RTRTR User Manual

Release 0.2.1

NLnet Labs

Mar 29, 2022

CONTENTS

A versatile toolbox RTRTR is an RPKI data proxy, designed to collect Validated ROA Payloads from one or more sources in multiple formats and dispatch it onwards. It provides the means to implement multiple distribution architectures for RPKI such as centralised RPKI validators that dispatch data to local caching RTR servers.

Secure and redundant RTR connections RTRTR can read RPKI data from multiple RPKI Relying Party packages via RTR and JSON and, in turn, provide an RTR service for routers to connect to. The HTTP server provides the validated data set in JSON format, as well as a monitoring endpoint in plain text and Prometheus format. TLS is supported on all connections.

Open source with community and professional support NLnet Labs offers professional support services with a service-level agreement. We also provide a mailing list and Discord server for community support and to exchange operational experiences. RTRTR is liberally licensed under the BSD 3-Clause license.

CHAPTER

ONE

INSTALLATION

1.1 System Requirements

When choosing a system to run RTRTR on, make sure you have 1GB of available memory and 1GB of disk space.

1.2 Binary Packages

Getting started with RTRTR is really easy by installing a binary package for either Debian and Ubuntu or for Red Hat Enterprise Linux (RHEL) and compatible systems such as Rocky Linux. Alternatively, you can run with Docker.

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

Debian

Ubuntu

RHEL/CentOS

Docker

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

- 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` architecture packages for Debian/Raspbian Bullseye, and `arm64` for Buster.

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/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 RTRTR with:

```
sudo apt install rtrtr
```

Configure RTRTR by editing `/etc/rtrtr.conf` and start it with:

```
sudo systemctl enable --now rtrtr
```

You can check the status of RTRTR with:

```
sudo systemctl status rtrtr
```

You can view the logs with:

```
sudo journalctl --unit=rtrtr
```

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

- 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 RTRTR with:

```
sudo apt install rtrtr
```

Configure RTRTR by editing `/etc/rtrtr.conf` and start it with:

```
sudo systemctl enable --now rtrtr
```

You can check the status of RTRTR with:

```
sudo systemctl status rtrtr
```

You can view the logs with:

```
sudo journalctl --unit=rtrtr
```

To install an RTRTR 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
```

Then run the following command to add the public key:

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

You can then install RTRTR by running:

```
sudo yum install -y rtrtr
```

Configure RTRTR by editing `/etc/rtrtr.conf` and start it with:

```
sudo systemctl enable --now rtrtr
```

You can check the status of RTRTR with:

```
sudo systemctl status rtrtr
```

You can view the logs with:

```
sudo journalctl --unit=rtrtr
```

RTRTR Docker images are built with Alpine Linux for `amd64/x86_64` architecture.

To run RTRTR with Docker you will first need to create an `rtrtr.conf` file somewhere on your host computer and make that available to the Docker container when you run it. For example if your config file is in `/etc/rtrtr.conf` on the host computer:

```
docker run -v /etc/rtrtr.conf:/etc/rtrtr.conf nlnetlabs/rtrtr -c /etc/rtrtr.conf
```

RTRTR will need network access to fetch and publish data according to the configured units and targets respectively. Explaining Docker networking is beyond the scope of this Quick Start, however below are a couple of examples to get you started.

If you need an RTRTR unit to fetch data from a source port on the host you will also need to give the Docker container access to the host network. For example one way to do this is with `--net=host`, where `...` represents the rest of the arguments to pass to Docker and RTRTR:

```
docker run --net=host ...
```

If you're not using `--net=host` you will need to tell Docker to expose the RTRTR target ports, either one by one using `-p`, or you can publish the default ports exposed by the Docker container (and at the same time remap them to high numbered ports) using `-P`:

```
docker run -p 8080:8080/tcp -p 9001:9001/tcp ...
```

Or:

```
docker run -P ...
```

1.3 Updating

Debian

Ubuntu

RHEL/CentOS

Docker

To update an existing RTRTR 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 rtrtr
```

