required.

Open-source with professional support services
NLnet Labs offers professional support and consultancy services with a service-level agreement. Rotonda is liberally licensed under the Mozilla Public License 2.0.



1.1 Installation

1.1.1 System Requirements

The system requirements of Rotonda depend highly on the volume of routes that you want to keep in memory, and the amount of BGP/BMP messages that flow through Rotonda. If you do not wish to keep any routes in memory (say you want to emit only MQTT messages based on incoming messages), the requirements would be fairly minimal, both in terms of memory and CPU power. If on the other hand, you want to keep all routes that you receive from multiple sessions that feed full tables, your memory requirements will surely run into the tens of gigabytes.

1.1.2 Binary Packages

Rotonda is packaged for several popular Linux distributions, and Docker images based on Alpine Linux are provided. Instructions for all these are below.

You can also build Rotonda from the source code using Cargo, Rust's build system and package manager. Cargo lets you build Rotonda on almost any operating system and CPU architecture. Refer to the *Building From Source* section to get started.

Debian

To install a Rotonda package, you need the 64-bit version of one of these Debian versions:

- Debian Bookworm 12
- Debian Bullseye 11
- Debian Buster 10
- Debian Stretch 9

Packages for the `amd64/x86_64` architecture are available for all listed versions. In addition, we offer `armhf/armv7` architecture packages for Debian/Raspbian Bullseye, and `arm64/aarch64` for Buster and Bookworm.

First update the **`apt`** package index:

```
$ sudo apt update
```

Then install packages to allow **`apt`** to use a repository over HTTPS:

```
$ sudo apt install curl gpg
```

Add the GPG key from NLnet Labs:

```
$ curl -fsSL https://packages.nlnetlabs.nl/aptkey.asc | sudo gpg --dearmor -o /usr/share/
↪keyrings/nlnetlabs-archive-keyring.gpg
```

Now, use the following command to set up the *main* repository:

```
$ echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/debian \
$(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/nlnetlabs.list > /dev/null
```

Update the **`apt`** package index once more:

```
$ sudo apt update
```

You can now install Rotonda with:

```
$ sudo apt install rotonda
```

After installation Rotonda will run immediately as the user *rotonda* and be configured to start at boot.

You can check the status of Rotonda with:

```
$ sudo systemctl status rotonda
```

You can view the logs with:

```
$ sudo journalctl --unit=rotonda
```

Ubuntu

To install a Rotonda package, you need the 64-bit version of one of these Ubuntu versions:

- Ubuntu Jammy 22.04 (LTS)
- Ubuntu Focal 20.04 (LTS)
- Ubuntu Bionic 18.04 (LTS)
- Ubuntu Xenial 16.04 (LTS)

Packages are available for the `amd64/x86_64` architecture only.

First update the **apt** package index:

```
$ sudo apt update
```

Then install packages to allow **apt** to use a repository over HTTPS:

```
$ sudo apt install \
  ca-certificates \
  curl \
  gnupg \
  lsb-release
```

Add the GPG key from NLnet Labs:

```
$ curl -fsSL https://packages.nlnetlabs.nl/aptkey.asc | sudo gpg --dearmor -o /usr/share/
↪keyrings/nlnetlabs-archive-keyring.gpg
```

Now, use the following command to set up the *main* repository:

```
$ echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/ubuntu \
$(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/nlnetlabs.list > /dev/null
```

Update the **apt** package index once more:

```
$ sudo apt update
```

You can now install Rotonda with:

```
$ sudo apt install rotonda
```

After installation Rotonda will run immediately as the user *rotonda* and be configured to start at boot.

You can check the status of Rotonda with:

```
$ sudo systemctl status rotonda
```

You can view the logs with:

```
$ sudo journalctl --unit=rotonda
```

RHEL/CentOS

To install a Rotonda package, you need Red Hat Enterprise Linux (RHEL) 7 or 8, or compatible operating system such as Rocky Linux. Packages are available for the `amd64/x86_64` architecture only.

First create a file named `/etc/yum.repos.d/nlnetlabs.repo`, enter this configuration and save it:

```
[nlnetlabs]
name=NLnet Labs
baseurl=https://packages.nlnetlabs.nl/linux/centos/$releasever/main/$basearch
enabled=1
```

Add the GPG key from NLnet Labs:

```
$ sudo rpm --import https://packages.nlnetlabs.nl/aptkey.asc
```

You can now install Rotonda with:

```
$ sudo yum install -y rotonda
```

After installation Rotonda will run immediately as the user *rotonda* and be configured to start at boot.

You can check the status of Rotonda with:

```
$ sudo systemctl status rotonda
```

You can view the logs with:

```
$ sudo journalctl --unit=rotonda
```

Docker

Rotonda Docker images are built with Alpine Linux. The supported CPU architectures are shown on the Docker Hub Rotonda page per Rotonda version (aka Docker "tag") in the `OS/ARCH` column.

1.1.3 Updating

Debian

To update an existing Rotonda installation, first update the repository using:

```
$ sudo apt update
```

You can use this command to get an overview of the available versions:

```
$ sudo apt policy rotonda
```

You can upgrade an existing Rotonda installation to the latest version using:

```
$ sudo apt --only-upgrade install rotonda
```

Ubuntu

To update an existing Rotonda installation, first update the repository using:

```
$ sudo apt update
```

You can use this command to get an overview of the available versions:

```
$ sudo apt policy rotonda
```

You can upgrade an existing Rotonda installation to the latest version using:

```
$ sudo apt --only-upgrade install rotonda
```

RHEL/CentOS

To update an existing Rotonda installation, you can use this command to get an overview of the available versions:

```
$ sudo yum --showduplicates list rotonda
```

You can update to the latest version using:

```
$ sudo yum update -y rotonda
```

Docker

Assuming that you run Docker with image *nlnetlabs/rotonda*, upgrading to the latest version can be done by running the following commands:

```
$ sudo docker pull nlnetlabs/rotonda
$ sudo docker rm --force rotonda
$ sudo docker run <your usual arguments> nlnetlabs/rotonda
```

1.1.4 Installing Specific Versions

Before every new release of Rotonda, one or more release candidates are provided for testing through every installation method. You can also install a specific version, if needed.

Debian

If you would like to try out release candidates of Rotonda you can add the *proposed* repository. This repository can live side by side wih the *main* repository.

If you have already installed the *main* repository you can skip the first three steps here, and go directly to **`Debian proposed repository`_**.

First update the **apt** package index:

```
$ sudo apt update
```

Then install packages to allow **apt** to use a repository over HTTPS:

```
$ sudo apt install curl gpg
```

Add the GPG key from NLnet Labs:

```
curl -fsSL https://packages.nlnetlabs.nl/aptkey.asc | sudo gpg --dearmor -o /usr/share/
↪keyrings/nlnetlabs-archive-keyring.gpg
```

Now, use the following command to set up the Debian proposed repository:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/debian \
```

(continues on next page)

(continued from previous page)

```
$(lsb_release -cs)-proposed main" | sudo tee /etc/apt/sources.list.d/nlnetlabs-proposed.
↪list > /dev/null
```

Make sure to update the **apt** package index:

```
$ sudo apt update
```

You can now use this command to get an overview of the available versions:

```
$ sudo apt policy rotonda
```

You can install a specific version using `<package name>=<version>`, e.g.:

```
$ sudo apt install rotonda=0.2.0~rc2-1buster
```

Ubuntu

If you would like to try out release candidates of Rotonda you can add the *proposed* repository to the existing *main* repository described earlier.

Assuming you already have followed the steps to install regular releases, run this command to add the additional repository:

```
$ echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/nlnetlabs-archive-
↪keyring.gpg] https://packages.nlnetlabs.nl/linux/ubuntu \
$(lsb_release -cs)-proposed main" | sudo tee /etc/apt/sources.list.d/nlnetlabs-proposed.
↪list > /dev/null
```

Make sure to update the **apt** package index:

```
$ sudo apt update
```

You can now use this command to get an overview of the available versions:

```
$ sudo apt policy rotonda
```

You can install a specific version using `<package name>=<version>`, e.g.:

```
$ sudo apt install rotonda=0.2.0~rc2-1bionic
```

RHEL/CentOS

To install release candidates of Rotonda, create an additional repo file named `/etc/yum.repos.d/nlnetlabs-testing.repo`, enter this configuration and save it:

```
[nlnetlabs-testing]
name=NLnet Labs Testing
baseurl=https://packages.nlnetlabs.nl/linux/centos/$releasever/proposed/$basearch
enabled=1
```

You can use this command to get an overview of the available versions:

```
$ sudo yum --showduplicates list rotonda
```

You can install a specific version using `<package name>-<version info>`, e.g.:

```
$ sudo yum install -y rotonda-0.2.0~rc2
```

Docker

All release versions of Rotonda, as well as release candidates and builds based on the latest main branch are available on Docker Hub.

For example, installing Rotonda 0.2.0 RC2 is as simple as:

```
$ sudo docker run <your usual arguments> nlnetlabs/rotonda:v0.2.0-rc2
```

1.2 Building From Source

Warning: Building from source is perfectly possible, but you have to ensure all the required configuration files to run Rotonda are in the right place. If you intend to follow the *Quick Tour*, it is recommend to use a packaged version if possible.

If you do build from source but run into problems afterwards, refer to *Introduction* or *Unable to load Roto scripts*.

In addition to meeting the *system requirements*, these are two things you need to build Rotonda:

- a C toolchain
- Rust

You can run Rotonda on any operating system and CPU architecture where you can fulfil these requirements.

1.2.1 Dependencies

Some of the libraries used by Rotonda require a C toolchain, most notably the MQTT client. You also need Rust because that's the programming language that Rotonda has been written in.

C Toolchain

Some of the libraries Rotonda depends on require a C toolchain to be present. Your system probably has some easy way to install the minimum set of packages to build from C sources. For example, this command will install everything you need on Debian/Ubuntu, provided the currently logged in user has enough privileges to install system packages:

```
$ sudo apt install curl build-essential gcc make
```

If you are unsure, try to run **cc** on a command line. If there is a complaint about missing input files, you are probably good to go.

Rust

The Rust compiler runs on, and compiles to, a great number of platforms, though not all of them are equally supported. The official Rust Platform Support page provides an overview of the various support levels.

While some system distributions include Rust as system packages, Rotonda relies on a relatively new version of Rust, currently 1.71.0 or newer. We therefore suggest to use the canonical Rust installation via a tool called **`rustup`**.

Assuming you already have **`curl`** installed, you can install **`rustup`** and Rust by simply entering:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

This will install the stable rust toolchain, on a per user basis. This means that you probably want to install this as the same user as you will be running the Rotonda application itself. This should most probably be a non-root user, so we suggest you create a *rotonda* user, log in as that user and then perform the *curl* command above.

Make sure to restart your shell, or add *$HOME/.cargo/bin* to your $PATH enviroment variable manually. This will make sure that the tools, most notably *cargo*, are reachable without specifying the full path.

Alternatively, visit the Rust website for other installation methods.

1.2.2 Building and Updating

In Rust, a library or executable program such as Rotonda is called a *crate*. Crates are published on crates.io, the Rust package registry. Cargo is the Rust package manager. It is a tool that allows Rust packages to declare their various dependencies and ensure that you'll always get a repeatable build.

Cargo fetches and builds Rotonda's dependencies into an executable binary for your platform. By default you install from crates.io, but you can for example also install from a specific Git URL, as explained below.

Installing the latest Rotonda release from crates.io is as simple as running:

```
$ cargo install rotonda --version 0.1.0 --locked
```

The command will build Rotonda and install it in the same directory that Cargo itself lives in, likely `$HOME/.cargo/bin`. This means Rotonda will be in your path, too. The version of Rotonda that was build corresponds to the version of this documentation you are reading. If for any reason you want to install another version of Rotonda, you should substitute the value after `--version` with the version you want. Omitting the whole `--version`` option will install the latest published version on `crates.io`.

Downloading the configuration files

Although Rotonda has a built-in configuration, and you can create a configuration file from scratch it's very useful to download the configuration files that come with Rotonda. These files are situated in the github repository of Rotonda. Provided you have a version of *git* higher than or equal to 2.25 installed, you can issue these commands to download them to a newly created directory, called `rotonda` in your current working directory:

```
$ git clone --no-checkout --depth 1 --branch v0.1.0 https://github.com/nlnetlabs/rotonda␣
↪&& cd rotonda/ && git sparse-checkout set etc && git checkout v0.1.0
```

Again, the version of the configuration files installed here matches with the Rotonda version you just installed, and this documentation. If you've installed another Rotonda version, you should also substitute the two version values with the version you used when installing Rotonda.

Updating

If you want to update to the latest version of Rotonda, it's recommended to update Rust itself as well, using:

```
$ rustup update
```

Use the `--force` option to overwrite an existing version with the latest Rotonda release:

```
$ cargo install --locked --force rotonda
```

Installing Rotonda from the main branch

All new features of Rotonda are built on a branch and merged via a pull request, allowing you to easily try them out using Cargo. If you want to try a specific branch from the repository you can use the `--git` and `--branch` options:

```
$ cargo install --git https://github.com/NLnetLabs/rotonda.git --branch main
```

Note that you will also have to download the correct configuration files with:

```
$ git clone --no-checkout --depth 1 --branch main https://github.com/nlnetlabs/rotonda &&
↪ cd rotonda/ && git sparse-checkout set etc && git checkout main
```

See also:

For more installation options refer to the Cargo book.

1.2.3 Platform Specific Instructions

For some platforms, **`rustup`** cannot provide binary releases to install directly. The Rust Platform Support page lists several platforms where official binary releases are not available, but Rust is still guaranteed to build. For these platforms, automated tests are not run so it's not guaranteed to produce a working build, but they often work to quite a good degree.

OpenBSD

On OpenBSD, patches are required to get Rust running correctly, but these are well maintained and offer the latest version of Rust quite quickly.

Rust can be installed on OpenBSD by running:

```
$ pkg_add rust
```

For this quick tour we assume that you have installed Rotonda via one of the packages (see *Getting Started*), putting the required configuration files in `/etc/rotonda`.

If you built Rotonda from source, you should create the directory `/etc/rotonda` yourself. You can copy the configuration need for this tour from the Rotonda GitHub repository.

We're only going to invoke the binary it installed directly in this quick tour. Most probably you can just invoke the binary without further ado, as `rotonda` on the command line. If that does not work you might have to restart your shell (to add the path to your default paths), or as a last resort figure out the full path to the binary, and use that.

It probably helps if you've read the *Why does this exist?* section and/or *Overview* section, but you can also learn on the job by following this tour, especially if you're a bit familiar with how a BGP speaker operates.

1.3 Starting a Rotonda instance

So, let's invoke the binary directly. By default the binary will run as daemon in the foreground, i.e. it will not exit unless you explicit kill by sending a SIGINT signal, normally invoked by issuing a `ctrl-c` in the shell instance that is running it. Let's try:

```
$ rotonda -c /etc/rotonda/rotonda.conf
```

Hopefully you'll see output like this:

```
Loading new Roto script /etc/filters/bmp-in-filter.roto
Loading new Roto script /etc/filters/rib-in-pre-filter.roto
Loading new Roto script /etc/filters/rib-in-post-filter.roto
Loading new Roto script /etc/filters/bgp-in-filter.roto
Listening for HTTP connections on 127.0.0.1:8080
Starting target 'null'
Starting unit 'rib-in-pre'
Starting unit 'bmp-in'
Starting unit 'bgp-in'
Starting unit 'rib-in-post'
All components are ready.
All components are running.
bmp-in: Listening for connections on 0.0.0.0:11019
bgp-in: Listening for connections on 0.0.0.0:11179
```

Note: If you see an error stating Rotonda failed to load Roto scripts however, please refer to *Unable to load Roto scripts*. You will not be able to complete the Quick Tour before fixing this.

Congratulations! You've successfully started the Rotonda daemon, but what did you just do? Well, you've started the Rotonda daemon with a configuration that we've created specially for the first, "minimal viable product" release. It creates a pipeline that can ingress data from BMP and BGP sources, has two RIBs and a few filters.

Before we go into details about that, let's go over the output to STDOUT a bit. The first few lines that mention *Roto* are messages about filters that are being loaded in various places in the Rotonda pipeline. The line following that is a HTTP web server that is started to host requests from the users issued to any of the RIBs. Then there is a line about a target 'null' being started, that's the endpoint of the pipeline, in this case it's basically send to `/dev/null`.

Then a few lines about the units that are being started. Units are the parts that together form the pipeline. They look innocuous.

Then following two lines for the ingress connector units, one for BGP (`bgp-in`), and one for BMP ingress (`bmp-in`). Lastly, a confirmation that everything's ready and that everything's successfully started.

We will learn more about all these components as we work our way through this Quick Tour. First, we'll have a brief look at the HTTP, verifying it is working as expected as we need it for the rest of this trip.

1.3.1 Using the HTTP service

Rotonda is now waiting for input on one of its configured and running ingress interfaces, not very exciting. We can however inspect some of its internal state. If you let the shell with Rotonda running, open another shell and issue this command:

```
$ curl http://localhost:8080/status
```

(Or open the URL in a browser; also make sure to **not** add a trailing slash)

Then you'll see a list of variables names with zeroes and minus ones as values. Again, not super exciting, but at least we are seeing the confirmation that it is running and waiting.

Now let's query another endpoint, preferably in a browser (since it outputs html): http://localhost:8080/bmp-routers/.

Hopefully, you'll see a (for now empty) table, with column headers hinting at the type of information it will present once occupied. So far, so good. Let's fill this thing with some data.

1.4 Injecting and querying for data

With the two ingress connectors up and running, our Rotonda instance is ready to ingest information via BMP and BGP. We will focus on BMP, and for this tutorial, we will be using artificial BMP data. If you have a (software) router capable of producing a BMP stream and prefer to use that, study *Configuration* and continue with *Querying the RIBs* afterwards.

To inject the artificial BMP data, we are going to use our `bmp-speaker` tool. In the next session, we will discuss how to install and use it.

1.4.1 Mocking Ingress Data

The `bmp-speaker` tool can be installed with `cargo`. We leave our Rotonda running and open a new shell:

```
$ cargo install routes --bin bmp-speaker --version 0.1.0-rc0 --git https://github.com/
↪NLnetLabs/routes
```

Note: If you do not have `cargo` available on your system, you probably want to check out the *Rust* section on here to acquire a working toolchain with everything you need, without hassle.

This should make the `bmp-speaker` tool available. We can connect to our Rotonda instance:

```
$ bmp-speaker --server localhost
```

You'll be presented with a prompt, waiting for your input. We must mimic a real BMP stream here, adhering to the BMP protocol standards, meaning that we:

1. First send information about who (which router) is connecting (line 1 below),
2. followed by information about one or more connected peers (line 2),
3. followed by route information for those connected peers (line 3+4).

Copy these lines, in this order, into the prompt, and don't worry if not all of these make sense (and yes, we are inserting /25's):

```
$ bmp-speaker --server localhost
> initiation my-bmp-router "Mock BMP monitored router"
> peer_up_notification global 0 10.0.0.1 65000 127.0.0.1 80 81 888 999 0 0
> route_monitoring global 0 10.0.0.1 65000 0 none "e [65001,65002,65003] 10.0.0.1 NO_
↪ADVERTISE 192.0.2.0/25"
> route_monitoring global 0 10.0.0.1 65001 0 none "e [65001,65002,65003] 10.0.0.1 NO_
↪EXPORT 192.0.2.128/25"
```

If all's well, you should not have gotten any errors, just a new prompt. Let's verify Rotonda received the exported routing information by visiting http://localhost:8080/bmp-routers/ in a browser. (**Note:** the trailing slash is required this time).

You should now see a table listing one monitored router, with the name and description we used in the first input to `bmp-speaker`.

Clicking on the *sysName* takes you to a details page for that connected router. It includes a table describing all the connected Peers, showing the number of prefixes obtained from each peer. Clicking on the number expands the table, showing the prefixes we just fed into Rotonda via `bmp-speaker`.

It also shows two links, `rib-in-pre` and `rib-in-post`. These are the two RIBs that Rotonda configured by default, referring to the contents of the *Adj-RIB-In* (i.e., the received routes) for that peer before and after applying any local policy, respectively. These links point to the RIB query endpoints and should give back JSON instead of HTML.

Looking at the URLs for those links gives an idea of how we can use these endpoints: 'search RIB $R for prefix $P'. In the next section, we look at how we can further refine such queries, and explain the output.

1.4.2 Querying the RIBs

To reiterate: Rotonda creates an HTTP endpoint for every RIB, which means the default configuration gives us http://localhost:8080/rib-in-pre and https://localhost:8080/rib-in-post. Note that when requested like this, these return an error. We need to create a proper query.

Before we dive into what these endpoints offer, make sure you are able to format the returned JSON answers such that they are readable. If your browser does such formatting (e.g. Firefox does), that's perfect. If you prefer to use the command line with for example `curl`, consider combining it with `jq` to format the JSON output, as in our examples below.

Note: If you opted to you another source of data than the mock data using `bmp-speaker` as described in the previous step, the outputs in these examples will be different.

Now, let's start out with a simple query, one that might still be in your browser form the previous steps after navigating through the HTML output:

```
$ curl -s http://localhost:8080/rib-in-post/192.0.2.0/25 | jq .
```

You should see output like this:

```
{
  "data": [
    {
      "route": {
        "prefix": "192.0.2.0/25",
        "as_path": [
          "AS65001",
```

(continues on next page)

(continued from previous page)

```
          "AS65002",
          "AS65003"
        ],
        "origin_type": "Egp",
        "next_hop": {
          "Ipv4": "10.0.0.1"
        },
        "atomic_aggregate": false,
        "communities": [
          {
            "rawFields": [
              "0xFFFFFF02"
            ],
            "type": "standard",
            "parsed": {
              "value": {
                "type": "well-known",
                "attribute": "NO_ADVERTISE"
              }
            }
          }
        ],
        "peer_ip": "10.0.0.1",
        "peer_asn": 65000,
        "router_id": "my-bmp-router"
      },
      "status": "InConvergence",
      "route_id": [
        0,
        0
      ]
    }
  ],
  "included": {}
}
```

In the `data` object of this JSON output you see one of the routes that was transmitted by our `bmp-speaker` to Rotonda. It contains the prefix, the BGP path attributes, and some metadata such as the `router_id` field.

Let’s query for a less-specific prefix that we did not explicitly fed into Rotonda, specifying that we want to have more-specifics included in the response:

```
$ curl -s http://localhost:8080/rib-in-post/192.0.2.0/24?include=moreSpecifics | jq .
```

```
{
  "data": [],
  "included": {
    "moreSpecifics": [
      {
        "route": {
          "prefix": "192.0.2.0/25",
          "as_path": [
```

(continues on next page)

(continued from previous page)

```
            "AS65001",
            "AS65002",
            "AS65003"
          ],
          "origin_type": "Egp",
          "next_hop": {
            "Ipv4": "10.0.0.1"
          },
          "atomic_aggregate": false,
          "communities": [
            {
              "rawFields": [
                "0xFFFFFF02"
              ],
              "type": "standard",
              "parsed": {
                "value": {
                  "type": "well-known",
                  "attribute": "NO_ADVERTISE"
                }
              }
            }
          ],
          "peer_ip": "10.0.0.1",
          "peer_asn": 65000,
          "router_id": "my-bmp-router"
        },
        "status": "InConvergence",
        "route_id": [
          0,
          0
        ]
      },
      {
        "route": {
          "prefix": "192.0.2.128/25",
          "as_path": [
            "AS65001",
            "AS65002",
            "AS65003"
          ],
          "origin_type": "Egp",
          "next_hop": {
            "Ipv4": "10.0.0.1"
          },
          "atomic_aggregate": false,
          "communities": [
            {
              "rawFields": [
                "0xFFFFFF01"
              ],
              "type": "standard",
```

(continues on next page)

(continued from previous page)

```
              "parsed": {
                "value": {
                  "type": "well-known",
                  "attribute": "NO_EXPORT"
                }
              }
            }
          ],
          "peer_ip": "10.0.0.1",
          "peer_asn": 65000,
          "router_id": "my-bmp-router"
        },
        "status": "InConvergence",
        "route_id": [
          0,
          0
        ]
      }
    ]
  }
}
```

In this output, the `data` block is an empty array, because there were no results found for the *exact* prefix we queried for. However, because we specified the query parameter `include=moreSpecifics` in the URL, the `included` field hosts an object `moreSpecifics` with an array containing the two routes we fed into Rotonda using `bmp-speaker`.

And yes, you guessed it, there's also a query parameter argument `lessSpecifics`, yielding similar results:

```
$ curl -s http://localhost:8080/rib-in-post/192.0.2.1/32?include=lessSpecifics | jq .
```

```
{
  "data": [],
  "included": {
    "lessSpecifics": [
      {
        "route": {
          "prefix": "192.0.2.0/25",
          "as_path": [
            "AS65001",
            "AS65002",
            "AS65003"
          ],
          "origin_type": "Egp",
          "next_hop": {
            "Ipv4": "10.0.0.1"
          },
          "atomic_aggregate": false,
          "communities": [
            {
              "rawFields": [
                "0xFFFFFF02"
              ],
```

(continues on next page)

(continued from previous page)

```
              "type": "standard",
              "parsed": {
                "value": {
                  "type": "well-known",
                  "attribute": "NO_ADVERTISE"
                }
              }
            }
          ],
          "peer_ip": "10.0.0.1",
          "peer_asn": 65000,
          "router_id": "my-bmp-router"
        },
        "status": "InConvergence",
        "route_id": [
          0,
          0
        ]
      }
    ]
  }
}
```

More details on the HTTP server and its endpoints for each RIB can be found in the section on *RIB unit*.

Now that we know how to get data out of Rotonda, let's have a look at filters and figure out how we can control what actually gets stored in the first place.

1.5 Using Filters

Note: Make sure your Rotonda instance found and loaded the filter scripts upon startup before working your way through this section.

If not, refer to *Unable to load Roto scripts* or *Introduction*.

The filter scripts Rotonda uses are located in `/etc/rotonda/filters` if you installed via a package directory. If you list the contents of that directory, you'll notice a bunch of files of type `.roto`, these are the files containing the filters. Open the file called `rib-in-pre-filter.roto` with your favourite text editor. It should look like this:

```
filter rib-in-pre-filter {
    define {
      rx msg: Route;
    }

    apply {
      accept;
    }
}
```

This is the filter that gets run on any route that flows into the `rib-in-pre` RIB in Rotonda, this filter decides whether to store the route, and subsequently pass it on to `rib-in-post`.

Let's change this filter a bit, so that it look likes this:

```
filter rib-in-pre-filter {
  define {
    rx route: Route;
  }

  term my-asn {
    match  {
      route.as-path.origin() == AS64512;
    }
  }

  apply {
    filter match my-asn matching {
      return reject;
    };
    accept;
  }
}
```

When your code looks good you can save it, and exit your text editor. So what did we just do? Well, as we saw earlier Rotonda configured a few RIBs for you out of the box. Each of these RIBs has a filter built in into it, in front of the storage mechanism of the RIB. So the payload comes into the filter, the filter creates a filtering decision based on the content of the payload and if that decision is a resounding `Accept`, it gets stored in the RIB. Each filter consists of a script in a language we dubbed *Roto*, so each filter inside a RIB is programmable. And so, we just re-programmed the filter inside the RIB called `rib-in-pre`.

1.5.1 Explaining our Filter

So what does this script do? First of all, in the `define` section, we defined the incoming *type* of our payload. For filters to be able to meaningfully create a filtering decision it needs to know how the contents of the payload can be parsed and this is exactly what specifying the type does. *Roto* has built-in types: Primitive ones, like various integer types, a string type and so on, and more complex ones especially for BGP/BMP purposes, like `BgpMessage` and `Route`. Finally, roto users can create their own types, based on a *Record* or a *List*. In our *define* section the keyword `rx` stands for the incoming payload ("receive"). We assign it to a variable called `route`, of type `Route`. `Route` is a built-in Roto type, that resembles a Record. This is the roto type that Rotonda extracts from a BGP UPDATE message (or a BGP UPDATE message carried in a BMP RouteMonitoring message), and is modeled after the way **RFC 4271** uses the term. It contains a prefix, the path attributes and some meta-data that were found in a BGP UPDATE message. So a single BGP UPDATE may get transformed into multiple routes, since a BGP UPDATE message can contain more than one prefix in its NLRI. You can read more about the roto `Route` type *here*. Suffices to say for now, that we can use the payload-as-a-route to make filtering decisions with, and that's exactly what we do in the rest of our roto script.

We have one `term` section in our script called *my-asn*. It contains one match rule, that features our `route` variable, that has as its value our incoming payload. With the expression `route.as-path.origin() == AS64512` we create a comparison with the value returned from a method that is being called on a field of the *route* variable. So this expression says: *if the origin of the AS PATH atttribute of the incoming payload equals AS64512 then return true`*.

In the *apply* section - a roto script can only have one `apply` section - *term* sections are bound to a filtering decision by means of one or more *filter* expressions. In our script we only have one `filter` expression. It states that the mentioned `term` should *match*, meaning it should return `true`. Then, inside that `filter` block, the *return reject;* statement is an early return from the whole script. The *accept* statement in the last line of the *apply* section is the fall-through return value from the script if nothing above it in the section matched. So our `filter` expression says: "if the `my-asn` term returns `true`, then return `reject` from our script. In all other cases return `accept`".

So, now we can assess the overall effect of our filter script, and that is: *drop all routes that have AS64512 as the origin of the AS PATH*. In our default BGP configuration AS64512 is defined as our ASN. In other words, this filter script is an example of an iBGP filter.

1.5.2 Activating the modified Filter

We have changed the filter, we know what it is supposed to do now, but we still have to activate the filter. We can do this by sending Rotonda the HUP signal. You can do this by issuing:

```
$ killall -HUP rotonda
```

in a shell. In the log output you should see the confirmation of Rotonda reloading the changed script:

```
[2023-12-11 13:34:42] INFO  SIGHUP signal received, re-loading roto scripts from␣
↪location "/etc/rotonda/filters"
[2023-12-11 13:34:42] INFO  Roto script /etc/rotonda/filters/bmp-in-filter.roto is␣
↪already loaded and unchanged. Skipping reload
[2023-12-11 13:34:42] INFO  Re-loading modified Roto script /etc/rotonda/filters/rib-in-
↪pre-filter.roto
[2023-12-11 13:34:42] INFO  Roto script etc/rotonda/filters/rib-in-post-filter.roto is␣
↪already loaded and unchanged. Skipping reload
[2023-12-11 13:34:42] INFO  Roto script etc/rotonda/filters/bgp-in-filter.roto is␣
↪already loaded and unchanged. Skipping reload
[2023-12-11 13:34:42] INFO  Done reloading roto scripts
```

In the first line we see the confirmation that Rotonda received our signal, and in the fourth line, we see confirmation that it is reloading our script.

Tip: If you don't see any new logging information, then maybe your process is not precisely called rotonda. You can try *pgrep rotonda | xargs kill* and see if that works.

1.5.3 Trying the modified Filter

If you now restart the `bmp-speaker` tool that we used earlier, we can try to send a few BMP messages and then see if our filter functions.

```
$ bmp-speaker --server localhost
> initiation my-bmp-router "Mock BMP monitored router"
> peer_up_notification global 0 10.0.0.1 65000 127.0.0.1 80 81 888 999 0 0
> route_monitoring global 0 10.0.0.1 65000 0 none "e [65001,65002,64512] 10.0.0.1 NO_
↪ADVERTISE 192.0.2.0/25"
> route_monitoring global 0 10.0.0.1 65001 0 none "e [65001,65002,65003] 10.0.0.1 NO_
↪EXPORT 192.0.2.128/25"
```

If you go to the HTTP/JSON interface of Rotonda then you can check that only one route has been filtered out, and that one has passed through our filter scripts and has been stored in the RIBs.

In the next chapter we will look at the configuration of the RIBs in Rotonda.

1.6 Composing the pipeline

So, as said earlier, Rotonda can be though of as a pipeline through which BGP data will flow. This pipeline consists of units that pass on data. All units emit data and most of them can specify from which other unit(s) they want to receive data. Furthermore some units can receive input from users and they can output data through additional channels.

Let's a look at a very simple pipeline:

Here you see an Ingress Unit on the left, the *west* side, and an egress unit on the right, the *east*. The BGP data flows from the west to the east. This pipeline would just take all the BGP data it gets

1.6.1 A Filter and RIB Pipeline

This pipeline however would just move all the data through its ingress unit to the the egress unit, and that's that.

1.6.2 Changing the Pipeline

As we've briefly explained in the <<gentle introduction>> and the <<configuration defualt>>, Rotonda consists of units that are connected to form a pipeline, where data flows from sources to targets.

When you start Rotonda without a configuration file, it will use its built-in configuration, that features a pipe-line that exists of five units, namely two connectors (`bmp-in` and `bgp-in`), two RIBs (`rib-in-pre` and `rib-in-post`) and a terminating connector called `null`.

The west-east flow of the default pipeline looks schematically like this:

In the last chapter we saw how we could query these RIBs through the HTTP interface, on their respective endpoints, i.e. https://localhost:8080/rib-in-pre and https://localhost:8080/rib-in-post/. But there's more that we can do with our pipeline. We can add and remove units at run-time. For example, we could add another RIB to our pipeline. Let's do that.

Suppose we want to create a RIB that stores routes that have AS PATHs that have origins only from certain autonomous systems (ASes). We'll break this down in a few steps:

1. Start Rotonda with the default configuration, in the right directory.

We will have to make sure first that we are running Rotonda with an on-disk configuration file, so not the built-in configuration, since the built-in configuration cannot be changed at run-time.

If you have a Rotonda running then stop that instance, by sending it a SIGKILL through any means, e.g. *ctrl-c* in the terminal that runs it.

If you have installed by building from source, using the *cargo install* method (<<here>>), you should first change your working directory to the directory mentioned in the *warning* at install time. The path probably looks something like this: */<USER_DIR>/.cargo/git/checkouts/rotonda-<HEX NUMBER>/<HEX NUMBER>*.

If you have installed from a package this directory is most likely, */etc/rotonda*. This */etc* directory can also be found in the Rotonda github repo. Change your working directory to one level above this *etc* directory.

You are now ready to start Rotonda by issuing:

```
rotonda -c etc/rotonda.example.conf
```

You should see output like this:

```
Loading new Roto script etc/bmp-in-filter.roto
Loading new Roto script etc/rib-in-post-filter.roto
Loading new Roto script etc/rib-in-pre-filter.roto
Loading new Roto script etc/bgp-in-filter.roto
Listening for HTTP connections on 127.0.0.1:8080
Starting target 'null'
Starting unit 'bgp-in'
Starting unit 'rib-in-post'
Starting unit 'bmp-in'
Starting unit 'rib-in-pre'
All components are ready.
All components are running.
bgp-in: Listening for connections on 0.0.0.0:11179
bmp-in: Listening for connections on 0.0.0.0:11019
```

If you have a browser present on the system you are running Rotonda on, you can navigate to http://localhost:8080/status/graph and see a graph that describes the pipeline that we just started.

1. Modify the configuration file that is being used.

Now for the cool stuff. While leaving Rotonda running, fire up your favourite text editor in another shell, and edit the file that we used for our configuration, *etc/rotonda.example.conf*. Add a unit at the end of the file like so:

```
[units.my-rib]
type = "rib"
sources = ["rib-in-pre"]
rib_type = "Physical"
filter_name = "my-rib-filter"
http_api_path = "/my-rib"
```

… and edit the the `[targets.null]` unit, and add *"my-rib"* to the `sources` field, like so:

```
[targets.null]
type = "null-out"
sources = ["rib-in-post","my-rib"]
```

… and save the file.

1. Create the roto filter script.

Now we must still create the roto script we referenced in our modified `rotonda.exmaple.conf`, namely the file `my_rib.roto`. So create that file, and fill it with this:

```
filter my-rib-filter {
    define {
        rx route: Route;
    }

    apply {
        accept;
    }
}
```

2. SIGHUP Rotonda.

Now with everything in place we can send the HUP signal to the rotonda process:

```
pgrep rotonda | xargs kill -HUP
```

You should get new log output like this in the console that is running your Rotonda:

```
SIGHUP signal received, re-reading configuration file '/home/rotonda/.cargo/git/
↪checkouts/rotonda-54306a42d783f077/8e4d152/etc/rotonda.example.conf'
Loading new Roto script etc/my-rib.roto
Roto script etc/bmp-in-filter.roto is already loaded and unchanged. Skipping reload
Roto script etc/rib-in-post-filter.roto is already loaded and unchanged. Skipping reload
Roto script etc/rib-in-pre-filter.roto is already loaded and unchanged. Skipping reload
Roto script etc/bgp-in-filter.roto is already loaded and unchanged. Skipping reload
Reconfiguring target 'null'
Reconfiguring unit 'rib-in-pre'
Starting unit 'my-rib'
Reconfiguring unit 'bgp-in'
Reconfiguring unit 'bmp-in'
Reconfiguring unit 'rib-in-post'
Configuration changes applied
All components are ready.
All components are running.
```

If you now refresh your browser tab that showed the pipeline graph, you'll see that our new *my-rib* was added!

1.7 Where to go from here

This was a very quick tour of the some of the current features of Rotonda. We've showed you how to run Rotonda with some mock data, do some configuration and create and modify a roto script.

You may want to run Rotonda, while inserting real BGP/BMP data, read about *connector* units *here*, to set up a BGP or BMP session.

When you've got one or more connectors working, you probably want to change one or more Roto filters. Read more about it *here*.

Finally you could read about the details of the various units *here*.

1.8 Introduction

Warning: The current configuration setup of Rotonda has serious shortcomings. It will very likely change drastically in the near future.

1.9 Built-in vs file-based configuration

When Rotonda is compiled, a configuration is built in. The built-in configuration is used when no configuration file is specified when invoking `rotonda` on the command line with the `-c` option. You can see the details of the built-in configuration by issuing `rotonda --print-config-and-exit` on the command line.

If you've installed Rotonda through a binary package, the built-in configuration is roughly equivalent to the file you will find in `/etc/rotonda/rotonda.conf`.

When running with the built-in configuration Rotonda will look for the roto filter scripts in the absolute path `/etc/rotonda/filters/`. If that directory can not be found, or the required filters cannot be found in that path, no filter scripts can be loaded resulting in a filter-less, accept-everything Rotonda instance.

In contrast: if you are starting Rotonda with an explicit configuration file, by invoking the `-c` option, and the `roto_scripts_path` in that configuration file cannot be found, or the required filters cannot be found, then **Rotonda will not start**.

Tip: the built-in configuration and the configuration files included in the package will look in the absolute path `/etc/rotonda/filters/` by default.

Tip: When compiling from source, the file in the repository in `etc/rotonda/rotonda.builtin.conf` is used to create the built-in configuration. Note that if you comment out the `[targets.mqtt]` section, mqtt functionality will not be available in the Rotonda build you are going to compile based on it.

This only applies to building with `cargo build --release`, not `cargo install`.

1.10 Completing the configuration file

The `/etc/rotonda/rotonda.conf` configuration file that comes with the packaged version of Rotonda is not complete, and Rotonda will abort if you try to use it as-is. You will have to edit it and at the very least fill out the values in the `[targets.mqtt]` section. Probably you'll also want to edit the fields `my_asn` and `my_bgp_id` in the `[units.bgp-in]` section.

1.11 The Global Configuration File

Warning: The current configuration setup of Rotonda has serious shortcomings. It will very likely change drastically in the near future.

This is the detailed description of what a global configuration file should look like.

Global configuration happens in a file that by convention is called `rotonda[.DESCRIPTION].conf`, e.g. a there is file called `rotonda.example.conf`, that describes an example configuration.

This file must be in TOML format (https://toml.io/) and is structured as follows:

- global settings
- 1 or more units
- 1 or more targets

Collectively units and targets are referred to as components.

Data flows from West to East beginning with at least one input unit, through zero or more intermediate units and out terminating at at least one target.

Additionally Rotonda has HTTP interfaces to the North and output stream interfaces to the South. The HTTP interfaces to the North may be used to inspect and interact with the application. Some types of units and target extend the HTTP interface with additional capabilities. The output stream interfaces to the South provide support for alternate forms of output such as MQTT event publication, logging/capture to file and proxying to external parties.

Taken together one can think of the flow of information like so:

Data can only be successfully passed from one component to another if the receiving component supports the value type output by the producing component. Consult the "Pipeline interaction" sections in the documentation below to ensure that your chosen inputs and outputs are compatible with each other.

1.11.1 A word about Components (Units & Targets)

A unit is an input or intermediate processing stage. A target is a final output stage. There must always be at least one unit with one downstream target.

Unit and target definitions have similar forms:

Names must be unique, types must be valid and any mandatory settings specific to the component type must be specified.

The currently available components are intended to be used like so:

Additionally there are some components intended for diagnostic use:

Each unit is able to process certain types of input and emit certain types of output. More information about each component type is given below.

1.11.2 A word about Roto Scripting

Some units and targets take one or more filter(-map) names which refer to "filter" or "filter-map" blocks in any loaded Roto script files. Roto files are written in the Rotonda Roto scripting language. Each file may contain a mix of "filter" and "filter-map" blocks. "filter" blocks accept or reject their Western input. "filter-map" blocks act the same but can also "map" the input from the West to a different output on the East. Both "filter" and "filter-map" blocks can also send data to one or more output streams to the South.

Roto scripts work with Roto Types (RTs). All Roto script inputs, outputs and intermediate values are Roto Types. Different units and targets accept and produce different Roto Types and for a Rotonda pipeline to work properly input and output types must be correctly aligned.

When Roto scripts send output to output streams to the South the data published to the stream is in the form of a Roto Record type which consists of key/value pairs, two of which have special meaning in Rotonda:

- name: This key should have a string value which identifies the name of the target which is intended to handle the output Roto value. That target must still receive the value.
- topic: This key should have a string value which may be used by a target that processes the output Roto value to determine what to do with it, e.g. in the case of the MQTT target it can be used to influence the eponymous MQTT topic to which a message will be published.

The following OPTIONAL settings MAY be specified if desired:

Note: In the diagrams below the term "RT" denotes any valid Roto scripting type.

1.12 HTTP API

The HTTP API offers endpoints for interacting with and monitoring Rotonda at runtime:

The following MANDATORY settings MUST be specified:

1.13 Unit: bgp-tcp-in

This unit listens on a specified TCP/IP address and port number for incoming connections from zero or more RFC 4271 [1] BGP speakers.

The following MANDATORY settings MUST be specified:

The following OPTIONAL settings MAY be specified if desired:

The following MANDATORY settings MUST be specified in a peers."address" table:

Pipeline Interaction

One Route value is output per prefix announced or withdrawn via a BGP UPDATE message received. Withdrawals may also be synthesized if the BGP session is disconnected or the TCP/IP connection to the remote BGP speaker is lost.

1.14 Unit: bmp-tcp-in

This unit implement an RFC 7854 "BGP Monitoring Protocol (BMP)" "monitoring station" [1] by listening on a specified TCP/IP address and port number for incoming connections from zero or more BMP capable routers. This unit processes the incoming raw BMP messages through a BMP state machine in order to extract, store and propagate downstream the route announcements and withdrawals.

This unit extends the HTTP API with endpoints that output HTML and text formatted information about the monitored routers currently streaming data into Rotonda. These endpoints are intended for operators as a diagnostic aid and not for automation purposes. The output format is not intended to be machine readable and may change without warning.

The following MANDATORY settings MUST be specified:

1.14.1 HTTP API Endpoints

Pipeline Interaction

One Route value is output per prefix announced or withdrawn via a BGP UPDATE message received as the payload of a BMP Route Monitoring message. Withdrawals may also be synthesized due to BMP Peer Down notification or loss of TCP/IP connection to the monitored BMP router.

1.15 Unit: filter

This unit runs a filter script that can be either a filter or a filter-map:

- A filter accepts or rejects the input Roto value that it receives.
- A filter-map does the same but the output Roto value can be different than the input value, i.e. as if the input was "mapped" to the output.
- Both filter and filter-map scripts can optionally emit additional Roto values for consumption by particular targets.

The following MANDATORY settings MUST be specified:

Pipeline Interaction

1.16 Unit: rib

This unit is a general purpose prefix store but is primarily intended to map prefixes to the details of the routes to those prefixes and the source from which they were received.

It offers a HTTP API for querying the set of known routes to a longest match to a given IP prefix address and length.

Upstream announcements cause routes to be added to the store. Upstream withdrawals cause routes to be flagged as withdrawn in the store.

The following MANDATORY settings MUST be specified:

Pipeline Interaction

In summary the flow looks like this:

Now lets break down the various different possible scenarios into more detail:

1. A single physical RIB with no Roto script filtering:
2. A single physical RIB with a Roto script filter:
3. A physical RIB and a virtual RIB, each with their own Roto script filter:

Tip: Queries to the HTTP API of a virtual RIB are submitted upstream to the physical RIB and the results flow back down the pipeline to the requesting virtual RIB and out via its HTTP API. Results are processed through each vRIB filter yielding the vRIB modified "view" of the result data.

Tip: Values emitted by output streams of vRIB filters when processing HTTP API query results are silently discarded, i.e. values emitted by output streams of vRIB filters are only honoured for input data that originated to the West of the pRIB, NOT for data that was the result of a HTTP API query.

Tip: The input to a physical RIB is usually a Route but can also be a Record with a "prefix" key, but only Route values support the notion of being "withdrawn". The entire record (all its keys and values) will be added to the set of values stored at the prefix in the RIB, with the rib_keys fields determining whether a new value is added to the set or replaces an existing item in the set.

1.17 Target: mqtt-out

This target publishes JSON events to an MQTT broker via a TCP connection.

Tip: The MQTT broker is not part of Rotonda, it is a separate service that must be deployed and operated separately to Rotonda.

Tested with the EMQX MQTT broker with both the free public MQTT 5 Broker [1] and with the EMQX Docker image [2].

This target ONLY accepts input data that:

- Was received from a configured upstream source unit.
- Was emitted by a Roto script output stream.
- Is of type Record with a "name" field whose value matches the name of this instance of the mqtt-out target.

So naming an instance of this unit in a Roto script output stream record is not sufficient to have this unit receive it, this unit must still be downstream of the producing unit to receive its output.

The JSON event structure produced by this target is a direct serialization of the received Roto type as JSON, i.e. a record with a set of key/value pairs.

The following MANDATORY settings MUST be specified:

Pipeline Interaction

1.18 Target: null-out

This target discards everything it receives.

Rotonda requires that there always be at least one target. Using this target allows you to run Rotonda for testing purposes without any "real" targets, or if the only output is via Roto script output stream messages.

The following MANDATORY settings MUST be specified:

Pipeline Interaction

1.19 Unit Configuration

1.19.1 The default Pipeline

Rotonda consists of units that are connected to form a pipeline, where data flows from sources to target. Conceptually the data flows from the ingress all the way to the west to the egress in the east, through user-defined units. A unit may also have a north-facing input, and a south-facing output. Each units has a type, and the available types are: `Connector`, `RIB`, and `Filter`. You can read about the details of these units <<here>>.

When you start Rotonda without a configuration file, it will use its built-in configuration, that features a pipe-line that exists of five units, namely two connectors (`bmp-in` and `bgp-in`), two RIBs (`rib-in-pre` and `rib-in-post`) and a terminating connector called `null`. We will refer to this configuration as the 'default configuration'.

The west-east flow of the default pipeline looks schematically like this:

Fig 1. The MVP default Pipeline

Let's go over this configuration a bit. On the west-side we see the two connectors that act as ingress connectors, one for BMP sessions, and one for BGP sessions. The BMP connector knows how to handle a BMP sessions, where it acts as a so-called `monitoring station` (**RFC 7854**), with one BMP peer per session, presumably a router. Likewise, the BGP connector can handle BGP sessions with BGP speakers. The packets received by these connectors are parsed by these connectors into BMP or BGP messages, and run through a user-defined roto filter, one per connector. If the BMP or BGP message passes the filter, then it is 'exploded' into one or several routes. Then these routes are broadcasted to all connected units on their east side, which in the case of the default configuration is the `rib-in-pre` RIB unit.

The `rib-in-pre` unit is a physical RIB, meaning the routes that it receives are stored in memory, and can be queried through the HTTP interface. This RIB also includes a filter in front of it, that determines whether the incoming route should be stored and passed on to the next RIB to the east.

From `rib-in-pre` the routes are broadcasted to `rib-in-post`, a virtual RIB, this means that the routes are *not* stored in this RIB. The RIB can still be queried through the HTTP interface, but when done so it will go back to the first virtual RIB to the west and fetch the requested routes from there, and then filter them through its own filter.

All `RIB` units in a pipeline are required to connect both on the west and on the east side, and therefore the `rib-in-post` unit must also have a east bound connection. To fulfil this requirement, we connect a `null`, or terminating connector to the east. This terminating connector acts somewhat like `/dev/null` does on UNIX, it accepts the packets, but simply discards them upon reception.

Both the RIBs can queried through a HTTP JSON API

1.19.2 API endpoints

/metrics /status /status/graph

/bmp-routers /rib-in-pre/{prefix}/[?include=moreSpecifics/lessSpecifics] /rib-in-post/

1.20 Connector Unit

1.21 Filter Unit

1.22 FilterMap Unit

1.23 Ingress Unit

1.24 introduction

Units

1.25 Rib Unit

1.26 On speaking BGP

Central to Rotonda is the observation that there is no one-size-fits-all application for a network's BGP routing needs any more. As networks are growing, so is their complexity, and as a consequence, BGP is being used in more places in the network, and it has turned into a generic container for information about network nodes, increasing its complexity. Pressure on BGP to offer better security further increased the complexity of BGP itself and its deployments.

By now, the notion of a "BGP speaker" (the wording of **RFC 4271**) with a fixed set of "Routing Information Bases" (RIBs) with prescribed behaviour is only one of many different *BGP functions*, as we would like to call them, that we can identify in a network. Rotonda aims to provide users with the building blocks to create their own application optimized for any of these BGP functions.

1.27 BGP Functions

So, what are these BGP functions anyway? The way we would like to describe them is "a specialised task within a network relying on BGP, that requires a subset of BGP's features". Often, this task is performed by a node in the network.

1.28 BGP's Features

Before we delve deeper into BGP functions, let's first establish what we mean with BGP's features, by posing the question: "What are the requirements for a fully-fledged BGP speaker?". **RFC 4271** and its updates describes BGP consisting of three components: the BGP packet format, the BGP state machine, and the Routing Information Base.

The specific required interaction of these components describe BGP's features, they boil down to:

"Listening" or "Passively speaking" BGP

- parsing BGP packets
- opening and maintaining sessions with the BGP state machine

This is the least amount of work a BGP function would need to do in order to be accepted as a functioning part of a network routed with BGP. Note that both the terms "listening" and "passively speaking" are colloquial, i.e. not mentioned anywhere in the RFCs.

"Keeping State in RIBs"

- storing incoming routes in "adj-RIB-in", "loc-RIB" and "adj-RIB-out"

In order for a BGP function to conform to an IETF standards-compliant "BGP speaker" it would have to feature these RIBs, at least, "logically" (again, you guessed it, **RFC 4271**).

"Speaking" BGP

- performing best-path selection
- propagating routes to peers

Based on the contents of its RIB-ins and its configured policies, it should implement the best-path selection rules from **RFC 4271** and propagate the selected routes to its peers.

These are the features which, if implemented correctly, allow for an application to be an IETF standards compliant "BGP Speaker". As we stated before, though, a BGP function does not have to implement all of them to fulfil their function. Even more so, for them to function correctly, they don't have to be standards-compliant for all these features.

It would be nice if the first feature, "Passively speaking", is implemented completely standards-compliant, since it would hamper the interaction with other BGP speakers if it wasn't. Since it is valid for a BGP speaker to have a (local) policy in place that discards all routes to "adj-RIB-out", a BGP speaker that never propagates any route could still be standards-compliant. From that, it follows that such a BGP speaker will not have to engage in best-path selection, and in turn, it wouldn't have to do any state-keeping. Strictly speaking, it would at that point not be standards-compliant any more, but there would be no way for an outside observer to establish its non-compliance. We shall see that many a BGP function does indeed not require one or both of the "Keeping state" or "Speaking BGP" features, hence the word "subset" in our definition.

1.29 Route Server

A Route Server (as mentioned in **RFC 7947**) would be a example of a BGP function. A Route Server requires the "Speaking BGP" feature of the BGP protocol and the BGP state machine. It does require some kind of best-path selection mechanism, but it deviate considerably from the mechanism specified in **RFC 4271** and its updates. See for example here.

1.30 Route Reflector

Likewise, a Route Reflector (**RFC 4456**) serves a specific function in an iBGP network. Again, it requires the "Speaking BGP" feature of the BGP protocol, but it doesn't have to engage necessarily in best-path selection. Very simple Route Reflectors would not have a need for RIBs, they would just reflect the announcements and withdrawals they receive to their iBGP peers.

1.31 Route Collector

