FRR Developer's Manual

Release latest

FRR

Feb 15, 2023

CONTENTS

CHAPTER

ONE

PROCESS & WORKFLOW

FRR is a large project developed by many different groups. This section documents standards for code style & quality, commit messages, pull requests and best practices that all contributors are asked to follow.

This chapter is "descriptive/post-factual" in that it documents pratices that are in use; it is not "definitive/pre-factual" in prescribing practices. This means that when a procedure changes, it is agreed upon, then put into practice, and then documented here. If this document doesn't match reality, it's the document that needs to be updated, not reality.

1.1 Mailing Lists

The FRR development group maintains multiple mailing lists for use by the community. Italicized lists are private.

Topic	List
Development	dev@lists.frrouting.org
Users & Operators	frog@lists.frrouting.org
Announcements	announce@lists.frrouting.org
Security	security@lists.frrouting.org
Technical Steering Committee	tsc@lists.frrouting.org

The Development list is used to discuss and document general issues related to project development and governance. The public Slack instance and weekly technical meetings provide a higher bandwidth channel for discussions. The results of such discussions must be reflected in updates, as appropriate, to code (i.e., merges), GitHub issues, and for governance or process changes, updates to the Development list and either this file or information posted at https://frrouting.org/.

1.2 Development & Release Cycle

1.2.1 Development

The master Git for FRR resides on GitHub.

There is one main branch for development, `master`. For each major release (2.0, 3.0 etc) a new release branch is created based on the master. Significant bugfixes should be backported to upcoming and existing release branches no more than 1 year old. As a general rule new features are not backported to release branches.

Subsequent point releases based on a major branch are handled with git tags.

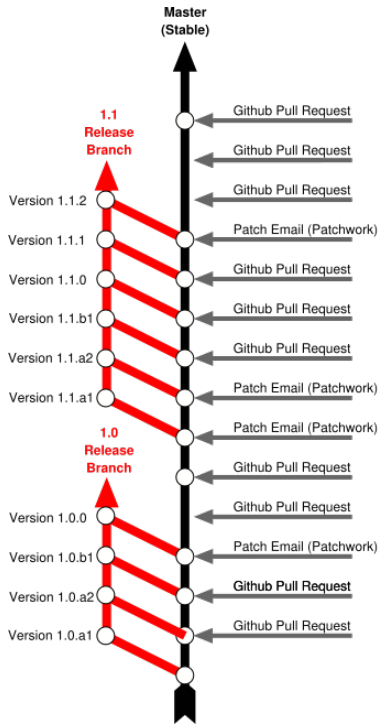


Fig. 1: Rough outline of FRR development workflow

1.2.2 Releases

FRR employs a `<MAJOR>.<MINOR>.<BUGFIX>` versioning scheme.

`MAJOR` Significant new features or multiple minor features. This should mostly cover any kind of disruptive change that is visible or "risky" to operators. New features or protocols do not necessarily trigger this. (This was changed for FRR 7.x after feedback from users that the pace of major version number increments was too high.)

`MINOR` General incremental development releases, excluding "major" changes mentioned above. Not necessarily fully backwards compatible, as smaller (but still visible) changes or deprecated feature removals may still happen. However, there shouldn't be any huge "surprises" between minor releases.

`BUGFIX` Fixes for actual bugs and/or security issues. Fully compatible.

Releases are scheduled in a 4-month cycle on the first Tuesday each March/July/November. Walking backwards from this date:

- 6 weeks earlier, `master` is frozen for new features, and feature PRs are considered lowest priority (regardless of when they were opened.)
- 4 weeks earlier, the stable branch separates from master (named `dev/MAJOR.MINOR` at this point) and tagged as `` `base_X.Y ``. Master is unfrozen and new features may again proceed.

  Part of unfreezing master is editing the `AC_INIT` statement in `configure.ac` to reflect the new development version that master now refers to. This is accompanied by a `frr-X.Y-dev` tag on master, which should always be on the first commit on master *after* the stable branch was forked (even if that is not the edit to `AC_INIT`; it's more important to have it on the very first commit on master after the fork.)

  (The `configure.ac` edit and tag push are considered git housekeeping and are pushed directly to `master`, not through a PR.)

  Below is the snippet of the commands to use in this step.

```
% git remote --verbose
upstream  git@github.com:frrouting/frr (fetch)
upstream  git@github.com:frrouting/frr (push)

% git checkout master
% git pull upstream master
% git checkout -b dev/8.2
% git tag base_8.2
% git push upstream base_8.2
% git push upstream dev/8.2
% git checkout master
% sed -i 's/8.2-dev/8.3-dev/' configure.ac
% git add configure.ac
% git commit -s -m "build: FRR 8.3 development version"
% git tag -a frr-8.3-dev -m "frr-8.3-dev"
% git push upstream master
% git push upstream frr-8.3-dev
```

In this step, we also have to update package versions to reflect the development version. Versions need to be updated using a standard way of development (Pull Requests) based on master branch.

Only change the version number with no other changes. This will produce packages with the a version number that is higher than any previous version. Once the release is done, whatever updates we make to changelog files on the release branch need to be cherry-picked to the master branch.

Update essential dates in advance for reference table (below) when the next freeze, dev/X.Y, RC, and release phases are scheduled. This should go in the `master` branch.

- 2 weeks earlier, a `frr-X.Y-rc` release candidate is tagged.

```
% git remote --verbose
upstream  git@github.com:frrouting/frr (fetch)
upstream  git@github.com:frrouting/frr (push)

% git checkout dev/8.2
% git tag frr-8.2-rc
% git push upstream frr-8.2-rc
```

- on release date, the branch is renamed to `stable/MAJOR.MINOR`.

The 2 week window between each of these events should be used to run any and all testing possible for the release in progress. However, the current intention is to stick to the schedule even if known issues remain. This would hopefully occur only after all avenues of fixing issues are exhausted, but to achieve this, an as exhaustive as possible list of issues needs to be available as early as possible, i.e. the first 2-week window.

For reference, the expected release schedule according to the above is:

Release	2023-03-07	2023-07-04	2023-10-31	2024-02-27	2024-06-25
RC	2023-02-21	2023-06-20	2023-10-17	2024-02-13	2024-06-11
dev/X.Y	2023-02-07	2023-06-06	2023-10-03	2024-01-30	2024-05-28
freeze	2023-01-24	2023-05-23	2023-09-19	2024-01-16	2024-05-14

Here is the hint on how to get the dates easily:

```
~$ # Last freeze date was 2023-09-19
~$ date +%F --date='2023-09-19 +119 days' # Next freeze date
2024-01-16
~$ date +%F --date='2024-01-16 +14 days'  # Next dev/X.Y date
2024-01-30
~$ date +%F --date='2024-01-30 +14 days'  # Next RC date
2024-02-13
~$ date +%F --date='2024-02-13 +14 days'  # Next Release date
2024-02-27
```

Each release is managed by one or more volunteer release managers from the FRR community. These release managers are expected to handle the branch for a period of one year. To spread and distribute this workload, this should be rotated for subsequent releases. The release managers are currently assumed/expected to run a release management meeting during the weeks listed above. Barring other constraints, this would be scheduled before the regular weekly FRR community call such that important items can be carried over into that call.

Bugfixes are applied to the two most recent releases. It is expected that each bugfix backported should include some reasoning for its inclusion as well as receiving approval by the release managers for that release before accepted into the release branch. This does not necessarily preclude backporting of bug fixes to older than the two most recent releases.

Security fixes are backported to all releases less than or equal to at least one year old. Security fixes may also be backported to older releases depending on severity.

For detailed instructions on how to produce an FRR release, refer to *FRR Release Procedure*.

1.2.3 Long term support branches (LTS)

This kind of branch is not yet officially supported, and need experimentation before being effective.

Previous definition of releases prevents long term support of previous releases. For instance, bug and security fixes are not applied if the stable branch is too old.

Because the FRR users have a need to backport bug and security fixes after the stable branch becomes too old, there is a need to provide support on a long term basis on that stable branch. If that support is applied on that stable branch, then that branch is a long term support branch.

Having a LTS branch requires extra-work and requires one person to be in charge of that maintenance branch for a certain amount of time. The amount of time will be by default set to 4 months, and can be increased. 4 months stands for the time between two releases, this time can be applied to the decision to continue with a LTS release or not. In all cases, that time period will be well-defined and published. Also, a self nomination from a person that proposes to handle the LTS branch is required. The work can be shared by multiple people. In all cases, there must be at least one person that is in charge of the maintenance branch. The person on people responsible for a maintenance branch must be a FRR maintainer. Note that they may choose to abandon support for the maintenance branch at any time. If no one takes over the responsibility of the LTS branch, then the support will be discontinued.

The LTS branch duties are the following ones:

- organise meetings on a (bi-)weekly or monthly basis, the handling of issues and pull requested relative to that branch. When time permits, this may be done during the regularly scheduled FRR meeting.
- ensure the stability of the branch, by using and eventually adapting the checking the CI tools of FRR (indeed, maintaining may lead to create maintenance branches for topotests or for CI).

It will not be possible to backport feature requests to LTS branches. Actually, it is a false good idea to use LTS for that need. Introducing feature requests may break the paradigm where all more recent releases should also include the feature request. This would require the LTS maintainer to ensure that all more recent releases have support for this feature request. Moreover, introducing features requests may result in breaking the stability of the branch. LTS branches are first done to bring long term support for stability.

1.2.4 Development Branches

Occassionally the community will desire the ability to work together on a feature that is considered useful to FRR. In this case the parties may ask the Maintainers for the creation of a development branch in the main FRR repository. Requirements for this to happen are:

- A one paragraph description of the feature being implemented to allow for the facilitation of discussion about the feature. This might include pointers to relevant RFC's or presentations that explain what is planned. This is intended to set a somewhat low bar for organization.
- A branch maintainer must be named. This person is responsible for keeping the branch up to date, and general communication about the project with the other FRR Maintainers. Additionally this person must already be a FRR Maintainer.
- Commits to this branch must follow the normal PR and commit process as outlined in other areas of this document. The goal of this is to prevent the current state where large features are submitted and are so large they are difficult to review.

After a development branch has completed the work together, a final review can be made and the branch merged into master. If a development branch is becomes un-maintained or not being actively worked on after three months then the Maintainers can decide to remove the branch.

1.2.5 Debian Branches

The Debian project contains "official" packages for FRR. While FRR Maintainers may participate in creating these, it is entirely the Debian project's decision what to ship and how to work on this.

As a courtesy and for FRR's benefit, this packaging work is currently visible in git branches named `debian/*` on the main FRR git repository. These branches are for the exclusive use by people involved in Debian packaging work for FRR. Direct commit access may be handed out and FRR git rules (review, testing, etc.) do not apply. Do not push to these branches without talking to the people noted under `Maintainer:` and `Uploaders:` in `debian/control` on the target branch – even if you are a FRR Maintainer.

1.2.6 Changelog

The changelog will be the base for the release notes. A changelog entry for your changes is usually not required and will be added based on your commit messages by the maintainers. However, you are free to include an update to the changelog with some better description.

1.3 Accords: non-code community consensus

The FRR repository has a place for "accords" - these are items of consideration for FRR that influence how we work as a community, but either haven't resulted in code *yet*, or may *never* result in code being written. They are placed in the `doc/accords/` directory.

The general idea is to simply pass small blurbs of text through our normal PR procedures, giving them the same visibility, comment and review mechanisms as code PRs - and changing them later is another PR. Please refer to the README file in `doc/accords/` for further details. The file names of items in that directory are hopefully helpful in determining whether some of them might be relevant to your work.

1.4 Submitting Patches and Enhancements

FRR accepts patches using GitHub pull requests.

The base branch for new contributions and non-critical bug fixes should be `master`. Please ensure your pull request is based on this branch when you submit it.

Code submitted by pull request will be automatically tested by one or more CI systems. Once the automated tests succeed, other developers will review your code for quality and correctness. After any concerns are resolved, your code will be merged into the branch it was submitted against.

The title of the pull request should provide a high level technical summary of the included patches. The description should provide additional details that will help the reviewer to understand the context of the included patches.

1.4.1 Squash commits

Before merging make sure a PR has squashed the following kinds of commits:

- Fixes/review feedback
- Typos
- Merges and rebases
- Work in progress

This helps to automatically generate human-readable changelog messages.

1.4.2 Commit Guidelines

There is a built-in commit linter. Basic rules:

- Commit messages must be prefixed with the name of the changed subsystem, followed by a colon and a space and start with an imperative verb.

  Check all the supported subsystems.

- Commit messages must start with a capital letter
- Commit messages must not end with a period `.`

1.4.3 Why was my pull request closed?

Pull requests older than 180 days will be closed. Exceptions can be made for pull requests that have active review comments, or that are awaiting other dependent pull requests. Closed pull requests are easy to recreate, and little work is lost by closing a pull request that subsequently needs to be reopened.

We want to limit the total number of pull requests in flight to:

- Maintain a clean project
- Remove old pull requests that would be difficult to rebase as the underlying code has changed over time
- Encourage code velocity

1.4.4 License for Contributions

FRR is under a "GPLv2 or later" license. Any code submitted must be released under the same license (preferred) or any license which allows redistribution under this GPLv2 license (eg MIT License). It is forbidden to push any code that prevents from using GPLv3 license. This becomes a community rule, as FRR produces binaries that links with Apache 2.0 libraries. Apache 2.0 and GPLv2 license are incompatible, if put together. Please see http://www.apache.org/licenses/GPL-compatibility.html for more information. This rule guarantees the user to distribute FRR binary code without any licensing issues.

1.4.5 Pre-submission Checklist

- Format code (see *Code Formatting*)
- Verify and acknowledge license (see *License for Contributions*)
- Ensure you have properly signed off (see *Signing Off*)
- Test building with various configurations:
  - `buildtest.sh`
- Verify building source distribution:
  - `make dist` (and try rebuilding from the resulting tar file)
- Run unit tests:
  - `make test`
- In the case of a major new feature or other significant change, document plans for continued maintenance of the feature. In addition it is a requirement that automated testing must be written that exercises the new feature within our existing CI infrastructure. Also the addition of automated testing to cover any pull request is encouraged.
- All new code must use the current latest version of acceptable code.
  - If a daemon is converted to YANG, then new code must use YANG.
  - DEFPY's must be used for new cli
  - Typesafe lists must be used
  - printf formatting changes must be used

1.4.6 Signing Off

Code submitted to FRR must be signed off. We have the same requirements for using the signed-off-by process as the Linux kernel. In short, you must include a `Signed-off-by` tag in every patch.

An easy way to do this is to use `git commit -s` where `-s` will automatically append a signed-off line to the end of your commit message. Also, if you commit and forgot to add the line you can use `git commit --amend -s` to add the signed-off line to the last commit.

`Signed-off-by` is a developer's certification that they have the right to submit the patch for inclusion into the project. It is an agreement to the *Developer's Certificate of Origin*. Code without a proper `Signed-off-by` line cannot and will not be merged.

If you are unfamiliar with this process, you should read the official policy at kernel.org. You might also find this article about participating in the Linux community on the Linux Foundation website to be a helpful resource.

In short, when you sign off on a commit, you assert your agreement to all of the following:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part by
    me, under the same open source license (unless I am permitted to
    submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

1.4.7 After Submitting Your Changes

- Watch for Continuous Integration (CI) test results
  - You should automatically receive an email with the test results within less than 2 hrs of the submission. If you don't get the email, then check status on the GitHub pull request.
  - Please notify the development mailing list if you think something doesn't work.
- If the tests failed:
  - In general, expect the community to ignore the submission until the tests pass.
  - It is up to you to fix and resubmit.
    - This includes fixing existing unit ("make test") tests if your changes broke or changed them.
    - It also includes fixing distribution packages for the failing platforms (ie if new libraries are required).
    - Feel free to ask for help on the development list.
  - Go back to the submission process and repeat until the tests pass.
- If the tests pass:
  - Wait for reviewers. Someone will review your code or be assigned to review your code.
  - Respond to any comments or concerns the reviewer has. Use e-mail or add a comment via github to respond or to let the reviewer know how their comment or concern is addressed.
  - An author must never delete or manually dismiss someone else's comments or review. (A review may be overridden by agreement in the weekly technical meeting.)
  - When you have addressed someone's review comments, please click the "re-request review" button (in the top-right corner of the PR page, next to the reviewer's name, an icon that looks like "reload")

  - The responsibility for keeping a PR moving rests with the author at least as long as there are either negative CI results or negative review comments. If you forget to mark a review comment as addressed (by clicking re-request review), the reviewer may very well not notice and won't come back to your PR.
  - Automatically generated comments, e.g., those generated by CI systems, may be deleted by authors and others when such comments are not the most recent results from that automated comment source.
  - After all comments and concerns are addressed, expect your patch to be merged.
- Watch out for questions on the mailing list. At this time there will be a manual code review and further (longer) tests by various community members.
- Your submission is done once it is merged to the master branch.

1.5 Programming Languages, Tools and Libraries

The core of FRR is written in C (gcc or clang supported) and makes use of GNU compiler extensions. A few non-essential scripts are implemented in Perl and Python. FRR requires the following tools to build distribution packages: automake, autoconf, texinfo, libtool and gawk and various libraries (i.e. libpam and libjson-c).

If your contribution requires a new library or other tool, then please highlight this in your description of the change. Also make sure it's supported by all FRR platform OSes or provide a way to build without the library (potentially without the new feature) on the other platforms.

Documentation should be written in reStructuredText. Sphinx extensions may be utilized but pure ReST is preferred where possible. See *Documentation*.

1.5.1 Use of C++

While C++ is not accepted for core components of FRR, extensions, modules or other distinct components may want to use C++ and include FRR header files. There is no requirement on contributors to work to retain C++ compatibility, but fixes for C++ compatibility are welcome.

This implies that the burden of work to keep C++ compatibility is placed with the people who need it, and they may provide it at their leisure to the extent it is useful to them. So, if only a subset of header files, or even parts of a header file are made available to C++, this is perfectly fine.

1.6 Code Reviews

Code quality is paramount for any large program. Consequently we require reviews of all submitted patches by at least one person other than the submitter before the patch is merged.

Because of the nature of the software, FRR's maintainer list (i.e. those with commit permissions) tends to contain employees / members of various organizations. In order to prevent conflicts of interest, we use an honor system in which submissions from an individual representing one company should be merged by someone unaffiliated with that company.

1.6.1 Guidelines for code review

- As a rule of thumb, the depth of the review should be proportional to the scope and / or impact of the patch.
- Anyone may review a patch.
- When using GitHub reviews, marking "Approve" on a code review indicates willingness to merge the PR.
- For individuals with merge rights, marking "Changes requested" is equivalent to a NAK.
- For a PR you marked with "Changes requested", please respond to updates in a timely manner to avoid impeding the flow of development.
- Rejected or obsolete PRs are generally closed by the submitter based on requests and/or agreement captured in a PR comment. The comment may originate with a reviewer or document agreement reached on Slack, the Development mailing list, or the weekly technical meeting.
- Reviewers may ask for new automated testing if they feel that the code change is large enough/significant enough to warrant such a requirement.

For project members with merge permissions, the following patterns have emerged:

- a PR with any reviews requesting changes may not be merged.
- a PR with any negative CI result may not be merged.
- an open "yellow" review mark ("review requested, but not done") should be given some time (a few days up to weeks, depending on the size of the PR), but is not a merge blocker.
- a "textbubble" review mark ("review comments, but not positive/negative") should be read through but is not a merge blocker.
- non-trivial PRs are generally given some time (again depending on the size) for people to mark an interest in reviewing. Trivial PRs may be merged immediately when CI is green.

1.7 Coding Practices & Style

1.7.1 Commit messages

Commit messages should be formatted in the same way as Linux kernel commit messages. The format is roughly:

```
dir: short summary

extended summary
```

`dir` should be the top level source directory under which the change was made. For example, a change in `bgpd/rfapi` would be formatted as:

```
bgpd: short summary

...
```

The first line should be no longer than 50 characters. Subsequent lines should be wrapped to 72 characters.

The purpose of commit messages is to briefly summarize what the commit is changing. Therefore, the extended summary portion should be in the form of an English paragraph. Brief examples of program output are acceptable but if present should be short (on the order of 10 lines) and clearly demonstrate what has changed. The goal should be that someone with only passing familiarity with the code in question can understand what is being changed.

Commit messages consisting entirely of program output are *unacceptable*. These do not describe the behavior changed. For example, putting VTYSH output or the result of test runs as the sole content of commit messages is unacceptable.

You must also sign off on your commit.

See also:

Signing Off

1.7.2 Source File Header

New files must have a copyright header (see *License for Contributions* above) added to the file. The header should be:

```
/*
 * Title/Function of file
 * Copyright (C) YEAR  Author's Name
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; see the file COPYING; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
 */

#include <zebra.h>
```

Please copy-paste this header verbatim. In particular:

- Do not replace "This program" with "FRR"
- Do not change the address of the FSF
- keep `#include <zebra.h>`. The absolute first header included in any C file **must** be either `zebra.h` or `config.h` (with HAVE_CONFIG_H guard)

1.7.3 Adding Copyright Claims to Existing Files

When adding copyright claims for modifications to an existing file, please add a `Portions:` section as shown below. If this section already exists, add your new claim at the end of the list.

```
/*
 * Title/Function of file
 * Copyright (C) YEAR  Author's Name
 * Portions:
 *     Copyright (C) 2010 Entity A ....
 *     Copyright (C) 2016 Your name [optional brief change description]
```

(continues on next page)

(continued from previous page)

```
 * ...
 */
```

1.7.4 Defensive coding requirements

In general, code submitted into FRR will be rejected if it uses unsafe programming practices. While there is no enforced overall ruleset, the following requirements have achieved consensus:

- `strcpy`, `strcat` and `sprintf` are unacceptable without exception. Use `strlcpy`, `strlcat` and `snprintf` instead. (Rationale: even if you know the operation cannot overflow the buffer, a future code change may inadvertedly introduce an overflow.)
- buffer size arguments, particularly to `strlcpy` and `snprintf`, must use `sizeof()` whereever possible. Particularly, do not use a size constant in these cases. (Rationale: changing a buffer to another size constant may leave the write operations on a now-incorrect size limit.)
- For stack allocated structs and arrays that should be zero initialized, prefer initializer expressions over `memset()` wherever possible. This helps prevent `memset()` calls being missed in branches, and eliminates the error class of an incorrect `size` argument to `memset()`.

  For example, instead of:

  ```
  struct foo mystruct;
  ...
  memset(&mystruct, 0x00, sizeof(struct foo));
  ```

  Prefer:

  ```
  struct foo mystruct = {};
  ```

- Do not zero initialize stack allocated values that must be initialized with a nonzero value in order to be used. This way the compiler and memory checking tools can catch uninitialized value use that would otherwise be suppressed by the (incorrect) zero initialization.

Other than these specific rules, coding practices from the Linux kernel as well as CERT or MISRA C guidelines may provide useful input on safe C code. However, these rules are not applied as-is; some of them expressly collide with established practice.

Container implementations

In particular to gain defensive coding benefits from better compiler type checks, there is a set of replacement container data structures to be found in `lib/typesafe.h`. They're documented under *Type-safe containers*.

Unfortunately, the FRR codebase is quite large, and migrating existing code to use these new structures is a tedious and far-reaching process (even if it can be automated with coccinelle, the patches would touch whole swaths of code and create tons of merge conflicts for ongoing work.) Therefore, little existing code has been migrated.

However, both **new code and refactors of existing code should use the new containers**. If there are any reasons this can't be done, please work to remove these reasons (e.g. by adding necessary features to the new containers) rather than falling back to the old code.

In order of likelyhood of removal, these are the old containers:

- `nhrpd/list.*`, `hlist_* DECLARE_LIST`
- `nhrpd/list.*`, `list_* DECLARE_DLIST`

- lib/skiplist.*, skiplist_* DECLARE_SKIPLIST
- lib/*_queue.h (BSD), SLIST_* DECLARE_LIST
- lib/*_queue.h (BSD), LIST_* DECLARE_DLIST
- lib/*_queue.h (BSD), STAILQ_* DECLARE_LIST
- lib/*_queue.h (BSD), TAILQ_* DECLARE_DLIST
- lib/hash.*, hash_* DECLARE_HASH
- lib/linklist.*, list_* DECLARE_DLIST
- open-coded linked lists DECLARE_LIST/DECLARE_DLIST

1.7.5 Code Formatting

C Code

For C code, FRR uses Linux kernel style except where noted below. Code which does not comply with these style guidelines will not be accepted.

The project provides multiple tools to allow you to correctly style your code as painlessly as possible, primarily built around `clang-format`.

clang-format In the project root there is a `.clang-format` configuration file which can be used with the `clang-format` source formatter tool from the LLVM project. Most of the time, this is the easiest and smartest tool to use. It can be run in a variety of ways. If you point it at a C source file or directory of source files, it will format all of them. In the LLVM source tree there are scripts that allow you to integrate it with `git`, `vim` and `emacs`, and there are third-party plugins for other editors. The `git` integration is particularly useful; suppose you have some changes in your git index. Then, with the integration installed, you can do the following:

```
git clang-format
```

This will format *only* the changes present in your index. If you have just made a few commits and would like to correctly style only the changes made in those commits, you can use the following syntax:

```
git clang-format HEAD~X
```

Where X is one more than the number of commits back from the tip of your branch you would like `clang-format` to look at (similar to specifying the target for a rebase).

The `vim` plugin is particularly useful. It allows you to select lines in visual line mode and press a key binding to invoke `clang-format` on only those lines.

When using `clang-format`, it is recommended to use the latest version. Each consecutive version generally has better handling of various edge cases. You may notice on occasion that two consecutive runs of `clang-format` over the same code may result in changes being made on the second run. This is an unfortunate artifact of the tool. Please check with the kernel style guide if in doubt.

One stylistic problem with the FRR codebase is the use of `DEFUN` macros for defining CLI commands. `clang-format` will happily format these macro invocations, but the result is often unsightly and difficult to read. Consequently, FRR takes a more relaxed position with how these are formatted. In general you should lean towards using the style exemplified in the section on *Command Line Interface*. Because `clang-format` mangles this style, there is a Python script named `tools/indent.py` that wraps `clang-format` and handles `DEFUN` macros as well as some other edge cases specific to FRR. If you are submitting a new file, it is recommended to run that script over the new file, preferably after ensuring that the latest stable release of `clang-format` is in your `PATH`.

Documentation on `clang-format` and its various integrations is maintained on the LLVM website.

https://clang.llvm.org/docs/ClangFormat.html

checkpatch.sh In the Linux kernel source tree there is a Perl script used to check incoming patches for style errors. FRR uses an adapted version of this script for the same purpose. It can be found at `tools/checkpatch.sh`. This script takes a git-formatted diff or patch file, applies it to a clean FRR tree, and inspects the result to catch potential style errors. Running this script on your patches before submission is highly recommended. The CI system runs this script as well and will comment on the PR with the results if style errors are found.

It is run like this:

```
./checkpatch.sh <patch> <tree>
```

Reports are generated on `stderr` and the exit code indicates whether issues were found (2, 1) or not (0).

Where `<patch>` is the path to the diff or patch file and `<tree>` is the path to your FRR source tree. The tree should be on the branch that you intend to submit the patch against. The script will make a best-effort attempt to save the state of your working tree and index before applying the patch, and to restore it when it is done, but it is still recommended that you have a clean working tree as the script does perform a hard reset on your tree during its run.

The script reports two classes of issues, namely WARNINGs and ERRORs. Please pay attention to both of them. The script will generally report WARNINGs where it cannot be 100% sure that a particular issue is real. In most cases WARNINGs indicate an issue that needs to be fixed. Sometimes the script will report false positives; these will be handled in code review on a case-by-case basis. Since the script only looks at changed lines, occasionally changing one part of a line can cause the script to report a style issue already present on that line that is unrelated to the change. When convenient it is preferred that these be cleaned up inline, but this is not required.

In general, a developer should heed the information reported by checkpatch. However, some flexibility is needed for cases where human judgement yields better clarity than the script. Accordingly, it may be appropriate to ignore some checkpatch.sh warnings per discussion among the submitter(s) and reviewer(s) of a change. Misreporting of errors by the script is possible. When this occurs, the exception should be handled either by patching checkpatch to correct the false error report, or by documenting the exception in this document under *Exceptions*. If the incorrect report is likely to appear again, a checkpatch update is preferred.

If the script finds one or more WARNINGs it will exit with 1. If it finds one or more ERRORs it will exit with 2.

Please remember that while FRR provides these tools for your convenience, responsibility for properly formatting your code ultimately lies on the shoulders of the submitter. As such, it is recommended to double-check the results of these tools to avoid delays in merging your submission.

In some cases, these tools modify or flag the format in ways that go beyond or even conflict[1] with the canonical documented Linux kernel style. In these cases, the Linux kernel style takes priority; non-canonical issues flagged by the tools are not compulsory but rather are opportunities for discussion among the submitter(s) and reviewer(s) of a change.

Whitespace changes in untouched parts of the code are not acceptable in patches that change actual code. To change/fix formatting issues, please create a separate patch that only does formatting changes and nothing else.

Kernel and BSD styles are documented externally:

- https://www.kernel.org/doc/html/latest/process/coding-style.html
- http://man.openbsd.org/style

For GNU coding style, use `indent` with the following invocation:

```
indent -nut -nfc1 file_for_submission.c
```

[1] For example, lines over 80 characters are allowed for text strings to make it possible to search the code for them: please see Linux kernel style (breaking long lines and strings) and Issue #1794.

Historically, FRR used fixed-width integral types that do not exist in any standard but were defined by most platforms at some point. Officially these types are not guaranteed to exist. Therefore, please use the fixed-width integral types introduced in the C99 standard when contributing new code to FRR. If you need to convert a large amount of code to use the correct types, there is a shell script in `tools/convert-fixedwidth.sh` that will do the necessary replacements.

Incorrect	Correct
u_int8_t	uint8_t
u_int16_t	uint16_t
u_int32_t	uint32_t
u_int64_t	uint64_t
u_char	uint8_t or unsigned char
u_short	unsigned short
u_int	unsigned int
u_long	unsigned long

FRR also uses unnamed struct fields, enabled with `-fms-extensions` (cf. https://gcc.gnu.org/onlinedocs/gcc/Unnamed-Fields.html). The following two patterns can/should be used where contextually appropriate:

```
struct outer {
        struct inner;
};
```

```
struct outer {
        union {
                struct inner;
                struct inner inner_name;
        };
};
```

Exceptions

FRR project code comes from a variety of sources, so there are some stylistic exceptions in place. They are organized here by branch.

For `master`:

BSD coding style applies to:

- `ldpd/`

`babeld` uses, approximately, the following style:

- K&R style braces
- Indents are 4 spaces
- Function return types are on their own line

For `stable/3.0` and `stable/2.0`:

GNU coding style apply to the following parts:

- `lib/`
- `zebra/`
- `bgpd/`

- `ospfd/`
- `ospf6d/`
- `isisd/`
- `ripd/`
- `ripngd/`
- `vtysh/`

BSD coding style applies to:

- `ldpd/`

Python Code

Format all Python code with black.

In a line:

```
python3 -m black <file.py>
```

Run this on any Python files you modify before committing.

FRR's Python code has been formatted with black version 19.10b.

YANG

FRR uses YANG to define data models for its northbound interface. YANG models should follow conventions used by the IETF standard models. From a practical standpoint, this corresponds to the output produced by the `yanglint` tool included in the `libyang` project, which is used by FRR to parse and validate YANG models. You should run the following command on all YANG documents you write:

```
yanglint -f yang <model>
```

The output of this command should be identical to the input file. The sole exception to this is comments. `yanglint` does not support comments and will strip them from its output. You may include comments in your YANG documents, but they should be indented appropriately (use spaces). Where possible, comments should be eschewed in favor of a suitable `description` statement.

In short, a diff between your input file and the output of `yanglint` should either be empty or contain only comments.

Specific Exceptions

Most of the time checkpatch errors should be corrected. Occasionally as a group maintainers will decide to ignore certain stylistic issues. Usually this is because correcting the issue is not possible without large unrelated code changes. When an exception is made, if it is unlikely to show up again and doesn't warrant an update to checkpatch, it is documented here.

Issue	Ignore Reason
DEFPY_HIDDEN, DEFPY_ATTR: complex macros should be wrapped in parentheses	DEF* macros cannot be wrapped in parentheses without updating all usages of the macro, which would be highly disruptive.

1.7.6 Types of configurables

Note: This entire section essentially just argues to not make configuration unnecessarily involved for the user. Rather than rules, this is more of a list of conclusions intended to help make FRR usable for operators.

Almost every feature FRR has comes with its own set of switches and options. There are several stages at which configuration can be applied. In order of preference, these are:

- at configuration/runtime, through YANG.

  This is the preferred way for all FRR knobs. Not all daemons and features are fully YANGified yet, so in some cases new features cannot rely on a YANG interface. If a daemon already implements a YANG interface (even partial), new CLI options must be implemented through a YANG model.

  Warning: Unlike everything else in this section being guidelines with some slack, implementing and using a YANG interface for new CLI options in (even partially!) YANGified daemons is a hard requirement.

- at configuration/runtime, through the CLI.

  The "good old" way for all regular configuration. More involved for users to automate *correctly* than YANG.

- at startup, by loading additional modules.

  If a feature introduces a dependency on additional libraries (e.g. libsnmp, rtrlib, etc.), this is the best way to encapsulate the dependency. Having a separate module allows the distribution to create a separate package with the extra dependency, so FRR can still be installed without pulling everything in.

  A module may also be appropriate if a feature is large and reasonably well isolated. Reducing the amount of running the code is a security benefit, so even if there are no new external dependencies, modules can be useful.

  While modules cannot currently be loaded at runtime, this is a tradeoff decision that was made to allow modules to change/extend code that is very hard to (re)adjust at runtime. If there is a case for runtime (un)loading of modules, this tradeoff can absolutely be reevaluated.

- at startup, with command line options.

  This interface is only appropriate for options that have an effect very early in FRR startup, i.e. before configuration is loaded. Anything that affects configuration load itself should be here, as well as options changing the environment FRR runs in.

  If a tunable can be changed at runtime, a command line option is only acceptable if the configured value has an effect before configuration is loaded (e.g. zebra reads routes from the kernel before loading config, so the netlink buffer size is an appropriate command line option.)

- at compile time, with `./configure` options.

  This is the absolute last preference for tunables, since the distribution needs to make the decision for the user and/or the user needs to rebuild FRR in order to change the option.

  "Good" configure options do one of three things:

  - set distribution-specific parameters, most prominently all the path options. File system layout is a distribution/packaging choice, so the user would hopefully never need to adjust these.
  - changing toolchain behavior, e.g. instrumentation, warnings, optimizations and sanitizers.
  - enabling/disabling parts of the build, especially if they need additional dependencies. Being able to build only parts of FRR, or without some library, is useful. **The only effect these options should have is adding**

or removing files from the build result. If a knob in this category causes the same binary to exist in different variants, it is likely implemented incorrectly!

Note: This last guideline is currently ignored by several configure options. `vtysh` in general depends on the entire list of enabled daemons, and options like `--enable-bgp-vnc` and `--enable-ospfapi` change daemons internally. Consider this more of an "ideal" than a "rule".

Whenever adding new knobs, please try reasonably hard to go up as far as possible on the above list. Especially `./configure` flags are often enough the "easy way out" but should be avoided when at all possible. To a lesser degree, the same applies to command line options.

1.7.7 Compile-time conditional code

Many users access FRR via binary packages from 3rd party sources; compile-time code puts inclusion/exclusion in the hands of the package maintainer. Please think very carefully before making code conditional at compile time, as it increases regression testing, maintenance burdens, and user confusion. In particular, please avoid gratuitous `--enable-...` switches to the configure script - in general, code should be of high quality and in working condition, or it shouldn't be in FRR at all.

When code must be compile-time conditional, try have the compiler make it conditional rather than the C pre-processor so that it will still be checked by the compiler, even if disabled. For example,

```
if (SOME_SYMBOL)
      frobnicate();
```

is preferred to

```
#ifdef SOME_SYMBOL
frobnicate ();
#endif /* SOME_SYMBOL */
```

Note that the former approach requires ensuring that `SOME_SYMBOL` will be defined (watch your `AC_DEFINE`s).