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

```
sudo apt --only-upgrade install rtrtr
```

To update an existing RTRTR 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 rtrtr
```

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

```
sudo apt --only-upgrade install rtrtr
```

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

```
sudo yum --showduplicates list rtrtr
```

You can update to the latest version using:

```
sudo yum update -y rtrtr
```

Upgrading to the latest version of RTRTR can be done with:

```
docker run -it nlnetlabs/rtrtr:latest
```

1.4 Installing Specific Versions

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

Debian

Ubuntu

RHEL/CentOS

Docker

If you would like to try out release candidates of RTRTR 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/debian \
$(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 rtrtr
```

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

```
sudo apt install rtrtr=0.1.1~rc2-1buster
```

If you would like to try out release candidates of RTRTR 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 rtrtr
```

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

```
sudo apt install rtrtr=0.1.1~rc2-1bionic
```

To install release candidates of RTRTR, 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 rtrtr
```

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

```
sudo yum install -y rtrtr-0.1.1
```

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

For example, installing RTRTR 0.1.1 is as simple as:

```
docker run -it nlnetlabs/rtrtr:v0.1.1
```

CHAPTER

TWO

BUILDING FROM SOURCE

In addition to meeting the *system requirements*, there are two things you need to build RTRTR: a C toolchain and Rust. You can run RTRTR on any operating system and CPU architecture where you can fulfil these requirements.

2.1 Dependencies

2.1.1 C Toolchain

Some of the libraries RTRTR 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:

```
apt install build-essential
```

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.

2.1.2 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, RTRTR relies on a relatively new version of Rust, currently 1.52 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
```

Alternatively, visit the Rust website for other installation methods.

2.2 Building and Updating

In Rust, a library or executable program such as RTRTR 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 RTRTR'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 RTRTR release from crates.io is as simple as running:

```
cargo install --locked rtrtr
```

The command will build RTRTR and install it in the same directory that Cargo itself lives in, likely `$HOME/.cargo/bin`. This means RTRTR will be in your path, too.

2.2.1 Updating

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

```
rustup update
```

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

```
cargo install --locked --force rtrtr
```

Once RTRTR is installed, you need to create a *Configuration* file that suits your needs. The config file to use needs to be passed to RTRTR via the `-c` option, i.e.:

```
rtrtr -c rtrtr.conf
```

2.2.2 Installing Specific Versions

If you want to install a specific version of RTRTR using Cargo, explicitly use the `--version` option. If needed, use the `--force` option to overwrite an existing version:

```
cargo install --locked --force rtrtr --version 0.2.0-rc2
```

All new features of RTRTR 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/rtrtr.git --branch main
```

See also:

For more installation options refer to the Cargo book.

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.

2.3.1 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
```

2.3.2 CentOS 6

The standard installation method does not work when using CentOS 6. Here, you will end up with a long list of error messages about missing assembler instructions. This is because the assembler shipped with CentOS 6 is too old.

You can get the necessary version by installing the Developer Toolset 6 from the Software Collections repository. On a virgin system, you can install Rust using these steps:

```
sudo yum install centos-release-scl
sudo yum install devtoolset-6
scl enable devtoolset-6 bash
curl https://sh.rustup.rs -sSf | sh
source $HOME/.cargo/env
```

CHAPTER

THREE

CONFIGURATION

RTRTR uses two classes of components: *units* and *targets*. Units take data from somewhere and produce a single, constantly updated data set. Targets take the data set from exactly one other unit and serve it in some specific way.

Both units and targets have a name — so that we can refer to them — and a type that defines which particular kind of unit or target this is. For each type, additional arguments need to be provided. Which these are and what they mean depends on the type.

Units and targets can be wired together in any way to achieve your specific goal. This is done in a configuration file, which also specifies several general parameters for logging, as well as status and Prometheus metrics endpoints via the built-in HTTP server.

Note: The configuration file is in TOML (Tom's Obvious Minimal Language) format, which is somewhat similar to INI files. You can find more information on the TOML website.

3.1 General Parameters

The configuration file starts out with a number of optional parameters to specify logging. The built-in HTTP server provides status information at the **/status** path and Prometheus metrics at the **/metrics** path. Note that details are provided for each unit and each target.