A Route Collector is, broadly speaking, a device that passively engages in BGP sessions with the purpose of storing the learned routes together with meta-data about the whereabouts of these routes. Some of the purposes of storing these routes would be troubleshooting, and (longitudinal) analysis.

"Passively engages" may mean that the collector wil connect over BMP (BGP Monitoring Protocol), out-of-band, with one or more BGP routers. In this case the collector, called a "BMP Station" (**RFC 7854**), will **not** be a node in the BGP network. It will only require the packet parsing features of BGP in order to be able to extract the routes, and to be able to gather metadata.

More commonly, though, Route Collectors **are** a node in the BGP network and the collector tries to "Passively Speak". As we saw, though, a passive speaker will have to engage in a minimum of speaking BGP. A Route Collector must never engage in best-path selection, or propagate routes to its peers whatsoever. Therefore, Route Collectors have no need for keeping state in RIBs as described in **RFC 4271**. Indeed, Route Collectors may entirely forego having RIBs, since they can be synthesised later from the stored routes and meta-data if need be.

1.32 Route Monitor

A Route Monitor is like a Route Collector in that it engages passively with BGP speakers through BMP or “Passively speaking”. However, instead of or in addition to storing, it will send signals to other systems and/or applications based on specific user-defined events or combinations of (accumulated) events occurring in the observed BGP network. Some purposes would be troubleshooting, post-mortem analysis and anomaly detection.

1.33 Other Functions

There are other numerous BGP functions that already exist in some shape or form or that could be extracted from current practices, to name just a few:

- Off-line Looking Glass
- Route Provisioning
- Route Policy Engine
- RPKI injection Filter
- Edge Sanitation Filter (“Edge Lord”)
- Route Optimizer

1.34 From BGP Function to BGP Application

All of the BGP functions mentioned here exist today, as hardware devices, or as software applications, be it open-source or proprietary. Many of these applications, though, were not intended to be used for these BGP functions, e.g. requiring patching, and/or requiring a multitude of applications, glued together with ad-hoc code.

Rotonda aims to alleviate this by offering the user the tools, a framework if you will, that allows users to build their own *BGP application* that may perform one or multiple, combined BGP functions, without aforementioned problems.

Secondly, Rotonda wants to be a tool that you can easily spin up to collect, experiment with and analyze BGP (and related) data.

Thirdly, Rotonda aspires to lower the barrier to implement new, experimental BGP (and routing) features, not only by offering this already-mentioned framework, but also by allowing plugins to be inserted into it easily. One area of development that jumps to mind would be improving the security features of BGP.

1.35 A BGP ‘Echo’ Application

So, from the previous chapter we learned that a BGP application does not have to do everything a BGP speaker does. The only must-have seems to be the ability to decipher BGP packets. For an application to do something useful it also probably must be able to output something. The minimal useful BGP application that we can come up with, then, looks something like this:

Fig.1 BGP ‘Echo’ Application

BGP packets ingress -> BGP packets egress

If our ‘Echo’ application was a UNIX utility, we would probably assume the default ingress would be STDIN and the default egress would be STDOUT. That would give us quite a bit of flexibility and extensibility: the user would be able to pipe the output of some other program into ours and likewise they could transform the output by piping into another program. There is significant drawback tough: our pipes would only ever be useful if our 'Echo' application and the programs at the other end of the pipe would use the same encoding. We could use, for example, the `mrt` (**RFC 6396**) format. Also, if we really want to output to STDOUT, we probably want to use a human-readable serialisation format there.

What if we could make our 'Echo' application a bit more flexible by creating an input 'thing' that could acquire BGP packets from different types of inputs and then transform that into BGP packets. If we do the same thing in reverse on the output side, it would something like this:

Fig.2 BGP 'Echo' Application 2

input from (socket, `mrt` file, stdin) -> BGP packets -> output to (socket, `mrt` file, stdout)

Neat! Now we can take input from various sources and pipe into another program, dump it to a file, or to stdout. Our application is still a bit simplistic, though, if you consider the socket input and output, for example. How would BGP packets appear on the socket? That would require a live BGP session on that socket at least. Also, an `mrt` file input resembles BGP packets, they are not the same, we would still need to transform the input into actual BGP packets that our application can internally handle.

Why don't we create a thing that can setup a BGP session on a socket and can then take BGP input from that session hand it to the rest of our application? Let's do that, and, while we're at it, we can also create a similar thing on the output side that can handle BGP sessions, or maybe open an output file, or something else that needs state. Since the format of the input and the output can differ, and both may not be actual BGP packets at all, and bot the input and the output handler have their own transform mechanisms, we are probably better off creating an internal BGP representation for our application.

Fig.3 BGP 'Echo' Application 3

input type handler -> into BGP transformer for input type -> internal BGP representation -> into output format transformer -> output type handler

We have now two 'things' on each end in our application, but we have a multitude of type of 'things', one type per format and input source combination that we would like to support. Also, in our 'thing' types the handler and the transformer are tightly coupled, so this would be a good time to start calling our handler and transformer 'things' *units*. We now have two units: a *ingress* unit that combines the input handler and its transformer and a *egress* unit that combines the output handler and its transformer.

Fig 4. BGP Echo Units Application

ingress unit -> egress unit

We can now create different ingress and egress units for all kinds of sources. Let's say that we create an ingress unit for a BGP session. BGP sessions are opened over TCP, so let's call our unit, `bgp-tcp-in`. We could also have a unit `mrt-file-in` for `mrt` files, and so on.

Almost done. We still have one problem, our BGP input contains lots of messages that we don't care about, let's say for now everything that is not a BGP UPDATE message. We don't want any other BGP messages in our output. We need to filter them out. So, in the middle let's create a filter called `update-filter`. That filter would simply take as its input an internal representation of a BGP packet and take a decision to let it pass or not, 'Accept' or 'Reject', based on what kind of BGP packet flows through it.

Fig. 5 Echo application

ingress unit -> update-filter -> egress unit

This is the pipeline of our minimal viable BGP 'Echo' application. And with that we have been introduced to two elemental parts of Rotonda: *Units* and *Pipelines*.

1.36 Beyond the ‘Echo’ Application

Let’s take a closer look at our filter. We have a filtering function that is hard-coded to only ‘Accept’ BGP UPDATE messages. That sounds a bit .. arbitrary. Wouldn’t it be nicer to let our users decide what they want to filter on? Then our filter could also be unit type! Let’s go with that plan and assume we’re going to offer ours users a generic filter that can contain some logic that they define to create the filtering decision. This filter would still take as its input our internal BGP representation and its output would still be the *Accept* or *Reject* value, the filtering decision. So, now we can offer a smarter *Echo* application, with a programmable filter, let’s call it the *FilteredEcho* application.

Fig. 5 The ‘FilteredEcho’ application

(configurable) ingress unit -> (programmable) filter unit -> (configurable) egress unit

1.37 Storing Data in the Pipeline

Our ‘FilteredEcho’ application slices, it dices, but as it goes with successful things, our users want more. Our users want to be able inspect the *current* state of a prefix that was received at one point in one or more BGP packets, they want to learn if it was announced with that community, not seen at all, or withdrawn. So how do we go about this? We could of course say, we gave you a BGP echo application, if you want that, you should accumulate that state yourself as a consumer of our stream. But we are not like that, we will help our users out, so let’s invent something for them.

As we saw, the Routing Information Base (RIB) is one of the fundamental elements of BGP. Could we reuse that for our ‘StatefulFilteredEcho’ application? Turns out we can. If we put a RIB to the right of our filter and assume that that RIB will be able to store routes that got the ‘accept’ for our filter, then we have the desired state.

Fig. 6 The preliminary *StatefulFilteredEcho* application

ingress unit -> filter unit -> RIB -> egress unit

Now we have the state we want, but if we assume that our filter is unchanged we have way too *much* state, we’re storing all the prefixes, because all the messages get through and will be ‘disassembled’ into route announcements or withdrawals. So if we assume that our filter could inspect the route announcements and withdrawals in the UPDATE packet, we could have it accept only the packets that contain the community that our users are interested in. The store would store all prefixes that have the community attached, and, assuming the RIB would have the logic to handle announcements and withdrawals we would have the desired state captured in our pipeline. After storing the routes that flow through the filter it will pass the routes on in our pipeline, as routes. As a matter of fact, we have changed the output type of our filter! We ’disassembled’ the packet, so the output type of our filter is already a bunch of routes, instead of the packet we had earlier on.

Fig 7. The *StatefulFilteredEcho* Application with types