1.7.8 Debug-guards in code

Debugging statements are an important methodology to allow developers to fix issues found in the code after it has been released. The caveat here is that the developer must remember that people will be using the code at scale and in ways that can be unexpected for the original implementor. As such debugs **MUST** be guarded in such a way that they can be turned off. FRR has the ability to turn on/off debugs from the CLI and it is expected that the developer will use this convention to allow control of their debugs.

1.7.9 Custom syntax-like block macros

FRR uses some macros that behave like the `for` or `if` C keywords. These macros follow these patterns:

- loop-style macros are named `frr_each_*` (and `frr_each`)
- single run macros are named `frr_with_*`
- to avoid confusion, `frr_with_*` macros must always use a `{ ... }` block even if the block only contains one statement. The `frr_each` constructs are assumed to be well-known enough to use normal `for` rules.
- `break`, `return` and `goto` all work correctly. For loop-style macros, `continue` works correctly too.

Both the `each` and `with` keywords are inspired by other (more higher-level) programming languages that provide these constructs.

There are also some older iteration macros, e.g. `ALL_LIST_ELEMENTS` and `FOREACH_AFI_SAFI`. These macros in some cases do **not** fulfill the above pattern (e.g. `break` does not work in `FOREACH_AFI_SAFI` because it expands to 2 nested loops.)

1.7.10 Static Analysis and Sanitizers

Clang/LLVM and GCC come with a variety of tools that can be used to help find bugs in FRR.

clang-analyze This is a static analyzer that scans the source code looking for patterns that are likely to be bugs. The tool is run automatically on pull requests as part of CI and new static analysis warnings will be placed in the CI results. FRR aims for absolutely zero static analysis errors. While the project is not quite there, code that introduces new static analysis errors is very unlikely to be merged.

AddressSanitizer This is an excellent tool that provides runtime instrumentation for detecting memory errors. As part of CI FRR is built with this instrumentation and run through a series of tests to look for any results. Testing your own code with this tool before submission is encouraged. You can enable it by passing:

```
--enable-address-sanitizer
```

to `configure`.

ThreadSanitizer Similar to AddressSanitizer, this tool provides runtime instrumentation for detecting data races. If you are working on or around multithreaded code, extensive testing with this instrumtation enabled is *highly* recommended. You can enable it by passing:

```
--enable-thread-sanitizer
```

to `configure`.

MemorySanitizer Similar to AddressSanitizer, this tool provides runtime instrumentation for detecting use of uninitialized heap memory. Testing your own code with this tool before submission is encouraged. You can enable it by passing:

```
--enable-memory-sanitizer
```

to `configure`.

All of the above tools are available in the Clang/LLVM toolchain since 3.4. AddressSanitizer and ThreadSanitizer are available in recent versions of GCC, but are no longer actively maintained. MemorySanitizer is not available in GCC.

Note: The different Sanitizers are mostly incompatible with each other. Please refer to GCC/LLVM documentation for details.

---

frr-format plugin This is a GCC plugin provided with FRR that does extended type checks for `%pFX`-style printfrr extensions. To use this plugin,

1. install GCC plugin development files, e.g.:

   ```
   apt-get install gcc-10-plugin-dev
   ```

2. **before** running `configure`, compile the plugin with:

```
make -C tools/gcc-plugins CXX=g++-10
```

(Edit the GCC version to what you're using, it should work for GCC 9 or newer.)

After this, the plugin should be automatically picked up by `configure`. The plugin does not change very frequently, so you can keep it around across work on different FRR branches. After a `git clean -x`, the `make` line will need to be run again. You can also add `--with-frr-format` to the `configure` line to make sure the plugin is used, otherwise if something is not set up correctly it might be silently ignored.

> **Warning:** Do **not** enable this plugin for package/release builds. It is intended for developer/debug builds only. Since it modifies the compiler, it may cause silent corruption of the executable files.
>
> Using the plugin also changes the string for `PRI[udx]64` from the system value to `%L[udx]` (normally `%ll[udx]` or `%l[udx]`.)

Additionally, the FRR codebase is regularly scanned with Coverity. Unfortunately Coverity does not have the ability to handle scanning pull requests, but after code is merged it will send an email notifying project members with Coverity access of newly introduced defects.

1.7.11 Executing non-installed dynamic binaries

Since FRR uses the GNU autotools build system, it inherits its shortcomings. To execute a binary directly from the build tree under a wrapper like *valgrind*, *gdb* or *strace*, use:

```
./libtool --mode=execute valgrind [--valgrind-opts] zebra/zebra [--zebra-opts]
```

While replacing valgrind/zebra as needed. The *libtool* script is found in the root of the build directory after *./configure* has completed. Its purpose is to correctly set up *LD_LIBRARY_PATH* so that libraries from the build tree are used. (On some systems, *libtool* is also available from PATH, but this is not always the case.)

1.7.12 CLI changes

CLI's are a complicated ugly beast. Additions or changes to the CLI should use a DEFPY to encapsulate one setting as much as is possible. Additionally as new DEFPY's are added to the system, documentation should be provided for the new commands.

1.7.13 Backwards Compatibility

As a general principle, changes to CLI and code in the lib/ directory should be made in a backwards compatible fashion. This means that changes that are purely stylistic in nature should be avoided, e.g., renaming an existing macro or library function name without any functional change. When adding new parameters to common functions, it is also good to consider if this too should be done in a backward compatible fashion, e.g., by preserving the old form in addition to adding the new form.

This is not to say that minor or even major functional changes to CLI and common code should be avoided, but rather that the benefit gained from a change should be weighed against the added cost/complexity to existing code. Also, that when making such changes, it is good to preserve compatibility when possible to do so without introducing maintenance overhead/cost. It is also important to keep in mind, existing code includes code that may reside in private repositories (and is yet to be submitted) or code that has yet to be migrated from Quagga to FRR.

That said, compatibility measures can (and should) be removed when either:

- they become a significant burden, e.g. when data structures change and the compatibility measure would need a complex adaptation layer or becomes flat-out impossible
- some measure of time (dependent on the specific case) has passed, so that the compatibility grace period is considered expired.

For CLI commands, the deprecation period is 1 year.

In all cases, compatibility pieces should be marked with compiler/preprocessor annotations to print warnings at compile time, pointing to the appropriate update path. A `-Werror` build should fail if compatibility bits are used. To avoid compilation issues in released code, such compiler/preprocessor annotations must be ignored non-development branches. For example:

```
#if CONFDATE > 20180403
CPP_NOTICE("Use of <XYZ> is deprecated, please use <ABC>")
#endif
```

Preferably, the shell script `tools/fixup-deprecated.py` will be updated along with making non-backwards compatible code changes, or an alternate script should be introduced, to update the code to match the change. When the script is updated, there is no need to preserve the deprecated code. Note that this does not apply to user interface changes, just internal code, macros and libraries.

1.7.14 Miscellaneous

When in doubt, follow the guidelines in the Linux kernel style guide, or ask on the development mailing list / public Slack instance.

JSON Output

New JSON output in FRR needs to be backed by schema, in particular a YANG model. When adding new JSON, first search for an existing YANG model, either in FRR or a standard model (e.g., IETF) and use that model as the basis for any JSON structure and *especially* for key names and canonical values formats.

If no YANG model exists to support the JSON then an FRR YANG model needs to be added to or created to support the JSON format.

- All JSON keys are to be `camelCased`, with no spaces. YANG modules almost always use `kebab-case` (i.e., all lower case with hyphens to separate words), so these identifiers need to be mapped to `camelCase` by removing the hyphen (or symbol) and capitalizing the following letter, for example "router-id" becomes "routerId"
- Commands which output JSON should produce `{}` if they have nothing to display
- In general JSON commands include a `json` keyword typically at the end of the CLI command (e.g., `show ip ospf json`)

Use of const

Please consider using `const` when possible: it's a useful hint to callers about the limits to side-effects from your apis, and it makes it possible to use your apis in paths that involve `const` objects. If you encounter existing apis that *could* be `const`, consider including changes in your own pull-request.

Help with specific warnings

FRR's configure script enables a whole batch of extra warnings, some of which may not be obvious in how to fix. Here are some notes on specific warnings:

- `-Wstrict-prototypes`: you probably just forgot the `void` in a function declaration with no parameters, i.e. `static void foo() {...}` rather than `static void foo(void) {...}`.

  Without the `void`, in C, it's a function with *unspecified* parameters (and varargs calling convention.) This is a notable difference to C++, where the `void` is optional and an empty parameter list means no parameters.

- `"strict match required"` from the frr-format plugin: check if you are using a cast in a printf parameter list. The frr-format plugin cannot access correct full type information for casts like `printfrr(... , (uint64_t)something, ...)` and will print incorrect warnings particularly if `uint64_t`, `size_t` or `ptrdiff_t` are involved. The problem is *not* triggered with a variable or function return value of the exact same type (without a cast).

  Since these cases are very rare, community consensus is to just work around the warning even though the code might be correct. If you are running into this, your options are:

  1. try to avoid the cast altogether, maybe using a different printf format specifier (e.g. `%lu` instead of `%zu` or `PRIu64`).
  2. fix the type(s) of the function/variable/struct member being printed
  3. create a temporary variable with the value and print that without a cast (this is the last resort and was not necessary anywhere so far.)

1.8 Documentation

FRR uses Sphinx+RST as its documentation system. The document you are currently reading was generated by Sphinx from RST source in `doc/developer/workflow.rst`. The documentation is structured as follows:

Directory	Contents
`doc/user`	User documentation; configuration guides; protocol overviews
`doc/developer`	Developer's documentation; API specs; datastructures; architecture overviews; project management procedure
`doc/manpages`	Source for manpages
`doc/figures`	Images and diagrams
`doc/extra`	Miscellaneous Sphinx extensions, scripts, customizations, etc.

Each of these directories, with the exception of `doc/figures` and `doc/extra`, contains a Sphinx-generated Makefile and configuration script `conf.py` used to set various document parameters. The makefile can be used for a variety of targets; invoke *make help* in any of these directories for a listing of available output formats. For convenience, there is a top-level `Makefile.am` that has targets for PDF and HTML documentation for both developer and user documentation, respectively. That makefile is also responsible for building manual pages packed with distribution builds.

Indent and styling should follow existing conventions:

- 3 spaces for indents under directives
- Cross references may contain only lowercase alphanumeric characters and hyphens ('-')
- Lines wrapped to 80 characters where possible

Characters for header levels should follow Python documentation guide:

- # with overline, for parts
- * with overline, for chapters
- =, for sections
- -, for subsections
- ^, for subsubsections
- ", for paragraphs

After you have made your changes, please make sure that you can invoke `make latexpdf` and `make html` with no warnings.

The documentation is currently incomplete and needs love. If you find a broken cross-reference, figure, dead hyperlink, style issue or any other nastiness we gladly accept documentation patches.

To build the docs, please ensure you have installed a recent version of Sphinx. If you want to build LaTeX or PDF docs, you will also need a full LaTeX distribution installed.

1.8.1 Code

FRR is a large and complex software project developed by many different people over a long period of time. Without adequate documentation, it can be exceedingly difficult to understand code segments, APIs and other interfaces. In the interest of keeping the project healthy and maintainable, you should make every effort to document your code so that other people can understand what it does without needing to closely read the code itself.

Some specific guidelines that contributors should follow are:

- Functions exposed in header files should have descriptive comments above their signatures in the header file. At a minimum, a function comment should contain information about the return value, parameters, and a general summary of the function's purpose. Documentation on parameter values can be omitted if it is (very) obvious what they are used for.

  Function comments must follow the style for multiline comments laid out in the kernel style guide.

  Example:

```
/*
 * Determines whether or not a string is cool.
 *
 * text
 *    the string to check for coolness
 *
 * is_clccfc
 *    whether capslock is cruise control for cool
 *
 * Returns:
 *    7 if the text is cool, 0 otherwise
 */
int check_coolness(const char *text, bool is_clccfc);
```

  Function comments should make it clear what parameters and return values are used for.

- Static functions should have descriptive comments in the same form as above if what they do is not immediately obvious. Use good engineering judgement when deciding whether a comment is necessary. If you are unsure, document your code.
- Global variables, static or not, should have a comment describing their use.

- **For new code in lib/, these guidelines are hard requirements.**

If you make significant changes to portions of the codebase covered in the Developer's Manual, add a major subsystem or feature, or gain arcane mastery of some undocumented or poorly documented part of the codebase, please document your work so others can benefit. If you add a major feature or introduce a new API, please document the architecture and API to the best of your abilities in the Developer's Manual, using good judgement when choosing where to place it.

Finally, if you come across some code that is undocumented and feel like going above and beyond, document it! We absolutely appreciate and accept patches that document previously undocumented code.

1.8.2 User

If you are contributing code that adds significant user-visible functionality please document how to use it in `doc/user`. Use good judgement when choosing where to place documentation. For example, instructions on how to use your implementation of a new BGP draft should go in the BGP chapter instead of being its own chapter. If you are adding a new protocol daemon, please create a new chapter.

1.8.3 FRR Specific Markup

FRR has some customizations applied to the Sphinx markup that go a long way towards making documentation easier to use, write and maintain.

CLI Commands

When documenting CLI please use the `.. clicmd::` directive. This directive will format the command and generate index entries automatically. For example, the command `show pony` would be documented as follows:

```
.. clicmd:: show pony

   Prints an ASCII pony. Example output::

            >>\.
           /_  )`.
          /  _)`^)`.   _.---. _
         (_,' \  `^-)""      `.\
               |  | \
               \              / |
              / \  /.___.'\  (\ (_
             < ,"||     \ |`. \`-'
              \\ ()      )|  )/
     hjw      |_>|>     /_] //
                /_]        /_]
```

When documented this way, CLI commands can be cross referenced with the `:clicmd:` inline markup like so:

```
:clicmd:`show pony`
```

This is very helpful for users who want to quickly remind themselves what a particular command does.

When documenting a cli that has a `no` form, please do not include the `no` form. I.e. `no show pony` would not be documented anywhere. Since most commands have `no` forms, users should be able to infer these or get help from vtysh's completions.

When documenting commands that have lots of possible variants, just document the single command in summary rather than enumerating each possible variant. E.g. for `show pony [foo|bar]`, do not:

```
.. clicmd:: show pony
.. clicmd:: show pony foo
.. clicmd:: show pony bar
```

Do:

```
.. clicmd:: show pony [foo|bar]
```

Configuration Snippets

When putting blocks of example configuration please use the `.. code-block::` directive and specify `frr` as the highlighting language, as in the following example. This will tell Sphinx to use a custom Pygments lexer to highlight FRR configuration syntax.

```
.. code-block:: frr

   !
   ! Example configuration file.
   !
   log file /tmp/log.log
   service integrated-vtysh-config
   !
   ip route 1.2.3.0/24 reject
   ipv6 route de:ea:db:ee:ff::/64 reject
   !
```

CHAPTER

TWO

BUILDING FRR

2.1 Static Linking

This document describes how to build FRR without hard dependencies on shared libraries. Note that it's not possible to build FRR *completely* statically. This document just covers how to statically link the dependencies that aren't likely to be present on a given platform - libfrr and libyang. The resultant binaries should still be fairly portable. For example, here is the DSO dependency list for *bgpd* after using these steps:

```
$ ldd bgpd
     linux-vdso.so.1 (0x00007ffe3a989000)
     libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f9dc10c0000)
     libcap.so.2 => /lib/x86_64-linux-gnu/libcap.so.2 (0x00007f9dc0eba000)
     libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f9dc0b1c000)
     libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9dc0918000)
     libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1 (0x00007f9dc06e0000)
     libjson-c.so.3 => /lib/x86_64-linux-gnu/libjson-c.so.3 (0x00007f9dc04d5000)
     librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f9dc02cd000)
     libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9dc00ae000)
     libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f9dbfe96000)
     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9dbfaa5000)
     /lib64/ld-linux-x86-64.so.2 (0x00007f9dc1449000)
```

2.1.1 Procedure

Note that these steps have only been tested with LLVM 9 / clang.

Today, libfrr can already be statically linked by passing these configure options:

```
--enable-static --enable-static-bin --enable-shared
```

libyang is more complicated. You must build and install libyang as a static library. To do this, follow the usual libyang build procedure as listed in the FRR developer docs, but set the `ENABLE_STATIC` option in your cmake invocation. You also need to build with PIC enabled, which today is disabled when building libyang statically.

The resultant cmake command is:

```
cmake -DENABLE_STATIC=ON -DENABLE_LYD_PRIV=ON \
      -DCMAKE_INSTALL_PREFIX:PATH=/usr \
      -DCMAKE_POSITION_INDEPENDENT_CODE=TRUE \
      -DCMAKE_BUILD_TYPE:String="Release" ..
```

This produces a bunch of `.a` static archives that need to ultimately be linked into FRR. However, not only is it 6 archives rather than the usual `libyang.so`, you will now also need to link FRR with `libpcre.a`. Ubuntu's `libpcre3-dev` package provides this, but it hasn't been built with PIC enabled, so it's not usable for our purposes. So download `libpcre` from SourceForge, and build it like this:

```
./configure --with-pic
make
```

Hopefully you get a nice, usable, PIC `libpcre.a`.

So now we have to link all these static libraries into FRR. Rather than modify FRR to accommodate this, the best option is to create an archive with all of libyang's dependencies. Then to avoid making any changes to FRR build foo, rename this `libyang.a` and copy it over the usual static library location. Ugly but it works. To do this, go into your libyang build directory, which should have a bunch of `.a` files. Copy `libpcre.a` into this directory. Write the following into a shell script and run it:

```
#!/bin/bash
ar -M <<EOM
  CREATE libyang_fat.a
  ADDLIB libyang.a
  ADDLIB libyangdata.a
  ADDLIB libmetadata.a
  ADDLIB libnacm.a
  ADDLIB libuser_inet_types.a
  ADDLIB libuser_yang_types.a
  ADDLIB libpcre.a
  SAVE
  END
EOM
ranlib libyang_fat.a
```

`libyang_fat.a` is your archive. Now copy this over your install `libyang.a`, which on my machine is located at `/usr/lib/x86_64-linux-gnu/libyang.a` (try `locate libyang.a` if not).

Now when you build FRR with the static options enabled as above, clang should pick up the static libyang and link it, leaving you with FRR binaries that have no hard DSO dependencies beyond common system libraries. To verify, run `ldd` over the resultant binaries.

2.2 Alpine Linux 3.7+

For building Alpine Linux dev packages, we use docker.

2.2.1 Install docker 17.05 or later

Depending on your host, there are different ways of installing docker. Refer to the documentation here for instructions on how to install a free version of docker: https://www.docker.com/community-edition

2.2.2 Pre-built packages and docker images

The master branch of https://github.com/frrouting/frr.git has a continuous delivery of docker images to docker hub at: https://hub.docker.com/r/ajones17/frr/. These images have the frr packages in /pkgs/apk and have the frr package pre-installed. To copy Alpine packages out of these images:

```
id=`docker create ajones17/frr:latest`
docker cp ${id}:/pkgs _some_directory_
docker rm $id
```

To run the frr daemons (see below for how to configure them):

```
docker run -it --rm --name frr ajones17/frr:latest
docker exec -it frr /bin/sh
```

2.2.3 Work with sources

```
git clone https://github.com/frrouting/frr.git frr
cd frr
```

2.2.4 Build apk packages

```
./docker/alpine/build.sh
```

This will put the apk packages in:

```
./docker/pkgs/apk/x86_64/
```

2.2.5 Usage

To create a base image with the frr packages installed:

```
docker build --rm -f docker/alpine/Dockerfile -t frr:latest .
```

Or, if you don't have a git checkout of the sources, you can build a base image directly off the github account:

```
docker build --rm -f docker/alpine/Dockerfile -t frr:latest \
    https://github.com/frrouting/frr.git
```

And to run the image:

```
docker run -it --rm --name frr frr:latest
```

In the default configuration, none of the frr daemons will be running. To configure the daemons, exec into the container and edit the configuration files or mount a volume with configuration files into the container on startup. To configure by hand:

```
docker exec -it frr /bin/sh
vi /etc/frr/daemons
/etc/init.d/frr start
```

Or, to configure the daemons using /etc/frr from a host volume, put the config files in, say, ./docker/etc and bind mount that into the container:

```
docker run -it --rm -v `pwd`/docker/etc:/etc/frr frr:latest
```

We can also build the base image directly from docker-compose, with a docker-compose.yml file like this one:

```
version: '2.2'

services:
   frr:
      build:
         context: https://github.com/frrouting/frr.git
         dockerfile: docker/alpine/Dockerfile
```

2.3 CentOS 6

This document describes installation from source. If you want to build an RPM, see *Packaging Red Hat*.

Instructions are tested with `CentOS 6.8` on `x86_64` platform

2.3.1 Warning:

`CentOS 6` is very old and not fully supported by the FRR community anymore. Building FRR takes multiple manual steps to update the build system with newer packages than what's available from the archives. However, the built packages can still be installed afterwards on a standard `CentOS 6` without any special packages.

Support for CentOS 6 is now on a best-effort base by the community.

2.3.2 CentOS 6 restrictions:

- PIMd is not supported on `CentOS 6`. Upgrade to `CentOS 7` if PIMd is needed
- MPLS is not supported on `CentOS 6`. MPLS requires Linux Kernel 4.5 or higher (LDP can be built, but may have limited use without MPLS)
- Zebra is unable to detect what bridge/vrf an interface is associated with (IFLA_INFO_SLAVE_KIND does not exist in the kernel headers, you can use a newer kernel + headers to get this functionality)
- frr_reload.py will not work, as this requires Python 2.7, and CentOS 6 only has 2.6. You can install Python 2.7 via IUS, but it won't work properly unless you compile and install the ipaddr package for it.
- Building the package requires Sphinx >= 1.1. Only a non-standard package provides a newer sphinx and requires manual installation (see below)

2.3.3 Install required packages

Add packages:

```
sudo yum install git autoconf automake libtool make \
   readline-devel texinfo net-snmp-devel groff pkgconfig \
   json-c-devel pam-devel flex epel-release c-ares-devel libcap-devel \
   elfutils-libelf-devel
```

Install newer version of bison (CentOS 6 package source is too old) from CentOS 7:

```
sudo yum install rpm-build
curl -O http://vault.centos.org/7.0.1406/os/Source/SPackages/bison-2.7-4.el7.src.rpm
rpmbuild --rebuild ./bison-2.7-4.el7.src.rpm
sudo yum install ./rpmbuild/RPMS/x86_64/bison-2.7-4.el6.x86_64.rpm
rm -rf rpmbuild
```

Install newer version of autoconf and automake (Package versions are too old):

```
curl -O http://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz
tar xvf autoconf-2.69.tar.gz
cd autoconf-2.69
./configure --prefix=/usr
make
sudo make install
cd ..

curl -O http://ftp.gnu.org/gnu/automake/automake-1.15.tar.gz
tar xvf automake-1.15.tar.gz
cd automake-1.15
./configure --prefix=/usr
make
sudo make install
cd ..
```

Install `Python 2.7` in parallel to default 2.6. Make sure you've install EPEL (`epel-release` as above). Then install current `python27`: `python27-devel` and `pytest`

```
sudo rpm -ivh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
sudo rpm -ivh https://centos6.iuscommunity.org/ius-release.rpm
sudo yum install python27 python27-pip python27-devel
sudo pip2.7 install pytest
```

Please note that `CentOS 6` needs to keep python pointing to version 2.6 for `yum` to keep working, so don't create a symlink for python2.7 to python.

Install newer `Sphinx-Build` based on `Python 2.7`.

Create a new repo `/etc/yum.repos.d/puias6.repo` with the following contents:

```
### Name: RPM Repository for RHEL 6 - PUIAS (used for Sphinx-Build)
### URL: http://springdale.math.ias.edu/data/puias/computational
[puias-computational]
name = RPM Repository for RHEL 6 - Sphinx-Build
baseurl = http://springdale.math.ias.edu/data/puias/computational/$releasever/$basearch
```

(continues on next page)

(continued from previous page)

```
#mirrorlist =
enabled = 1
protect = 0
gpgkey =
gpgcheck = 0
```

Update rpm database & Install newer sphinx

```
sudo yum update
sudo yum install python27-sphinx
```

Install libyang and its dependencies:

```
sudo yum install pcre-devel doxygen cmake
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout 090926a89d59a3c4000719505d563aaf6ac60f2
mkdir build ; cd build
cmake -DENABLE_LYD_PRIV=ON -DCMAKE_INSTALL_PREFIX:PATH=/usr -D CMAKE_BUILD_TYPE:String=
↪"Release" ..
make build-rpm
sudo yum install ./rpms/RPMS/x86_64/libyang-0.16.111-0.x86_64.rpm ./rpms/RPMS/x86_64/
↪libyang-devel-0.16.111-0.x86_64.rpm
cd ../..
```

2.3.4 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr groups and user

```
sudo groupadd -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo useradd -u 92 -g 92 -M -r -G frrvty -s /sbin/nologin \
   -c "FRR FRRouting suite" -d /var/run/frr frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example.)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --bindir=/usr/bin \
    --sbindir=/usr/lib/frr \
    --sysconfdir=/etc/frr \
    --libdir=/usr/lib/frr \
    --libexecdir=/usr/lib/frr \
```

(continues on next page)

(continued from previous page)

```
    --localstatedir=/var/run/frr \
    --with-moduledir=/usr/lib/frr/modules \
    --disable-pimd \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --disable-ldpd \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
make check
sudo make install
```

Create empty FRR configuration files

```
sudo mkdir /var/log/frr
sudo mkdir /etc/frr
```

For integrated config file:

```
sudo touch /etc/frr/frr.conf
```

For individual config files:

Note: Integrated config is preferred to individual config.

```
sudo touch /etc/frr/babeld.conf
sudo touch /etc/frr/bfdd.conf
sudo touch /etc/frr/bgpd.conf
sudo touch /etc/frr/eigrpd.conf
sudo touch /etc/frr/isisd.conf
sudo touch /etc/frr/ldpd.conf
sudo touch /etc/frr/nhrpd.conf
sudo touch /etc/frr/ospf6d.conf
sudo touch /etc/frr/ospfd.conf
sudo touch /etc/frr/pbrd.conf
sudo touch /etc/frr/pimd.conf
sudo touch /etc/frr/ripd.conf
sudo touch /etc/frr/ripngd.conf
sudo touch /etc/frr/staticd.conf
sudo touch /etc/frr/zebra.conf
sudo chown -R frr:frr /etc/frr/
sudo touch /etc/frr/vtysh.conf
sudo chown frr:frrvty /etc/frr/vtysh.conf
sudo chmod 640 /etc/frr/*.conf
```

Install daemon config file

```
sudo install -p -m 644 tools/etc/frr/daemons /etc/frr/
sudo chown frr:frr /etc/frr/daemons
```

Edit /etc/frr/daemons as needed to select the required daemons

Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`

Enable IP & IPv6 forwarding

Edit `/etc/sysctl.conf` and set the following values (ignore the other settings):

```
# Controls IP packet forwarding
net.ipv4.ip_forward = 1
net.ipv6.conf.all.forwarding=1

# Controls source route verification
net.ipv4.conf.default.rp_filter = 0
```

Load the modified sysctl's on the system:

```
sudo sysctl -p /etc/sysctl.d/90-routing-sysctl.conf
```

Add init.d startup file

```
sudo install -p -m 755 tools/frr /etc/init.d/frr
sudo chkconfig --add frr
```

Enable FRR daemon at startup

```
sudo chkconfig frr on
```

Start FRR manually (or reboot)

```
sudo /etc/init.d/frr start
```

2.4 CentOS 7

This document describes installation from source. If you want to build an RPM, see *Packaging Red Hat*.

2.4.1 CentOS 7 restrictions:

- MPLS is not supported on `CentOS 7` with default kernel. MPLS requires Linux Kernel 4.5 or higher (LDP can be built, but may have limited use without MPLS)

2.4.2 Install required packages

Add packages:

```
sudo yum install git autoconf automake libtool make \
  readline-devel texinfo net-snmp-devel groff pkgconfig \
  json-c-devel pam-devel bison flex pytest c-ares-devel \
  python-devel python-sphinx libcap-devel \
  elfutils-libelf-devel libunwind-devel
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.4.3 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr groups and user

```
sudo groupadd -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo useradd -u 92 -g 92 -M -r -G frrvty -s /sbin/nologin \
  -c "FRR FRRouting suite" -d /var/run/frr frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example.)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --bindir=/usr/bin \
    --sbindir=/usr/lib/frr \
    --sysconfdir=/etc/frr \
    --libdir=/usr/lib/frr \
    --libexecdir=/usr/lib/frr \
    --localstatedir=/var/run/frr \
    --with-moduledir=/usr/lib/frr/modules \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --disable-ldpd \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion \
    SPHINXBUILD=/usr/bin/sphinx-build
make
make check
sudo make install
```

Create empty FRR configuration files

```
sudo mkdir /var/log/frr
sudo mkdir /etc/frr
sudo touch /etc/frr/zebra.conf
sudo touch /etc/frr/bgpd.conf
sudo touch /etc/frr/ospfd.conf
sudo touch /etc/frr/ospf6d.conf
sudo touch /etc/frr/isisd.conf
sudo touch /etc/frr/ripd.conf
sudo touch /etc/frr/ripngd.conf
sudo touch /etc/frr/pimd.conf
sudo touch /etc/frr/nhrpd.conf
sudo touch /etc/frr/eigrpd.conf
sudo touch /etc/frr/babeld.conf
sudo chown -R frr:frr /etc/frr/
sudo touch /etc/frr/vtysh.conf
sudo chown frr:frrvty /etc/frr/vtysh.conf
sudo chmod 640 /etc/frr/*.conf
```

Install daemon config file

```
sudo install -p -m 644 tools/etc/frr/daemons /etc/frr/
sudo chown frr:frr /etc/frr/daemons
```

Edit /etc/frr/daemons as needed to select the required daemons

Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`

Enable IP & IPv6 forwarding

Create a new file `/etc/sysctl.d/90-routing-sysctl.conf` with the following content:

```
# Sysctl for routing
#
# Routing: We need to forward packets
net.ipv4.conf.all.forwarding=1
net.ipv6.conf.all.forwarding=1
```

Load the modified sysctl's on the system:

```
sudo sysctl -p /etc/sysctl.d/90-routing-sysctl.conf
```

Install frr Service

```
sudo install -p -m 644 tools/frr.service /usr/lib/systemd/system/frr.service
```

Register the systemd files

```
sudo systemctl preset frr.service
```

Enable required frr at startup

```
sudo systemctl enable frr
```

Reboot or start FRR manually

```
sudo systemctl start frr
```

2.5 CentOS 8

This document describes installation from source. If you want to build an RPM, see *Packaging Red Hat*.

2.5.1 Install required packages

Add packages:

```
sudo dnf install --enablerepo=PowerTools git autoconf pcre-devel \
  automake libtool make readline-devel texinfo net-snmp-devel pkgconfig \
  groff pkgconfig json-c-devel pam-devel bison flex python2-pytest \
  c-ares-devel python2-devel libcap-devel \
  elfutils-libelf-devel libunwind-devel
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.5.2 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr groups and user

```
sudo groupadd -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo useradd -u 92 -g 92 -M -r -G frrvty -s /sbin/nologin \
  -c "FRR FRRouting suite" -d /var/run/frr frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example.)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --bindir=/usr/bin \
    --sbindir=/usr/lib/frr \
    --sysconfdir=/etc/frr \
    --libdir=/usr/lib/frr \
    --libexecdir=/usr/lib/frr \
```

(continues on next page)

(continued from previous page)

```
    --localstatedir=/var/run/frr \
    --with-moduledir=/usr/lib/frr/modules \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --disable-ldpd \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion \
    SPHINXBUILD=/usr/bin/sphinx-build
make
make check
sudo make install
```

Create empty FRR configuration files

```
sudo mkdir /var/log/frr
sudo mkdir /etc/frr
sudo touch /etc/frr/zebra.conf
sudo touch /etc/frr/bgpd.conf
sudo touch /etc/frr/ospfd.conf
sudo touch /etc/frr/ospf6d.conf
sudo touch /etc/frr/isisd.conf
sudo touch /etc/frr/ripd.conf
sudo touch /etc/frr/ripngd.conf
sudo touch /etc/frr/pimd.conf
sudo touch /etc/frr/nhrpd.conf
sudo touch /etc/frr/eigrpd.conf
sudo touch /etc/frr/babeld.conf
sudo chown -R frr:frr /etc/frr/
sudo touch /etc/frr/vtysh.conf
sudo chown frr:frrvty /etc/frr/vtysh.conf
sudo chmod 640 /etc/frr/*.conf
```

Install daemon config file

```
sudo install -p -m 644 tools/etc/frr/daemons /etc/frr/
sudo chown frr:frr /etc/frr/daemons
```

Edit /etc/frr/daemons as needed to select the required daemons

Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`

Enable IP & IPv6 forwarding

Create a new file `/etc/sysctl.d/90-routing-sysctl.conf` with the following content:

```
# Sysctl for routing
#
# Routing: We need to forward packets
net.ipv4.conf.all.forwarding=1
net.ipv6.conf.all.forwarding=1
```

Load the modified sysctl's on the system:

```
sudo sysctl -p /etc/sysctl.d/90-routing-sysctl.conf
```

Install frr Service

```
sudo install -p -m 644 tools/frr.service /usr/lib/systemd/system/frr.service
```

Register the systemd files

```
sudo systemctl preset frr.service
```

Enable required frr at startup

```
sudo systemctl enable frr
```

Reboot or start FRR manually

```
sudo systemctl start frr
```

2.6 Debian 8

2.6.1 Debian 8 restrictions:

- MPLS is not supported on `Debian 8` with default kernel. MPLS requires Linux Kernel 4.5 or higher (LDP can be built, but may have limited use without MPLS)

2.6.2 Install required packages

Add packages:

```
sudo apt-get install git autoconf automake libtool make \
   libreadline-dev texinfo libjson-c-dev pkg-config bison flex python3-pip \
   libc-ares-dev python3-dev python3-sphinx build-essential \
   libsnmp-dev libcap-dev libelf-dev
```

Install newer pytest (>3.0) from pip

```
sudo pip3 install pytest
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
```

(continues on next page)

(continued from previous page)

```
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.6.3 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr groups and user

```
sudo addgroup --system --gid 92 frr
sudo addgroup --system --gid 85 frrvty
sudo adduser --system --ingroup frr --home /var/run/frr/ \
   --gecos "FRR suite" --shell /bin/false frr
sudo usermod -a -G frrvty frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example.)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --localstatedir=/var/run/frr \
    --sbindir=/usr/lib/frr \
    --sysconfdir=/etc/frr \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
make check
sudo make install
```

Create empty FRR configuration files

```
sudo install -m 755 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/zebra.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/bgpd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ospfd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ospf6d.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/isisd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ripd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ripngd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/pimd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ldpd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/nhrpd.conf
sudo install -m 640 -o frr -g frrvty /dev/null /etc/frr/vtysh.conf
```

Enable IP & IPv6 forwarding

Edit `/etc/sysctl.conf` and uncomment the following values (ignore the other settings)

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system

Troubleshooting

Local state directory

The local state directory must exist and have the correct permissions applied for the frrouting daemons to start. In the above ./configure example the local state directory is set to /var/run/frr (–localstatedir=/var/run/frr) Debian considers /var/run/frr to be temporary and this is removed after a reboot.

When using a different local state directory you need to create the new directory and change the ownership to the frr user, for example:

```
mkdir /var/opt/frr
chown frr /var/opt/frr
```

Shared library error

If you try and start any of the frrouting daemons you may see the below error due to the frrouting shared library directory not being found:

```
./zebra: error while loading shared libraries: libfrr.so.0: cannot open shared object␣
↪file: No such file or directory
```

The fix is to add the following line to /etc/ld.so.conf which will continue to reference the library directory after the system reboots. To load the library directory path immediately run the ldconfig command after adding the line to the file eg:

```
echo include /usr/local/lib >> /etc/ld.so.conf
ldconfig
```

2.7 Debian 9

2.7.1 Install required packages

Add packages:

```
sudo apt-get install git autoconf automake libtool make \
  libreadline-dev texinfo libjson-c-dev pkg-config bison flex \
  libc-ares-dev python3-dev python3-pytest python3-sphinx build-essential \
  libsnmp-dev libcap-dev libelf-dev libunwind-dev
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.7.2 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr groups and user

```
sudo addgroup --system --gid 92 frr
sudo addgroup --system --gid 85 frrvty
sudo adduser --system --ingroup frr --home /var/opt/frr/ \
   --gecos "FRR suite" --shell /bin/false frr
sudo usermod -a -G frrvty frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example.)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --localstatedir=/var/opt/frr \
    --sbindir=/usr/lib/frr \
    --sysconfdir=/etc/frr \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
make check
sudo make install
```

Create empty FRR configuration files

```
sudo install -m 755 -o frr -g frr -d /var/log/frr
sudo install -m 755 -o frr -g frr -d /var/opt/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/zebra.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/bgpd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ospfd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ospf6d.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/isisd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ripd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ripngd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/pimd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/ldpd.conf
sudo install -m 640 -o frr -g frr /dev/null /etc/frr/nhrpd.conf
sudo install -m 640 -o frr -g frrvty /dev/null /etc/frr/vtysh.conf
```

Enable IP & IPv6 forwarding

Edit `/etc/sysctl.conf` and uncomment the following values (ignore the other settings)

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system

2.7.3 Troubleshooting

Shared library error

If you try and start any of the frrouting daemons you may see the below error due to the frrouting shared library directory not being found:

```
./zebra: error while loading shared libraries: libfrr.so.0: cannot open
shared object file: No such file or directory
```

The fix is to add the following line to /etc/ld.so.conf which will continue to reference the library directory after the system reboots. To load the library directory path immediately run the ldconfig command after adding the line to the file eg:

```
echo include /usr/local/lib >> /etc/ld.so.conf
ldconfig
```

2.8 Fedora 24+

This document describes installation from source. If you want to build an RPM, see *Packaging Red Hat*.