```
# The minimum log level to consider.
log_level = "debug"

# The target for logging. This can be "syslog", "stderr", "file", or "default".
log_target = "stderr"

# If syslog is used, the syslog facility can be given.
log_facility = "daemon"

# If file logging is used, the log file must be given.
log_file = "/var/log/rtrtr.log"

# Where should the HTTP server listen on?
http-listen = ["127.0.0.1:8080"]
```

3.2 Units

RTRTR currently has four types of units. Each unit gets its own section in the configuration. The name of the section, given in square brackets, starts with `units.` and is followed by a descriptive name you set, which you can later refer to from other units, or a target.

3.2.1 RTR Unit

The unit of the type `rtr` takes a feed of Validated ROA Payloads (VRPs) from a Relying Party software instance via the RTR protocol. Along with a unique name, the only required argument is the IP or hostname of the instance to connect to, along with the port.

Because the RTR protocol uses sessions and state, we don't need to specify a refresh interval for this unit. Should the server close the connection, by default RTRTR will retry every 60 seconds. This value is configurable wih the `retry` option.

```
[units.rtr-unit-name]
type = "rtr"
remote = "validator.example.net:3323"
```

It's also possible to configure RTR over TLS, using the `rtr-tls` unit type. When using this unit type, there is an additional configuration option, `cacerts`, which specifies a list of paths to files that contain one or more PEM encoded certificates that should be trusted when verifying a TLS server certificate.

The `rtr-tls` unit also uses the usual set of web trust anchors, so this option is only necessary when the RTR server doesn't use a server certificate that would be trusted by web browser. This is, for instance, the case if the server uses a self-signed certificate in which case this certificate needs to be added via this option.

3.2.2 JSON Unit

Most Relying Party software packages can produce the Validated ROA Payload set in JSON format as well, either as a file on disk or at an HTTP endpoint. RTRTR can use this format as a data source too, using units of the type `json`. Along with specifying a name, you must specify the URI to fetch the VRP set from, as well as the refresh interval in seconds.

```
[units.json-unit-name]
type = "json"
uri = "http://validator.example.net/vrps.json"
refresh = 60
```

3.2.3 Any Unit

The `any` unit type is given any number of *other* units and picks the data set from one of them. Units can signal that they currently don't have an up-to-date data set available, allowing the `any` unit to skip those. This ensures there is always an up-to-date data set available.

Important: The `any` unit uses a single data source at a time. RTRTR does **not** attempt to make a union or intersection of multiple VRPs sets, to avoid the risk of making a route *invalid* that would otherwise be *unknown*.

To configure this unit, specify a name, set the type to `any` and list the sources that should be used. Lastly, specify if a random unit should be selected every time it needs to switch or whether it should go through the list in order.

```
[units.any-unit-name]
type = "any"
sources = [ "unit-1", "unit-2", "unit-3" ]
random = false
```

3.2.4 SLURM Unit

In some cases, you may want to override the global RPKI data set with your own local exceptions. You can do this by specifying route origins that should be filtered out of the output, as well as origins that should be added, in a file using JSON notation according to the SLURM (Simplified Local Internet Number Resource Management with the RPKI) standard specified in **RFC 8416**.

You can refer to the JSON file you created with a unit of the type `slurm`. As the source to which the exceptions should be applied, you must specify any of the other units you have created. Note that the `files` attribute is an array and can take multiple values as input.