```
    query prefix
         |

ingress unit<BGP p> – filter unit<routes> – RIB<routes> – egress unit
         |

    query result
```

Our RIB now contains everything we need, but we’re not done yet, because our users need a way to get the actual state of their prefixes out. Up until this point we used our RIB basically the way it was defined in RFC 4271, but now we going to extend it a bit: we’re going to add query capabilities to our RIB. So now we have a west-to-east pipeline with our BGP data and one north-to-south flow, where are users can provide a query for a prefix on the north side of the RIB, and get a result out at the south side of the same RIB.

Our filter is fairly tightly coupled with the RIB now, the output type of the filter should exactly fit the type that the RIB stores, so before we declare our RIB also a unit type, we should consider this. So let's make a RIB unit that simply includes a 'filter' to express that. Our RIB **unit** would look like this:

Fig. 8 RIB Unit

query prefix

|

filter<routes> – RIB<routes>

|

query result

1.38 Transforming Data

Our users are happy now, but they think they would be even more happy, if the output from their queries would only show its current status (never seen, announced, or withdrawn) and the AS path. The rest doesn't interest them, and they think it's a waste of space. Hard to disagree there, nobody wants to waste space, right? But what can we do, our RIB can only store complete routes, so that would be something like the tuple `(prefix, status, path_attributes)`, and they want `(prefix, status, as_path)`. Note that we can omit the actual community, because it's implied in the existence of the tuple in our RIB.

In line with RFC 4271 we defined our RIB to store routes, with the prefix as the key. But what if we defined our RIB to be able to store an arbitrary type, let's call it *metadata*. Our tuples that we store in our RIB would then look like: `(prefix, metadata)`. Since `metadata` is an arbitrary type, that in itself could be a tuple `(status, as_path)` for example.

Since the output type of the filter must be the storage type of the store our filter in the RIB should output the same type, `(prefix, metadata)`. But that is a problem! Remember, filters cannot change the output, as a matter of fact, they don't even output anything else but the filtering decision. For that we will have to invent another type of unit: the `FilterMap`. A `FilterMap` can both make filtering decision *and* transform the payload it receives and output the transformed payload. A *FilterMap* acts also as a normal filter if the input payload and the output payload are the same.

So let's change our RIB unit, and replace the `Filter` with a `FilterMap`, and integrate it into our new, all-singing and dancing *StatusAsPathStorage* application based on our *StatefulFilteredEcho*, but with significant changes.

Fig. 9 'StatusAsPathStorage' Application

query prefix

|

ingress unit<BGP p> – filtermap<T> – RIB<T> – egress<T> unit

|

query result<T>

where T: (status, as_path)

Besides using our new RIB type, we've changed the input type to that RIB to `(status, as_path)`, and therefore it also outputs that same type. The user can query the application, and will also get instances of that same type out. Finally, to the east we also get instances out of that type.

1.39 Egressing other data

Still our users are not satisfied! In spite of their latest BGP application being able to open BGP connections, ingress, transform and store the relevant data, while making it queryable, they want the east egress unit to output *all* announcements and withdrawals from the UPDATE messages it receives on the ingress unit, while keeping all other functionality. Ok, let's present the solution without further ado:

Fig. 10 'StatusAsPathStorage2' Application

ingress<BGP packet> unit – filter<routes> unit ——— egress<routes> | unit

|

RIB<(status, as_path)> unit

We have a filter *unit* that sits before the egress and the RIB unit. That filter unit filters to let only UPDATE messages flow through. From there it *splits* into a flow direct at the RIB, which in its latest incarnation has a filtermap of its own, that filters on the community our users want and transforms the routes into `(prefix, (status, as_path))` pairs and stores those into the RIB. The RIB can be queried by our users. The other branch flows to the east from the filter unit as routes, where they can be picked by our users at the egress unit.

1.40 A Virtual RIB

Ok, now our users have *one* last request: they want to be able to query the current status and all attributes for all the routes that come out of the UPDATE messages. Simultaneously they want to be able to query the RIB for prefixes with the particular community and retrieve their status and as_path. On the east they want the routes from the UPDATE messages echoed, both announcements and withdrawals.

Now, we could solve this by creating a pipeline that contains two RIBs, one for all the routes and one for the routes containing the particular community. That would waste space, though, since we would store the routes with the community *twice*, one time in each RIB. A better pipeline plan would include something we call a 'Virtual RIB', that's a RIB that does not store anything itself, but instead goes back to the closest RIB on its west-side and filter on prefixes present in that RIB.

Our whole pipeline plan would look like:

Fig. 11 'VirtualRIB1' Application

ingress<BGP packet> unit – pRIB<routes> unit ——— egress<routes> unit

|

vRIB<(status, as_path)> unit

We have now called the RIB we already had, *pRIB' as an abbreviation for `physical RIB* to set it apart from the virtual RIB, 'vRIB'. Our vRIB here is dependent on the pRIB from which routes flow come flowing in. It adds additional filtering and transforms the incoming routes, while being fully queryable, like the physical RIB.

1.41 Wrapping it up for now

And this is some of the stuff users will be able to construct with Rotonda. There are more types of units that we haven't discussed here, we didn't mention `Roto` yet, the language that allows users to program Filter(Maps). The rest of this documentation is about those.

1.42 A Word about the Minimal Viable Product ('MVP')

Not everything described above is possible with the first release, the "Minimum Viable Product", of Rotonda, partly because some details are still lacking in the Rotonda code, partly because not all configuration options are there yet. All the unit types, as well as the BMP and BGP ingress connectors do exist though.

In the first release, Rotonda has a fixed default pipeline, that is described in detail in this documentation <<<here>>> TODO. That configuration can be changed however to resemble the pipelines mentioned, but they are largely untested, so at this point we don't know if they actually work.

1.43 Where to go from here?

When you've read of all this introduction and you like what you've read, you may want to actually install Rotonda. See those instructions <<<here>>> TODO.

We have a practical <<quicktour>> for you, that talks through a working setup and suggesting some modifications you can try and see their effect.

Then, we have a more in-depth chapter on <<configuring Rotonda>>.

Finally, we have the full reference of the *Roto* filter (and query) language, and the full description of all the units.

1.44 Introduction

`Roto` is the filter language used by Rotonda. Eventually it will also be used as a query and configuration language.

`Roto` is a strongly typed, compiled language, i.e. every expression has to return a value that is an instance of a well defined, unambiguous type. Variable assignments, Constant definitions and method invocations are all expressions in `Roto`. This does not mean that the user has to specify types everywhere, most types can be inferred by the `Roto` compiler. Next to that Roto allows for automatic type conversions where possible. `Roto` has a fair amount of builtin types, both generic ones, e.g. unsigned 32-bits integers (`U32`), strings, etc, and types specific to BGP and routing, e.g. it has a `Prefix` type, a `AsPath` type and even a `Route` type. `Roto` also has the compound types, `List` and `Record`. Users can create their own types based on these two types.

`Roto` has no facilities for users to create loops of any kind. This is on purpose, and the reasoning behind this is similar to embedded languages like eBPF. The host application, in our case Rotonda, and in the eBPF case the Linux kernel, needs to be sure that the embedded program will actually finish and return a value, and in a reasonable - maybe even predictable - timeframe. As such, `Roto` aims to be non-turing complete.

The `Roto` toolchain consists of a compiler and a virtual machine, both of which are embedded in the `Rotonda` application. This means that the only requirements to create filters from source code is a running Rotonda instance. `Roto` does not rely on Rust or any part of its toolchain. The compiler takes `Roto` source code and compiles it down to a so-called *Mid-level Intermediate Representation* (`MIR`). This MIR code is then fed by Rotonda into a virtual machine. This MIR code is then executed by the virtual machine. The virtual machine itself consists of a stack machine and a bunch of registers.

1.44.1 The Future

In the future the virtual machine will probably will take a low-level IR, to facilate running user-created plug-ins, written in a general purpose language of the user's choice. These plug-ins would allow users to create more complex, turing-complete programs. Rotonda can compile, load MIR Code, and execute the code in virtual machines transparant to the user, and as a consequence the user can create, modify and remove Roto filter on-the-fly in a running Rotonda system.

1.45 Structure and scoping

1.45.1 Structure

The top-level of a `Roto` script can contain definitions of blocks. The type of these blocks can be:

- `filter-map`
- `filter`
- `rib`
- `table`
- `output-stream`

The syntax for all definitions of blocks looks like this:

```
filter my-filter { }
```

In this example, the type of the block is `filter`, the name is *my-filter*, and its body (the code between the curly braces) is empty.

The top-level can also contain type definitions for user-defined types. These look very similar to block definitions:

A type definition looks like this:

```
type MyRec {
    asn: Asn
}
```

Im this example, a type with the name `MyRec` is created, and is defined as *Record* with a field called `asn`, which is of type `Asn`.

Lastly, the top-level can contain single-line comments. Comments start with `//`.

1.45.2 Roto Global namespace and Block Scopes

For now, `Roto` only has one namespace, namely the Global namespace. Rotonda loads all available *Roto* source code from all files in a directory on the local file system specified in a Rotonda *configuration* file and puts all that source code in said global namespace. For the roto user this means that all identifiers, i.e. names of blocks, in the top-level of all loaded scripts must be unique across all these files. A failure to do so will result in a compilation error, and the compilation process will abort.

Although `Roto` only has one namespace, it does have multiple scopes. It has a global scope, which lives in the global namespace. All built-in types and user-defined types are situated in this global scope. Moreover, some constants are defined in the global namespace. All other scopes are nested inside the global scope, and nothing that is defined in the global scope with a name ("identifier"), be it block definitions, or type definitions, or constants, or built-in types, can be redefined ("shadowed") in the nested scopes. An attempt to do so will result in a compilation error, and the compilation process will be aborted.

Each block defined in the global namespace will have its own nested scope in its body. This means that a variable assignment in one block has no effect whatsoever on other blocks.

Consider this example:

```
// Global Scope is active here

filter a {
    // Nested Scope for filter a is active here
    define {
        pfx = 192.0.2.0/24;
        my_rec = MyRec {
            asn: 65534,
            prefix: pfx
        };
    }
}

filter b {
    // Nested Scope for filter b is active here
    define {
        pfx = 192.0.3.0/24;
        my_rec = MyRec {
            asn: 65535,
            prefix: pfx
        };
    }
}

// Global Scope is active here
type MyRec {
    asn: Asn,
    prefix: Prefix,
}
```

In this example we defined a type `MyRec`. That type is defined in the global scope and is therefore usable in all scopes. So in the nested scopes of filter a and filter b we can create new instances of this type. Filter a and filter b use the same variable names while creating these instances, but that does not matter: they are scoped to their own block, so they don't interfere.

1.46 Pattern Matching

1.47 Block `filter`

A `filter` block represents a filter that Rotonda will invoke in a Rib or Connector Unit, as specified in the *configuration* file of Rotonda.

A filter will be run for each payload that Rotonda receives in the unit where the filter is installed. The filter can access that payload and it must return a `filtering decision`. A filtering decision is either a `accept` or a `reject` expression.

The body of a `filter` block must contain only sections. It must contain exactly one `define` section, and exactly one `apply` section. Optionally it may contain one or more `term` sections and one or more `action` sections.

1.47.1 Section `define`

A `define` section in a `filter` contains all the variable declarations and assignments that can be used through-out the `filter` block. It must contain at least one declaration, and that is the declaration of the payload to a type and variable name, like so:

```
define {
    rx msg: BmpMessage;
}
```

The Roto `rx` expression is used to define the declaration of the payload. It is followed by the variable name that can be used to refer to it through-out the `Roto` script. The name is followed by a colon and the type of the payload. So, in this example we are referring to the payload with the variable name `msg` and its type is `BmpMessage`.

1.47.2 Section `term`

1.47.3 Section `action`

1.47.4 Section `apply`

1.48 Block `filter-map`

1.49 Block `rib`

1.50 Block `table`

1.51 Block `output-stream`

1.52 Unable to load Roto scripts

Upon starting Rotonda, an error similar to the one below is printed:

```
Unable to load Roto scripts:
read directory error for path /Users/jasper/Projects/rotonda/etc/filters:
No such file or directory (os error 2).
Roto filters 'rib-in-post-filter', 'bmp-in-filter', 'bgp-in-filter',
'rib-in-pre-filter' are referenced by your configuration but do not exist
because no .roto scripts could be loaded from the configured
`roto_scripts_path` directory '/etc/rotondafilters'. These filters will be ignored.
```

This is likely to happen when running `rotonda` by hand from a shell (as opposed to running it as a service after installing it from a package), without specifying a configuration file (with the `-c` option). In that case Rotonda falls back to the built-in configuration. This configuration expects to find specific filters in the absolute path `/etc/rotonda/`.

To obtain the required configuration files, refer to *Downloading the configuration files*.

INDEX