These instructions have been tested on Fedora 24+.

2.8.1 Installing Dependencies

```
sudo dnf install git autoconf automake libtool make \
  readline-devel texinfo net-snmp-devel groff pkgconfig json-c-devel \
  pam-devel python3-pytest bison flex c-ares-devel python3-devel \
  python3-sphinx perl-core patch libcap-devel \
  elfutils-libelf-devel libunwind-devel
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
```

(continues on next page)

(continued from previous page)

```
make
sudo make install
```

2.8.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo useradd -u 92 -g 92 -M -r -G frrvty -s /sbin/nologin \
  -c "FRR FRRouting suite" -d /var/run/frr frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Create a new file `/etc/sysctl.d/90-routing-sysctl.conf` with the following content:

```
#
# Enable packet forwarding
#
net.ipv4.conf.all.forwarding=1
net.ipv6.conf.all.forwarding=1
#
# Enable MPLS Label processing on all interfaces
#
#net.mpls.conf.eth0.input=1
#net.mpls.conf.eth1.input=1
#net.mpls.conf.eth2.input=1
#net.mpls.platform_labels=100000
```

Note: MPLS must be invidividually enabled on each interface that requires it. See the example in the config block above.

Load the modified sysctls on the system:

```
sudo sysctl -p /etc/sysctl.d/90-routing-sysctl.conf
```

Create a new file `/etc/modules-load.d/mpls.conf` with the following content:

```
# Load MPLS Kernel Modules
mpls-router
mpls-iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

Note: Fedora ships with the `firewalld` service enabled. You may run into some issues with the iptables rules it installs by default. If you wish to just stop the service and clear *ALL* rules do these commands:

```
sudo systemctl disable firewalld.service
sudo systemctl stop firewalld.service
sudo iptables -F
```

Install frr Service

```
sudo install -p -m 644 tools/frr.service /usr/lib/systemd/system/frr.service
sudo systemctl enable frr
```

Enable daemons

Open `/etc/frr/daemons` with your text editor of choice. Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`.

Start FRR

```
sudo systemctl start frr
```

2.9 openSUSE

This document describes installation from source.

These instructions have been tested on openSUSE Tumbleweed in a Raspberry Pi 400.

2.9.1 Installing Dependencies

```
zypper in  git autoconf automake libtool make   \
  readline-devel texinfo net-snmp-devel groff pkgconfig libjson-c-devel\
  pam-devel python3-pytest bison flex c-ares-devel python3-devel\
  python3-Sphinx perl patch libcap-devel libyang-devel \
  libelf-devel libunwind-devel
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

2.9.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo useradd -u 92 -g 92 -M -r -G frrvty -s /sbin/nologin \
  -c "FRR FRRouting suite" -d /var/run/frr frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Note: In some platforms like raspberry for performance reasons some directories are in file systems (/var/run, ...) mounted with tempfs so will disapear after every reboot. In frr the /var/run/frr is used to store pid files for every daemon.

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Create a new file `/etc/sysctl.d/90-routing-sysctl.conf` with the following content:

```
#
# Enable packet forwarding
#
net.ipv4.conf.all.forwarding=1
net.ipv6.conf.all.forwarding=1
#
# Enable MPLS Label processing on all interfaces
#
#net.mpls.conf.eth0.input=1
#net.mpls.conf.eth1.input=1
#net.mpls.conf.eth2.input=1
#net.mpls.platform_labels=100000
```

Note: MPLS must be invidividually enabled on each interface that requires it. See the example in the config block above.

Load the modified sysctls on the system:

```
sudo sysctl -p /etc/sysctl.d/90-routing-sysctl.conf
```

Create a new file `/etc/modules-load.d/mpls.conf` with the following content:

```
# Load MPLS Kernel Modules
mpls-router
mpls-iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

Note: The `firewalld` service could be enabled. You may run into some issues with the iptables rules it installs by default. If you wish to just stop the service and clear *ALL* rules do these commands:

```
sudo systemctl disable firewalld.service
sudo systemctl stop firewalld.service
sudo iptables -F
```

Install frr Service

```
sudo install -p -m 644 tools/frr.service /usr/lib/systemd/system/frr.service
sudo systemctl enable frr
```

Enable daemons

Open `/etc/frr/daemons` with your text editor of choice. Look for the section with `bgpd=no` etc. Enable the daemons as required by changing the value to `yes`.

Start FRR

```
sudo systemctl start frr
```

Check the starting messages of frr with

```
journalctl -u frr --follow
```

2.10 FreeBSD 9

2.10.1 FreeBSD 9 restrictions:

- MPLS is not supported on `FreeBSD`. MPLS requires a Linux Kernel (4.5 or higher). LDP can be built, but may have limited use without MPLS

2.10.2 Install required packages

Add packages: (Allow the install of the package management tool if this is first package install and asked)

```
pkg install -y git autoconf automake libtool gmake \
    pkgconf texinfo json-c bison flex py36-pytest c-ares \
    python3 py36-sphinx libexecinfo
```

Make sure there is no /usr/bin/flex preinstalled (and use the newly installed in /usr/local/bin): (FreeBSD frequently provides a older flex as part of the base OS which takes preference in path)

```
rm -f /usr/bin/flex
```

For building with clang (instead of gcc), upgrade clang from 3.4 default to 3.6 *This is needed to build FreeBSD packages as well - for packages clang is default* (Clang 3.4 as shipped with FreeBSD 9 crashes during compile)

```
pkg install clang36
pkg delete clang34
mv /usr/bin/clang /usr/bin/clang34
ln -s /usr/local/bin/clang36 /usr/bin/clang
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

> **Warning:** `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.10.3 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr group and user

```
pw groupadd frr -g 101
pw groupadd frrvty -g 102
pw adduser frr -g 101 -u 101 -G 102 -c "FRR suite" \
    -d /usr/local/etc/frr -s /usr/sbin/nologin
```

(You may prefer different options on configure statement. These are just an example)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
export MAKE=gmake
export LDFLAGS="-L/usr/local/lib"
export CPPFLAGS="-I/usr/local/include"
```

(continues on next page)

(continued from previous page)

```
./configure \
    --sysconfdir=/usr/local/etc/frr \
    --enable-pkgsrcrcdir=/usr/pkg/share/examples/rc.d \
    --localstatedir=/var/run/frr \
    --prefix=/usr/local \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
gmake
gmake check
sudo gmake install
```

Create empty FRR configuration files

```
sudo mkdir /usr/local/etc/frr
```

For integrated config file:

```
sudo touch /usr/local/etc/frr/frr.conf
```

For individual config files:

Note: Integrated config is preferred to individual config.

```
sudo touch /usr/local/etc/frr/babeld.conf
sudo touch /usr/local/etc/frr/bfdd.conf
sudo touch /usr/local/etc/frr/bgpd.conf
sudo touch /usr/local/etc/frr/eigrpd.conf
sudo touch /usr/local/etc/frr/isisd.conf
sudo touch /usr/local/etc/frr/ldpd.conf
sudo touch /usr/local/etc/frr/nhrpd.conf
sudo touch /usr/local/etc/frr/ospf6d.conf
sudo touch /usr/local/etc/frr/ospfd.conf
sudo touch /usr/local/etc/frr/pbrd.conf
sudo touch /usr/local/etc/frr/pimd.conf
sudo touch /usr/local/etc/frr/ripd.conf
sudo touch /usr/local/etc/frr/ripngd.conf
sudo touch /usr/local/etc/frr/staticd.conf
sudo touch /usr/local/etc/frr/zebra.conf
sudo chown -R frr:frr /usr/local/etc/frr/
sudo touch /usr/local/etc/frr/vtysh.conf
sudo chown frr:frrvty /usr/local/etc/frr/vtysh.conf
sudo chmod 640 /usr/local/etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/sysctl.conf`:

```
# Routing: We need to forward packets
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

Reboot or use `sysctl` to apply the same config to the running system.

2.11 FreeBSD 10

2.11.1 FreeBSD 10 restrictions:

- MPLS is not supported on `FreeBSD`. MPLS requires a Linux Kernel (4.5 or higher). LDP can be built, but may have limited use without MPLS

2.11.2 Install required packages

Add packages: (Allow the install of the package management tool if this is first package install and asked)

```
pkg install git autoconf automake libtool gmake json-c pkgconf \
    bison flex py36-pytest c-ares python3.6 py36-sphinx libunwind
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

Make sure there is no /usr/bin/flex preinstalled (and use the newly installed in /usr/local/bin): (FreeBSD frequently provides a older flex as part of the base OS which takes preference in path)

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

```
rm -f /usr/bin/flex
```

2.11.3 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr group and user

```
pw groupadd frr -g 101
pw groupadd frrvty -g 102
pw adduser frr -g 101 -u 101 -G 102 -c "FRR suite" \
    -d /usr/local/etc/frr -s /usr/sbin/nologin
```

(You may prefer different options on configure statement. These are just an example)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
export MAKE=gmake
export LDFLAGS="-L/usr/local/lib"
export CPPFLAGS="-I/usr/local/include"
./configure \
    --sysconfdir=/usr/local/etc/frr \
    --enable-pkgsrcrcdir=/usr/pkg/share/examples/rc.d \
    --localstatedir=/var/run/frr \
    --prefix=/usr/local \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
```

(continues on next page)

(continued from previous page)

```
gmake
gmake check
sudo gmake install
```

Create empty FRR configuration files

```
sudo mkdir /usr/local/etc/frr
```

For integrated config file:

```
sudo touch /usr/local/etc/frr/frr.conf
```

For individual config files:

Note: Integrated config is preferred to individual config.

```
sudo touch /usr/local/etc/frr/babeld.conf
sudo touch /usr/local/etc/frr/bfdd.conf
sudo touch /usr/local/etc/frr/bgpd.conf
sudo touch /usr/local/etc/frr/eigrpd.conf
sudo touch /usr/local/etc/frr/isisd.conf
sudo touch /usr/local/etc/frr/ldpd.conf
sudo touch /usr/local/etc/frr/nhrpd.conf
sudo touch /usr/local/etc/frr/ospf6d.conf
sudo touch /usr/local/etc/frr/ospfd.conf
sudo touch /usr/local/etc/frr/pbrd.conf
sudo touch /usr/local/etc/frr/pimd.conf
sudo touch /usr/local/etc/frr/ripd.conf
sudo touch /usr/local/etc/frr/ripngd.conf
sudo touch /usr/local/etc/frr/staticd.conf
sudo touch /usr/local/etc/frr/zebra.conf
sudo chown -R frr:frr /usr/local/etc/frr/
sudo touch /usr/local/etc/frr/vtysh.conf
sudo chown frr:frrvty /usr/local/etc/frr/vtysh.conf
sudo chmod 640 /usr/local/etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/sysctl.conf`:

```
# Routing: We need to forward packets
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

Reboot or use `sysctl` to apply the same config to the running system.

2.12 FreeBSD 11

2.12.1 FreeBSD 11 restrictions:

- MPLS is not supported on `FreeBSD`. MPLS requires a Linux Kernel (4.5 or higher). LDP can be built, but may have limited use without MPLS

2.12.2 Install required packages

Add packages: (Allow the install of the package management tool if this is first package install and asked)

```
pkg install git autoconf automake libtool gmake json-c pkgconf \
   bison flex py36-pytest c-ares python3.6 py36-sphinx texinfo libunwind
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

Make sure there is no /usr/bin/flex preinstalled (and use the newly installed in /usr/local/bin): (FreeBSD frequently provides a older flex as part of the base OS which takes preference in path)

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
```

(continues on next page)

(continued from previous page)

```
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

```
rm -f /usr/bin/flex
```

2.12.3 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr group and user

```
pw groupadd frr -g 101
pw groupadd frrvty -g 102
pw adduser frr -g 101 -u 101 -G 102 -c "FRR suite" \
   -d /usr/local/etc/frr -s /usr/sbin/nologin
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
setenv MAKE gmake
setenv LDFLAGS -L/usr/local/lib
setenv CPPFLAGS -I/usr/local/include
ln -s /usr/local/bin/sphinx-build-3.6 /usr/local/bin/sphinx-build
./configure \
    --sysconfdir=/usr/local/etc/frr \
    --enable-pkgsrcrcdir=/usr/pkg/share/examples/rc.d \
    --localstatedir=/var/run/frr \
    --prefix=/usr/local \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
gmake
gmake check
sudo gmake install
```

Create empty FRR configuration files

```
sudo mkdir /usr/local/etc/frr
```

For integrated config file:

```
sudo touch /usr/local/etc/frr/frr.conf
```

For individual config files:

Note: Integrated config is preferred to individual config.

```
sudo touch /usr/local/etc/frr/babeld.conf
sudo touch /usr/local/etc/frr/bfdd.conf
sudo touch /usr/local/etc/frr/bgpd.conf
sudo touch /usr/local/etc/frr/eigrpd.conf
sudo touch /usr/local/etc/frr/isisd.conf
sudo touch /usr/local/etc/frr/ldpd.conf
sudo touch /usr/local/etc/frr/nhrpd.conf
sudo touch /usr/local/etc/frr/ospf6d.conf
sudo touch /usr/local/etc/frr/ospfd.conf
sudo touch /usr/local/etc/frr/pbrd.conf
sudo touch /usr/local/etc/frr/pimd.conf
sudo touch /usr/local/etc/frr/ripd.conf
sudo touch /usr/local/etc/frr/ripngd.conf
sudo touch /usr/local/etc/frr/staticd.conf
sudo touch /usr/local/etc/frr/zebra.conf
sudo chown -R frr:frr /usr/local/etc/frr/
sudo touch /usr/local/etc/frr/vtysh.conf
sudo chown frr:frrvty /usr/local/etc/frr/vtysh.conf
sudo chmod 640 /usr/local/etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/sysctl.conf`:

```
# Routing: We need to forward packets
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

Reboot or use `sysctl` to apply the same config to the running system.

2.13 FreeBSD 13

2.13.1 FreeBSD 13 restrictions:

- MPLS is not supported on FreeBSD. MPLS requires a Linux Kernel (4.5 or higher). LDP can be built, but may have limited use without MPLS
- PIM for IPv6 is not currently supported on FreeBSD.

2.13.2 Install required packages

Add packages: (Allow the install of the package management tool if this is first package install and asked)

```
pkg install git autoconf automake libtool gmake json-c pkgconf \
   bison py39-pytest c-ares py39-sphinx texinfo libunwind libyang2
```

Note: The libunwind library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

2.13.3 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr group and user

```
pw groupadd frr -g 101
pw groupadd frrvty -g 102
pw adduser frr -g 101 -u 101 -G 102 -c "FRR suite" \
   -d /usr/local/etc/frr -s /usr/sbin/nologin
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
export MAKE=gmake LDFLAGS=-L/usr/local/lib CPPFLAGS=-I/usr/local/include
./configure \
    --sysconfdir=/usr/local/etc/frr \
    --enable-pkgsrcrcdir=/usr/pkg/share/examples/rc.d \
    --localstatedir=/var/run/frr \
    --prefix=/usr/local \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
```

(continues on next page)

(continued from previous page)

```
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
gmake
gmake check
sudo gmake install
```

Create empty FRR configuration files

```
sudo mkdir /usr/local/etc/frr
```

For integrated config file:

```
sudo touch /usr/local/etc/frr/frr.conf
```

For individual config files:

Note: Integrated config is preferred to individual config.

```
sudo touch /usr/local/etc/frr/babeld.conf
sudo touch /usr/local/etc/frr/bfdd.conf
sudo touch /usr/local/etc/frr/bgpd.conf
sudo touch /usr/local/etc/frr/eigrpd.conf
sudo touch /usr/local/etc/frr/isisd.conf
sudo touch /usr/local/etc/frr/ldpd.conf
sudo touch /usr/local/etc/frr/nhrpd.conf
sudo touch /usr/local/etc/frr/ospf6d.conf
sudo touch /usr/local/etc/frr/ospfd.conf
sudo touch /usr/local/etc/frr/pbrd.conf
sudo touch /usr/local/etc/frr/pimd.conf
sudo touch /usr/local/etc/frr/ripd.conf
sudo touch /usr/local/etc/frr/ripngd.conf
sudo touch /usr/local/etc/frr/staticd.conf
sudo touch /usr/local/etc/frr/zebra.conf
sudo chown -R frr:frr /usr/local/etc/frr/
sudo touch /usr/local/etc/frr/vtysh.conf
sudo chown frr:frrvty /usr/local/etc/frr/vtysh.conf
sudo chmod 640 /usr/local/etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/sysctl.conf`:

```
# Routing: We need to forward packets
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

Reboot or use `sysctl` to apply the same config to the running system.

2.14 NetBSD 6

2.14.1 NetBSD 6 restrictions:

- MPLS is not supported on `NetBSD`. MPLS requires a Linux Kernel (4.5 or higher). LDP can be built, but may have limited use without MPLS

2.14.2 Install required packages

Configure Package location:

```
PKG_PATH="ftp://ftp.NetBSD.org/pub/pkgsrc/packages/NetBSD/`uname -m`/`uname -r`/All"
export PKG_PATH
```

Add packages:

```
sudo pkg_add git autoconf automake libtool gmake openssl \
   pkg-config json-c py36-test python36 py36-sphinx
```

Install SSL Root Certificates (for git https access):

```
sudo pkg_add mozilla-rootcerts
sudo touch /etc/openssl/openssl.cnf
sudo mozilla-rootcerts install
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

> **Warning:** `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.14.3 Get FRR, compile it and install it (from Git)

Add frr groups and user

```
sudo groupadd -g 92 frr
sudo groupadd -g 93 frrvty
sudo useradd -g 92 -u 92 -G frrvty -c "FRR suite" \
    -d /nonexistent -s /sbin/nologin frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
MAKE=gmake
export LDFLAGS="-L/usr/pkg/lib -R/usr/pkg/lib"
export CPPFLAGS="-I/usr/pkg/include"
./configure \
    --sysconfdir=/usr/pkg/etc/frr \
    --enable-pkgsrcrcdir=/usr/pkg/share/examples/rc.d \
    --localstatedir=/var/run/frr \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
```

(continues on next page)

(continued from previous page)

```
gmake
gmake check
sudo gmake install
```

Create empty FRR configuration files

```
sudo mkdir /var/log/frr
sudo mkdir /usr/pkg/etc/frr
sudo touch /usr/pkg/etc/frr/zebra.conf
sudo touch /usr/pkg/etc/frr/bgpd.conf
sudo touch /usr/pkg/etc/frr/ospfd.conf
sudo touch /usr/pkg/etc/frr/ospf6d.conf
sudo touch /usr/pkg/etc/frr/isisd.conf
sudo touch /usr/pkg/etc/frr/ripd.conf
sudo touch /usr/pkg/etc/frr/ripngd.conf
sudo touch /usr/pkg/etc/frr/pimd.conf
sudo chown -R frr:frr /usr/pkg/etc/frr
sudo touch /usr/local/etc/frr/vtysh.conf
sudo chown frr:frrvty /usr/pkg/etc/frr/*.conf
sudo chmod 640 /usr/pkg/etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/sysctl.conf`:

```
# Routing: We need to forward packets
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

Reboot or use `sysctl` to apply the same config to the running system

Install rc.d init files

```
cp pkgsrc/*.sh /etc/rc.d/
chmod 555 /etc/rc.d/*.sh
```

Enable FRR processes

(Enable the required processes only)

```
echo "zebra=YES" >> /etc/rc.conf
echo "bgpd=YES" >> /etc/rc.conf
echo "ospfd=YES" >> /etc/rc.conf
echo "ospf6d=YES" >> /etc/rc.conf
echo "isisd=YES" >> /etc/rc.conf
echo "ripngd=YES" >> /etc/rc.conf
echo "ripd=YES" >> /etc/rc.conf
echo "pimd=YES" >> /etc/rc.conf
```

2.15 NetBSD 7

2.15.1 NetBSD 7 restrictions:

- MPLS is not supported on `NetBSD`. MPLS requires a Linux Kernel (4.5 or higher). LDP can be built, but may have limited use without MPLS

2.15.2 Install required packages

```
sudo pkgin install git autoconf automake libtool gmake openssl \
   pkg-config json-c python36 py36-test py36-sphinx
```

Install SSL Root Certificates (for git https access):

```
sudo pkgin install mozilla-rootcerts
sudo touch /etc/openssl/openssl.cnf
sudo mozilla-rootcerts install
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
```

(continues on next page)

(continued from previous page)

```
make
sudo make install
```

2.15.3 Get FRR, compile it and install it (from Git)

Add frr groups and user

```
sudo groupadd -g 92 frr
sudo groupadd -g 93 frrvty
sudo useradd -g 92 -u 92 -G frrvty -c "FRR suite" \
    -d /nonexistent -s /sbin/nologin frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example)

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
MAKE=gmake
export LDFLAGS="-L/usr/pkg/lib -R/usr/pkg/lib"
export CPPFLAGS="-I/usr/pkg/include"
./configure \
    --sysconfdir=/usr/pkg/etc/frr \
    --enable-pkgsrcrcdir=/usr/pkg/share/examples/rc.d \
    --localstatedir=/var/run/frr \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
gmake
gmake check
sudo gmake install
```

Create empty FRR configuration files

```
sudo mkdir /usr/pkg/etc/frr
sudo touch /usr/pkg/etc/frr/zebra.conf
sudo touch /usr/pkg/etc/frr/bgpd.conf
sudo touch /usr/pkg/etc/frr/ospfd.conf
sudo touch /usr/pkg/etc/frr/ospf6d.conf
sudo touch /usr/pkg/etc/frr/isisd.conf
sudo touch /usr/pkg/etc/frr/ripd.conf
```

(continues on next page)

(continued from previous page)

```
sudo touch /usr/pkg/etc/frr/ripngd.conf
sudo touch /usr/pkg/etc/frr/pimd.conf
sudo chown -R frr:frr /usr/pkg/etc/frr
sudo touch /usr/local/etc/frr/vtysh.conf
sudo chown frr:frrvty /usr/pkg/etc/frr/*.conf
sudo chmod 640 /usr/pkg/etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/sysctl.conf`:

```
# Routing: We need to forward packets
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

Reboot or use `sysctl` to apply the same config to the running system

Install rc.d init files

```
cp pkgsrc/*.sh /etc/rc.d/
chmod 555 /etc/rc.d/*.sh
```

Enable FRR processes

(Enable the required processes only)

```
echo "zebra=YES" >> /etc/rc.conf
echo "bgpd=YES" >> /etc/rc.conf
echo "ospfd=YES" >> /etc/rc.conf
echo "ospf6d=YES" >> /etc/rc.conf
echo "isisd=YES" >> /etc/rc.conf
echo "ripngd=YES" >> /etc/rc.conf
echo "ripd=YES" >> /etc/rc.conf
echo "pimd=YES" >> /etc/rc.conf
```

2.16 OpenBSD 6

2.16.1 Install required packages

Configure PKG_PATH

```
export PKG_PATH=http://ftp5.usa.openbsd.org/pub/OpenBSD/$(uname -r)/packages/$(machine -
↪a)/
```

Add packages:

```
pkg_add clang libcares python3
pkg_add git autoconf-2.69p2 automake-1.15.1 libtool bison
pkg_add gmake json-c py-test py-sphinx libexecinfo
```

Select Python2.7 as default (required for pytest)

```
ln -s /usr/local/bin/python2.7 /usr/local/bin/python
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.16.2 Get FRR, compile it and install it (from Git)

This assumes you want to build and install FRR from source and not using any packages

Add frr group and user

```
groupadd -g 525 _frr
groupadd -g 526 _frrvty
useradd -g 525 -u 525 -c "FRR suite" -G _frrvty \
    -d /nonexistent -s /sbin/nologin _frr
```

Download Source, configure and compile it

(You may prefer different options on configure statement. These are just an example)

> **Warning:** In openbsd the proper links for the libyang library may not have been created.

```
ln -s /usr/lib/libyang.so.1.10.17 /usr/lib/libyang.so
```

> **Warning:** openbsd since version 6.2 has `clang` as the default compiler so to build frr, clang must be used (the included gcc version is very old).

```
git clone https://github.com/frrouting/frr.git frr
cd frr
export AUTOCONF_VERSION="2.69"
export AUTOMAKE_VERSION="1.15"
./bootstrap.sh
export LDFLAGS="-L/usr/local/lib"
export CPPFLAGS="-I/usr/local/include"
./configure \
    --sysconfdir=/etc/frr \
    --localstatedir=/var/frr \
    --enable-multipath=64 \
    --enable-user=_frr \
    --enable-group=_frr \
    --enable-vty-group=_frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion \
    CC=clang
gmake
gmake check
doas gmake install
```

Create empty FRR configuration files

```
doas mkdir /var/frr
doas chown _frr:_frr /var/frr
doas chmod 755 /var/frr
doas mkdir /etc/frr
doas touch /etc/frr/zebra.conf
doas touch /etc/frr/bgpd.conf
doas touch /etc/frr/ospfd.conf
doas touch /etc/frr/ospf6d.conf
doas touch /etc/frr/isisd.conf
doas touch /etc/frr/ripd.conf
doas touch /etc/frr/ripngd.conf
doas touch /etc/frr/pimd.conf
doas touch /etc/frr/ldpd.conf
doas touch /etc/frr/nhrpd.conf
doas chown -R _frr:_frr /etc/frr
doas touch /etc/frr/vtysh.conf
doas chown -R _frr:_frrvty /etc/frr/vtysh.conf
doas chmod 750 /etc/frr
doas chmod 640 /etc/frr/*.conf
```

Enable IP & IPv6 forwarding

Add the following lines to the end of `/etc/rc.conf`:

```
net.inet6.ip6.forwarding=1      # 1=Permit forwarding of IPv6 packets
net.inet6.ip6.mforwarding=1     # 1=Permit forwarding of IPv6 multicast packets
net.inet6.ip6.multipath=1       # 1=Enable IPv6 multipath routing
```

Reboot to apply the config to the system

Enable MPLS Forwarding

To enable MPLS forwarding on a given interface, use the following command:

```
doas ifconfig em0 mpls
```

Alternatively, to make MPLS forwarding persistent across reboots, add the "mpls" keyword in the hostname.* files of the desired interfaces. Example:

```
cat /etc/hostname.em0
inet 10.0.1.1 255.255.255.0 mpls
```

Install rc.d init files

(create them in /etc/rc.d - no example are included at this time with FRR source)

Example (for zebra - store as `/etc/rc.d/frr_zebra.sh`)

```
#!/bin/sh
#
# $OpenBSD: frr_zebra.rc,v 1.1 2013/04/18 20:29:08 sthen Exp $

daemon="/usr/local/sbin/zebra -d"

. /etc/rc.d/rc.subr

rc_cmd $1
```

Enable FRR processes

(Enable the required processes only)

```
echo "frr_zebra=YES" >> /etc/rc.conf
echo "frr_bgpd=YES" >> /etc/rc.conf
echo "frr_ospfd=YES" >> /etc/rc.conf
echo "frr_ospf6d=YES" >> /etc/rc.conf
echo "frr_isisd=YES" >> /etc/rc.conf
echo "frr_ripngd=YES" >> /etc/rc.conf
echo "frr_ripd=YES" >> /etc/rc.conf
echo "frr_pimd=YES" >> /etc/rc.conf
echo "frr_ldpd=YES" >> /etc/rc.conf
```

2.17 OpenWrt

General info about OpenWrt buildsystem: link.

2.17.1 Prepare build environment

For Debian based distributions, run:

```
sudo apt-get install git build-essential libssl-dev libncurses5-dev \
   unzip zlib1g-dev subversion mercurial
```

For other environments, instructions can be found in the official documentation.

2.17.2 Get OpenWrt Sources (from Git)

Note: The OpenWrt build will fail if you run it as root. So take care to run it as a nonprivileged user.

Clone the OpenWrt sources and retrieve the package feeds

```
git clone https://github.com/openwrt/openwrt.git
cd openwrt
./scripts/feeds update -a
./scripts/feeds install -a
```

Configure OpenWrt for your target and select the needed FRR packages in Network -> Routing and Redirection -> frr, exit and save

```
make menuconfig
```

Then, to compile either a complete OpenWrt image, or the FRR packages, run:

```
make or make package/frr/compile
```

It may be possible that on first build `make package/frr/compile` not to work and it may be needed to run a `make` for the entire build environment. Add `V=s` to get more debugging output.

More information about OpenWrt buildsystem can be found here.

2.17.3 Work with sources

To update to a newer version, or change other options, you need to edit the `feeds/packages/frr/Makefile`.

More information about working with patches in OpenWrt buildsystem can be found here.

2.17.4 Usage

Edit `/usr/sbin/frr.init` and add/remove the daemons name in section `DAEMONS=` or don't install unneeded packages For example: zebra bgpd ldpd isisd nhrpd ospfd ospf6d pimd ripd ripngd

Enable the service

- `service frr enable`

Start the service

- `service frr start`

2.18 Ubuntu 14.04 LTS

This document describes installation from source. If you want to build a `deb`, see *Packaging Debian*.

2.18.1 Installing Dependencies

```
apt-get update
apt-get install \
   git autoconf automake libtool make libreadline-dev texinfo \
   pkg-config libpam0g-dev libjson-c-dev bison flex python3-pytest \
   libc-ares-dev python3-dev python3-sphinx install-info build-essential \
   libsnmp-dev perl libcap-dev libelf-dev
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.18.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -r -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo adduser --system --ingroup frr --home /var/run/frr/ \
   --gecos "FRR suite" --shell /sbin/nologin frr
sudo usermod -a -G frrvty frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Edit `/etc/sysctl.conf` and uncomment the following values (ignore the other settings):

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system.

Add MPLS kernel modules

Warning: MPLS is not supported on Ubuntu 14.04 with the default kernel. MPLS requires kernel 4.5 or higher. LDPD can be built, but may have limited use without MPLS. For an updated Ubuntu Kernel, see http://kernel.ubuntu.com/~kernel-ppa/mainline/

Ubuntu 18.04 ships with kernel 4.15. MPLS modules are present by default. To enable, add the following lines to `/etc/modules-load.d/modules.conf`:

```
# Load MPLS Kernel Modules
mpls_router
mpls_iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

Enable MPLS Forwarding

Edit `/etc/sysctl.conf` and the following lines. Make sure to add a line equal to `net.mpls.conf.eth0.input` for each interface used with MPLS.

```
# Enable MPLS Label processing on all interfaces
net.mpls.conf.eth0.input=1
net.mpls.conf.eth1.input=1
net.mpls.conf.eth2.input=1
net.mpls.platform_labels=100000
```

Install the init.d service

```
sudo install -m 755 tools/frr /etc/init.d/frr
```

Enable daemons

Open `/etc/frr/daemons` with your text editor of choice. Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`.

Start the init.d service

```
/etc/init.d/frr start
```

Use `/etc/init.d/frr status` to check its status.

2.19 Ubuntu 16.04 LTS

This document describes installation from source. If you want to build a `deb`, see *Packaging Debian*.

2.19.1 Installing Dependencies

```
apt-get update
apt-get install \
   git autoconf automake libtool make libreadline-dev texinfo \
   pkg-config libpam0g-dev libjson-c-dev bison flex python3-pytest \
   libc-ares-dev python3-dev python-ipaddress python3-sphinx \
   install-info build-essential libsnmp-dev perl libcap-dev \
   libelf-dev
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

> **Warning:** `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

2.19.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -r -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo adduser --system --ingroup frr --home /var/run/frr/ \
   --gecos "FRR suite" --shell /sbin/nologin frr
sudo usermod -a -G frrvty frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
```

(continues on next page)

(continued from previous page)

```
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Edit `/etc/sysctl.conf` and uncomment the following values (ignore the other settings):

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system.

Add MPLS kernel modules

Warning: MPLS is not supported on Ubuntu 16.04 with the default kernel. MPLS requires kernel 4.5 or higher. LDPD can be built, but may have limited use without MPLS. For an updated Ubuntu Kernel, see http://kernel.ubuntu.com/~kernel-ppa/mainline/

Ubuntu 18.04 ships with kernel 4.15. MPLS modules are present by default. To enable, add the following lines to `/etc/modules-load.d/modules.conf`:

```
# Load MPLS Kernel Modules
mpls_router
mpls_iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

Enable MPLS Forwarding

Edit `/etc/sysctl.conf` and the following lines. Make sure to add a line equal to `net.mpls.conf.eth0.input` for each interface used with MPLS.

```
# Enable MPLS Label processing on all interfaces
net.mpls.conf.eth0.input=1
net.mpls.conf.eth1.input=1
net.mpls.conf.eth2.input=1
net.mpls.platform_labels=100000
```

Install service files

```
sudo install -m 644 tools/frr.service /etc/systemd/system/frr.service
sudo systemctl enable frr
```

Enable daemons

Open `/etc/frr/daemons` with your text editor of choice. Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`.

Start FRR

```
systemctl start frr
```

2.20 Ubuntu 18.04 LTS

This document describes installation from source. If you want to build a `deb`, see *Packaging Debian*.

2.20.1 Installing Dependencies

```
sudo apt update
sudo apt-get install \
   git autoconf automake libtool make libreadline-dev texinfo \
   pkg-config libpam0g-dev libjson-c-dev bison flex \
   libc-ares-dev python3-dev python3-sphinx \
   install-info build-essential libsnmp-dev perl libcap-dev \
   libelf-dev libunwind-dev
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

> **Warning:** `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

Protobuf

```
sudo apt-get install protobuf-c-compiler libprotobuf-c-dev
```

ZeroMQ

```
sudo apt-get install libzmq5 libzmq3-dev
```

2.20.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -r -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo adduser --system --ingroup frr --home /var/run/frr/ \
   --gecos "FRR suite" --shell /sbin/nologin frr
sudo usermod -a -G frrvty frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Edit `/etc/sysctl.conf` and uncomment the following values (ignore the other settings):

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system.

Add MPLS kernel modules

Ubuntu 18.04 ships with kernel 4.15. MPLS modules are present by default. To enable, add the following lines to `/etc/modules-load.d/modules.conf`:

```
# Load MPLS Kernel Modules
mpls_router
mpls_iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

If the above command returns an error, you may need to install the appropriate or latest linux-modules-extra-<kernel-version>-generic package. For example `apt-get install linux-modules-extra-`uname -r`-generic`

Enable MPLS Forwarding

Edit `/etc/sysctl.conf` and the following lines. Make sure to add a line equal to `net.mpls.conf.eth0.input` for each interface used with MPLS.

```
# Enable MPLS Label processing on all interfaces
net.mpls.conf.eth0.input=1
net.mpls.conf.eth1.input=1
net.mpls.conf.eth2.input=1
net.mpls.platform_labels=100000
```

Install service files

```
sudo install -m 644 tools/frr.service /etc/systemd/system/frr.service
sudo systemctl enable frr
```

Enable daemons

Open `/etc/frr/daemons` with your text editor of choice. Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`.