```
[units.slurm]
type = "slurm"
source = "source-unit-name"
files = [ "/var/lib/rtrtr/local-expections.json" ]
```

The Local Exceptions page in the Routinator documentation has more information on the format and syntax of SLURM files.

3.3 Targets

RTRTR currently has two types of targets. As with units, each unit gets its own section in the configuration. And also here, the name of the section starts with `targets.` and is followed by a descriptive name you set, all enclosed in square brackets.

3.3.1 RTR Target

Targets of the type `rtr` let you serve the data you collected with your units via the RPKI-to-Router (RTR) protocol. You must give your target a name and specify the host name or IP address it should listen on, along with the port. As the RTR target can listen on multiple addresses, the listen argument is a list. Lastly, you must specify the name of the unit the target should receive its data from.

```
[targets.rtr-target-name]
type = "rtr"
listen = [ "127.0.0.1:9001" ]
unit = "source-unit-name"
```

This target also supports TLS connections, via the `rtr-tls` type. This target has two additional configuration options. First, the `certificate` option, which is a string value providing a path to a file containing the PEM-encoded certificate to be used as the TLS server certificate. And secondly, there is the `key` option, which provides a path to a file containing the PEM-encoded certificate to be used as the private key by the TLS server.

3.3.2 HTTP Target

Targets of the type `http` let you serve the collected data via HTTP, which is currently only possible in `json` format. You can us this data stream for monitoring, provisioning, your IP address management, or any other purpose that you require. To use this target, specify a name and a path, as well as the name of the unit the target should receive its data from.

```
[targets.http-target-name]
type = "http"
path = "/json"
format = "json"
unit = "source-unit-name"
```

CHAPTER

FOUR

EXAMPLE SCENARIO

To make it clearer how you can deploy RTRTR, below is an example scenario. This flow may not be entirely realistic, but it intends to show all the different ways you can wire units and targets together using a visual representation and the configuration file needed to accomplish it.

In this example, there is routing infrastructure in a data centre labeled as `dc1`. To ensure redundancy, it gets Validated ROA Payloads (VRPs) primarily from relying party software running in the `eu-west-3` location, using the RTR protocol. There are two backups configured: a validator serving RTR in `ap-south-1` and an instance from another vendor offering a feed in JSON format in `us-east-2`. A unit of the type `any` is configured to get a feed from all three and, should the first one fail, do a round robin to the next available one.

To make the management of some statically configured routes for this location easy, the `slurm` unit gets its data from the `any` unit so only a single file has to be kept up-to-date.

Finally, an `http` target is configured to get the VRPs without the SLURM exceptions, to be fed into internal tooling and an `rtr` unit is defined to serve the routing infrastructure.

4.1 Configuration File

```
log_level = "debug"
log_target = "stderr"
log_facility = "daemon"
log_file = "/var/log/rtrtr.log"

http-listen = ["dc1.http.example.net:8080"]

# RTR UNITS

[units.eu-west-3]
type = "rtr"
remote = "paris.validator.example.net:3323"

[units.ap-south-1]
type = "rtr"
remote = "mumbai.validator.example.net:3323"