Start FRR

```
systemctl start frr
```

2.21 Ubuntu 20.04 LTS

This document describes installation from source. If you want to build a `deb`, see *Packaging Debian*.

2.21.1 Installing Dependencies

```
sudo apt update
sudo apt-get install \
   git autoconf automake libtool make libreadline-dev texinfo \
   pkg-config libpam0g-dev libjson-c-dev bison flex \
   libc-ares-dev python3-dev python3-sphinx \
   install-info build-essential libsnmp-dev perl \
   libcap-dev python2 libelf-dev libunwind-dev
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

Note that Ubuntu 20 no longer installs python 2.x, so it must be installed explicitly. Ensure that your system has a symlink named `/usr/bin/python` pointing at `/usr/bin/python3`.

In addition, `pip` for python2 must be installed if you wish to run the FRR topotests. That version of `pip` is not available from the ubuntu apt repositories; in order to install it:

```
curl https://bootstrap.pypa.io/pip/2.7/get-pip.py --output get-pip.py
sudo python2 ./get-pip.py

# And verify the installation
pip2 --version
```

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

Protobuf

```
sudo apt-get install protobuf-c-compiler libprotobuf-c-dev
```

ZeroMQ

```
sudo apt-get install libzmq5 libzmq3-dev
```

2.21.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -r -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo adduser --system --ingroup frr --home /var/run/frr/ \
   --gecos "FRR suite" --shell /sbin/nologin frr
sudo usermod -a -G frrvty frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Edit `/etc/sysctl.conf` and uncomment the following values (ignore the other settings):

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
```

(continues on next page)

(continued from previous page)

```
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system.

Add MPLS kernel modules

Ubuntu 20.04 ships with kernel 5.4; MPLS modules are present by default. To enable, add the following lines to `/etc/modules-load.d/modules.conf`:

```
# Load MPLS Kernel Modules
mpls_router
mpls_iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

If the above command returns an error, you may need to install the appropriate or latest linux-modules-extra-<kernel-version>-generic package. For example `apt-get install linux-modules-extra-`uname -r`-generic`

Enable MPLS Forwarding

Edit `/etc/sysctl.conf` and the following lines. Make sure to add a line equal to `net.mpls.conf.eth0.input` for each interface used with MPLS.

```
# Enable MPLS Label processing on all interfaces
net.mpls.conf.eth0.input=1
net.mpls.conf.eth1.input=1
net.mpls.conf.eth2.input=1
net.mpls.platform_labels=100000
```

Install service files

```
sudo install -m 644 tools/frr.service /etc/systemd/system/frr.service
sudo systemctl enable frr
```

Enable daemons

Open `/etc/frr/daemons` with your text editor of choice. Look for the section with `watchfrr_enable=...` and `zebra=...` etc. Enable the daemons as required by changing the value to `yes`.

Start FRR

```
systemctl start frr
```

2.22 Arch Linux

2.22.1 Installing Dependencies

```
sudo pacman -Syu
sudo pacman -S \
   git autoconf automake libtool make cmake pcre readline texinfo \
   pkg-config pam json-c bison flex python-pytest \
   c-ares python python2-ipaddress python-sphinx \
   net-snmp perl libcap libelf libunwind
```

Note: The `libunwind` library is optional but highly recommended, as it improves backtraces printed for crashes and debugging. However, if it is not available for some reason, it can simply be left out without any loss of functionality.

FRR depends on the relatively new `libyang` library to provide YANG/NETCONF support. Unfortunately, most distributions do not yet offer a `libyang` package from their repositories. Therefore we offer two options to install this library.

Option 1: Binary Install

The FRR project builds some binary `libyang` packages.

RPM packages are at our RPM repository.

DEB packages are available as CI artifacts here.

Warning: `libyang` version 2.0.0 or newer is required to build FRR.

Note: The `libyang` development packages need to be installed in addition to the libyang core package in order to build FRR successfully. Make sure to download and install those from the link above alongside the binary packages.

Depending on your platform, you may also need to install the PCRE development package. Typically this is `libpcre2-dev` or `pcre2-devel`.

Option 2: Source Install

Note: Ensure that the libyang build requirements are met before continuing. Usually this entails installing `cmake` and `libpcre2-dev` or `pcre2-devel`.

```
git clone https://github.com/CESNET/libyang.git
cd libyang
git checkout v2.0.0
mkdir build; cd build
```

(continues on next page)

(continued from previous page)

```
cmake -D CMAKE_INSTALL_PREFIX:PATH=/usr \
      -D CMAKE_BUILD_TYPE:String="Release" ..
make
sudo make install
```

Protobuf

```
sudo pacman -S protobuf-c
```

ZeroMQ

```
sudo pacman -S zeromq
```

2.22.2 Building & Installing FRR

Add FRR user and groups

```
sudo groupadd -r -g 92 frr
sudo groupadd -r -g 85 frrvty
sudo useradd --system -g frr --home-dir /var/run/frr/ \
-c "FRR suite" --shell /sbin/nologin frr
sudo usermod -a -G frrvty frr
```

Compile

Clone the FRR git repo and use the included `configure` script to configure FRR's build time options to your liking. The full option listing can be obtained by running `./configure -h`. The options shown below are examples.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --includedir=\${prefix}/include \
    --bindir=\${prefix}/bin \
    --sbindir=\${prefix}/lib/frr \
    --libdir=\${prefix}/lib/frr \
    --libexecdir=\${prefix}/lib/frr \
    --localstatedir=/var/run/frr \
    --sysconfdir=/etc/frr \
    --with-moduledir=\${prefix}/lib/frr/modules \
    --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-snmp=agentx \
    --enable-multipath=64 \
```

(continues on next page)

(continued from previous page)

```
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
make
sudo make install
```

Install FRR configuration files

```
sudo install -m 775 -o frr -g frr -d /var/log/frr
sudo install -m 775 -o frr -g frrvty -d /etc/frr
sudo install -m 640 -o frr -g frrvty tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons.conf /etc/frr/daemons.conf
sudo install -m 640 -o frr -g frr tools/etc/frr/daemons /etc/frr/daemons
```

Tweak sysctls

Some sysctls need to be changed in order to enable IPv4/IPv6 forwarding and MPLS (if supported by your platform). If your platform does not support MPLS, skip the MPLS related configuration in this section.

Edit `/etc/sysctl.conf` [*Create the file if it doesn't exist*] and append the following values (ignore the other settings):

```
# Enable packet forwarding for IPv4
net.ipv4.ip_forward=1

#  Enable packet forwarding for IPv6
net.ipv6.conf.all.forwarding=1
```

Reboot or use `sysctl -p` to apply the same config to the running system.

Add MPLS kernel modules

To enable, add the following lines to `/etc/modules-load.d/modules.conf`:

```
# Load MPLS Kernel Modules
mpls_router
mpls_iptunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-iptunnel
```

Enable MPLS Forwarding

Edit `/etc/sysctl.conf` and the following lines. Make sure to add a line equal to `net.mpls.conf.eth0.input` for each interface used with MPLS.

```
# Enable MPLS Label processing on all interfaces
net.mpls.conf.eth0.input=1
net.mpls.conf.eth1.input=1
net.mpls.conf.eth2.input=1
net.mpls.platform_labels=100000
```

Install service files

```
sudo install -m 644 tools/frr.service /etc/systemd/system/frr.service
sudo systemctl enable frr
```

Start FRR

```
systemctl start frr
```

2.23 Docker

This page covers how to build FRR Docker images.

2.23.1 Images

FRR has Docker build infrastructure to produce Docker images containing source-built FRR on the following base platforms:

- Alpine
- Centos 7
- Centos 8

The following platform images are used to support Travis CI and can also be used to reproduce topotest failures when the docker host is Ubuntu (tested on 18.04 and 20.04):

- Ubuntu 18.04
- Ubuntu 20.04

The following platform images may also be built, but these simply install a binary package from an existing repository and do not perform source builds:

- Debian 10

Some of these are available on DockerHub.

There is no guarantee on what is and is not available from DockerHub at time of writing.

2.23.2 Scripts

Some platforms contain an included build script that may be run from the host. This will set appropriate packaging environment variables and clean up intermediate build images.

These scripts serve another purpose. They allow building platform packages without needing the platform. For example, the Centos 8 docker image can also be leveraged to build Centos 8 RPMs that can then be used separately from Docker.

If you are only interested in the Docker images and don't want the cleanup functionality of the scripts you can ignore them and perform a normal Docker build. If you want to build multi-arch docker images this is required as the scripts do not support using Buildkit for multi-arch builds.

Building Alpine Image

Script:

```
./docker/alpine/build.sh
```

No script:

```
docker build -f docker/alpine/Dockerfile .
```

No script, multi-arch (ex. amd64, arm64, armv7):

```
docker buildx build --platform linux/amd64,linux/arm64,linux/arm/v7 -f docker/alpine/
↪Dockerfile -t frr:latest .
```

Building Debian Image

```
cd docker/debian
docker build .
```

Multi-arch (ex. amd64, arm64, armv7):

```
cd docker/debian
docker buildx build --platform linux/amd64,linux/arm64,linux/arm/v7 -t frr-debian:latest␣
↪.
```

Building Centos 7 Image

Script:

```
./docker/centos-7/build.sh
```

No script:

```
docker build -f docker/centos-7/Dockerfile .
```

No script, multi-arch (ex. amd64, arm64):

```
docker buildx build --platform linux/amd64,linux/arm64 -f docker/centos-7/Dockerfile -t␣
↪frr-centos7:latest .
```

Building Centos 8 Image

Script:

```
./docker/centos-8/build.sh
```

No script:

```
docker build -f docker/centos-8/Dockerfile .
```

No script, multi-arch (ex. amd64, arm64):

```
docker buildx build --platform linux/amd64,linux/arm64 -f docker/centos-8/Dockerfile -t␣
↪frr-centos8:latest .
```

Building ubi 8 Image

Script:

```
./docker/ubi-8/build.sh
```

Script with params, an example could be this (all that info will go to docker label)

```
./docker/ubi-8/build.sh  frr:ubi-8-my-test "$(git rev-parse --short=10 HEAD)" my_release␣
↪my_name my_vendor
```

No script:

```
docker build -f docker/ubi-8/Dockerfile .
```

No script, multi-arch (ex. amd64, arm64):

```
docker buildx build --platform linux/amd64,linux/arm64 -f docker/ubi-8/Dockerfile -t frr-
↪ubi-8:latest .
```

Building Ubuntu 18.04 Image

Build image (from project root directory):

```
docker build -t frr-ubuntu18:latest  -f docker/ubuntu18-ci/Dockerfile .
```

Start the container:

```
docker run -d --privileged --name frr-ubuntu18 --mount type=bind,source=/lib/modules,
↪target=/lib/modules frr-ubuntu18:latest
```

Running a topotest (when the docker host is Ubuntu):

```
docker exec frr-ubuntu18 bash -c 'cd ~/frr/tests/topotests/ospf-topo1 ; sudo pytest test_
↪ospf_topo1.py'
```

Starting an interactive bash session:

```
docker exec -it frr-ubuntu18 bash
```

Stopping an removing a container:

```
docker stop frr-ubuntu18 ; docker rm frr-ubuntu18
```

Removing the built image:

```
docker rmi frr-ubuntu18:latest
```

Building Ubuntu 20.04 Image

Build image (from project root directory):

```
docker build -t frr-ubuntu20:latest  -f docker/ubuntu20-ci/Dockerfile .
```

Start the container:

```
docker run -d --privileged --name frr-ubuntu20 --mount type=bind,source=/lib/modules,
↪target=/lib/modules frr-ubuntu20:latest
```

Running a topotest (when the docker host is Ubuntu):

```
docker exec frr-ubuntu20 bash -c 'cd ~/frr/tests/topotests/ospf-topo1 ; sudo pytest test_
↪ospf_topo1.py'
```

Starting an interactive bash session:

```
docker exec -it frr-ubuntu20 bash
```

Stopping an removing a container:

```
docker stop frr-ubuntu20 ; docker rm frr-ubuntu20
```

Removing the built image:

```
docker rmi frr-ubuntu20:latest
```

2.24 Cross-Compiling

FRR is capable of being cross-compiled to a number of different architectures. With an adequate toolchain this process is fairly straightforward, though one must exercise caution to validate this toolchain's correctness before attempting to compile FRR or its dependencies; small oversights in the construction of the build tools may lead to problems which quickly become difficult to diagnose.

2.24.1 Toolchain Preliminary

The first step to cross-compiling any program is to identify the system which the program (FRR) will run on. From here on this will be called the "host" machine, following autotools' convention, while the machine building FRR will be called the "build" machine. The toolchain will of course be installed onto the build machine and be leveraged to build FRR for the host machine to run.

Note: The build machine used while writing this guide was `x86_64-pc-linux-gnu` and the target machine was `arm-linux-gnueabihf` (a Raspberry Pi 3B+). Replace this with your targeted tuple below if you plan on running the commands from this guide:

export HOST_ARCH="arm-linux-gnueabihf"

For your given target, the build system's OS may have some support for building cross compilers natively, or may even offer binary toolchains built upstream for the target architecture. Check your package manager or OS documentation before committing to building a toolchain from scratch.

This guide will not detail *how* to build a cross-compiling toolchain but will instead assume one already exists and is installed on the build system. The methods for building the toolchain itself may differ between operating systems so consult the OS documentation for any particulars regarding cross-compilers. The OSDev wiki has a pleasant tutorial on cross-compiling in the context of operating system development which bootstraps from only the native GCC and binutils on the build machine. This may be useful if the build machine's OS does not offer existing tools to build a cross-compiler targeting the host.

This guide will also not demonstrate how to build all of FRR's dependencies for the target architecture. Instead, general instructions for using a cross-compiling toolchain to compile packages using CMake, Autotools, and Makefiles are provided; these three cases apply to almost all FRR dependencies.

Warning: Ensure the versions and implementations of the C standard library (glibc or what have you) match on the host and the build toolchain. `ldd --version` will help you here. Upgrade one or the other if the they do not match.

2.24.2 Testing the Toolchain

Before any cross-compilation begins it would be prudent to test the new toolchain by writing, compiling and linking a simple program.

```
# A small program
cat > nothing.c <<EOF
int main() { return 0; }
EOF

# Build and link with the cross-compiler
${HOST_ARCH}-gcc -o nothing nothing.c

# Inspect the resulting binary, results may vary
file ./nothing

# nothing: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV),
```

(continues on next page)

(continued from previous page)

```
# dynamically linked, interpreter /lib/ld-linux-armhf.so.3,
# for GNU/Linux 3.2.0, not stripped
```

If this produced no errors then the installed toolchain is probably ready to start compiling the build dependencies and eventually FRR itself. There still may be lurking issues but fundamentally the toolchain can produce binaries and that's good enough to start working with it.

Warning: If any errors occurred during the previous functional test please look back and address them before moving on; this indicates your cross-compiling toolchain is *not* in a position to build FRR or its dependencies. Even if everything was fine, keep in mind that many errors from here on *may still be related* to your toolchain (e.g. libstdc++.so or other components) and this small test is not a guarantee of complete toolchain coherence.

2.24.3 Cross-compiling Dependencies

When compiling FRR it is necessary to compile some of its dependencies alongside it on the build machine. This is so symbols from the shared libraries (which will be loaded at run-time on the host machine) can be linked to the FRR binaries at compile time; additionally, headers for these libraries are needed during the compile stage for a successful build.

Sysroot Overview

All build dependencies should be installed into a "root" directory on the build computer, hereafter called the "sysroot". This directory will be prefixed to paths while searching for requisite libraries and headers during the build process. Often this may be set via a `--prefix` flag when building the dependent packages, meaning a `make install` will copy compiled libraries into (e.g.) `/usr/${HOST_ARCH}/usr`.

If the toolchain was built on the build machine then there is likely already a sysroot where those tools and standard libraries were installed; it may be helpful to use that directory as the sysroot for this build as well.

Basic Workflow

Before compiling or building any dependencies, make note of which daemons are being targeted and which libraries will be needed. Not all dependencies are necessary if only building with a subset of the daemons.

The following workflow will compile and install any libraries which can be built with Autotools. The resultant library will be installed into the sysroot `/usr/${HOST_ARCH}`.

```
./configure \
   CC=${HOST_ARCH}-gcc \
   CXX=${HOST_ARCH}-g++ \
   --build=${HOST_ARCH} \
   --prefix=/usr/${HOST_ARCH}
make
make install
```

Some libraries like `json-c` and `libyang` are packaged with CMake and can be built and installed generally like:

```
mkdir build
cd build
CC=${HOST_ARCH}-gcc \
```

(continues on next page)

(continued from previous page)

```
CXX=${HOST_ARCH}-g++ \
cmake \
    -DCMAKE_INSTALL_PREFIX=/usr/${HOST_ARCH} \
    ..
make
make install
```

For programs with only a Makefile (e.g. `libcap`) the process may look still a little different:

```
CC=${HOST_ARCH}-gcc make
make install DESTDIR=/usr/${HOST_ARCH}
```

These three workflows should handle the bulk of building and installing the build-time dependencies for FRR. Verify that the installed files are being placed correctly into the sysroot and were actually built using the cross-compile toolchain, not by the native toolchain by accident.

Dependency Notes

There are a lot of things that can go wrong during a cross-compilation. Some of the more common errors and a few special considerations are collected below for reference.

libyang

`-DENABLE_LYD_PRIV=ON` should be provided during the CMake step.

Ensure also that the version of `libyang` being installed corresponds to the version required by the targeted FRR version.

gRPC

This piece is requisite only if the `--enable-grpc` flag will be passed later on to FRR. One may get burned when compiling gRPC if the `protoc` version on the build machine differs from the version of `protoc` being linked to during a gRPC build. The error messages from this defect look like:

```
gens/src/proto/grpc/channelz/channelz.pb.h: In member function ‘void␣
↪grpc::channelz::v1::ServerRef::set_name(const char*, size_t)’:
gens/src/proto/grpc/channelz/channelz.pb.h:9127:64: error: ‘EmptyDefault’ is not a␣
↪member of ‘google::protobuf::internal::ArenaStringPtr’
 9127 |   name_.Set(::PROTOBUF_NAMESPACE_ID::internal::ArenaStringPtr::EmptyDefault{},␣
↪::std::string(
```

This happens because protocol buffer code generation uses `protoc` to create classes with different getters and setters corresponding to the protobuf data defined by the source tree's `.proto` files. Clearly the cross-compiled `protoc` cannot be used for this code generation because that binary is built for a different CPU.

The solution is to install matching versions of native and cross-compiled protocol buffers; this way the native binary will generate code and the cross-compiled library will be linked to by gRPC and these versions will not disagree.

The `-latomic` linker flag may also be necessary here if using `libstdc++` since GCC's C++11 implementation makes library calls in certain cases for `<atomic>` so `-latomic` cannot be assumed.

2.24.4 Cross-compiling FRR Itself

With all the necessary libraries cross-compiled and installed into the sysroot, the last thing to actually build is FRR itself:

```
# Clone and bootstrap the build
git clone 'https://github.com/FRRouting/frr.git'
# (e.g.) git checkout stable/7.5
./bootstrap.sh

# Build clippy using the native toolchain
mkdir build-clippy
cd build-clippy
../configure --enable-clippy-only
make clippy-only
cd ..

# Next, configure FRR and use the clippy we just built
./configure \
   CC=${HOST_ARCH}-gcc \
   CXX=${HOST_ARCH}-g++ \
   --host=${HOST_ARCH} \
   --with-sysroot=/usr/${HOST_ARCH} \
   --with-clippy=./build-clippy/lib/clippy \
   --sysconfdir=/etc/frr \
   --sbindir="\${prefix}/lib/frr" \
   --localstatedir=/var/run/frr \
   --prefix=/usr \
   --enable-user=frr \
   --enable-group=frr \
   --enable-vty-group=frrvty \
   --disable-doc \
   --enable-grpc

# Send it
make
```

2.24.5 Installation to Host Machine

If no errors were observed during the previous steps it is safe to `make install` FRR into its own directory.

```
# Install FRR its own "sysroot"
make install DESTDIR=/some/path/to/sysroot
```

After running the above command, FRR binaries, modules and example configuration files will be installed into some path on the build machine. The directory will have folders like `/usr` and `/etc`; this "root" should now be copied to the host and installed on top of the root directory there.

```
# Tar this sysroot (preserving permissions)
tar -C /some/path/to/sysroot -cpvf frr-${HOST_ARCH}.tar .

# Transfer tar file to host machine
```

(continues on next page)

(continued from previous page)

```
scp frr-${HOST_ARCH}.tar me@host-machine:

# Overlay the tarred sysroot on top of the host machine's root
ssh me@host-machine <<-EOF
   # May need to elevate permissions here
   tar -C / -xpvf frr-${HOST_ARCH}.tar.gz .
EOF
```

Now FRR should be installed just as if `make install` had been run on the host machine. Create configuration files and assign permissions as needed. Lastly, ensure the correct users and groups exist for FRR on the host machine.

2.24.6 Troubleshooting

Even when every precaution has been taken some things may still go wrong! This section details some common runtime problems.

Mismatched Libraries

If you see something like this after installing on the host:

```
/usr/lib/frr/zebra: error while loading shared libraries: libyang.so.1: cannot open␣
↪shared object file: No such file or directory
```

… at least one of FRR's dependencies which was linked to the binary earlier is not available on the host OS. Even if it has been installed the host repository's version may lag what is needed for more recent FRR builds (this is likely to happen with YANG at the moment).

If the matching library is not available from the host OS package manager it may be possible to compile them using the same toolchain used to compile FRR. The library may have already been built earlier when compiling FRR on the build machine, in which case it may be as simple as following the same workflow laid out during the *Installation to Host Machine*.

Mismatched Glibc Versions

The version and implementation of the C standard library must match on both the host and build toolchain. The error corresponding to this misconfiguration will look like:

```
/usr/lib/frr/zebra: /lib/${HOST_ARCH}/libc.so.6: version `GLIBC_2.32' not found␣
↪(required by /usr/lib/libfrr.so.0)
```

See the earlier warning about preventing a *glibc mismatch*.

CHAPTER

THREE

RELEASES & PACKAGING

3.1 FRR Release Procedure

<version> - version to be released, e.g. 7.3 `origin` - FRR upstream repository

3.1.1 Stage 1 - Preparation

1. Prepare changelog for the new release

   Note: use `tools/release_notes.py` to help draft release notes changelog

2. Checkout the existing `dev/<version>` branch.

   ```
   git checkout dev/<version>
   ```

3. Create and push a new branch called `stable/<version>` based on the `dev/<version>` branch.

   ```
   git checkout -b stable/<version>
   ```

4. Remove the development branch called `dev/<version>`

   ```
   git push origin --delete dev/<version>
   ```

5. Update Changelog for Red Hat Packages:

   Edit `redhat/frr.spec.in` and look for the `%changelog` section:

   - Change last (top of list) entry from `%{version}` to the **last** released version number. For example, if <version> is `7.3` and the last public release was `7.2`, you would use `7.2`, changing the file like so:

     ```
     * Tue Nov  7 2017 Martin Winter <mwinter@opensourcerouting.org> - %{version}
     ```

     to:

     ```
     * Tue Nov  7 2017 Martin Winter <mwinter@opensourcerouting.org> - 7.2
     ```

   - Add new entry to the top of the list with `%{version}` tag. Make sure to watch the format, i.e. the day is always 2 characters, with the 1st character being a space if the day is one digit.
   - Add the changelog text below this entry.

6. Update Changelog for Debian Packages:

   Update `debian/changelog`:

- Run following with **last** release version number and debian revision (usually -1) as argument to `dch --newversion VERSION`. For example, if <version> is `7.3` then you will run `dch --newversion 7.3-1`.
- The `dch` will run an editor, and you should add the changelog text below this entry, usually that would be: **New upstream version**.
- Verify the changelog format using `dpkg-parsechangelog`. In the repository root:

```
dpkg-parsechangelog
```

You should see output like this:

```
vagrant@local ~/frr> dpkg-parsechangelog
Source: frr
Version: 7.3-dev-0
Distribution: UNRELEASED
Urgency: medium
Maintainer: FRRouting-Dev <dev@lists.frrouting.org>
Timestamp: 1540478210
Date: Thu, 25 Oct 2018 16:36:50 +0200
Changes:
 frr (7.3-dev-0) RELEASED; urgency=medium
 .
   * Your Changes Here
```

7. Commit the changes, adding the changelog to the commit message. Follow all existing commit guidelines. The commit message should be akin to:

```
debian, redhat: updating changelog for new release
```

8. Change main version number:
   - Edit `configure.ac` and change version in the `AC_INIT` command to <version>

   Add and commit this change. This commit should be separate from the commit containing the changelog. The commit message should be:

```
FRR Release <version>
```

   The version field should be complete; i.e. for `8.0.0`, the version should be `8.0.0` and not `8.0` or `8`.

3.1.2 Stage 2 - Staging

1. Push the stable branch to a new remote branch prefixed with `rc`:

```
git push origin stable/<version>:rc/version
```

   This will trigger the NetDEF CI, which serve as a sanity check on the release branch. Verify that all tests pass and that all package builds are successful. To do this, go to the NetDEF CI located here:

   https://ci1.netdef.org/browse/FRR-FRR

   In the top left, look for `rc-`<version> in the "Plan branch" dropdown. Select this version. Note that it may take a few minutes for the CI to kick in on this new branch and appear in the list.

2. Push the stable branch:

```
git push origin stable/<version>:refs/heads/stable/<version>
```

3. Create and push a git tag for the version:

```
git tag -a frr-<version> -m "FRRouting Release <version>"
git push origin frr-<version>
```

4. Create a new branch based on `master`, cherry-pick the commit made earlier that added the changelogs, and use it to create a PR against `master`. This way `master` has the latest changelog for the next cycle.
5. Kick off the "Release" build plan on the CI system for the correct release. Contact Martin Winter for this step. Ensure all release packages build successfully.
6. Kick off the Snapcraft build plan for the release.

3.1.3 Stage 3 - Publish

1. Upload both the Debian and RPM packages to their respective repositories.
2. Coordinate with the maintainer of FRR's RPM repository to publish the RPM packages on that repository. Update the repository webpage. Verify that the instructions on the webpage work and that FRR is installable from the repository on a Red Hat system.

   Current maintainer: *Martin Winter*
3. Coordinate with the maintainer of FRR Debian package to publish the Debian packages on that repository. Update the repository webpage. Verify that the instructions on the webpage work and that FRR is installable from the repository on a Debian system.

   Current maintainer: *Jafar Al-Gharaibeh*
4. Log in to the Read The Docs instance. in the "FRRouting" project, navigate to the "Overview" tab. Ensure there is a `stable-<version>` version listed and that it is enabled. Go to "Admin" and then "Advanced Settings". Change "Default version" to the new version. This ensures that the documentation shown to visitors is that of the latest release by default.

   This step must be performed by someone with administrative access to the Read the Docs instance.
5. On GitHub, go to the <https://github.com/FRRouting/frr/releases>_ and click "Draft a new release". Write a release announcement. The release announcement should follow the template in `release-announcement-template.md`, located next to this document. Check for spelling errors, and optionally (but preferably) have other maintainers proofread the announcement text.

   Do not attach any packages or source tarballs to the GitHub release.

   Publish the release once it is reviewed.
6. Deploy Snapcraft release. Remember that this will automatically upgrade Snap users.

   Current maintainer: *Martin Winter*
7. Build and publish the Docker containers.

   Current maintainer: *Quentin Young*
8. Clone the `frr-www` repository:

```
git clone https://github.com/FRRouting/frr-www.git
```

9. Add a new release announcement, using a previous announcement as template:

```
cp content/release/<old-version>.md content/release/<new-version>.md
```

Paste the GitHub release announcement text into this document, and **remove line breaks**. In other words, this:

```
This is one continuous
sentence that should be
rendered on one line
```

Needs to be changed to this:

```
This is one continuous sentence that should be rendered on one line
```

This is very important otherwise the announcement will be unreadable on the website.

To get the number of commiters and commits, here is a couple of handy commands:

```
# The number of commits
% git log --oneline --no-merges base_8.2...base_8.1 | wc -l

# The number of commiters
% git shortlog --summary --no-merges base_8.2...base_8.1 | wc -l
```

Make sure to add a link to the GitHub releases page at the top.

10. Deploy the updated `frr-www` on the frrouting.org web server and verify that the announcement text is visible.
11. Update readthedocs.org (Default Version) for https://docs.frrouting.org to be the version of this latest release.
12. Send an email to `announce@lists.frrouting.org`. The text of this email should include text as appropriate from the GitHub release and a link to the GitHub release, Debian repository, and RPM repository.

3.2 Packaging Debian

(Tested on Ubuntu 14.04, 16.04, 17.10, 18.04, Debian jessie, stretch and buster.)

1. Install the Debian packaging tools:

```
sudo apt install fakeroot debhelper devscripts
```

2. Checkout FRR under an **unprivileged** user account:

```
git clone https://github.com/frrouting/frr.git frr
cd frr
```

If you wish to build a package for a branch other than master:

```
git checkout <branch>
```

3. Install build dependencies using the *mk-build-deps* tool from the *devscripts* package:

```
sudo mk-build-deps --install --remove debian/control
```

Alternatively, you can manually install build dependencies for your platform as outlined in *Building FRR*.

4. Install *git-buildpackage* package:

```
sudo apt-get install git-buildpackage
```

5. (optional) Append a distribution identifier if needed (see below under *Multi-Distribution builds*.)
6. Build Debian Binary and/or Source Packages:

```
gbp buildpackage --git-builder=dpkg-buildpackage --git-debian-branch="$(git rev-
↪parse --abbrev-ref HEAD)" $options
```

   Where *$options* may contain any or all of the following items:

   - build profiles specified with `-P`, e.g. `-Ppkg.frr.nortrlib,pkg.frr.rtrlib`. Multiple values are separated by commas and there must not be a space after the `-P`.

     The following build profiles are currently available:

Profile	Negation	Effect
pkg.frr.rtrlib	pkg.frr.nortrlib	builds frr-rpki-rtrlib package (or not)
pkg.frr.lua	pkg.frr.nolua	builds lua scripting extension
pkg.frr.pim6d	pkg.frr.nopim6d	builds pim6d (default enabled)

   - the `-uc -us` options to disable signing the packages with your GPG key

     (git builds of the *master* or *stable/X.X* branches won't be signed by default since their target release is set to `UNRELEASED`.)

   - the `--build=type` accepts following options (see `dpkg-buildpackage` manual page):
     - `source` builds the source package
     - `any` builds the architecture specific binary packages
     - `all` build the architecture independent binary packages
     - `binary` build the architecture specific and independent binary packages (alias for `any,all`)
     - `full` builds everything (alias for `source,any,all`)

   Alternatively, you might want to replace `dpkg-buildpackage` with `debuild` wrapper that also runs `lintian` and `debsign` on the final packages.

7. Done!

   If all worked correctly, then you should end up with the Debian packages in the parent directory of where *debuild* ran. If distributed, please make sure you distribute it together with the sources (`frr_*.orig.tar.xz`, `frr_*.debian.tar.xz` and `frr_*.dsc`)

Note: A package created from *master* or *stable/X.X* is slightly different from a package created from the *debian* branch. The changelog for the former is autogenerated and sets the Debian revision to `-0`, which causes an intentional lintian warning. The *debian* branch on the other hand has a manually maintained changelog that contains proper Debian release versioning.

3.3 Multi-Distribution builds

You can optionally append a distribution identifier in case you want to make multiple versions of the package available in the same repository.