# JSON UNIT

[units.us-east-2]
type = "json"
uri = "https://ohio.validator.example.net/rpki.json"
```

(continues on next page)

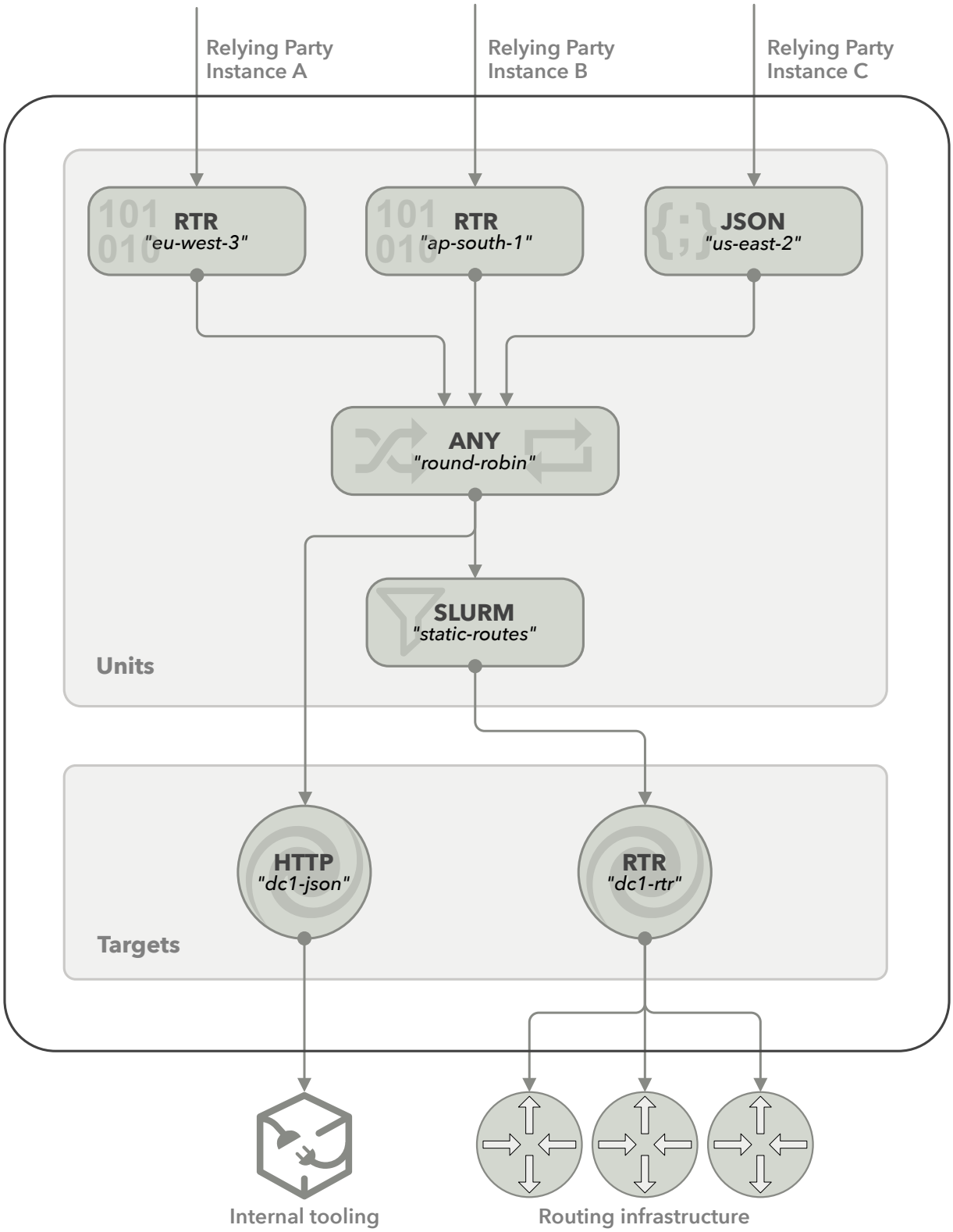

Fig. 1: Example of an RTRTR data flow

(continued from previous page)

```
refresh = 60

# ANY UNIT

[units.round-robin]
type = "any"
sources = [ "eu-west-3", "ap-south-1", "us-east-2" ]
random = false

# SLURM

[units.static-routes]
type = "slurm"
source = "round-robin"
files = [ "/var/lib/rtrtr/local-expections.json" ]

# RTR TARGET

[targets.dc1-rtr]
type = "rtr"
listen = [ "dc1.rtr.example.net:9001" ]
unit = "static-routes"

# JSON TARGET

[targets.dc1-json]
type = "http"
path = "/json"
format = "json"
unit = "round-robin"
```

CHAPTER

FIVE

MANUAL PAGE

5.1 Synopsis

rtrtr [options]

5.2 Description

RTRTR is an RPKI data proxy, designed to collect Validated ROA Payloads from one or more sources in multiple formats and dispatch it onwards. It provides the means to implement multiple distribution architectures for RPKI such as centralised RPKI validators that dispatch data to local caching RTR servers.

RTRTR can read RPKI data from multiple RPKI Relying Party packages via RTR and JSON and, in turn, provide an RTR service for routers to connect to. The HTTP server provides the validated data set in JSON format, as well as a monitoring endpoint in plain text and Prometheus format.

5.3 Options

-c path, **--config**=path
: Provides the path to a file containing the configuration for RTRTR. See *CONFIGURATION FILE* below for more information on the format and contents of the file.

  This option is required.

-v, **--verbose**
: Print more information. If given twice, even more information is printed.

  More specifically, a single `-v` increases the log level from the default of warn to *info*, specifying it more than once increases it to *debug*.

  See *LOGGING* below for more information on what information is logged at the different levels.

-q, **--quiet**
: Print less information. Given twice, print nothing at all.

  A single `-q` will drop the log level to *error*. Repeating `-q` more than once turns logging off completely.

--syslog
: Redirect logging output to syslog.

  This option is implied if a command is used that causes Routinator to run in daemon mode.

--syslog-facility=facility
: If logging to syslog is used, this option can be used to specify the syslog facility to use. The default is *daemon*.

`--logfile`=path
: Redirect logging output to the given file.

`-h, --help`
: Print some help information.

`-V, --version`
: Print version information.

5.4 Configuration File

The configuration file describes how and from where RTRTR is collecting data, how it processes it and how it should provide access to the resulting data set or data sets.

The configuration file is a file in TOML format. It consists of a sequence of key-value pairs, each on its own line. Strings are to be enclosed in double quotes. Lists can be given by enclosing a comma-separated list of values in square brackets. The file contains multiple sections, each started with a name enclosed in square brackets.

The first section without a name at the beginning of the file provides general configuration for RTRTR as a whole. It is followed by a single section for each component to be started.

There are two types of components: *units* and *targets*. Units take data from somewhere and produce a single, constantly updated data set. Targets take the data set from exactly one other unit and serve it in some specific way.

Both units and targets have a name and a type that defines which particular kind of unit or target this is. For each type, additional arguments need to be provided. Which these are and what they mean depends on the type.

The section of a component is named by appending the name of the component to its class. I.e., a unit named `foo` would have a section name of `[unit.foo]` while a target `bar` would have a section name of `[target.bar]`.

The following reference lists all configuration options for the global section as well as all options for each currently defined unit and target type. For each option it states the name, type, and purpose. Any relative path given as a configuration value is interpreted relative to the directory the configuration file is located in.

5.5 Global Options

http-listen A list of string values each specifying an address and port the HTTP server should listen on. Address and port should be separated by a colon. IPv6 address should be enclosed in square brackets.

RTRTR will listen on all address port combinations specified. All HTTP endpoints will be available on all of them.

log-level A string value specifying the maximum log level for which log messages should be emitted. The default is warn.

log A string specifying where to send log messages to. This can be one of the following values:

default Log messages will be sent to standard error if Routinator stays attached to the terminal or to syslog if it runs in daemon mode.

stderr Log messages will be sent to standard error.

syslog Log messages will be sent to syslog.

file Log messages will be sent to the file specified through the log-file configuration file entry.

The default if this value is missing is, unsurprisingly, default.

log-file A string value containing the path to a file to which log messages will be appended if the log configuration value is set to file. In this case, the value is mandatory.

syslog-facility A string value specifying the syslog facility to use for logging to syslog. The default value if this entry is missing is daemon.

5.6 RTR Units

There are two units that download RPKI data sets from an upstream server using the RPKI-to-Router protocol (RTR). The unit of type `"rtr"` uses unencrypted RTR while `"rtr-tls"` uses RTR over TLS.

The RTR units have the following configuration options:

remote A string value specifying the remote server to connect to. The string must contain both an address and a port separated by a colon. The address can be given as a an IP address, enclosed in square brackets for IPv6, or a host name.

For the `"rtr-tls"` unit, the address portion will be used to verify the server certificate against.