```
dch -l '~deb8u' 'build for Debian 8 (jessie)'
dch -l '~deb9u' 'build for Debian 9 (stretch)'
dch -l '~ubuntu14.04.' 'build for Ubuntu 14.04 (trusty)'
dch -l '~ubuntu16.04.' 'build for Ubuntu 16.04 (xenial)'
dch -l '~ubuntu18.04.' 'build for Ubuntu 18.04 (bionic)'
```

Between building packages for specific distributions, the only difference in the package itself lies in the automatically generated shared library dependencies, e.g. libjson-c2 or libjson-c3. This means that the architecture independent packages should **not** have a suffix appended. Also, the current Debian testing/unstable releases should not have any suffix appended.

For example, at the end of 2018 (i.e. `buster`/Debian 10 is the current "testing" release), the following is a complete list of *.deb* files for Debian 8, 9 and 10 packages for FRR 6.0.1-1 with RPKI support:

```
frr_6.0.1-1_amd64.deb
frr_6.0.1-1~deb8u1_amd64.deb
frr_6.0.1-1~deb9u1_amd64.deb
frr-dbg_6.0.1-1_amd64.deb
frr-dbg_6.0.1-1~deb8u1_amd64.deb
frr-dbg_6.0.1-1~deb9u1_amd64.deb
frr-rpki-rtrlib_6.0.1-1_amd64.deb
frr-rpki-rtrlib_6.0.1-1~deb8u1_amd64.deb
frr-rpki-rtrlib_6.0.1-1~deb9u1_amd64.deb
frr-doc_6.0.1-1_all.deb
frr-pythontools_6.0.1-1_all.deb
```

Note that there are no extra versions of the *frr-doc* and *frr-pythontools* packages (because they are for architecture `all`, not `amd64`), and the version for Debian 10 does **not** have a `~deb10u1` suffix.

> **Warning:** Do not use the `-` character in the version suffix. The last `-` in the version number is the separator between upstream version and Debian version. `6.0.1-1~foobar-2` means upstream version `6.0.1-1~foobar`, Debian version `2`. This is not what you want.
>
> The only allowed characters in the Debian version are `0-9 A-Z a-z + . ~`

Note: The separating character for the suffix **must** be the tilde (~) because the tilde is ordered in version-comparison before the empty string. That means the order of the above packages is the following:

`6.0.1-1` newer than `6.0.1-1~deb9u1` newer than `6.0.1-1~deb8u1`

If you use another character (e.g. +), the untagged version will be regarded as the "oldest"!

3.4 Packaging Red Hat

Tested on CentOS 6, CentOS 7, CentOS 8 and Fedora 24.

1. On CentOS 6, refer to *CentOS 6* for details on installing sufficiently up-to-date package versions to enable building FRR.

   Newer automake/autoconf/bison is only needed to build the RPM and is **not** needed to install the binary RPM package.

2. Install the build dependencies for your platform. Refer to the platform-specific build documentation on how to do this.

3. Install the following additional packages:

   ```
   yum install rpm-build net-snmp-devel pam-devel libcap-devel
   ```

   For CentOS 7 and CentOS 8, the package will be built using python3 and requires additional python3 packages:

   ```
   yum install python3-devel python3-sphinx
   ```

   Note: For CentOS 8 you need to install `platform-python-devel` package to provide `/usr/bin/pathfix.py`:

   ```
   yum install platform-python-devel
   ```

   If `yum` is not present on your system, use `dnf` instead.

   You should enable `PowerTools` repo if using CentOS 8 which is disabled by default.

4. Checkout FRR:

   ```
   git clone https://github.com/frrouting/frr.git frr
   ```

5. Run Bootstrap and make distribution tar.gz:

   ```
   cd frr
   ./bootstrap.sh
   ./configure --with-pkg-extra-version=-MyRPMVersion
   make dist
   ```

   Note: The only `configure` option respected when building RPMs is `--with-pkg-extra-version`.

   ---

6. Create RPM directory structure and populate with sources:

   ```
   mkdir rpmbuild
   mkdir rpmbuild/SOURCES
   mkdir rpmbuild/SPECS
   cp redhat/*.spec rpmbuild/SPECS/
   cp frr*.tar.gz rpmbuild/SOURCES/
   ```

7. Edit `rpm/SPECS/frr.spec` with configuration as needed.

   Look at the beginning of the file and adjust the following parameters to enable or disable features as required:

```
############### FRRouting (FRR) configure options #################
# with-feature options
%{!?with_pam:           %global  with_pam           0 }
%{!?with_ospfclient:    %global  with_ospfclient    1 }
%{!?with_ospfapi:       %global  with_ospfapi       1 }
%{!?with_irdp:          %global  with_irdp          1 }
%{!?with_rtadv:         %global  with_rtadv         1 }
%{!?with_ldpd:          %global  with_ldpd          1 }
%{!?with_nhrpd:         %global  with_nhrpd         1 }
%{!?with_eigrp:         %global  with_eigrpd        1 }
%{!?with_shared:        %global  with_shared        1 }
%{!?with_multipath:     %global  with_multipath     256 }
%{!?frr_user:           %global  frr_user           frr }
%{!?vty_group:          %global  vty_group          frrvty }
%{!?with_fpm:           %global  with_fpm           0 }
%{!?with_watchfrr:      %global  with_watchfrr      1 }
%{!?with_bgp_vnc:       %global  with_bgp_vnc       0 }
%{!?with_pimd:          %global  with_pimd          1 }
%{!?with_pim6d:         %global  with_pim6d         1 }
%{!?with_rpki:          %global  with_rpki          0 }
```

8. Build the RPM:

```
rpmbuild --define "_topdir `pwd`/rpmbuild" -ba rpmbuild/SPECS/frr.spec
```

   If building with RPKI, then download and install the additional RPKI packages from https://ci1.netdef.org/browse/RPKI-RTRLIB/latestSuccessful/artifact

If all works correctly, then you should end up with the RPMs under `rpmbuild/RPMS` and the source RPM under `rpmbuild/SRPMS`.

CHAPTER

FOUR

PROCESS ARCHITECTURE

FRR is a suite of daemons that serve different functions. This document describes internal architecture of daemons, focusing their general design patterns, and especially how threads are used in the daemons that use them.

4.1 Overview

The fundamental pattern used in FRR daemons is an event loop. Some daemons use kernel threads. In these daemons, each kernel thread runs its own event loop. The event loop implementation is constructed to be thread safe and to allow threads other than its owning thread to schedule events on it. The rest of this document describes these two designs in detail.

4.2 Terminology

Because this document describes the architecture for kernel threads as well as the event system, a digression on terminology is in order here.

Historically Quagga's loop system was viewed as an implementation of userspace threading. Because of this design choice, the names for various datastructures within the event system are variations on the term "thread". The primary datastructure that holds the state of an event loop in this system is called a "threadmaster". Events scheduled on the event loop - what would today be called an 'event' or 'task' in systems such as libevent - are called "threads" and the datastructure for them is `struct thread`. To add to the confusion, these "threads" have various types, one of which is "event". To hopefully avoid some of this confusion, this document refers to these "threads" as a 'task' except where the datastructures are explicitly named. When they are explicitly named, they will be formatted `like this` to differentiate from the conceptual names. When speaking of kernel threads, the term used will be "pthread" since FRR's kernel threading implementation uses the POSIX threads API.

4.3 Event Architecture

This section presents a brief overview of the event model as currently implemented in FRR. This doc should be expanded and broken off into its own section. For now it provides basic information necessary to understand the interplay between the event system and kernel threads.

The core event system is implemented in `lib/thread.[ch]`. The primary structure is `struct thread_master`, hereafter referred to as a `threadmaster`. A `threadmaster` is a global state object, or context, that holds all the tasks currently pending execution as well as statistics on tasks that have already executed. The event system is driven by adding tasks to this data structure and then calling a function to retrieve the next task to execute. At initialization, a daemon will typically create one `threadmaster`, add a small set of initial tasks, and then run a loop to fetch each task and execute it.

These tasks have various types corresponding to their general action. The types are given by integer macros in `thread.h` and are:

THREAD_READ Task which waits for a file descriptor to become ready for reading and then executes.

THREAD_WRITE Task which waits for a file descriptor to become ready for writing and then executes.

THREAD_TIMER Task which executes after a certain amount of time has passed since it was scheduled.

THREAD_EVENT Generic task that executes with high priority and carries an arbitrary integer indicating the event type to its handler. These are commonly used to implement the finite state machines typically found in routing protocols.

THREAD_READY Type used internally for tasks on the ready queue.

THREAD_UNUSED Type used internally for `struct thread` objects that aren't being used. The event system pools `struct thread` to avoid heap allocations; this is the type they have when they're in the pool.

THREAD_EXECUTE Just before a task is run its type is changed to this. This is used to show `X` as the type in the output of `show thread cpu`.

The programmer never has to work with these types explicitly. Each type of task is created and queued via special-purpose functions (actually macros, but irrelevant for the time being) for the specific type. For example, to add a `THREAD_READ` task, you would call

```
thread_add_read(struct thread_master *master, int (*handler)(struct thread *), void *arg,
↪ int fd, struct thread **ref);
```

The `struct thread` is then created and added to the appropriate internal datastructure within the `threadmaster`. Note that the `READ` and `WRITE` tasks are independent - a `READ` task only tests for readability, for example.

4.3.1 The Event Loop

To use the event system, after creating a `threadmaster` the program adds an initial set of tasks. As these tasks execute, they add more tasks that execute at some point in the future. This sequence of tasks drives the lifecycle of the program. When no more tasks are available, the program dies. Typically at startup the first task added is an I/O task for VTYSH as well as any network sockets needed for peerings or IPC.

To retrieve the next task to run the program calls `thread_fetch()`. `thread_fetch()` internally computes which task to execute next based on rudimentary priority logic. Events (type `THREAD_EVENT`) execute with the highest priority, followed by expired timers and finally I/O tasks (type `THREAD_READ` and `THREAD_WRITE`). When scheduling a task a function and an arbitrary argument are provided. The task returned from `thread_fetch()` is then executed with `thread_call()`.

The following diagram illustrates a simplified version of this infrastructure.

The series of "task" boxes represents the current ready task queue. The various other queues for other types are not shown. The fetch-execute loop is illustrated at the bottom.

Mapping the general names used in the figure to specific FRR functions:

- `task` is `struct thread *`
- `fetch` is `thread_fetch()`
- `exec()` is `thread_call`
- `cancel()` is `thread_cancel()`
- `schedule()` is any of the various task-specific `thread_add_*` functions

Adding tasks is done with various task-specific function-like macros. These macros wrap underlying functions in `thread.c` to provide additional information added at compile time, such as the line number the task was scheduled

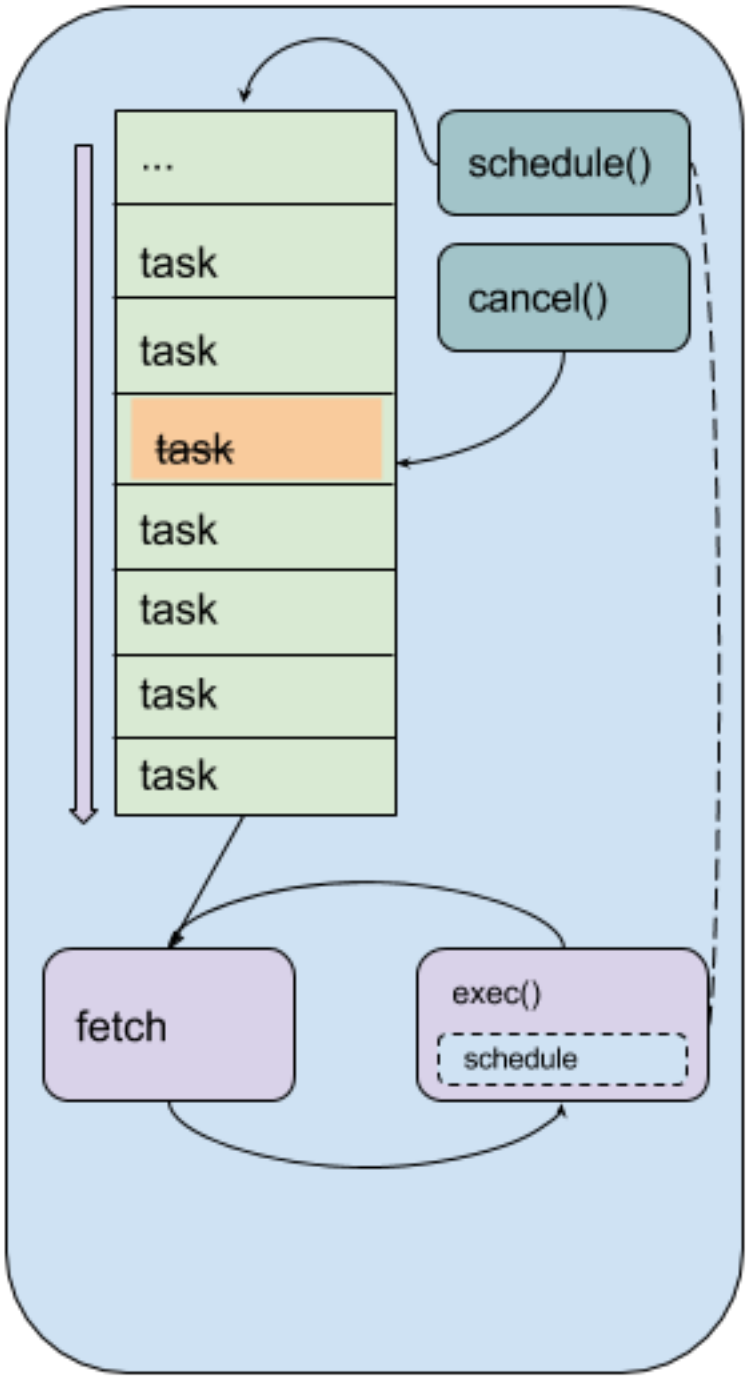


Fig. 1: Lifecycle of a program using a single threadmaster.

from, that can be accessed at runtime for debugging, logging and informational purposes. Each task type has its own specific scheduling function that follow the naming convention `thread_add_<type>`; see `thread.h` for details.

There are some gotchas to keep in mind:

- I/O tasks are keyed off the file descriptor associated with the I/O operation. This means that for any given file descriptor, only one of each type of I/O task (`THREAD_READ` and `THREAD_WRITE`) can be scheduled. For example, scheduling two write tasks one after the other will overwrite the first task with the second, resulting in total loss of the first task and difficult bugs.
- Timer tasks are only as accurate as the monotonic clock provided by the underlying operating system.
- Memory management of the arbitrary handler argument passed in the schedule call is the responsibility of the caller.

4.4 Kernel Thread Architecture

Efforts have begun to introduce kernel threads into FRR to improve performance and stability. Naturally a kernel thread architecture has long been seen as orthogonal to an event-driven architecture, and the two do have significant overlap in terms of design choices. Since the event model is tightly integrated into FRR, careful thought has been put into how pthreads are introduced, what role they fill, and how they will interoperate with the event model.

4.4.1 Design Overview

Each kernel thread behaves as a lightweight process within FRR, sharing the same process memory space. On the other hand, the event system is designed to run in a single process and drive serial execution of a set of tasks. With this consideration, a natural choice is to implement the event system within each kernel thread. This allows us to leverage the event-driven execution model with the currently existing task and context primitives. In this way the familiar execution model of FRR gains the ability to execute tasks simultaneously while preserving the existing model for concurrency.

The following figure illustrates the architecture with multiple pthreads, each running their own `threadmaster`-based event loop.

Each roundrect represents a single pthread running the same event loop described under *Event Architecture*. Note the arrow from the `exec()` box on the right to the `schedule()` box in the middle pthread. This illustrates code running in one pthread scheduling a task onto another pthread's threadmaster. A global lock for each `threadmaster` is used to synchronize these operations. The pthread names are examples.

4.4.2 Kernel Thread Wrapper

The basis for the integration of pthreads and the event system is a lightweight wrapper for both systems implemented in `lib/frr_pthread.[ch]`. The header provides a core datastructure, `struct frr_pthread`, that encapsulates structures from both POSIX threads and `thread.[ch]`. In particular, this datastructure has a pointer to a `threadmaster` that runs within the pthread. It also has fields for a name as well as start and stop functions that have signatures similar to the POSIX arguments for `pthread_create()`.

Calling `frr_pthread_new()` creates and registers a new `frr_pthread`. The returned structure has a pre-initialized `threadmaster`, and its `start` and `stop` functions are initialized to defaults that will run a basic event loop with the given threadmaster. Calling `frr_pthread_run` starts the thread with the `start` function. From there, the model is the same as the regular event model. To schedule tasks on a particular pthread, simply use the regular `thread.c` functions as usual and provide the `threadmaster` pointed to from the `frr_pthread`. As part of implementing the wrapper,

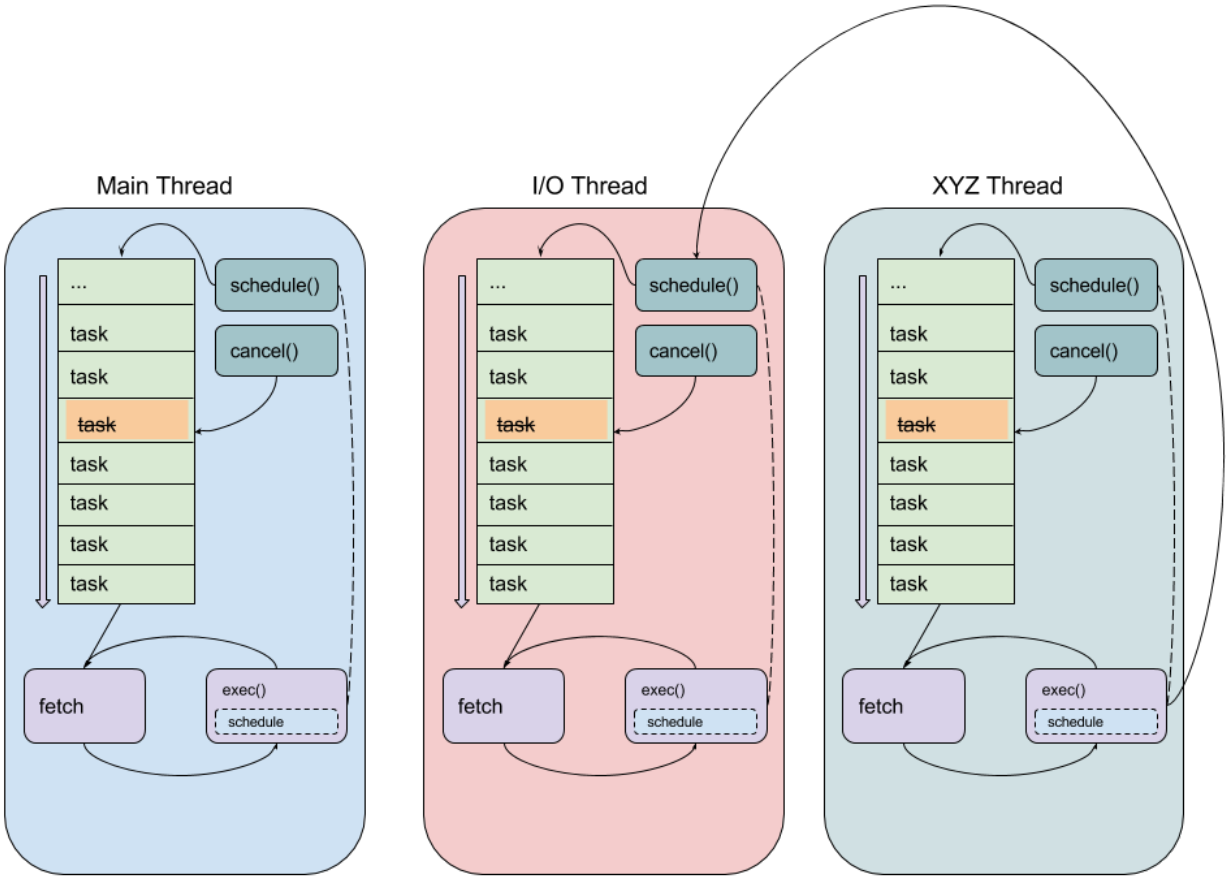


Fig. 2: Lifecycle of a program using multiple pthreads, each running their own `threadmaster`

the `thread.c` functions were made thread-safe. Consequently, it is safe to schedule events on a `threadmaster` belonging both to the calling thread as well as *any other pthread*. This serves as the basis for inter-thread communication and boils down to a slightly more complicated method of message passing, where the messages are the regular task events as used in the event-driven model. The only difference is thread cancellation, which requires calling `thread_cancel_async()` instead of `thread_cancel` to cancel a task currently scheduled on a `threadmaster` belonging to a different pthread. This is necessary to avoid race conditions in the specific case where one pthread wants to guarantee that a task on another pthread is cancelled before proceeding.

In addition, the existing commands to show statistics and other information for tasks within the event driven model have been expanded to handle multiple pthreads; running `show thread cpu` will display the usual event breakdown, but it will do so for each pthread running in the program. For example, *BGPD* runs a dedicated I/O pthread and shows the following output for `show thread cpu`:

```
frr# show thread cpu

Thread statistics for bgpd:

Showing statistics for pthread main
------------------------------------
                     CPU (user+system): Real (wall-clock):
Active   Runtime(ms)   Invoked Avg uSec Max uSecs Avg uSec Max uSecs  Type  Thread
    0       1389.000        10   138900    248000   135549    255349   T   subgroup_
↪coalesce_timer
    0          0.000         1        0         0       18        18   T   bgp_startup_
↪timer_expire
    0        850.000        18    47222    222000    47795    233814   T   work_queue_run
    0          0.000        10        0         0        6        14   T   update_
↪subgroup_merge_check_thread_cb
    0          0.000         8        0         0      117       160  W    zclient_flush_
↪data
    2          2.000         1     2000      2000      831       831 R     bgp_accept
    0          1.000         1     1000      1000     2832      2832    E  zclient_
↪connect
    1      42082.000    240574      174     37000      178     72810 R     vtysh_read
    1        152.000      1885       80      2000       96      6292 R     zclient_read
    0     549346.000   2997298      183      7000      153     20242    E  bgp_event
    0       2120.000       300     7066     14000     6813     22046   T   (bgp_holdtime_
↪timer)
    0          0.000         2        0         0       57        59   T   update_group_
↪refresh_default_originate_route_map
    0         90.000         1    90000     90000    73729     73729   T   bgp_route_map_
↪update_timer
    0       1417.000      9147      154     48000      132     61998   T   bgp_process_
↪packet
  300      71807.000   2995200       23      3000       24     11066   T   (bgp_connect_
↪timer)
    0       1894.000     12713      148     45000      112     33606   T   (bgp_generate_
↪updgrp_packets)
    0          0.000         1        0         0      105       105  W    vtysh_write
    0         52.000       599       86      2000      138      6992   T   (bgp_start_
↪timer)
    1          1.000         8      125      1000      164       593 R     vtysh_accept
    0         15.000       600       25      2000       15       153   T   (bgp_routeadv_
↪timer)
```

(continues on next page)

(continued from previous page)

```
    0         11.000       299        36       3000        53       3128 RW    bgp_connect_
↪check


Showing statistics for pthread BGP I/O thread
----------------------------------------------
                     CPU (user+system): Real (wall-clock):
Active   Runtime(ms)   Invoked Avg uSec Max uSecs Avg uSec Max uSecs  Type  Thread
    0       1611.000      9296      173     13000      188     13685 R     bgp_process_
↪reads
    0       2995.000     11753      254     26000      182     29355  W    bgp_process_
↪writes


Showing statistics for pthread BGP Keepalives thread
-----------------------------------------------------
                     CPU (user+system): Real (wall-clock):
Active   Runtime(ms)   Invoked Avg uSec Max uSecs Avg uSec Max uSecs  Type  Thread
No data to display yet.
```

Attentive readers will notice that there is a third thread, the Keepalives thread. This thread is responsible for – surprise – generating keepalives for peers. However, there are no statistics showing for that thread. Although the pthread uses the `frr_pthread` wrapper, it opts not to use the embedded `threadmaster` facilities. Instead it replaces the `start` and `stop` functions with custom functions. This was done because the `threadmaster` facilities introduce a small but significant amount of overhead relative to the pthread's task. In this case since the pthread does not need the event-driven model and does not need to receive tasks from other pthreads, it is simpler and more efficient to implement it outside of the provided event facilities. The point to take away from this example is that while the facilities to make using pthreads within FRR easy are already implemented, the wrapper is flexible and allows usage of other models while still integrating with the rest of the FRR core infrastructure. Starting and stopping this pthread works the same as it does for any other `frr_pthread`; the only difference is that event statistics are not collected for it, because there are no events.

4.5 Notes on Design and Documentation

Because of the choice to embed the existing event system into each pthread within FRR, at this time there is not integrated support for other models of pthread use such as divide and conquer. Similarly, there is no explicit support for thread pooling or similar higher level constructs. The currently existing infrastructure is designed around the concept of long-running worker threads responsible for specific jobs within each daemon. This is not to say that divide and conquer, thread pooling, etc. could not be implemented in the future. However, designs in this direction must be very careful to take into account the existing codebase. Introducing kernel threads into programs that have been written under the assumption of a single thread of execution must be done very carefully to avoid insidious errors and to ensure the program remains understandable and maintainable.

In keeping with these goals, future work on kernel threading should be extensively documented here and FRR developers should be very careful with their design choices, as poor choices tightly integrated can prove to be catastrophic for development efforts in the future.

CHAPTER

FIVE

LIBRARY FACILITIES (LIBFRR)

5.1 Memtypes

FRR includes wrappers around `malloc()` and `free()` that count the number of objects currently allocated, for each of a defined `MTYPE`.

To this extent, there are *memory groups* and *memory types*. Each memory type must belong to a memory group, this is used just to provide some basic structure.

Example:

Listing 1: mydaemon.h

```
DECLARE_MGROUP(MYDAEMON);
DECLARE_MTYPE(MYNEIGHBOR);
```

Listing 2: mydaemon.c

```
DEFINE_MGROUP(      MYDAEMON, "My daemon's memory");
DEFINE_MTYPE(       MYDAEMON, MYNEIGHBOR,     "Neighbor entry");
DEFINE_MTYPE_STATIC(MYDAEMON, MYNEIGHBORNAME, "Neighbor name");

struct neigh *neighbor_new(const char *name)
{
        struct neigh *n = XMALLOC(MYNEIGHBOR, sizeof(*n));
        n->name = XSTRDUP(MYNEIGHBORNAME, name);
        return n;
}

void neighbor_free(struct neigh *n)
{
        XFREE(MYNEIGHBORNAME, n->name);
        XFREE(MYNEIGHBOR, n);
}
```

5.1.1 Definition

struct **memtype**

This is the (internal) type used for MTYPE definitions. The macros below should be used to create these, but in some cases it is useful to pass a `struct memtype *` pointer to some helper function.

The `MTYPE_name` created by the macros is declared as a pointer, i.e. a function taking a `struct memtype *` argument can be called with an `MTYPE_name` argument (as opposed to `&MTYPE_name`.)

Note: As `MTYPE_name` is a variable assigned from `&_mt_name` and not a constant expression, it cannot be used as initializer for static variables. In the case please fall back to `&_mt_name`.

DECLARE_MGROUP(name)

This macro forward-declares a memory group and should be placed in a `.h` file. It expands to an `extern struct memgroup` statement.

DEFINE_MGROUP(mname, description)

Defines/implements a memory group. Must be placed into exactly one `.c` file (multiple inclusion will result in a link-time symbol conflict).

Contains additional logic (constructor and destructor) to register the memory group in a global list.

DECLARE_MTYPE(name)

Forward-declares a memory type and makes `MTYPE_name` available for use. Note that the `MTYPE_` prefix must not be included in the name, it is automatically prefixed.

`MTYPE_name` is created as a *static const* symbol, i.e. a compile-time constant. It refers to an `extern struct memtype _mt_name`, where *name* is replaced with the actual name.

DEFINE_MTYPE(group, name, description)

Define/implement a memory type, must be placed into exactly one `.c` file (multiple inclusion will result in a link-time symbol conflict).

Like `DEFINE_MGROUP`, this contains actual code to register the MTYPE under its group.

DEFINE_MTYPE_STATIC(group, name, description)
Same as `DEFINE_MTYPE`, but the `DEFINE_MTYPE_STATIC` variant places the C `static` keyword on the definition, restricting the MTYPE's availability to the current source file. This should be appropriate in >80% of cases.

Todo: Daemons currently have `daemon_memory.[ch]` files listing all of their MTYPEs. This is not how it should be, most of these types should be moved into the appropriate files where they are used. Only a few MTYPEs should remain non-static after that.

5.1.2 Usage

void ***XMALLOC**(struct *memtype* *mtype, size_t size)

void ***XCALLOC**(struct *memtype* *mtype, size_t size)

void ***XSTRDUP**(struct *memtype* *mtype, const char *name)
Allocation wrappers for malloc/calloc/realloc/strdup, taking an extra mtype parameter.

void ***XREALLOC**(struct *memtype* *mtype, void *ptr, size_t size)
Wrapper around realloc() with MTYPE tracking. Note that `ptr` may be NULL, in which case the function does the same as XMALLOC (regardless of whether the system realloc() supports this.)

void **XFREE**(struct *memtype* *mtype, void *ptr)
Wrapper around free(), again taking an extra mtype parameter. This is actually a macro, with the following additional properties:

- the macro contains `ptr = NULL`
- if ptr is NULL, no operation is performed (as is guaranteed by system implementations.) Do not surround XFREE with `if (ptr != NULL)` checks.

void **XCOUNTFREE**(struct *memtype* *mtype, void *ptr)
This macro is used to count the `ptr` as freed without actually freeing it. This may be needed in some very specific cases, for example, when the `ptr` was allocated using any of the above wrappers and will be freed by some external library using simple `free()`.

5.2 RCU

5.2.1 Introduction

RCU (Read-Copy-Update) is, fundamentally, a paradigm of multithreaded operation (and not a set of APIs.) The core ideas are:

- longer, complicated updates to structures are made only on private, "invisible" copies. Other threads, when they access the structure, see an older (but consistent) copy.
- once done, the updated copy is swapped in a single operation so that other threads see either the old or the new data but no inconsistent state between.
- the old instance is only released after making sure that it is impossible any other thread might still be reading it.

For more information, please search for general or Linux kernel RCU documentation; there is no way this doc can be comprehensive in explaining the interactions:

- https://en.wikipedia.org/wiki/Read-copy-update
- https://www.kernel.org/doc/html/latest/kernel-hacking/locking.html#avoiding-locks-read-copy-update
- https://lwn.net/Articles/262464/
- http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf
- http://lse.sourceforge.net/locking/rcupdate.html

RCU, the TL;DR

1. data structures are always consistent for reading. That's the "R" part.
2. reading never blocks / takes a lock.
3. rcu_read_lock is not a lock in the traditional sense. Think of it as a "reservation"; it notes what the *oldest* possible thing the thread might be seeing is, and which thus can't be deleted yet.
4. you create some object, finish it up, and then publish it.
5. publishing is an `atomic_*` call with `memory_order_release`, which tells the compiler to make sure prior memory writes have completed before doing the atomic op.
6. `ATOMLIST_* add` operations do the `memory_order_release` for you.
7. you can't touch the object after it is published, except with atomic ops.
8. because you can't touch it, if you want to change it you make a new copy, work on that, and then publish the new copy. That's the "CU" part.
9. deleting the object is also an atomic op.
10. other threads that started working before you published / deleted an object might not see the new object / still see the deleted object.
11. because other threads may still see deleted objects, the `free()` needs to be delayed. That's what *rcu_free()* is for.

When (not) to use RCU

RCU is designed for read-heavy workloads where objects are updated relatively rarely, but frequently accessed. Do *not* indiscriminately replace locking by RCU patterns.

The "copy" part of RCU implies that, while updating, several copies of a given object exist in parallel. Even after the updated copy is swapped in, the old object remains queued for freeing until all other threads are guaranteed to not be accessing it anymore, due to passing a sequence point. In addition to the increased memory usage, there may be some bursted (due to batching) malloc contention when the RCU cleanup thread does its thing and frees memory.

Other useful patterns

In addition to the full "copy object, apply changes, atomically update" approach, there are 2 "reduced" usage cases that can be done:

- atomically updating single pieces of a particular object, e.g. some flags or configuration piece
- straight up read-only / immutable objects

Both of these cases can be considered RCU "subsets". For example, when maintaining an atomic list of items, but these items only have a single integer value that needs to be updated, that value can be atomically updated without copying the entire object. However, the object still needs to be free'd through *rcu_free()* since reading/updating and deleting might be happening concurrently. The same applies for immutable objects; deletion might still race with reading so they need to be free'd through RCU.

5.2.2 FRR API

Before diving into detail on the provided functions, it is important to note that the FRR RCU API covers the **cleanup part of RCU, not the read-copy-update paradigm itself**. These parts are handled by standard C11 atomic operations, and by extension through the atomic data structures (ATOMLIST, ATOMSORT & co.)

The `rcu_*` functions only make sense in conjunction with these RCU access patterns. If you're calling the RCU API but not using these, something is wrong. The other way around is not necessarily true; it is possible to use atomic ops & datastructures with other types of locking, e.g. rwlocks.

void **rcu_read_lock**()

void **rcu_read_unlock**()

These functions acquire / release the RCU read-side lock. All access to RCU-guarded data must be inside a block guarded by these. Any number of threads may hold the RCU read-side lock at a given point in time, including both no threads at all and all threads.

The functions implement a depth counter, i.e. can be nested. The nested calls are cheap, since they only increment/decrement the counter. Therefore, any place that uses RCU data and doesn't have a guarantee that the caller holds RCU (e.g. `lib/` code) should just have its own rcu_read_lock/rcu_read_unlock pair.

At the "root" level (e.g. un-nested), these calls can incur the cost of one syscall (to `futex()`). That puts them on about the same cost as a mutex lock/unlock.

The `thread_master` code currently always holds RCU everywhere, except while doing the actual `poll()` syscall. This is both an optimization as well as an "easement" into getting RCU going. The current implementation contract is that any `struct thread *` callback is called with a RCU holding depth of 1, and that this is owned by the thread so it may (should) drop and reacquire it when doing some longer-running work.

> **Warning:** The RCU read-side lock must be held **continuously** for the entire time any piece of RCU data is used. This includes any access to RCU data after the initial `atomic_load`. If the RCU read-side lock is released, any RCU-protected pointers as well as the data they refer to become invalid, as another thread may have called *rcu_free()* on them.

struct **rcu_head**

struct **rcu_head_close**

struct **rcu_action**

The `rcu_head` structures are small (16-byte) bits that contain the queueing machinery for the RCU sweeper/cleanup mechanisms.

Any piece of data that is cleaned up by RCU needs to have a matching `rcu_head` embedded in it. If there is more than one cleanup operation to be done (e.g. closing a file descriptor), more than one `rcu_head` may be embedded.

> **Warning:** It is not possible to reuse a `rcu_head`. It is owned by the RCU code as soon as `rcu_*` is called on it.

The `_close` variant carries an extra `int fd` field to store the fd to be closed.

To minimize the amount of memory used for `rcu_head`, details about the RCU operation to be performed are moved into the `rcu_action` structure. It contains e.g. the MTYPE for *rcu_free()* calls. The pointer to be freed is stored as an offset relative to the `rcu_head`, which means it must be embedded as a struct field so the offset is constant.

The `rcu_action` structure is an implementation detail. Using `rcu_free` or `rcu_close` will set it up correctly without further code needed.

The `rcu_head` may be put in an union with other data if the other data is only used during "life" of the data, since the `rcu_head` is used only for the "death" of data. But note that other threads may still be reading a piece of data while a thread is working to free it.

void **rcu_free**(struct *memtype* *mtype, struct X *ptr, field)
Free a block of memory after RCU has ensured no other thread can be accessing it anymore. The pointer remains valid for any other thread that has called *rcu_read_lock()* before the `rcu_free` call.

> **Warning:** In some other RCU implementations, the pointer remains valid to the *calling* thread if it is holding the RCU read-side lock. This is not the case in FRR, particularly when running single-threaded. Enforcing this rule also allows static analysis to find use-after-free issues.

`mtype` is the libfrr `MTYPE_FOO` allocation type to pass to *XFREE()*.

`field` must be the name of a `struct rcu_head` member field in `ptr`. The offset of this field (which must be constant) is used to reduce the memory size of `struct rcu_head`.

Note: `rcu_free` (and `rcu_close`) calls are more efficient if they are put close to each other. When freeing several RCU'd resources, try to move the calls next to each other (even if the data structures do not directly point to each other.)

Having the calls bundled reduces the cost of adding the `rcu_head` to the RCU queue; the RCU queue is an atomic data structure whose usage will require the CPU to acquire an exclusive hold on relevant cache lines.

---

void **rcu_close**(struct *rcu_head_close* *head, int fd)
Close a file descriptor after ensuring no other thread might be using it anymore. Same as *rcu_free()*, except it calls `close` instead of `free`.

Internals

struct **rcu_thread**
Per-thread state maintained by the RCU code, set up by the following functions. A pointer to a thread's own `rcu_thread` is saved in thread-local storage.

struct *rcu_thread* ***rcu_thread_prepare**(void)

void **rcu_thread_unprepare**(struct *rcu_thread* *rcu_thread)

void **rcu_thread_start**(struct *rcu_thread* *rcu_thread)
Since the RCU code needs to have a list of all active threads, these functions are used by the `frr_pthread` code to set up threads. Teardown is automatic. It should not be necessary to call these functions.

Any thread that accesses RCU-protected data needs to be registered with these functions. Threads that do not access RCU-protected data may call these functions but do not need to.

Note that passing a pointer to RCU-protected data to some library which accesses that pointer makes the library "access RCU-protected data". In that case, either all of the library's threads must be registered for RCU, or the code must instead pass a (non-RCU) copy of the data to the library.

void **rcu_shutdown**(void)
Stop the RCU sweeper thread and make sure all cleanup has finished.

This function is called on daemon exit by the libfrr code to ensure pending RCU operations are completed. This is mostly to get a clean exit without memory leaks from queued RCU operations. It should not be necessary to call this function as libfrr handles this.

5.2.3 FRR specifics and implementation details

The FRR RCU infrastructure has the following characteristics:

- it is Epoch-based with a 32-bit wrapping counter. (This is somewhat different from other Epoch-based approaches which may be designed to only use 3 counter values, but works out to a simple implementation.)
- instead of tracking CPUs as the Linux kernel does, threads are tracked. This has exactly zero semantic impact, RCU just cares about "threads of execution", which the kernel can optimize to CPUs but we can't. But it really boils down to the same thing.
- there are no `rcu_dereference` and `rcu_assign_pointer` - use `atomic_load` and `atomic_store` instead. (These didn't exist when the Linux RCU code was created.)
- there is no `synchronize_rcu`; this is a design choice but may be revisited at a later point. `synchronize_rcu` blocks a thread until it is guaranteed that no other threads might still be accessing data structures that they may have access to at the beginning of the function call. This is a blocking design and probably not appropriate for FRR. Instead, `rcu_call` can be used to have the RCU sweeper thread make a callback after the same constraint is fulfilled in an asynchronous way. Most needs should be covered by `rcu_free` and `rcu_close`.

5.3 Type-safe containers

Note: This section previously used the term *list*; it was changed to *container* to be more clear.

5.3.1 Common container interface

FRR includes a set of container implementations with abstracted common APIs. The purpose of this is easily allow swapping out one data structure for another while also making the code easier to read and write. There is one API for unsorted containers and a similar but not identical API for sorted containers - and heaps use a middle ground of both.

For unsorted containers, the following implementations exist:

- single-linked list with tail pointer (e.g. STAILQ in BSD)
- double-linked list
- atomic single-linked list with tail pointer

Being partially sorted, the oddball structure:

- an 8-ary heap

For sorted containers, these data structures are implemented:

- single-linked list

- atomic single-linked list
- skiplist
- red-black tree (based on OpenBSD RB_TREE)
- hash table (note below)

Except for hash tables, each of the sorted data structures has a variant with unique and non-unique items. Hash tables always require unique items and mostly follow the "sorted" API but use the hash value as sorting key. Also, iterating while modifying does not work with hash tables. Conversely, the heap always has non-unique items, but iterating while modifying doesn't work either.

The following sorted structures are likely to be implemented at some point in the future:

- atomic skiplist
- atomic hash table (note below)

The APIs are all designed to be as type-safe as possible. This means that there will be a compiler warning when an item doesn't match the container, or the return value has a different type, or other similar situations. **You should never use casts with these APIs.** If a cast is necessary in relation to these APIs, there is probably something wrong with the overall design.

Only the following pieces use dynamically allocated memory:

- the hash table itself is dynamically grown and shrunk
- skiplists store up to 4 next pointers inline but will dynamically allocate memory to hold an item's 5th up to 16th next pointer (if they exist)
- the heap uses a dynamically grown and shrunk array of items

5.3.2 Cheat sheet

Available types:

```
DECLARE_LIST
DECLARE_ATOMLIST
DECLARE_DLIST

DECLARE_HEAP

DECLARE_SORTLIST_UNIQ
DECLARE_SORTLIST_NONUNIQ
DECLARE_ATOMLIST_UNIQ
DECLARE_ATOMLIST_NONUNIQ
DECLARE_SKIPLIST_UNIQ
DECLARE_SKIPLIST_NONUNIQ
DECLARE_RBTREE_UNIQ
DECLARE_RBTREE_NONUNIQ

DECLARE_HASH
```

Functions provided:

Function	LIST	HEAP	HASH	*_UNIQ	*_NONUNIQ
_init, _fini	yes	yes	yes	yes	yes
_first, _next, _next_safe, _const_first, _const_next	yes	yes	yes	yes	yes
_last, _prev, _prev_safe, _const_last, _const_prev	DLIST only	–	–	RB only	RB only
_swap_all	yes	yes	yes	yes	yes
_anywhere	yes	–	–	–	–
_add_head, _add_tail, _add_after	yes	–	–	–	–
_add	–	yes	yes	yes	yes
_member	yes	yes	yes	yes	yes
_del, _pop	yes	yes	yes	yes	yes
_find, _const_find	–	–	yes	yes	–
_find_lt, _find_gteq, _const_find_lt, _const_find_gteq	–	–	–	yes	yes
use with frr_each() macros	yes	yes	yes	yes	yes

5.3.3 Datastructure type setup

Each of the data structures has a `PREDECL_*` and a `DECLARE_*` macro to set up an "instantiation" of the container. This works somewhat similar to C++ templating, though much simpler.

In all following text, the Z prefix is replaced with a name chosen for the instance of the datastructure.

The common setup pattern will look like this:

```
#include <typesafe.h>

PREDECL_XXX(Z);
struct item {
    int otherdata;
    struct Z_item mylistitem;
}

struct Z_head mylisthead;

/* unsorted: */
DECLARE_XXX(Z, struct item, mylistitem);

/* sorted, items that compare as equal cannot be added to list */
int compare_func(const struct item *a, const struct item *b);
DECLARE_XXX_UNIQ(Z, struct item, mylistitem, compare_func);

/* sorted, items that compare as equal can be added to list */
int compare_func(const struct item *a, const struct item *b);
DECLARE_XXX_NONUNIQ(Z, struct item, mylistitem, compare_func);

/* hash tables: */
int compare_func(const struct item *a, const struct item *b);
uint32_t hash_func(const struct item *a);
DECLARE_XXX(Z, struct item, mylistitem, compare_func, hash_func);
```

`XXX` is replaced with the name of the data structure, e.g. `SKIPLIST` or `ATOMLIST`. The `DECLARE_XXX` invocation can

either occur in a *.h* file (if the container needs to be accessed from several C files) or it can be placed in a *.c* file (if the container is only accessed from that file.) The `PREDECL_XXX` invocation defines the `struct Z_item` and `struct Z_head` types and must therefore occur before these are used.

To switch between compatible data structures, only these two lines need to be changes. To switch to a data structure with a different API, some source changes are necessary.

5.3.4 Common iteration macros

The following iteration macros work across all data structures:

frr_each(Z, head, item)
Equivalent to:

```
for (item = Z_first(&head); item; item = Z_next(&head, item))
```

Note that this will fail if the container is modified while being iterated over.

frr_each_safe(Z, head, item)
Same as the previous, but the next element is pre-loaded into a "hidden" variable (named `Z_safe`.) Equivalent to:

```
for (item = Z_first(&head); item; item = next) {
    next = Z_next_safe(&head, item);
    ...
}
```

> **Warning:** Iterating over hash tables while adding or removing items is not possible. The iteration position will be corrupted when the hash tables is resized while iterating. This will cause items to be skipped or iterated over twice.

frr_each_from(Z, head, item, from)
Iterates over the container, starting at item `from`. This variant is "safe" as in the previous macro. Equivalent to:

```
for (item = from; item; item = from) {
    from = Z_next_safe(&head, item);
    ...
}
```

> **Note:** The `from` variable is written to. This is intentional - you can resume iteration after breaking out of the loop by keeping the `from` value persistent and reusing it for the next loop.

frr_rev_each(Z, head, item)

frr_rev_each_safe(Z, head, item)

frr_rev_each_from(Z, head, item, from)
Reverse direction variants of the above. Only supported on containers that implement `_last` and `_prev` (i.e. `RBTREE` and `DLIST`).

To iterate over `const` pointers, add `_const` to the name of the datastructure (`Z` above), e.g. `frr_each (mylist, head, item)` becomes `frr_each (mylist_const, head, item)`.

5.3.5 Common API

The following documentation assumes that a container has been defined using `Z` as the name, and `itemtype` being the type of the items (e.g. `struct item`.)

void **Z_init**(struct Z_head*)
: Initializes the container for use. For most implementations, this just sets some values. Hash tables are the only implementation that allocates memory in this call.

void **Z_fini**(struct Z_head*)
: Reverse the effects of *Z_init()*. The container must be empty when this function is called.

  Warning: This function may `assert()` if the container is not empty.

size_t **Z_count**(const struct Z_head*)
: Returns the number of items in a structure. All structures store a counter in their *Z_head* so that calling this function completes in O(1).

  Note: For atomic containers with concurrent access, the value will already be outdated by the time this function returns and can therefore only be used as an estimate.

bool **Z_member**(const struct Z_head*, const itemtype*)
: Determines whether some item is a member of the given container. The item must either be valid on some container, or set to all zeroes.

  On some containers, if no faster way to determine membership is possible, this is simply `item == Z_find(head, item)`.

  Not currently available for atomic containers.

const itemtype ***Z_const_first**(const struct Z_head*)

itemtype ***Z_first**(struct Z_head*)
: Returns the first item in the structure, or `NULL` if the structure is empty. This is O(1) for all data structures except red-black trees where it is O(log n).

const itemtype ***Z_const_last**(const struct Z_head*)

itemtype ***Z_last**(struct Z_head*)
: Last item in the structure, or `NULL`. Only available on containers that support reverse iteration (i.e. `RBTREE` and `DLIST`).

itemtype ***Z_pop**(struct Z_head*)
: Remove and return the first item in the structure, or `NULL` if the structure is empty. Like *Z_first()*, this is O(1) for all data structures except red-black trees where it is O(log n) again.

  This function can be used to build queues (with unsorted structures) or priority queues (with sorted structures.)

  Another common pattern is deleting all container items:

```
while ((item = Z_pop(head)))
    item_free(item);
```

Note: This function can - and should - be used with hash tables. It is not affected by the "modification while iterating" problem. To remove all items from a hash table, use the loop demonstrated above.

const itemtype ***Z_const_next**(const struct Z_head*, const itemtype *prev)

itemtype ***Z_next**(struct Z_head*, itemtype *prev)
Return the item that follows after `prev`, or `NULL` if `prev` is the last item.

Warning: `prev` must not be `NULL`! Use *Z_next_safe()* if `prev` might be `NULL`.

itemtype ***Z_next_safe**(struct Z_head*, itemtype *prev)
Same as *Z_next()*, except that `NULL` is returned if `prev` is `NULL`.

const itemtype ***Z_const_prev**(const struct Z_head*, const itemtype *next)

itemtype ***Z_prev**(struct Z_head*, itemtype *next)

itemtype ***Z_prev_safe**(struct Z_head*, itemtype *next)
As above, but preceding item. Only available on structures that support reverse iteration (i.e. `RBTREE` and `DLIST`).

itemtype ***Z_del**(struct Z_head*, itemtype *item)
Remove `item` from the container and return it.

Note: This function's behaviour is undefined if `item` is not actually on the container. Some structures return `NULL` in this case while others return `item`. The function may also call `assert()` (but most don't.)

itemtype ***Z_swap_all**(struct Z_head*, struct Z_head*)
Swap the contents of 2 containers (of identical type). This exchanges the contents of the two head structures and updates pointers if necessary for the particular data structure. Fast for all structures.

(Not currently available on atomic containers.)

Todo: `Z_del_after()` / `Z_del_hint()`?

5.3.6 API for unsorted structures

Since the insertion position is not pre-defined for unsorted data, there are several functions exposed to insert data:

Note: `item` must not be `NULL` for any of the following functions.

DECLARE_XXX(Z, type, field)

Parameters

- **XXX** (*listtype*) – `LIST`, `DLIST` or `ATOMLIST` to select a data structure implementation.

- **Z** (*token*) – Gives the name prefix that is used for the functions created for this instantiation. `DECLARE_XXX(foo, ...)` gives `struct foo_item`, `foo_add_head()`, `foo_count()`, etc. Note that this must match the value given in `PREDECL_XXX(foo)`.
- **type** (*typename*) – Specifies the data type of the list items, e.g. `struct item`. Note that `struct` must be added here, it is not automatically added.
- **field** (*token*) – References a struct member of `type` that must be typed as `struct foo_item`. This struct member is used to store "next" pointers or other data structure specific data.

void **Z_add_head**(struct Z_head*, itemtype *item)
Insert an item at the beginning of the structure, before the first item. This is an O(1) operation for non-atomic lists.

void **Z_add_tail**(struct Z_head*, itemtype *item)
Insert an item at the end of the structure, after the last item. This is also an O(1) operation for non-atomic lists.

void **Z_add_after**(struct Z_head*, itemtype *after, itemtype *item)
Insert `item` behind `after`. If `after` is `NULL`, the item is inserted at the beginning of the list as with *Z_add_head()*. This is also an O(1) operation for non-atomic lists.

A common pattern is to keep a "previous" pointer around while iterating:

```
itemtype *prev = NULL, *item;

frr_each_safe(Z, head, item) {
    if (something) {
        Z_add_after(head, prev, item);
        break;
    }
    prev = item;
}
```

Todo: maybe flip the order of `item` & `after`? `Z_add_after(head, item, after)`

bool **Z_anywhere**(const itemtype*)
Returns whether an item is a member of *any* container of this type. The item must either be valid on some container, or set to all zeroes.

Guaranteed to be fast (pointer compare or similar.)

Not currently available for sorted and atomic containers. Might be added for sorted containers at some point (when needed.)

5.3.7 API for sorted structures

Sorted data structures do not need to have an insertion position specified, therefore the insertion calls are different from unsorted containers. Also, sorted containers can be searched for a value.

DECLARE_XXX_UNIQ(Z, type, field, compare_func)

Parameters

- **XXX** (*listtype*) – One of the following: `SORTLIST` (single-linked sorted list), `SKIPLIST` (skiplist), `RBTREE` (RB-tree) or `ATOMSORT` (atomic single-linked list).

- **Z** (*token*) – Gives the name prefix that is used for the functions created for this instantiation. `DECLARE_XXX(foo, ...)` gives `struct foo_item`, `foo_add()`, `foo_count()`, etc. Note that this must match the value given in `PREDECL_XXX(foo)`.
- **type** (*typename*) – Specifies the data type of the items, e.g. `struct item`. Note that `struct` must be added here, it is not automatically added.
- **field** (*token*) – References a struct member of `type` that must be typed as `struct foo_item`. This struct member is used to store "next" pointers or other data structure specific data.
- **compare_func** (*funcptr*) – Item comparison function, must have the following function signature: `int function(const itemtype *, const itemtype*)`. This function may be static if the container is only used in one file.

DECLARE_XXX_NONUNIQ(Z, type, field, compare_func)
Same as above, but allow adding multiple items to the container that compare as equal in `compare_func`. Ordering between these items is undefined and depends on the container implementation.

itemtype ***Z_add**(struct Z_head*, itemtype *item)
Insert an item at the appropriate sorted position. If another item exists in the container that compares as equal (`compare_func() == 0`), `item` is not inserted and the already-existing item in the container is returned. Otherwise, on successful insertion, `NULL` is returned.

For `_NONUNIQ` containers, this function always returns NULL since `item` can always be successfully added to the container.

const itemtype ***Z_const_find**(const struct Z_head*, const itemtype *ref)

itemtype ***Z_find**(struct Z_head*, const itemtype *ref)
Search the container for an item that compares equal to `ref`. If no equal item is found, return `NULL`.

This function is likely used with a temporary stack-allocated value for `ref` like so:

```
itemtype searchfor = { .foo = 123 };

itemtype *item = Z_find(head, &searchfor);
```

Note: The `Z_find()` function is only available for containers that contain unique items (i.e. `DECLARE_XXX_UNIQ`.) This is because on a container with non-unique items, more than one item may compare as equal to the item that is searched for.

const itemtype ***Z_const_find_gteq**(const struct Z_head*, const itemtype *ref)

itemtype ***Z_find_gteq**(struct Z_head*, const itemtype *ref)
Search the container for an item that compares greater or equal to `ref`. See *Z_find()* above.

const itemtype ***Z_const_find_lt**(const struct Z_head*, const itemtype *ref)

itemtype ***Z_find_lt**(struct Z_head*, const itemtype *ref)
Search the container for an item that compares less than `ref`. See *Z_find()* above.

5.3.8 API for hash tables

DECLARE_HASH(Z, type, field, compare_func, hash_func)

Parameters

- **HASH** (*listtype*) – Only `HASH` is currently available.
- **Z** (*token*) – Gives the name prefix that is used for the functions created for this instantiation. `DECLARE_XXX(foo, ...)` gives `struct foo_item`, `foo_add()`, `foo_count()`, etc. Note that this must match the value given in `PREDECL_XXX(foo)`.
- **type** (*typename*) – Specifies the data type of the items, e.g. `struct item`. Note that `struct` must be added here, it is not automatically added.
- **field** (*token*) – References a struct member of `type` that must be typed as `struct foo_item`. This struct member is used to store "next" pointers or other data structure specific data.
- **compare_func** (*funcptr*) – Item comparison function, must have the following function signature: `int function(const itemtype *, const itemtype*)`. This function may be static if the container is only used in one file. For hash tables, this function is only used to check for equality, the ordering is ignored.
- **hash_func** (*funcptr*) – Hash calculation function, must have the following function signature: `uint32_t function(const itemtype *)`. The hash value for items stored in a hash table is cached in each item, so this value need not be cached by the user code.

> **Warning:** Items that compare as equal cannot be inserted. Refer to the notes about sorted structures in the previous section.

void **Z_init_size**(struct Z_head*, size_t size)
Same as *Z_init()* but preset the minimum hash table to `size`.

Hash tables also support *Z_add()* and *Z_find()* with the same semantics as noted above. *Z_find_gteq()* and *Z_find_lt()* are **not** provided for hash tables.

Hash table invariants

There are several ways to injure yourself using the hash table API.

First, note that there are two functions related to computing uniqueness of objects inserted into the hash table. There is a hash function and a comparison function. The hash function computes the hash of the object. Our hash table implementation uses chaining. This means that your hash function does not have to be perfect; multiple objects having the same computed hash will be placed into a linked list corresponding to that key. The closer to perfect the hash function, the better performance, as items will be more evenly distributed and the chain length will not be long on any given lookup, minimizing the number of list operations required to find the correct item. However, the comparison function *must* be perfect, in the sense that any two unique items inserted into the hash table must compare not equal. At insertion time, if you try to insert an item that compares equal to an existing item the insertion will not happen and `hash_get()` will return the existing item. However, this invariant *must* be maintained while the object is in the hash table. Suppose you insert items `A` and `B` into the hash table which both hash to the same value `1234` but do not compare equal. They will be placed in a chain like so:

```
1234 : A -> B
```

Now suppose you do something like this elsewhere in the code:

```
*A = *B
```

I.e. you copy all fields of `B` into `A`, such that the comparison function now says that they are equal based on their contents. At this point when you look up `B` in the hash table, `hash_get()` will search the chain for the first item that compares equal to `B`, which will be `A`. This leads to insidious bugs.

Warning: Never modify the values looked at by the comparison or hash functions after inserting an item into a hash table.

A similar situation can occur with the hash allocation function. `hash_get()` accepts a function pointer that it will call to get the item that should be inserted into the list if the provided item is not already present. There is a builtin function, `hash_alloc_intern`, that will simply return the item you provided; if you always want to store the value you pass to `hash_get` you should use this one. If you choose to provide a different one, that function *must* return a new item that hashes and compares equal to the one you provided to `hash_get()`. If it does not the behavior of the hash table is undefined.

Warning: Always make sure your hash allocation function returns a value that hashes and compares equal to the item you provided to `hash_get()`.

Finally, if you maintain pointers to items you have inserted into a hash table, then before deallocating them you must release them from the hash table. This is basic memory management but worth repeating as bugs have arisen from failure to do this.

5.3.9 API for heaps

Heaps provide the same API as the sorted data structures, except:

- none of the find functions (*Z_find()*, *Z_find_gteq()* or *Z_find_lt()*) are available.
- iterating over the heap yields the items in semi-random order, only the first item is guaranteed to be in order and actually the "lowest" item on the heap. Being a heap, only the rebalancing performed on removing the first item (either through *Z_pop()* or *Z_del()*) causes the new lowest item to bubble up to the front.
- all heap modifications are O(log n). However, cacheline efficiency and latency is likely quite a bit better than with other data structures.

5.3.10 Atomic lists

atomlist.h provides an unsorted and a sorted atomic single-linked list. Since atomic memory accesses can be considerably slower than plain memory accessses (depending on the CPU type), these lists should only be used where necessary.

The following guarantees are provided regarding concurrent access:

- the operations are lock-free but not wait-free.

  Lock-free means that it is impossible for all threads to be blocked. Some thread will always make progress, regardless of what other threads do. (This even includes a random thread being stopped by a debugger in a random location.)

Wait-free implies that the time any single thread might spend in one of the calls is bounded. This is not provided here since it is not normally relevant to practical operations. What this means is that if some thread is hammering a particular list with requests, it is possible that another thread is blocked for an extended time. The lock-free guarantee still applies since the hammering thread is making progress.

- without a RCU mechanism in place, the point of contention for atomic lists is memory deallocation. As it is, **a rwlock is required for correct operation**. The *read* lock must be held for all accesses, including reading the list, adding items to the list, and removing items from the list. The *write* lock must be acquired and released before deallocating any list element. If this is not followed, an use-after-free can occur as a MT race condition when an element gets deallocated while another thread is accessing the list.

  Note: The *write* lock does not need to be held for deleting items from the list, and there should not be any instructions between the `pthread_rwlock_wrlock` and `pthread_rwlock_unlock`. The write lock is used as a sequence point, not as an exclusion mechanism.

- insertion operations are always safe to do with the read lock held. Added items are immediately visible after the insertion call returns and should not be touched anymore.
- when removing a *particular* (pre-determined) item, the caller must ensure that no other thread is attempting to remove that same item. If this cannot be guaranteed by architecture, a separate lock might need to be added.
- concurrent *pop* calls are always safe to do with only the read lock held. This does not fall under the previous rule since the *pop* call will select the next item if the first is already being removed by another thread.

  Deallocation locking still applies. Assume another thread starts reading the list, but gets task-switched by the kernel while reading the first item. *pop* will happily remove and return that item. If it is deallocated without acquiring and releasing the write lock, the other thread will later resume execution and try to access the now-deleted element.
- the list count should be considered an estimate. Since there might be concurrent insertions or removals in progress, it might already be outdated by the time the call returns. No attempt is made to have it be correct even for a nanosecond.

Overall, atomic lists are well-suited for MT queues; concurrent insertion, iteration and removal operations will work with the read lock held.

Code snippets

Iteration:

```
struct item *i;

pthread_rwlock_rdlock(&itemhead_rwlock);
frr_each(itemlist, &itemhead, i) {
  /* lock must remain held while iterating */
  ...
}
pthread_rwlock_unlock(&itemhead_rwlock);
```

Head removal (pop) and deallocation:

```
struct item *i;

pthread_rwlock_rdlock(&itemhead_rwlock);
i = itemlist_pop(&itemhead);
```

(continues on next page)

(continued from previous page)

```
pthread_rwlock_unlock(&itemhead_rwlock);

/* i might still be visible for another thread doing an
 * frr_each() (but won't be returned by another pop()) */
...

pthread_rwlock_wrlock(&itemhead_rwlock);
pthread_rwlock_unlock(&itemhead_rwlock);
/* i now guaranteed to be gone from the list.
 * note nothing between wrlock() and unlock() */
XFREE(MTYPE_ITEM, i);
```

5.3.11 FAQ

What are the semantics of `const` in the container APIs? `const` pointers to list heads and/or items are interpreted to mean that both the container itself as well as the data items are read-only.

Why is it `PREDECL` + `DECLARE` instead of `DECLARE` + `DEFINE`? The rule is that a `DEFINE` must be in a `.c` file, and linked exactly once because it defines some kind of global symbol. This is not the case for the data structure macros; they only define `static` symbols and it is perfectly fine to include both `PREDECL` and `DECLARE` in a header file. It is also perfectly fine to have the same `DECLARE` statement in 2 `.c` files, but only **if the macro arguments are identical.** Maybe don't do that unless you really need it.

5.3.12 FRR lists

Todo: document

5.3.13 BSD lists

Todo: refer to external docs

5.4 Logging

One of the most frequent decisions to make while writing code for FRR is what to log, what level to log it at, and when to log it. Here is a list of recommendations for these decisions.

5.4.1 printfrr()

`printfrr()` is FRR's modified version of `printf()`, designed to make life easier when printing nontrivial datastructures. The following variants are available:

ssize_t **snprintfrr**(char *buf, size_t len, const char *fmt, ...)

ssize_t **vsnprintfrr**(char *buf, size_t len, const char *fmt, va_list)
: These correspond to `snprintf`/`vsnprintf`. If you pass NULL for buf or 0 for len, no output is written but the return value is still calculated.

  The return value is always the full length of the output, unconstrained by *len*. It does **not** include the terminating `\0` character. A malformed format string can result in a `-1` return value.

ssize_t **csnprintfrr**(char *buf, size_t len, const char *fmt, ...)

ssize_t **vcsnprintfrr**(char *buf, size_t len, const char *fmt, va_list)
: Same as above, but the `c` stands for "continue" or "concatenate". The output is appended to the string instead of overwriting it.

char ***asprintfrr**(struct *memtype* *mt, const char *fmt, ...)

char ***vasprintfrr**(struct *memtype* *mt, const char *fmt, va_list)
: These functions allocate a dynamic buffer (using MTYPE *mt*) and print to that. If the format string is malformed, they return a copy of the format string, so the return value is always non-NULL and always dynamically allocated with *mt*.

char ***asnprintfrr**(struct *memtype* *mt, char *buf, size_t len, const char *fmt, ...)

char ***vasnprintfrr**(struct *memtype* *mt, char *buf, size_t len, const char *fmt, va_list)
: This variant tries to use the static buffer provided, but falls back to dynamic allocation if it is insufficient.

  The return value can be either *buf* or a newly allocated string using *mt*. You MUST free it like this:

```
char *ret = asnprintfrr(MTYPE_FOO, buf, sizeof(buf), ...);
if (ret != buf)
   XFREE(MTYPE_FOO, ret);
```

ssize_t **bprintfrr**(struct fbuf *fb, const char *fmt, ...)

ssize_t **vbprintfrr**(struct fbuf *fb, const char *fmt, va_list)
: These are the "lowest level" functions, which the other variants listed above use to implement their functionality on top. Mainly useful for implementing printfrr extensions since those get a `struct fbuf *` to write their output to.

FMT_NSTD(expr)
: This macro turns off/on format warnings as needed when non-ISO-C compatible printfrr extensions are used (e.g. `%.*p` or `%Ld`.):

```
vty_out(vty, "standard compatible %pI4\n", &addr);
FMT_NSTD(vty_out(vty, "non-standard %-47.*pHX\n", (int)len, buf));
```

  When the frr-format plugin is in use, this macro is a no-op since the frr-format plugin supports all printfrr extensions. Since the FRR CI includes a system with the plugin enabled, this means format errors will not slip by undetected even with FMT_NSTD.

Note: `printfrr()` does not support the `%n` format.

AS-Safety

`printfrr()` are AS-Safe under the following conditions:

- the `[v]as[n]printfrr` variants are not AS-Safe (allocating memory)
- floating point specifiers are not AS-Safe (system printf is used for these)
- the positional `%1$d` syntax should not be used (8 arguments are supported while AS-Safe)
- extensions are only AS-Safe if their printer is AS-Safe

5.4.2 printfrr Extensions

`printfrr()` format strings can be extended with suffixes after *%p* or *%d*. Printf features like field lengths can be used normally with these extensions, e.g. `%-15pI4` works correctly, **except if the extension consumes the width or precision**. Extensions that do so are listed below as `%*pXX` rather than `%pXX`.

The extension specifier after `%p` or `%d` is always an uppercase letter; by means of established pattern uppercase letters and numbers form the type identifier which may be followed by lowercase flags.

You can grep the FRR source for `printfrr_ext_autoreg` to see all extended printers and what exactly they do. More printers are likely to be added as needed/useful, so the list here may be outdated.

Note: The `zlog_*/flog_*` and `vty_out` functions all use printfrr internally, so these extensions are available there. However, they are **not** available when calling `snprintf` directly. You need to call `snprintfrr` instead.

Networking data types

%**pI4** (*struct in_addr **, *in_addr_t **)
`1.2.3.4`

`%pI4s`: * — print star instead of `0.0.0.0` (for multicast)

%**pI6** (*struct in6_addr **)
`fe80::1234`

`%pI6s`: * — print star instead of `::` (for multicast)

%**pEA** (*struct ethaddr **)
`01:23:45:67:89:ab`

%**pIA** (*struct ipaddr **)
`1.2.3.4` / `fe80::1234`

`%pIAs`: — print star instead of zero address (for multicast)

%**pFX** (*struct prefix **)
`1.2.3.0/24` / `fe80::1234/64`

This accepts the following types:

- `prefix`

- `prefix_ipv4`
- `prefix_ipv6`
- `prefix_eth`
- `prefix_evpn`
- `prefix_fs`

It does **not** accept the following types:

- `prefix_ls`
- `prefix_rd`
- `prefix_sg` (use *%pPSG4*)
- `prefixptr` (dereference to get `prefix`)
- `prefixconstptr` (dereference to get `prefix`)

Options:

`%pFXh`: (address only) `1.2.3.0` / `fe80::1234`

%**pPSG4** (*struct prefix_sg **)
`(*,1.2.3.4)`

This is *(S,G)* output for use in zebra. (Note prefix_sg is not a prefix "subclass" like the other prefix_* structs.)

%**pSU** (*union sockunion **)
`%pSU`: `1.2.3.4` / `fe80::1234`

`%pSUs`: `1.2.3.4` / `fe80::1234%89` (adds IPv6 scope ID as integer)

`%pSUp`: `1.2.3.4:567` / `[fe80::1234]:567` (adds port)

`%pSUps`: `1.2.3.4:567` / `[fe80::1234%89]:567` (adds port and scope ID)

%**pRN** (*struct route_node **, *struct bgp_node **, *struct agg_node **)
`192.168.1.0/24` (dst-only node)

`2001:db8::/32 from fe80::/64` (SADR node)

%**pNH** (*struct nexthop **)
`%pNHvv`: `via 1.2.3.4, eth0` — verbose zebra format

`%pNHv`: `1.2.3.4, via eth0` — slightly less verbose zebra format

`%pNHs`: `1.2.3.4 if 15` — same as `nexthop2str()`

`%pNHcg`: `1.2.3.4` — compact gateway only

`%pNHci`: `eth0` — compact interface only

%**dPF** (*int*)
`AF_INET`

Prints an *AF_* / PF_** constant. `PF` is used here to avoid confusion with *AFI* constants, even though the FRR codebase prefers *AF_INET* over *PF_INET* & co.

%**dSO** (*int*)
`SOCK_STREAM`

Time/interval formats

%**pTS** (*struct timespec* *)

%**pTV** (*struct timeval* *)

%**pTT** (*time_t* *)

Above 3 options internally result in the same code being called, support the same flags and produce equal output with one exception: `%pTT` has no sub-second precision and the formatter will never print a (nonsensical) `.000`.

Exactly one of `I`, `M` or `R` must immediately follow after `TS`/`TV`/`TT` to specify whether the input is an interval, monotonic timestamp or realtime timestamp:

`%pTVI`: input is an interval, not a timestamp. Print interval.

`%pTVIs`: input is an interval, convert to wallclock by subtracting it from current time (i.e. interval has passed **s**ince.)

`%pTVIu`: input is an interval, convert to wallclock by adding it to current time (i.e. **u**ntil interval has passed.)

`%pTVM` - input is a timestamp on CLOCK_MONOTONIC, convert to wallclock time (by grabbing current CLOCK_MONOTONIC and CLOCK_REALTIME and doing the math) and print calendaric date.

`%pTVMs` - input is a timestamp on CLOCK_MONOTONIC, print interval **s**ince that timestamp (elapsed.)

`%pTVMu` - input is a timestamp on CLOCK_MONOTONIC, print interval **u**ntil that timestamp (deadline.)

`%pTVR` - input is a timestamp on CLOCK_REALTIME, print calendaric date.

`%pTVRs` - input is a timestamp on CLOCK_REALTIME, print interval **s**ince that timestamp.

`%pTVRu` - input is a timestamp on CLOCK_REALTIME, print interval **u**ntil that timestamp.

`%pTVA` - reserved for CLOCK_TAI in case a PTP implementation is interfaced to FRR. Not currently implemented.

Note: If `%pTVRs` or `%pTVRu` are used, this is generally an indication that a CLOCK_MONOTONIC timestamp should be used instead (or added in parallel.) CLOCK_REALTIME might be adjusted by NTP, PTP or similar procedures, causing bogus intervals to be printed.

`%pTVM` on first look might be assumed to have the same problem, but on closer thought the assumption is always that current system time is correct. And since a CLOCK_MONOTONIC interval is also quite safe to assume to be correct, the (past) absolute timestamp to be printed from this can likely be correct even if it doesn't match what CLOCK_REALTIME would have indicated at that point in the past. This logic does, however, not quite work for *future* times.

Generally speaking, almost all use cases in FRR should (and do) use CLOCK_MONOTONIC (through `monotime()`.)

Flags common to printing calendar times and intervals:

`p`: include spaces in appropriate places (depends on selected format.)

`%p.3TV...`: specify sub-second resolution (use with `FMT_NSTD` to suppress gcc warning.) As noted above, `%pTT` will never print sub-second digits since there are none. Only some formats support printing sub-second digits and the default may vary.

The following flags are available for printing calendar times/dates:

(no flag): `Sat Jan 1 00:00:00 2022` - print output from `ctime()`, in local time zone. Since FRR does not currently use/enable locale support, this is always the C locale. (Locale support getting added is unlikely for the time being and would likely break other things worse than this.)

`i`: `2022-01-01T00:00:00.123` - ISO8601 timestamp in local time zone (note there is no `Z` or `+00:00` suffix.) Defaults to millisecond precision.

`ip`: `2022-01-01 00:00:00.123` - use readable form of ISO8601 with space instead of `T` separator.

The following flags are available for printing intervals:

(no flag): `9w9d09:09:09.123` - does not match any preexisting format; added because it does not lose precision (like `t`) for longer intervals without printing huge numbers (like `h`/`m`). Defaults to millisecond precision. The week/day fields are left off if they're zero, `p` adds a space after the respective letter.

`t`: `9w9d09h`, `9d09h09m`, `09:09:09` - this replaces `frrtime_to_interval()`. `p` adds spaces after week/day/hour letters.

`d`: print decimal number of seconds. Defaults to millisecond precision.

`x` / `tx` / `dx`: Like no flag / `t` / `d`, but print `-` for zero or negative intervals (for use with unset timers.)

`h`: `09:09:09`

`hx`: `09:09:09`, `--:--:--` - this replaces `pim_time_timer_to_hhmmss()`.

`m`: `09:09`

`mx`: `09:09`, `--:--` - this replaces `pim_time_timer_to_mmss()`.

FRR library helper formats

%**pTH** (*struct thread* *)
: Print remaining time on timer thread. Interval-printing flag characters listed above for `%pTV` can be added, e.g. `%pTHtx`.

  `NULL` pointers are printed as `-`.

%**pTHD** (*struct thread* *)
: Print debugging information for given thread. Sample output:

```
{(thread *)NULL}
{(thread *)0x55a3b5818910 arg=0x55a3b5827c50 timer  r=7.824      mld_t_query() &mld_
↪ifp->t_query from pimd/pim6_mld.c:1369}
{(thread *)0x55a3b5827230 arg=0x55a3b5827c50 read   fd=16        mld_t_recv() &mld_
↪ifp->t_recv from pimd/pim6_mld.c:1186}
```

  (The output is aligned to some degree.)

FRR daemon specific formats

The following formats are only available in specific daemons, as the code implementing them is part of the daemon, not the library.

zebra

%**pZN** (*struct route_node **)
Print information for a RIB node, including zebra-specific data.

`::/0 src fe80::/64 (MRIB)` (`%pZN`)

`1234` (`%pZNt` - table number)

bgpd

%**pBD** (*struct bgp_dest **)
Print prefix for a BGP destination.

`fe80::1234/64`

%**pBP** (*struct peer **)
`192.168.1.1(leaf1.frrouting.org)`

Print BGP peer's IP and hostname together.

pimd/pim6d

%**pPA** (*pim_addr **)
Format IP address according to IP version (pimd vs. pim6d) being compiled.

`fe80::1234` / `10.0.0.1`

* (`%pPAs` - replace 0.0.0.0/:: with star)

%**pSG** (*pim_sgaddr **)
Format S,G pair according to IP version (pimd vs. pim6d) being compiled. Braces are included.

`(*,224.0.0.0)`

General utility formats

%**m** (*no argument*)
`Permission denied`

Prints `strerror(errno)`. Does **not** consume any input argument, don't pass `errno`!

(This is a GNU extension not specific to FRR. FRR guarantees it is available on all systems in printfrr, though BSDs support it in printf too.)

%**pSQ** (*char **)
([S]tring [Q]uote.) Like `%s`, but produce a quoted string. Options:

`n` - treat `NULL` as empty string instead.

`q` - include `""` quotation marks. Note: `NULL` is printed as `(null)`, not `"(null)"` unless `n` is used too. This is intentional.

`s` - use escaping suitable for RFC5424 syslog. This means `]` is escaped too.

If a length is specified (`%*pSQ` or `%.*pSQ`), null bytes in the input string do not end the string and are just printed as `\x00`.

%**pSE** (*char* *)
([S]tring [E]scape.) Like `%s`, but escape special characters. Options:

`n` - treat `NULL` as empty string instead.

Unlike *%pSQ*, this escapes many more characters that are fine for a quoted string but not on their own.

If a length is specified (`%*pSE` or `%.*pSE`), null bytes in the input string do not end the string and are just printed as `\x00`.

%**pVA** (*struct va_format* *)
Recursively invoke printfrr, with arguments passed in through:

struct **va_format**

const char ***fmt**
Format string to use for the recursive printfrr call.

va_list ***va**
Formatting arguments. Note this is passed as a pointer, not - as in most other places - a direct struct reference. Internally uses `va_copy()` so repeated calls can be made (e.g. for determining output length.)

%**pFB** (*struct fbuf* *)
Insert text from a `struct fbuf *`, i.e. the output of a call to *bprintfrr()*.

%***pHX** (*void* *, *char* *, *unsigned char* *)
`%pHX`: `12 34 56 78`

`%pHXc`: `12:34:56:78` (separate with [c]olon)

`%pHXn`: `12345678` (separate with [n]othing)

Insert hexdump. This specifier requires a precision or width to be specified. A precision (`%.*pHX`) takes precedence, but generates a compiler warning since precisions are undefined for `%p` in ISO C. If no precision is given, the width is used instead (and normal handling of the width is suppressed).

Note that width and precision are `int` arguments, not `size_t`. Use like:

```
char *buf;
size_t len;

snprintfrr(out, sizeof(out), "... %*pHX ...", (int)len, buf);

/* with padding to width - would generate a warning due to %.*p */
FMT_NSTD(snprintfrr(out, sizeof(out), "... %-47.*pHX ...", (int)len, buf));
```

%***pHS** (*void* *, *char* *, *unsigned char* *)
`%pHS`: `hex.dump`

This is a complementary format for *%*pHX* to print the text representation for a hexdump. Non-printable characters are replaced with a dot.

Integer formats

Note: These formats currently only exist for advanced type checking with the `frr-format` GCC plugin. They should not be used directly since they will cause compiler warnings when used without the plugin. Use with *FMT_NSTD* if necessary.

It is possible ISO C23 may introduce another format for these, possibly `%w64d` discussed in JTC 1/SC 22/WG 14/N2680.

%**Lu** (*uint64_t*)
`12345`

%**Ld** (*int64_t*)
`-12345`

5.4.3 Log levels

Errors and warnings

If it is something that the user will want to look at and maybe do something, it is either an **error** or a **warning**.

We're expecting that warnings and errors are in some way visible to the user (in the worst case by looking at the log after the network broke, but maybe by a syslog collector from all routers.) Therefore, anything that needs to get the user in the loop—and only these things—are warnings or errors.

Note that this doesn't necessarily mean the user needs to fix something in the FRR instance. It also includes when we detect something else needs fixing, for example another router, the system we're running on, or the configuration. The common point is that the user should probably do *something*.

Deciding between a warning and an error is slightly less obvious; the rule of thumb here is that an error will cause considerable fallout beyond its direct effect. Closing a BGP session due to a malformed update is an error since all routes from the peer are dropped; discarding one route because its attributes don't make sense is a warning.

This also loosely corresponds to the kind of reaction we're expecting from the user. An error is likely to need immediate response while a warning might be snoozed for a bit and addressed as part of general maintenance. If a problem will self-repair (e.g. by retransmits), it should be a warning—unless the impact until that self-repair is very harsh.

Examples for warnings:

- a BGP update, LSA or LSP could not be processed, but operation is proceeding and the broken pieces are likely to self-fix later
- some kind of controller cannot be reached, but we can work without it
- another router is using some unknown or unsupported capability

Examples for errors:

- dropping a BGP session due to malformed data
- a socket for routing protocol operation cannot be opened
- desynchronization from network state because something went wrong
- *everything that we as developers would really like to be notified about, i.e. some assumption in the code isn't holding up*

Informational messages

Anything that provides introspection to the user during normal operation is an **info** message.

This includes all kinds of operational state transitions and events, especially if they might be interesting to the user during the course of figuring out a warning or an error.

By itself, these messages should mostly be statements of fact. They might indicate the order and relationship in which things happened. Also covered are conditions that might be "operational issues" like a link failure due to an unplugged cable. If it's pretty much the point of running a routing daemon for, it's not a warning or an error, just business as usual.

The user should be able to see the state of these bits from operational state output, i.e. *show interface* or *show foobar neighbors*. The log message indicating the change may have been printed weeks ago, but the state can always be viewed. (If some state change has an info message but no "show" command, maybe that command needs to be added.)

Examples:

- all kinds of up/down state changes
  - interface coming up or going down
  - addresses being added or deleted
  - peers and neighbors coming up or going down
- rejection of some routes due to user-configured route maps
- backwards compatibility handling because another system on the network has a different or smaller feature set

---

Note: The previously used **notify** priority is replaced with *info* in all cases. We don't currently have a well-defined use case for it.

Debug messages and asserts

Everything that is only interesting on-demand, or only while developing, is a **debug** message. It might be interesting to the user for a particularly evasive issue, but in general these are details that an average user might not even be able to make sense of.

Most (or all?) debug messages should be behind a *debug foobar* category switch that controls which subset of these messages is currently interesting and thus printed. If a debug message doesn't have such a guard, there should be a good explanation as to why.

Conversely, debug messages are the only thing that should be guarded by these switches. Neither info nor warning or error messages should be hidden in this way.

Asserts should only be used as pretty crashes. We are expecting that asserts remain enabled in production builds, but please try to not use asserts in a way that would cause a security problem if the assert wasn't there (i.e. don't use them for length checks.)

The purpose of asserts is mainly to help development and bug hunting. If the daemon crashes, then having some more information is nice, and the assert can provide crucial hints that cut down on the time needed to track an issue. That said, if the issue can be reasonably handled and/or isn't going to crash the daemon, it shouldn't be an assert.

For anything else where internal constraints are violated but we're not breaking due to it, it's an error instead (not a debug.) These require "user action" of notifying the developers.

Examples:

- mismatched `prev`/`next` pointers in lists
- some field that is absolutely needed is `NULL`

- any other kind of data structure corruption that will cause the daemon to crash sooner or later, one way or another

5.4.4 Thread-local buffering

The core logging code in `lib/zlog.c` allows setting up per-thread log message buffers in order to improve logging performance. The following rules apply for this buffering:

- Only messages of priority *DEBUG* or *INFO* are buffered.
- Any higher-priority message causes the thread's entire buffer to be flushed, thus message ordering is preserved on a per-thread level.
- There is no guarantee on ordering between different threads; in most cases this is arbitrary to begin with since the threads essentially race each other in printing log messages. If an order is established with some synchronization primitive, add calls to *zlog_tls_buffer_flush()*.
- The buffers are only ever accessed by the thread they are created by. This means no locking is necessary.

Both the main/default thread and additional threads created by `frr_pthread_new()` with the default `frr_run()` handler will initialize thread-local buffering and call *zlog_tls_buffer_flush()* when idle.

If some piece of code runs for an extended period, it may be useful to insert calls to *zlog_tls_buffer_flush()* in appropriate places:

void **zlog_tls_buffer_flush**(void)
: Write out any pending log messages that the calling thread may have in its buffer. This function is safe to call regardless of the per-thread log buffer being set up / in use or not.

When working with threads that do not use the `thread_master` event loop, per-thread buffers can be managed with:

void **zlog_tls_buffer_init**(void)
: Set up thread-local buffering for log messages. This function may be called repeatedly without adverse effects, but remember to call *zlog_tls_buffer_fini()* at thread exit.

> **Warning:** If this function is called, but *zlog_tls_buffer_flush()* is not used, log message output will lag behind since messages will only be written out when the buffer is full.
>
> Exiting the thread without calling *zlog_tls_buffer_fini()* will cause buffered log messages to be lost.

void **zlog_tls_buffer_fini**(void)
: Flush pending messages and tear down thread-local log message buffering. This function may be called repeatedly regardless of whether *zlog_tls_buffer_init()* was ever called.

5.4.5 Log targets

The actual logging subsystem (in `lib/zlog.c`) is heavily separated from the actual log writers. It uses an atomic linked-list (*zlog_targets*) with RCU to maintain the log targets to be called. This list is intended to function as "backend" only, it **is not used for configuration**.

Logging targets provide their configuration layer on top of this and maintain their own capability to enumerate and store their configuration. Some targets (e.g. syslog) are inherently single instance and just stuff their config in global variables. Others (e.g. file/fd output) are multi-instance capable. There is another layer boundary here between these and the VTY configuration that they use.

Basic internals

struct **zlog_target**
This struct needs to be filled in by any log target and then passed to *zlog_target_replace()*. After it has been registered, **RCU semantics apply**. Most changes to associated data should make a copy, change that, and then replace the entire struct.

Additional per-target data should be "appended" by embedding this struct into a larger one, for use with *containeroff()*, and *zlog_target_clone()* and *zlog_target_free()* should be used to allocate/free the entire container struct.

Do not use this structure to maintain configuration. It should only contain (a copy of) the data needed to perform the actual logging. For example, the syslog target uses this:

```
struct zlt_syslog {
    struct zlog_target zt;
    int syslog_facility;
};

static void zlog_syslog(struct zlog_target *zt, struct zlog_msg *msgs[], size_t␣
↪nmsgs)
{
    struct zlt_syslog *zte = container_of(zt, struct zlt_syslog, zt);
    size_t i;

    for (i = 0; i < nmsgs; i++)
        if (zlog_msg_prio(msgs[i]) <= zt->prio_min)
            syslog(zlog_msg_prio(msgs[i]) | zte->syslog_facility, "%s",
                   zlog_msg_text(msgs[i], NULL));
}
```

struct *zlog_target* ***zlog_target_clone**(struct *memtype* *mt, struct *zlog_target* *oldzt, size_t size)
Allocates a logging target struct. Note that the `oldzt` argument may be `NULL` to allocate a "from scratch". If `oldzt` is not `NULL`, the generic bits in *zlog_target* are copied. **Target specific bits are not copied.**

struct *zlog_target* ***zlog_target_replace**(struct *zlog_target* *oldzt, struct *zlog_target* *newzt)
Adds, replaces or deletes a logging target (either `oldzt` or `newzt` may be `NULL`.)

Returns `oldzt` for freeing. The target remains possibly in use by other threads until the RCU cycle ends. This implies you cannot release resources (e.g. memory, file descriptors) immediately.

The replace operation is not atomic; for a brief period it is possible that messages are delivered on both `oldzt` and `newzt`.

> **Warning:** `oldzt` must remain **functional** until the RCU cycle ends.

void **zlog_target_free**(struct *memtype* *mt, struct *zlog_target* *zt)
Counterpart to *zlog_target_clone()*, frees a target (using RCU.)

void (**zlog_target*.**logfn**)(struct *zlog_target* *zt, struct zlog_msg *msgs[], size_t nmsg)
Called on a target to deliver "normal" logging messages. `msgs` is an array of opaque structs containing the actual message. Use `zlog_msg_*` functions to access message data (this is done to allow some optimizations, e.g. lazy formatting the message text and timestamp as needed.)

Note: `logfn()` must check each individual message's priority value against the configured `prio_min`. While

the `prio_min` field is common to all targets and used by the core logging code to early-drop unneeded log messages, the array is **not** filtered for each `logfn()` call.

void (*_zlog_target_.**logfn_sigsafe**)(struct *zlog_target* *zt, const char *text, size_t len)
Called to deliver "exception" logging messages (i.e. SEGV messages.) Must be Async-Signal-Safe (may not allocate memory or call "complicated" libc functions.) May be `NULL` if the log target cannot handle this.

Standard targets

`lib/zlog_targets.c` provides the standard file / fd / syslog targets. The syslog target is single-instance while file / fd targets can be instantiated as needed. There are 3 built-in targets that are fully autonomous without any config:

- startup logging to *stderr*, until either `zlog_startup_end()` or `zlog_aux_init()` is called.
- stdout logging for non-daemon programs using `zlog_aux_init()`
- crashlogs written to `/var/tmp/frr.daemon.crashlog`

The regular CLI/command-line logging setup is handled by `lib/log_vty.c` which makes the appropriate instantiations of syslog / file / fd targets.

Todo: `zlog_startup_end()` should do an explicit switchover from startup stderr logging to configured logging. Currently, configured logging starts in parallel as soon as the respective setup is executed. This results in some duplicate logging.

5.5 Introspection (xrefs)

The FRR library provides an introspection facility called "xrefs." The intent is to provide structured access to annotated entities in the compiled binary, such as log messages and thread scheduling calls.

5.5.1 Enabling and use

Support for emitting an xref is included in the macros for the specific entities, e.g. `zlog_info()` contains the relevant statements. The only requirement for the system to work is a GNU compatible linker that supports section start/end symbols. (The only known linker on any system FRR supports that does not do this is the Solaris linker.)

To verify xrefs have been included in a binary or dynamic library, run `readelf -n binary`. For individual object files, it's `readelf -S object.o | grep xref_array` instead.

5.5.2 Structure and contents

As a slight improvement to security and fault detection, xrefs are divided into a `const struct xref *` and an optional `struct xrefdata *`. The required const part contains:

enum xref_type `xref.`**`type`**
Identifies what kind of object the xref points to.

int **`line`**

const char *`xref.`**`file`**

const char *`xref.`**`func`**
: Source code location of the xref. `func` will be `<global>` for xrefs outside of a function.

struct *xrefdata* *`xref.`**`xrefdata`**
: The optional writable part of the xref. NULL if no non-const part exists.

The optional non-const part has:

const struct *xref* *`xrefdata.`**`xref`**
: Pointer back to the constant part. Since circular pointers are close to impossible to emit from inside a function body's static variables, this is initialized at startup.

char `xrefdata.`**`uid`**[16]
: Unique identifier, see below.

const char *`xrefdata.`**`hashstr`**

uint32_t `xrefdata.`**`hashu32`**[2]
: Input to unique identifier calculation. These should encompass all details needed to make an xref unique. If more than one string should be considered, use string concatenation for the initializer.

Both structures can be extended by embedding them in a larger type-specific struct, e.g. `struct xref_logmsg *`.

5.5.3 Unique identifiers

All xrefs that have a writable `struct xrefdata *` part are assigned an unique identifier, which is formed as base32 (crockford) SHA256 on:

- the source filename
- the `hashstr` field
- the `hashu32` fields

Note: Function names and line numbers are intentionally not included to allow moving items within a file without affecting the identifier.

For running executables, this hash is calculated once at startup. When directly reading from an ELF file with external tooling, the value must be calculated when necessary.

The identifiers have the form `AXXXX-XXXXX` where `X` is `0-9, A-Z except I,L,O,U` and `A` is `G-Z except I,L,O,U` (i.e. the identifiers always start with a letter.) When reading identifiers from user input, `I` and `L` should be replaced with `1` and `O` should be replaced with `0`. There are 49 bits of entropy in this identifier.

5.5.4 Underlying machinery

Xrefs are nothing other than global variables with some extra glue to make them possible to find from the outside by looking at the binary. The first non-obvious part is that they can occur inside of functions, since they're defined as `static`. They don't have a visible name – they don't need one.

To make finding these variables possible, another global variable, a pointer to the first one, is created in the same way. However, it is put in a special ELF section through `__attribute__((section("xref_array")))`. This is the section you can see with readelf.

Finally, on the level of a whole executable or library, the linker will stuff the individual pointers consecutive to each other since they're in the same section — hence the array. Start and end of this array is given by the linker-autogenerated `__start_xref_array` and `__stop_xref_array` symbols. Using these, both a constructor to run at startup as well as an ELF note are created.

The ELF note is the entrypoint for externally retrieving xrefs from a binary without having to run it. It can be found by walking through the ELF data structures even if the binary has been fully stripped of debug and section information. SystemTap's SDT probes & LTTng's trace points work in the same way (though they emit 1 note for each probe, while xrefs only emit one note in total which refers to the array.) Using xrefs does not impact SystemTap or LTTng, the notes have identifiers they can be distinguished by.

The ELF structure of a linked binary (library or executable) will look like this:

```
$ readelf --wide -l -n lib/.libs/libfrr.so

Elf file type is DYN (Shared object file)
Entry point 0x67d21
There are 12 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg␣
↪Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0002a0 0x0002a0 R   0x8
  INTERP         0x125560 0x0000000000125560 0x0000000000125560 0x00001c 0x00001c R  ␣
↪0x10
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x02aff0 0x02aff0 R  ␣
↪0x1000
  LOAD           0x02b000 0x000000000002b000 0x000000000002b000 0x0b2889 0x0b2889 R E␣
↪0x1000
  LOAD           0x0de000 0x00000000000de000 0x00000000000de000 0x070048 0x070048 R  ␣
↪0x1000
  LOAD           0x14e428 0x000000000014f428 0x000000000014f428 0x00fb70 0x01a2b8 RW ␣
↪0x1000
  DYNAMIC        0x157a40 0x0000000000158a40 0x0000000000158a40 0x000270 0x000270 RW  0x8
  NOTE           0x0002e0 0x00000000000002e0 0x00000000000002e0 0x00004c 0x00004c R   0x4
  TLS            0x14e428 0x000000000014f428 0x000000000014f428 0x000000 0x000008 R   0x8
  GNU_EH_FRAME   0x12557c 0x000000000012557c 0x000000000012557c 0x00819c 0x00819c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW ␣
↪0x10
  GNU_RELRO      0x14e428 0x000000000014f428 0x000000000014f428 0x009bd8 0x009bd8 R   0x1

(...)

Displaying notes found in: .note.gnu.build-id
  Owner                Data size      Description
  GNU                  0x00000014     NT_GNU_BUILD_ID (unique build ID bitstring)       ␣
↪ Build ID: 6a1f66be38b523095ebd6ec13cc15820cede903d

Displaying notes found in: .note.FRR
  Owner                Data size      Description
  FRRouting            0x00000010     Unknown note type: (0x46455258)    description␣
↪data: 6c eb 15 00 00 00 00 00 74 ec 15 00 00 00 00 00
```

Where 0x15eb6c…0x15ec74 are the offsets (relative to the note itself) where the xref array is in the file. Also note the owner is clearly marked as "FRRouting" and the type is "XREF" in hex.

For SystemTap's use of ELF notes, refer to https://libstapsdt.readthedocs.io/en/latest/how-it-works/internals.html as an entry point.

Note: Due to GCC bug 41091, the "xref_array" section is not correctly generated for C++ code when compiled by GCC. A workaround is present for runtime functionality, but to extract the xrefs from a C++ source file, it needs to be built with clang (or a future fixed version of GCC) instead.

5.5.5 Extraction tool

The FRR source contains a matching tool to extract xref data from compiled ELF binaries in `python/xrelfo.py`. This tool uses CPython extensions implemented in `clippy` and must therefore be executed with that.

`xrelfo.py` processes input from one or more ELF file (.o, .so, executable), libtool object (.lo, .la, executable wrapper script) or JSON (output from `xrelfo.py`) and generates an output JSON file. During standard FRR build, it is invoked on all binaries and libraries and the result is combined into `frr.json`.

ELF files from any operating system, CPU architecture and endianness can be processed on any host. Any issues with this are bugs in `xrelfo.py` (or clippy's ELF code.)

`xrelfo.py` also performs some sanity checking, particularly on log messages. The following options are available:

`-o` `OUTPUT`
: Filename to write JSON output to. As a convention, a `.xref` filename extension is used.

`-Wlog-format`
: Performs extra checks on log message format strings, particularly checks for `\t` and `\n` characters (which should not be used in log messages).

`-Wlog-args`
: Generates cleanup hints for format string arguments where `printfrr()` extensions could be used, e.g. replacing `inet_ntoa` with `%pI4`.

`--profile`
: Runs the Python profiler to identify hotspots in the `xrelfo.py` code.

`xrelfo.py` uses information about C structure definitions saved in `python/xrefstructs.json`. This file is included with the FRR sources and only needs to be regenerated when some of the `struct xref_*` definitions are changed (which should be almost never). The file is written by `python/tiabwarfo.py`, which uses `pahole` to extract the necessary data from DWARF information.

5.6 Locking

FRR ships two small wrappers around `pthread_mutex_lock()` / `pthread_mutex_unlock`. Use `#include "frr_pthread.h"` to get these macros.

`frr_with_mutex`(mutex)
: (With `pthread_mutex_t *mutex`.)

  Begin a C statement block that is executed with the mutex locked. Any exit from the block (`break`, `return`, `goto`, end of block) will cause the mutex to be unlocked:

```
int somefunction(int option)
{
    frr_with_mutex (&my_mutex) {
        /* mutex will be locked */

        if (!option)
```

(continues on next page)

(continued from previous page)

```
            /* mutex will be unlocked before return */
            return -1;

        if (something(option))
            /* mutex will be unlocked before goto */
            goto out_err;

        somethingelse();

        /* mutex will be unlocked at end of block */
    }

    return 0;

out_err:
    somecleanup();
    return -1;
}
```

This is a macro that internally uses a `for` loop. It is explicitly acceptable to use `break` to get out of the block. Even though a single statement works correctly, FRR coding style requires that this macro always be used with a `{ ... }` block.

frr_mutex_lock_autounlock(mutex)
(With `pthread_mutex_t *mutex`.)

Lock mutex and unlock at the end of the current C statement block:

```
int somefunction(int option)
{
    frr_mutex_lock_autounlock(&my_mutex);
    /* mutex will be locked */

    ...
    if (error)
      /* mutex will be unlocked before return */
      return -1;
    ...

    /* mutex will be unlocked before return */
    return 0;
}
```

This is a macro that internally creates a variable with a destructor. When the variable goes out of scope (i.e. the block ends), the mutex is released.

> **Warning:** This macro should only used when *frr_with_mutex()* would result in excessively/weirdly nested code. This generally is an indicator that the code might be trying to do too many things with the lock held. Try any possible venues to reduce the amount of code covered by the lock and move to *frr_with_mutex()*.

5.7 Hooks

Libfrr provides type-safe subscribable hook points where other pieces of code can add one or more callback functions. "type-safe" in this case applies to the function pointers used for subscriptions. The implementations checks (at compile-time) whether a callback to be added has the appropriate function signature (parameters) for the hook.

Example:

Listing 3: mydaemon.h

```
#include "hook.h"
DECLARE_HOOK(some_update_event, (struct eventinfo *info), (info));
```

Listing 4: mydaemon.c

```
#include "mydaemon.h"
DEFINE_HOOK(some_update_event, (struct eventinfo *info), (info));
...
hook_call(some_update_event, info);
```

Listing 5: mymodule.c

```
#include "mydaemon.h"
static int event_handler(struct eventinfo *info);
...
hook_register(some_update_event, event_handler);
```

Do not use parameter names starting with "hook", these can collide with names used by the hook code itself.

5.7.1 Return values

Callbacks to be placed on hooks always return "int" for now; hook_call will sum up the return values from each called function. (The default is 0 if no callbacks are registered.)

There are no pre-defined semantics for the value, in most cases it is ignored. For success/failure indication, 0 should be success, and handlers should make sure to only return 0 or 1 (not -1 or other values).

There is no built-in way to abort executing a chain after a failure of one of the callbacks. If this is needed, the hook can use an extra `bool *aborted` argument.

5.7.2 Priorities

Hooks support a "priority" value for ordering registered calls relative to each other. The priority is a signed integer where lower values are called earlier. There are also "Koohs", which is hooks with reverse priority ordering (for cleanup/deinit hooks, so you can use the same priority value).

Recommended priority value ranges are:

Range	Usage
-999 … 0 … 999	main executable / daemon, or library
-1999 … -1000	modules registering calls that should run before the daemon's bits
1000 … 1999	modules' calls that should run after daemon's (includes default value: 1000)

Note: the default value is 1000, based on the following 2 expectations:

- most hook_register() usage will be in loadable modules
- usage of hook_register() in the daemon itself may need relative ordering to itself, making an explicit value the expected case

The priority value is passed as extra argument on hook_register_prio() / hook_register_arg_prio(). Whether a hook runs in reverse is determined solely by the code defining / calling the hook. (DECLARE_KOOH is actually the same thing as DECLARE_HOOK, it's just there to make it obvious.)

5.7.3 Definition

DECLARE_HOOK(name, arglist, passlist)

DECLARE_KOOH(name, arglist, passlist)

Parameters

- **name** – Name of the hook to be defined
- **arglist** – Function definition style parameter list in braces.
- **passlist** – List of the same parameters without their types.

Note: the second and third macro args must be the hook function's parameter list, with the same names for each parameter. The second macro arg is with types (used for defining things), the third arg is just the names (used for passing along parameters).

This macro must be placed in a header file; this header file must be included to register a callback on the hook.

Examples:

```
DECLARE_HOOK(foo, (), ());
DECLARE_HOOK(bar, (int arg), (arg));
DECLARE_HOOK(baz, (const void *x, in_addr_t y), (x, y));
```

DEFINE_HOOK(name, arglist, passlist)

Implements an hook. Each DECLARE_HOOK must have be accompanied by exactly one DEFINE_HOOK, which needs to be placed in a source file. **The hook can only be called from this source file.** This is intentional to avoid overloading and/or misusing hooks for distinct purposes.

The compiled source file will include a global symbol with the name of the hook prefixed by *_hook_*. Trying to register a callback for a hook that doesn't exist will therefore result in a linker error, or a module load-time error for dynamic modules.

DEFINE_KOOH(name, arglist, passlist)

Same as DEFINE_HOOK, but the sense of priorities / order of callbacks is reversed. This should be used for cleanup hooks.

int **hook_call**(name, ...)

Calls the specified named hook. Parameters to the hook are passed right after the hook name, e.g.:

```
hook_call(foo);
hook_call(bar, 0);
hook_call(baz, NULL, INADDR_ANY);
```

Returns the sum of return values from all callbacks. The `DEFINE_HOOK` statement for the hook must be placed in the file before any `hook_call` use of the hook.

5.7.4 Callback registration

void **hook_register**(name, int (*callback)(...))

void **hook_register_prio**(name, int priority, int (*callback)(...))

void **hook_register_arg**(name, int (*callback)(void *arg, ...), void *arg)

void **hook_register_arg_prio**(name, int priority, int (*callback)(void *arg, ...), void *arg)
Register a callback with an hook. If the caller needs to pass an extra argument to the callback, the _arg variant can be used and the extra parameter will be passed as first argument to the callback. There is no typechecking for this argument.

The priority value is used as described above. The variants without a priority parameter use 1000 as priority value.

void **hook_unregister**(name, int (*callback)(...))

void **hook_unregister_arg**(name, int (*callback)(void *arg, ...), void *arg)
Removes a previously registered callback from a hook. Note that there is no _prio variant of these calls. The priority value is only used during registration.

5.8 Command Line Interface

FRR features a flexible modal command line interface. Often when adding new features or modifying existing code it is necessary to create or modify CLI commands. FRR has a powerful internal CLI system that does most of the heavy lifting for you.

5.8.1 Modes

FRR's CLI is organized by modes. Each mode is associated with some set of functionality, e.g. EVPN, or some underlying object such as an interface. Each mode contains a set of commands that control the associated functionality or object. Users move between the modes by entering a command, which is usually different for each source and destination mode.

A summary of the modes is given in the following figure.

See also:

Data Structures

Walkup

FRR exhibits, for historical reasons, a peculiar behavior called 'walkup'. Suppose a user is in `OSPF_NODE`, which contains only OSPF-specific commands, and enters the following command:

```
ip route 192.168.100.0/24 10.0.2.2
```

This command is not defined in `OSPF_NODE`, so the matcher will fail to match the command in that node. The matcher will then check "parent" nodes of `OSPF_NODE`. In this case the direct parent of `OSPF_NODE` is `CONFIG_NODE`, so the current node switches to `CONFIG_NODE` and the command is tried in that node. Since static route commands are defined in `CONFIG_NODE` the command succeeds. The procedure of attempting to execute unmatched commands by sequentially "walking up" to parent nodes only happens in children (direct and indirect) below `CONFIG_NODE` and stops at `CONFIG_NODE`.

Unfortunately, the internal representation of the various modes is not actually a graph. Instead, there is an array. The parent-child relationships are not explicitly defined in any datastructure but instead are hard-coded into the specific commands that switch nodes. For walkup, there is a function that takes a node and returns the parent of the node. This interface causes all manner of insidious problems, even for experienced developers, and needs to be fixed at some point in the future.

5.8.2 Deprecation of old style of commands

There are currently 2 styles of defining commands within a FRR source file. `DEFUN` and `DEFPY`. `DEFPY` should be used for all new commands that a developer is writing. This is because it allows for much better handling of command line arguments as well as ensuring that input is correct. `DEFUN` is listed here for historical reasons as well as for ensuring that existing code can be understood by new developers.

5.8.3 Defining Commands

All definitions for the CLI system are exposed in `lib/command.h`. In this header there are a set of macros used to define commands. These macros are collectively referred to as "DEFUNs", because of their syntax:

```
DEFUN(command_name,
      command_name_cmd,
      "example command FOO...",
      "Examples\n"
      "CLI command\n"
      "Argument\n")
{
    // ...command handler...
}
```

DEFUNs generally take four arguments which are expanded into the appropriate constructs for hooking into the CLI. In order these are:

- **Function name** - the name of the handler function for the command
- **Command name** - the identifier of the `struct cmd_element` for the command. By convention this should be the function name with `_cmd` appended.
- **Command definition** - an expression in FRR's CLI grammar that defines the form of the command and its arguments, if any
- **Doc string** - a newline-delimited string that documents each element in the command definition

In the above example, `command_name` is the function name, `command_name_cmd` is the command name, `"example..."` is the definition and the last argument is the doc string. The block following the macro is the body of the handler function, details on which are presented later in this section.

In order to make the command show up to the user it must be installed into the CLI graph. To do this, call:

```
install_element(NODE, &command_name_cmd);
```

This will install the command into the specified CLI node. Usually these calls are grouped together in a CLI initialization function for a set of commands, and the DEFUNs themselves are grouped into the same source file to avoid cluttering the codebase. The names of these files follow the form `*_vty.[ch]` by convention. Please do not scatter individual CLI commands in the middle of source files; instead expose the necessary functions in a header and place the command definition in a `*_vty.[ch]` file.

Note: Please see *CLI changes* for requirements when creating CLI commands (e.g., JSON structure and formatting).

Definition Grammar

FRR uses its own grammar for defining CLI commands. The grammar draws from syntax commonly seen in *nix manpages and should be fairly intuitive. The parser is implemented in Bison and the lexer in Flex. These may be found in `lib/command_parse.y` and `lib/command_lex.l`, respectively.

> **ProTip**: if you define a new command and find that the parser is throwing syntax or other errors, the parser is the last place you want to look. Bison is very stable and if it detects a syntax error, 99% of the time it will be a syntax error in your definition.

The formal grammar in BNF is given below. This is the grammar implemented in the Bison parser. At runtime, the Bison parser reads all of the CLI strings and builds a combined directed graph that is used to match and interpret user input.

Human-friendly explanations of how to use this grammar are given a bit later in this section alongside information on the *Data Structures* constructed by the parser.

```
command                 ::=   cmd_token_seq
                              cmd_token_seq placeholder_token "..."
cmd_token_seq           ::=   *empty*
                              cmd_token_seq cmd_token
cmd_token               ::=   simple_token
                              selector
simple_token            ::=   literal_token
                              placeholder_token
literal_token           ::=   WORD varname_token
varname_token           ::=   "$" WORD
placeholder_token       ::=   placeholder_token_real varname_token
placeholder_token_real  ::=   IPV4
                              IPV4_PREFIX
                              IPV6
                              IPV6_PREFIX
                              VARIABLE
                              RANGE
                              MAC
                              MAC_PREFIX
selector                ::=   "<" selector_seq_seq ">" varname_token
                              "{" selector_seq_seq "}" varname_token
                              "[" selector_seq_seq "]" varname_token
                              "![" selector_seq_seq "]" varname_token
selector_seq_seq        ::=   selector_seq_seq "|" selector_token_seq
                              selector_token_seq
selector_token_seq      ::=   selector_token_seq selector_token
                              selector_token
selector_token          ::=   selector
                              simple_token
```

Tokens

The various capitalized tokens in the BNF above are in fact themselves placeholders, but not defined as such in the formal grammar; the grammar provides the structure, and the tokens are actually more like a type system for the strings you write in your CLI definitions. A CLI definition string is broken apart and each piece is assigned a type by the lexer based on a set of regular expressions. The parser uses the type information to verify the string and determine the structure of the CLI graph; additional metadata (such as the raw text of each token) is encoded into the graph as it is constructed by the parser, but this is merely a dumb copy job.

Here is a brief summary of the various token types along with examples.

Token type	Syntax	Description
`WORD`	`show ip bgp`	Matches itself. In the given example every token is a WORD.
`IPV4`	`A.B.C.D`	Matches an IPv4 address.
`IPV6`	`X:X::X:X`	Matches an IPv6 address.
`IPV4_PREFIX`	`A.B.C.D/M`	Matches an IPv4 prefix in CIDR notation.
`IPV6_PREFIX`	`X:X::X:X/M`	Matches an IPv6 prefix in CIDR notation.
`MAC`	`X:X:X:X:X:X`	Matches a 48-bit mac address.
`MAC_PREFIX`	`X:X:X:X:X:X/M`	Matches a 48-bit mac address with a mask.
`VARIABLE`	`FOOBAR`	Matches anything.
`RANGE`	`(X-Y)`	Matches numbers in the range X..Y inclusive.

When presented with user input, the parser will search over all defined commands in the current context to find a match. It is aware of the various types of user input and has a ranking system to help disambiguate commands. For instance, suppose the following commands are defined in the user's current context:

```
example command FOO
example command (22-49)
example command A.B.C.D/X
```

The following table demonstrates the matcher's choice for a selection of possible user input.

Input	Matched command	Reason
`example command eLi7eH4xx0r`	example command FOO	`eLi7eH4xx0r` is not an integer or IPv4 prefix, but FOO is a variable and matches all input.
`example command 42`	example command (22-49)	`42` is not an IPv4 prefix. It does match both `(22-49)` and `FOO`, but RANGE tokens are more specific and have a higher priority than VARIABLE tokens.
`example command 10.3.3.0/24`	example command A.B.C.D/X	The user entered an IPv4 prefix, which is best matched by the last command.

Rules

There are also constructs which allow optional tokens, mutual exclusion, one-or-more selection and repetition.

- `<angle|brackets>` – Contain sequences of tokens separated by pipes and provide mutual exclusion. User input matches at most one option.
- `[square brackets]` – Contains sequences of tokens that can be omitted. `[<a|b>]` can be shortened to `[a|b]`.
- `![exclamation square brackets]` – same as `[square brackets]`, but only allow skipping the contents if the command input starts with `no`. (For cases where the positive command needs a parameter, but the parameter is optional for the negative case.)
- `{curly|braces}` – similar to angle brackets, but instead of mutual exclusion, curly braces indicate that one or more of the pipe-separated sequences may be provided in any order.
- `VARIADICS...` – Any token which accepts input (anything except WORD) which occurs as the last token of a line may be followed by an ellipsis, which indicates that input matching the token may be repeated an unlimited number of times.
- `$name` – Specify a variable name for the preceding token. See "Variable Names" below.

Some general notes:

- Options are allowed at the beginning of the command. The developer is entreated to use these extremely sparingly. They are most useful for implementing the 'no' form of configuration commands. Please think carefully before using them for anything else. There is usually a better solution, even if it is just separating out the command definition into separate ones.
- The developer should judiciously apply separation of concerns when defining commands. CLI definitions for two unrelated or vaguely related commands or configuration items should be defined in separate commands. Clarity is preferred over LOC (within reason).
- The maximum number of space-separated tokens that can be entered is presently limited to 256. Please keep this limit in mind when implementing new CLI.

Variable Names

The parser tries to fill the "varname" field on each token. This can happen either manually or automatically. Manual specifications work by appending `$name` after the input specifier:

```
foo bar$cmd WORD$name A.B.C.D$ip
```

Note that you can also assign variable names to fixed input tokens, this can be useful if multiple commands share code. You can also use "$name" after a multiple-choice option:

```
foo bar <A.B.C.D|X:X::X:X>$addr [optionA|optionB]$mode
```

The variable name is in this case assigned to the last token in each of the branches.

Automatic assignment of variable names works by applying the following rules:

- manual names always have priority
- a `[no]` at the beginning receives `no` as varname on the `no` token
- `VARIABLE` tokens whose text is not `WORD` or `NAME` receive a cleaned lowercase version of the token text as varname, e.g. `ROUTE-MAP` becomes `route_map`.
- other variable tokens (i.e. everything except "fixed") receive the text of the preceding fixed token as varname, if one can be found. E.g. `ip route A.B.C.D/M INTERFACE` assigns "route" to the `A.B.C.D/M` token.

These rules should make it possible to avoid manual varname assignment in 90% of the cases.

Doc Strings

Each token in a command definition should be documented with a brief doc string that informs a user of the meaning and/or purpose of the subsequent command tree. These strings are provided as the last parameter to DEFUN macros, concatenated together and separated by an escaped newline (\n). These are best explained by example.