This option is mandatory.

retry An integer value specifying the number of seconds to wait before trying to reconnect to the server if it closed the connection.

If this option is missing, the default of 60 seconds is used.

cacerts Only used with the `"rtr-tls"` type, a list of paths to files that contain one or more PEM encoded certificates that should be trusted when verifying a TLS server certificate.

The `"rtr-tls"` unit also uses the usual set of web trust anchors, so this option is only necessary when the RTR server doesn't use a server certificate that would be trusted by web browser. This is, for instance, the case if the server uses a self-signed certificate in which case this certificate needs to be added via this option.

5.7 JSON Unit

A unit of type `"json"` imports and updates an RPKI data set through a JSON-encoded file. It accepts the JSON format used by most relying party packages.

The `"json"` unit has the following configuration options:

uri A string value specifying the location of the JSON file expressed as a URI.

If this is an `http:` or `https:` URI, the unit will download the file from the given location.

If this is a `file:` URI, the unit will load the given local file. Note that the unit just uses the path as given, so relative paths will interpreted relative to the current directory, whatever that may be.

refresh An integer value specifying the number of seconds to wait before attempting to re-fetch the file.

This value is used independently of whether the previous fetch has succeeded or not.

5.8 Any Unit

A unit of type `"any"` will pick one data set from one of a number of source units. The unit will only pick a source if it has an updated data set and can therefore be used to fall back to a different unit if one fails.

The `"any"` unit has the following configuration options:

sources A list of strings each containing the name of a unit to use as a source.

random A boolean value specifying whether the unit should pick a source unit at random. If the value is `false` or not given, the source units are picked in the order given.

5.9 SLURM Unit

A unit of type `"slurm"` will apply local exception rules to a data set provided by another unit. These rules are defined through local JSON files as described in **RFC 8416**. They allow to both filter out existing entries in a data set as well as add new entries.

The `"slurm"` unit has the following configuration options:

source A string value specifying the name of the unit that provides the data set to apply the local exceptions to.

files A list of strings each specifying the path to a local exception file.

The files are continously checked for updates, so RTRTR does not need to be restarted if the files are updated.

5.10 RTR Targets

There are two types of targets that provide a data set as an RTR server. The target of type `"rtr"` provides the data set over unencrypted RTR while the type `"rtr-tls"` offers the set through RTR over TLS.

The RTR targets have the following configuration options:

listen A list of string values each specifying an address and port the RTR target should listen on. Address and port should be separated by a colon. IPv6 address should be enclosed in square brackets.

unit A string value specifying the name of the unit that provides the data set for the RTR target to offer.

history-size An integer value specifying the number of diffs the target should keep in order to process RTR serial queries, i.e., the number of updates to the data set a client may fall behind before having to fetch the full data set again.

If this value is missing, it defaults to 10.

The `"rtr-tls"` target has the following *additional* configuration options:

certificate A string value providing a path to a file containing the PEM-encoded certificate to be used as the TLS server certificate.

key A string value providing a path to a file containing the PEM-encoded certificate to be used as the private key by the TLS server.

5.11 HTTP Target

A target of type `"http"` will offer the data set provided by a unit for download through the HTTP server.

The `"http"` target has the following configuration options:

path A string value specifying the path in the HTTP server under which the target should offer its data.

All HTTP targets share the same name space in RTRTR's global HTTP server. This value provides the path portion of HTTP URIs. It should start with a slash.

format A string value specifying the format of the data set to be offered. Currently, this has to be `"json"` for the JSON format.

unit A string value specifying the name of the unit that provides the data set for the RTR target to offer.

5.12 Logging

In order to allow diagnosis of the operation as well as its overall health, RTRTR logs an extensive amount of information. The log levels used by syslog are utilized to allow filtering this information for particular use cases.

The log levels represent the following information:

error Information related to events that prevent RTRTR from continuing to operate at all as well as all issues related to local configuration even if RTRTR will continue to run.

warn Information about events and data that influences the data sets produced by RTRTR. This includes failures to communicate with upstream servers, or encountering invalid data.

info Information about events and data that could be considered abnormal but do not influence the data set.

debug Information about the internal state of RTRTR that may be useful for debugging.

INDEX