```
DEFUN (config_terminal,
       config_terminal_cmd,
       "configure terminal",
       "Configuration from vty interface\n"
       "Configuration terminal\n")
```

The last parameter is split into two lines for readability. Two newline delimited doc strings are present, one for each token in the command. The second string documents the functionality of the `terminal` command in the `configure` subtree.

Note that the first string, for `configure` does not contain documentation for 'terminal'. This is because the CLI is best envisioned as a tree, with tokens defining branches. An imaginary `start` token is the root of every command in a CLI node. Each subsequent written token descends into a subtree, so the documentation for that token ideally summarizes all the functionality contained in the subtree.

A consequence of this structure is that the developer must be careful to use the same doc strings when defining multiple commands that are part of the same tree. Commands which share prefixes must share the same doc strings for those prefixes. On startup the parser will generate warnings if it notices inconsistent doc strings. Behavior is undefined; the same token may show up twice in completions, with different doc strings, or it may show up once with a random doc string. Parser warnings should be heeded and fixed to avoid confusing users.

The number of doc strings provided must be equal to the amount of tokens present in the command definition, read left to right, ignoring any special constructs.

In the examples below, each arrowed token needs a doc string.

```
"show ip bgp"
 ^    ^  ^

"command <foo|bar> [example]"
 ^        ^   ^     ^
```

DEFPY

`DEFPY(...)` is an enhanced version of `DEFUN()` which is preprocessed by `python/clidef.py`. The python script parses the command definition string, extracts variable names and types, and generates a C wrapper function that parses the variables and passes them on. This means that in the CLI function body, you will receive additional parameters with appropriate types.

This is best explained by an example. Invoking `DEFPY` like this:

```
DEFPY(func, func_cmd, "[no] foo bar A.B.C.D (0-99)$num", "...help...")
```

defines the handler function like this:

```
func(self, vty, argc, argv,  /* standard CLI arguments */
     const char *no,         /* unparsed "no" */
     struct in_addr bar,     /* parsed IP address */
     const char *bar_str,    /* unparsed IP address */
     long num,               /* parsed num */
     const char *num_str)    /* unparsed num */
```

Note that as documented in the previous section, `bar` is automatically applied as variable name for `A.B.C.D`. The Python script then detects this as an IP address argument and generates code to parse it into a `struct in_addr`, passing it in `bar`. The raw value is passed in `bar_str`. The range/number argument works in the same way with the explicitly given variable name.

Type rules

Token(s)	Type	Value if omitted by user
`A.B.C.D`	`struct in_addr`	`0.0.0.0`
`X:X::X:X`	`struct in6_addr`	`::`
`A.B.C.D + X:X::X:X`	`const union sockunion *`	`NULL`
`A.B.C.D/M`	`const struct prefix_ipv4 *`	`all-zeroes struct`
`X:X::X:X/M`	`const struct prefix_ipv6 *`	`all-zeroes struct`
`A.B.C.D/M + X:X::X:X/M`	`const struct prefix *`	`all-zeroes struct`
`(0-9)`	`long`	`0`
`VARIABLE`	`const char *`	`NULL`
`word`	`const char *`	`NULL`
all other	`const char *`	`NULL`

Note the following details:

- Not all parameters are pointers, some are passed as values.
- When the type is not `const char *`, there will be an extra `_str` argument with type `const char *`.
- You can give a variable name not only to `VARIABLE` tokens but also to `word` tokens (e.g. constant words). This is useful if some parts of a command are optional. The type will be `const char *`.
- `[no]` will be passed as `const char *no`.
- Most pointers will be `NULL` when the argument is optional and the user did not supply it. As noted in the table above, some prefix struct type arguments are passed as pointers to all-zeroes structs, not as `NULL` pointers.
- If a parameter is not a pointer, but is optional and the user didn't use it, the default value will be passed. Check the `_str` argument if you need to determine whether the parameter was omitted.
- If the definition contains multiple parameters with the same variable name, they will be collapsed into a single function parameter. The python code will detect if the types are compatible (i.e. IPv4 + IPv6 variants) and choose a corresponding C type.
- The standard DEFUN parameters (`self, vty, argc, argv`) are still present and can be used. A DEFUN can simply be **edited into a DEFPY without further changes and it will still work**; this allows easy forward migration.
- A file may contain both `DEFUN` and `DEFPY` statements.

Getting a parameter dump

The clidef.py script can be called to get a list of DEFUNs/DEFPYs with the parameter name/type list:

```
lib/clippy python/clidef.py --all-defun --show lib/plist.c > /dev/null
```

The generated code is printed to stdout, the info dump to stderr. The `--all-defun` argument will make it process DEFUN blocks as well as DEFPYs, which is useful prior to converting some DEFUNs. **The dump does not list the ``_str`` arguments** to keep the output shorter.

Note that the `clidef.py` script cannot be run with python directly, it needs to be run with *clippy* since the latter makes the CLI parser available.

Include & Makefile requirements

A source file that uses DEFPY needs to include the `*_clippy.c` file **before all DEFPY statements**:

```
/* GPL header */
#include ...
...
#include "daemon/filename_clippy.c"

DEFPY(...)
DEFPY(...)

install_element(...)
```

This dependency needs to be marked in `Makefile.am` or `subdir.am`: (there is no ordering requirement)

```
# ...

# if linked into a LTLIBRARY (.la/.so):
filename.lo: filename_clippy.c

# if linked into an executable or static library (.a):
filename.o: filename_clippy.c
```

Handlers

The block that follows a CLI definition is executed when a user enters input that matches the definition. Its function signature looks like this:

```
int (*func) (const struct cmd_element *, struct vty *, int, struct cmd_token *[]);
```

The first argument is the command definition struct. The last argument is an ordered array of tokens that correspond to the path taken through the graph, and the argument just prior to that is the length of the array.

The arrangement of the token array has changed from Quagga's CLI implementation. In the old system, missing arguments were padded with `NULL` so that the same parts of a command would show up at the same indices regardless of what was entered. The new system does not perform such padding and therefore it is generally *incorrect* to assume consistent indices in this array. As a simple example:

Command definition:

```
command [foo] <bar|baz>
```

User enters:

```
command foo bar
```

Array:

```
[0] -> command
[1] -> foo
[2] -> bar
```

User enters:

```
command baz
```

Array:

```
[0] -> command
[1] -> baz
```

5.8.4 Data Structures

On startup, the CLI parser sequentially parses each command string definition and constructs a directed graph with each token forming a node. This graph is the basis of the entire CLI system. It is used to match user input in order to generate command completions and match commands to functions.

There is one graph per CLI node (not the same as a graph node in the CLI graph). The CLI node struct keeps a reference to its graph (see `lib/command.h`).

While most of the graph maintains the form of a tree, special constructs outlined in the Rules section introduce some quirks. <>, [] and {} form self-contained 'subgraphs'. Each subgraph is a tree except that all of the 'leaves' actually share a child node. This helps with minimizing graph size and debugging.

As a working example, here is the graph of the following command:

```
show [ip] bgp neighbors [<A.B.C.D|X:X::X:X|WORD>] [json]
```

`FORK` and `JOIN` nodes are plumbing nodes that don't correspond to user input. They're necessary in order to deduplicate these constructs where applicable.

Options follow the same form, except that there is an edge from the `FORK` node to the `JOIN` node. Since all of the subgraphs in the example command are optional, all of them have this edge.

Keywords follow the same form, except that there is an edge from `JOIN` to `FORK`. Because of this the CLI graph cannot be called acyclic. There is special logic in the input matching code that keeps a stack of paths already taken through the node in order to disallow following the same path more than once.

Variadics are a bit special; they have an edge back to themselves, which allows repeating the same input indefinitely.

The leaves of the graph are nodes that have no out edges. These nodes are special; their data section does not contain a token, as most nodes do, or `NULL`, as in `FORK`/`JOIN` nodes, but instead has a pointer to a `cmd_element`. All paths through the graph that terminate on a leaf are guaranteed to be defined by that command. When a user enters a complete command, the command matcher tokenizes the input and executes a DFS on the CLI graph. If it is simultaneously able to exhaust all input (one input token per graph node), and then find exactly one leaf connected to the last node it reaches, then the input has matched the corresponding command and the command is executed. If it finds more than

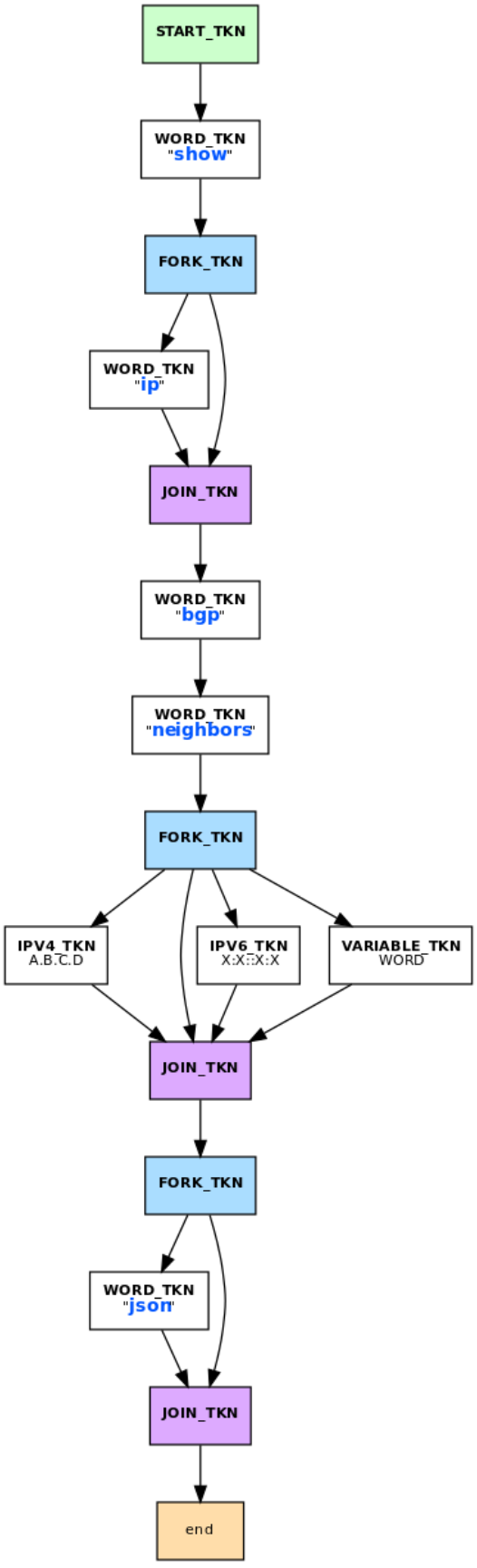


Fig. 1: Graph of example CLI command

one node, then the command is ambiguous (more on this in deduplication). If it cannot exhaust all input, the command is unknown. If it exhausts all input but does not find an edge node, the command is incomplete.

The parser uses an incremental strategy to build the CLI graph for a node. Each command is parsed into its own graph, and then this graph is merged into the overall graph. During this merge step, the parser makes a best-effort attempt to remove duplicate nodes. If it finds a node in the overall graph that is equal to a node in the corresponding position in the command graph, it will intelligently merge the properties from the node in the command graph into the already-existing node. Subgraphs are also checked for isomorphism and merged where possible. The definition of whether two nodes are 'equal' is based on the equality of some set of token properties; read the parser source for the most up-to-date definition of equality.

When the parser is unable to deduplicate some complicated constructs, this can result in two identical paths through separate parts of the graph. If this occurs and the user enters input that matches these paths, they will receive an 'ambiguous command' error and will be unable to execute the command. Most of the time the parser can detect and warn about duplicate commands, but it will not always be able to do this. Hence care should be taken before defining a new command to ensure it is not defined elsewhere.

struct cmd_token

```
/* Command token struct. */
struct cmd_token
{
        enum cmd_token_type type; // token type
        uint8_t attr;             // token attributes
        bool allowrepeat;         // matcher can match token repetitively?

        char *text;               // token text
        char *desc;               // token description
        long long min, max;       // for ranges
        char *arg;                // user input that matches this token
        char *varname;            // variable name
};
```

This struct is used in the CLI graph to match input against. It is also used to pass user input to command handler functions, as it is frequently useful for handlers to have access to that information. When a command is matched, the sequence of `cmd_tokens` that form the matching path are duplicated and placed in order into `*argv[]`. Before this happens the `->arg` field is set to point at the snippet of user input that matched it.

For most nontrivial commands the handler function will need to determine which of the possible matching inputs was entered. Previously this was done by looking at the first few characters of input. This is now considered an anti-pattern and should be avoided. Instead, the `->type` or `->text` fields for this logic. The `->type` field can be used when the possible inputs differ in type. When the possible types are the same, use the `->text` field. This field has the full text of the corresponding token in the definition string and using it makes for much more readable code. An example is helpful.

Command definition:

```
command <(1-10)|foo|BAR>
```

In this example, the user may enter any one of: - an integer between 1 and 10 - "foo" - anything at all

If the user enters "command f", then:

```
argv[1]->type == WORD_TKN
argv[1]->arg  == "f"
argv[1]->text == "foo"
```

Range tokens have some special treatment; a token with `->type == RANGE_TKN` will have the `->min` and `->max` fields set to the bounding values of the range.

struct cmd_element

```
struct cmd_node {
        /* Node index. */
        enum node_type node;

        /* Prompt character at vty interface. */
        const char *prompt;

        /* Is this node's configuration goes to vtysh ? */
        int vtysh;

        /* Node's configuration write function */
        int (*func)(struct vty *);

        /* Node's command graph */
        struct graph *cmdgraph;

        /* Vector of this node's command list. */
        vector cmd_vector;

        /* Hashed index of command node list, for de-dupping primarily */
        struct hash *cmd_hash;
};
```

This struct corresponds to a CLI mode. The last three fields are most relevant here.

cmdgraph This is a pointer to the command graph that was described in the first part of this section. It is the datastructure used for matching user input to commands.

cmd_vector This is a list of all the `struct cmd_element` defined in the mode.

cmd_hash This is a hash table of all the `struct cmd_element` defined in the mode. When `install_element` is called, it checks that the element it is given is not already present in the hash table as a safeguard against duplicate calls resulting in a command being defined twice, which renders the command ambiguous.

All `struct cmd_node` are themselves held in a static vector defined in `lib/command.c` that defines the global CLI space.

5.8.5 Command Abbreviation & Matching Priority

It is possible for users to elide parts of tokens when the CLI matcher does not need them to make an unambiguous match. This is best explained by example.

Command definitions:

```
command dog cow
command dog crow
```

User input:

```
c d c          -> ambiguous command
c d co         -> match "command dog cow"
```

The parser will look ahead and attempt to disambiguate the input based on tokens later on in the input string.

Command definitions:

```
show ip bgp A.B.C.D
show ipv6 bgp X:X::X:X
```

User enters:

```
s i b 4.3.2.1          -> match "show ip bgp A.B.C.D"
s i b ::e0             -> match "show ipv6 bgp X:X::X:X"
```

Reading left to right, both of these commands would be ambiguous since 'i' does not explicitly select either 'ip' or 'ipv6'. However, since the user later provides a token that matches only one of the commands (an IPv4 or IPv6 address) the parser is able to look ahead and select the appropriate command. This has some implications for parsing the `*argv[]` that is passed to the command handler.

Now consider a command definition such as:

```
command <foo|VAR>
```

'foo' only matches the string 'foo', but 'VAR' matches any input, including 'foo'. Who wins? In situations like this the matcher will always choose the 'better' match, so 'foo' will win.

Consider also:

```
show <ip|ipv6> foo
```

User input:

```
show ip foo
```

`ip` partially matches `ipv6` but exactly matches `ip`, so `ip` will win.

5.8.6 Adding a CLI Node

To add a new CLI node, you should:

- define a new numerical node constant
- define a node structure in the relevant daemon
- call `install_node()` in the relevant daemon
- define and install the new node in vtysh
- define corresponding node entry commands in daemon and vtysh
- add a new entry to the `ctx_keywords` dictionary in `tools/frr-reload.py`

Defining the numerical node constant

Add your new node value to the enum before `NODE_TYPE_MAX` in `lib/command.h`:

```
enum node_type {
     AUTH_NODE,                // Authentication mode of vty interface.
     VIEW_NODE,                // View node. Default mode of vty interface.
     [...]
     MY_NEW_NODE,
     NODE_TYPE_MAX, // maximum
};
```

Defining a node structure

In your daemon-specific code where you define your new commands that attach to the new node, add a node definition:

```
static struct cmd_node my_new_node = {
     .name = "my new node name",
     .node = MY_NEW_NODE, // enum node_type lib/command.h
     .parent_node = CONFIG_NODE,
     .prompt = "%s(my-new-node-prompt)# ",
     .config_write = my_new_node_config_write,
};
```

You will need to define `my_new_node_config_write(struct vty \*vty)` (or omit this field if you have no relevant configuration to save).

Calling `install_node()`

In the daemon's initialization function, before installing your new commands with `install_element()`, add a call `install_node(&my_new_node)`.

Defining and installing the new node in vtysh

The build tools automatically collect command definitions for vtysh. However, new nodes must be coded in vtysh specifically.

In `vtysh/vtysh.c`, define a stripped-down node structure and call `install_node()`:

```
static struct cmd_node my_new_node = {
     .name = "my new node name",
     .node = MY_NEW_NODE, /* enum node_type lib/command.h */
     .parent_node = CONFIG_NODE,
     .prompt = "%s(my-new-node-prompt)# ",
};
[...]
void vtysh_init_vty(void)
{
   [...]
   install_node(&my_new_node)
   [...]
}
```

Defining corresponding node entry commands in daemon and vtysh

The command that descends into the new node is typically programmed with `VTY_PUSH_CONTEXT` or equivalent in the daemon's CLI handler function. (If the CLI has been updated to use the new northbound architecture, `VTY_PUSH_XPATH` is used instead.)

In vtysh, you must implement a corresponding node change so that vtysh tracks the daemon's movement through the node tree.

Although the build tools typically scan daemon code for CLI definitions to replicate their parsing in vtysh, the node-descent function in the daemon must be blocked from this replication so that a hand-coded skeleton can be written in `vtysh.c`.

Accordingly, use one of the `*_NOSH` macros such as `DEFUN_NOSH`, `DEFPY_NOSH`, or `DEFUN_YANG_NOSH` for the daemon's node-descent CLI definition, and use `DEFUNSH` in `vtysh.c` for the vtysh equivalent.

See also:

Special DEFUNs

Examples:

`zebra_whatever.c`

```
DEFPY_NOSH(my_new_node,
     my_new_node_cmd,
     "my-new-node foo",
     "New Thing\n"
     "A foo\n")
{
   [...]
   VTY_PUSH_CONTEXT(MY_NEW_NODE, bar);
   [...]
}
```

`ripd_whatever.c`

```
DEFPY_YANG_NOSH(my_new_node,
     my_new_node_cmd,
     "my-new-node foo",
     "New Thing\n"
     "A foo\n")
{
   [...]
   VTY_PUSH_XPATH(MY_NEW_NODE, xbar);
   [...]
}
```

vtysh.c

```
DEFUNSH(VTYSH_ZEBRA, my_new_node,
     my_new_node_cmd,
     "my-new-node foo",
     "New Thing\n"
     "A foo\n")
{
     vty->node = MY_NEW_NODE;
     return CMD_SUCCESS;
}
[...]
install_element(CONFIG_NODE, &my_new_node_cmd);
```

Adding a new entry to the `ctx_keywords` dictionary

In file `tools/frr-reload.py`, the `ctx_keywords` dictionary describes the various node relationships. Add a new node entry at the appropriate level in this dictionary.

```
ctx_keywords = {
    [...]
    "key chain ": {
        "key ": {}
    },
    [...]
    "my-new-node": {},
    [...]
}
```

5.8.7 Inspection & Debugging

Permutations

It is sometimes useful to check all the possible combinations of input that would match an arbitrary definition string. There is a tool in `tools/permutations` that reads CLI definition strings on `stdin` and prints out all matching input permutations. It also dumps a text representation of the graph, which is more useful for debugging than anything else. It looks like this:

```
$ ./permutations "show [ip] bgp [<view|vrf> WORD]"

show ip bgp view WORD
show ip bgp vrf WORD
show ip bgp
show bgp view WORD
show bgp vrf WORD
show bgp
```

This functionality is also built into VTY/VTYSH; `list permutations` will list all possible matching input permutations in the current CLI node.

Graph Inspection

When in the Telnet or VTYSH console, `show cli graph` will dump the entire command space of the current mode in the DOT graph language. This can be fed into one of the various GraphViz layout engines, such as `dot`, `neato`, etc.

For example, to generate an image of the entire command space for the top-level mode (`ENABLE_NODE`):

```
sudo vtysh -c 'show cli graph' | dot -Tjpg -Grankdir=LR > graph.jpg
```

To do the same for the BGP mode:

```
sudo vtysh -c 'conf t' -c 'router bgp' -c 'show cli graph' | dot -Tjpg -Grankdir=LR >␣
↪bgpgraph.jpg
```

This information is very helpful when debugging command resolution, tracking down duplicate / ambiguous commands, and debugging patches to the CLI graph builder.

5.9 Modules

FRR has facilities to load DSOs at startup via `dlopen()`. These are used to implement modules, such as SNMP and FPM.

5.9.1 Limitations

- can't load, unload, or reload during runtime. This just needs some work and can probably be done in the future.
- doesn't fix any of the "things need to be changed in the code in the library" issues. Most prominently, you can't add a CLI node because CLI nodes are listed in the library…
- if your module crashes, the daemon crashes. Should be obvious.
- **does not provide a stable API or ABI**. Your module must match a version of FRR and you may have to update it frequently to match changes.
- **does not create a license boundary**. Your module will need to link libzebra and include header files from the daemons, meaning it will be GPL-encumbered.

5.9.2 Installation

Look for `moduledir` in `configure.ac`, default is normally `/usr/lib64/frr/modules` but depends on `--libdir` / `--prefix`.

The daemon's name is prepended when looking for a module, e.g. "snmp" tries to find "zebra_snmp" first when used in zebra. This is just to make it nicer for the user, with the snmp module having the same name everywhere.

Modules can be packaged separately from FRR. The SNMP and FPM modules are good candidates for this because they have dependencies (net-snmp / protobuf) that are not FRR dependencies. However, any distro packages should have an "exact-match" dependency onto the FRR package. Using a module from a different FRR version will probably blow up nicely.

For snapcraft (and during development), modules can be loaded with full path (e.g. -M `$SNAP/lib/frr/modules/zebra_snmp.so`). Note that libtool puts output files in the .libs directory, so during development you have to use `./zebra -M .libs/zebra_snmp.so`.

5.9.3 Creating a module

... best to look at the existing SNMP or FPM modules.

Basic boilerplate:

```
#include "hook.h"
#include "module.h"
#include "libfrr.h"
#include "thread.h"

static int module_late_init(struct thread_master *master)
{
    /* Do initialization stuff here */
    return 0;
}

static int
module_init (void)
{
  hook_register(frr_late_init, module_late_init);
  return 0;
}

FRR_MODULE_SETUP(
    .name = "my module",
    .version = "0.0",
    .description = "my module",
    .init = module_init,
);
```

The `frr_late_init` hook will be called after the daemon has finished its other startup and is about to enter the main event loop; this is the best place for most initialisation.

5.9.4 Compiler & Linker magic

There's a `THIS_MODULE` (like in the Linux kernel), which uses `visibility` attributes to restrict it to the current module. If you get a linker error with `_frrmod_this_module`, there is some linker SNAFU. This shouldn't be possible, though one way to get it would be to not include libzebra (which provides a fallback definition for the symbol).

libzebra and the daemons each have their own `THIS_MODULE`, as do all loadable modules. In any other libraries (e.g. `libfrrsnmp`), `THIS_MODULE` will use the definition in libzebra; same applies if the main executable doesn't use `FRR_DAEMON_INFO` (e.g. all testcases).

The deciding factor here is "what dynamic linker unit are you using the symbol from." If you're in a library function and want to know who called you, you can't use `THIS_MODULE` (because that'll just tell you you're in the library). Put a macro around your function that adds `THIS_MODULE` in the *caller's code calling your function*.

The idea is to use this in the future for module unloading. Hooks already remember which module they were installed by, as groundwork for a function that removes all of a module's installed hooks.

There's also the `frr_module` symbol in modules, pretty much a standard entry point for loadable modules.

5.9.5 Command line parameters

Command line parameters can be passed directly to a module by appending a colon to the module name when loading it, e.g. `-M mymodule:myparameter`. The text after the colon will be accessible in the module's code through `THIS_MODULE->load_args`. For example, see how the format parameter is configured in the `zfpm_init()` function inside `zebra_fpm.c`.

5.9.6 Hooks

Hooks are just points in the code where you can register your callback to be called. The parameter list is specific to the hook point. Since there is no stable API, the hook code has some extra type safety checks making sure you get a compiler warning when the hook parameter list doesn't match your callback. Don't ignore these warnings.

5.9.7 Relation to MTYPE macros

The MTYPE macros, while primarily designed to decouple MTYPEs from the library and beautify the code, also work very nicely with loadable modules – both constructors and destructors are executed when loading/unloading modules.

This means there is absolutely no change required to MTYPEs, you can just use them in a module and they will even clean up themselves when we implement module unloading and an unload happens. In fact, it's impossible to create a bug where unloading fails to de-register a MTYPE.

5.10 Scripting

See also:

User docs for scripting

5.10.1 Overview

FRR has the ability to call Lua scripts to perform calculations, make decisions, or otherwise extend builtin behavior with arbitrary user code. This is implemented using the standard Lua C bindings. The supported version of Lua is 5.3.

C objects may be passed into Lua and Lua objects may be retrieved by C code via a encoding/decoding system. In this way, arbitrary data from FRR may be passed to scripts.

The Lua environment is isolated from the C environment; user scripts cannot access FRR's address space unless explicitly allowed by FRR.

For general information on how Lua is used to extend C, refer to Part IV of "Programming in Lua".

https://www.lua.org/pil/contents.html#24

5.10.2 Design

Why Lua

Lua is designed to be embedded in C applications. It is very small; the standard library is 220K. It is relatively fast. It has a simple, minimal syntax that is relatively easy to learn and can be understood by someone with little to no programming experience. Moreover it is widely used to add scripting capabilities to applications. In short it is designed for this task.

Reasons against supporting multiple scripting languages:

- Each language would require different FFI methods, and specifically different object encoders; a lot of code
- Languages have different capabilities that would have to be brought to parity with each other; a lot of work
- Languages have vastly different performance characteristics; this would create alot of basically unfixable issues, and result in a single de facto standard scripting language (the fastest)
- Each language would need a dedicated maintainer for the above reasons; this is pragmatically difficult
- Supporting multiple languages fractures the community and limits the audience with which a given script can be shared

5.10.3 General

FRR's scripting functionality is provided in the form of Lua functions in Lua scripts (`.lua` files). One Lua script may contain many Lua functions. These are respectively encapsulated in the following structures:

```
struct frrscript {
    /* Lua file name */
    char *name;

    /* hash of lua_function_states */
    struct hash *lua_function_hash;
};

struct lua_function_state {
    /* Lua function name */
    char *name;

    lua_State *L;
};
```

struct frrscript: Since all Lua functions are contained within scripts, the following APIs manipulates this structure. `name` contains the Lua script name and a hash of Lua functions to their function names.

struct lua_function_state is an internal structure, but it essentially contains the name of the Lua function and its state (a stack), which is run using Lua library functions.

In general, to run a Lua function, these steps must take place:

- Initialization
- Load
- Call
- Delete

Initialization

The `frrscript` object encapsulates the Lua function state(s) from one Lua script file. To create, use `frrscript_new()` which takes the name of the Lua script. The string ".lua" is appended to the script name, and the resultant filename will be used to look for the script when we want to load a Lua function from it.

For example, to create `frrscript` for `/etc/frr/scripts/bingus.lua`:

```
struct frrscript *fs = frrscript_new("bingus");
```

The script is *not* read at this stage. This function cannot be used to test for a script's presence.

Load

The function to be called must first be loaded. Use `frrscript_load()` which takes a `frrscript` object, the name of the Lua function and a callback function. The script file will be read to load and compile the function.

For example, to load the Lua function `on_foo` in `/etc/frr/scripts/bingus.lua`:

```
int ret = frrscript_load(fs, "on_foo", NULL);
```

This function returns 0 if and only if the Lua function was successfully loaded. A non-zero return could indicate either a missing Lua script, a missing Lua function, or an error when loading the function.

During loading the script is validated for syntax and its environment is set up. By default this does not include the Lua standard library; there are security issues to consider, though for practical purposes untrusted users should not be able to write the scripts directory anyway.

Call

After loading, a Lua function can be called any number of times.

Input

Inputs to the Lua script should be given by providing a list of parenthesized pairs, where the first and second field identify the name of the variable and the value it is bound to, respectively. The types of the values must have registered encoders (more below); the compiler will warn you otherwise.

These variables are first encoded in-order, then provided as arguments to the Lua function. In the example, note that `c` is passed in as a value while `a` and `b` are passed in as pointers.

```
int a = 100, b = 200, c = 300;
frrscript_call(fs, "on_foo", ("a", &a), ("b", &b), ("c", c));
```

```
function on_foo(a, b, c)
  -- a is 100, b is 200, c is 300
  ...
```

Output

```
int a = 100, b = 200, c = 300;
frrscript_call(fs, "on_foo", ("a", &a), ("b", &b), ("c", c));
// a is 500, b is 200, c is 300

int* d = frrscript_get_result(fs, "on_foo", "d", lua_tointegerp);
// d is 800
```

```
function on_foo(a, b, c)
  b = 600
  return { ["a"] = 500, ["c"] = 700, ["d"] = 800 }
end
```

Lua functions being called must return a single table of string names to values. (Lua functions should return an empty table if there is no output.) The keys of the table are mapped back to names of variables in C. Note that the values in the table can also be tables. Since tables are Lua's primary data structure, this design lets us return any Lua value.

After the Lua function returns, the names of variables to `frrscript_call()` are matched against keys of the returned table, and then decoded. The types being decoded must have registered decoders (more below); the compiler will warn you otherwise.

In the example, since `a` was in the returned table and `b` was not, `a` was decoded and its value modified, while `b` was not decoded. `c` was decoded as well, but its decoder is a noop. What modifications happen given a variable depends whether its name was in the returned table and the decoder's implementation.

Warning: Always keep in mind that non const-qualified pointers in `frrscript_call()` may be modified - this may be a source of bugs. On the other hand, const-qualified pointers and other values cannot be modified.

Tip: You can make a copy of a data structure and pass that in instead, so that modifications only happen to that copy.

`frrscript_call()` returns 0 if and only if the Lua function was successfully called. A non-zero return could indicate either a missing Lua script, a missing Lua function, or an error from the Lua interpreter.

In the above example, `d` was not an input to `frrscript_call()`, so its value must be explicitly retrieved with `frrscript_get_result`.

`frrscript_get_result()` takes a decoder and string name which is used as a key to search the returned table. Returns the pointer to the decoded value, or NULL if it was not found. In the example, `d` is a "new" value in C space, so memory allocation might take place. Hence the caller is responsible for memory deallocation.

`frrscript_call()` may be called multiple times without re-loading with `frrscript_load()`. Results are not preserved between consecutive calls.

```
frrscript_load(fs, "on_foo");

frrscript_call(fs, "on_foo");
frrscript_get_result(fs, "on_foo", ...);
frrscript_call(fs, "on_foo");
frrscript_get_result(fs, "on_foo", ...);
```

Delete

To delete a script and the all Lua states associated with it:

```
frrscript_delete(fs);
```

A complete example

So, a typical execution call, with error checking, looks something like this:

```
struct frrscript *fs = frrscript_new("my_script"); // name *without* .lua

int ret = frrscript_load(fs, "on_foo", NULL);
if (ret != 0)
    goto DONE; // Lua script or function might have not been found

int a = 100, b = 200, c = 300;
ret = frrscript_call(fs, "on_foo", ("a", &a), ("b", &b), ("c", c));
if (ret != 0)
    goto DONE; // Lua function might have not successfully run

// a and b might be modified
assert(a == 500);
assert(b == 200);

// c could not have been modified
assert(c == 300);

// d is new
int* d = frrscript_get_result(fs, "on_foo", "d", lua_tointegerp);

if (!d)
    goto DONE; // "d" might not have been in returned table

assert(*d == 800);
```

(continues on next page)

(continued from previous page)

```
XFREE(MTYPE_SCRIPT_RES, d); // caller responsible for free

DONE:
frrscript_delete(fs);
```

```
function on_foo(a, b, c)
  b = 600
  return { a = 500, c = 700, d = 800 }
end
```

Note that `{ a = ...` is same as `{ ["a"] = ...`; it is Lua shorthand to use the variable name as the key in a table.

Encoding and Decoding

Earlier sections glossed over the types of values that can be passed into `frrscript_call()` and how data is passed between C and Lua. Lua, as a dynamically typed, garbage collected language, cannot directly use C values without some kind of encoding / decoding system to translate types between the two runtimes.

Lua communicates with C code using a stack. C code wishing to provide data to Lua scripts must provide a function that encodes the C data into a Lua representation and pushes it on the stack. C code wishing to retrieve data from Lua must provide a corresponding decoder function that retrieves a Lua value from the stack and converts it to the corresponding C type.

Encoders and decoders are provided for common data types. Developers wishing to pass their own data structures between C and Lua need to create encoders and decoders for that data type.

We try to keep them named consistently. There are three kinds of encoders and decoders:

1. lua_push*: encodes a value onto the Lua stack. Required for `frrscript_call`.
2. lua_decode*: decodes a value from the Lua stack. Required for `frrscript_call`. Only non const-qualified pointers may be actually decoded (more below).
3. lua_to*: allocates memory and decodes a value from the Lua stack. Required for `frrscript_get_result`.

This design allows us to combine typesafe *modification* of C values as well as *allocation* of new C values.

In the following sections, we will use the encoders/decoders for `struct prefix` as an example.

Encoding

An encoder function takes a `lua_State *`, a C type and pushes that value onto the Lua state (a stack). For C structs, the usual case, this will typically be encoded to a Lua table, then pushed onto the Lua stack.

Here is the encoder function for `struct prefix`:

```
void lua_pushprefix(lua_State *L, struct prefix *prefix)
{
        char buffer[PREFIX_STRLEN];

        lua_newtable(L);
        lua_pushstring(L, prefix2str(prefix, buffer, PREFIX_STRLEN));
        lua_setfield(L, -2, "network");
        lua_pushinteger(L, prefix->prefixlen);
        lua_setfield(L, -2, "length");
```

(continues on next page)

(continued from previous page)

```
        lua_pushinteger(L, prefix->family);
        lua_setfield(L, -2, "family");
}
```

This function pushes a single value, a table, onto the Lua stack, whose equivalent in Lua is:

```
{ ["network"] = "1.2.3.4/24", ["prefixlen"] = 24, ["family"] = 2 }
```

Decoding

Decoders are a bit more involved. They do the reverse; a decoder function takes a `lua_State *`, pops a value off the Lua stack and converts it back into its C type.

There are two: `lua_decode*` and `lua_to*`. The former does no mememory allocation and is needed for `frrscript_call`. The latter performs allocation and is optional.

A `lua_decode_*` function takes a `lua_State*`, an index, and a pointer to a C data structure, and directly modifies the structure with values from the Lua stack. Note that only non const-qualified pointers may be modified; `lua_decode_*` for other types will be noops.

Again, for `struct prefix *`:

```
void lua_decode_prefix(lua_State *L, int idx, struct prefix *prefix)
{
     lua_getfield(L, idx, "network");
     (void)str2prefix(lua_tostring(L, -1), prefix);
     /* pop the network string */
     lua_pop(L, 1);
     /* pop the prefix table */
     lua_pop(L, 1);
}
```

Note:

- Before `lua_decode*` is run, the "prefix" table is already on the top of the stack. `frrscript_call` does this for us.
- However, at the end of `lua_decode*`, the "prefix" table should be popped.
- The other two fields in the "network" table are disregarded, meaning that any modification to them is discarded in C space. In this case, this is desired behavior.

Warning: `lua_decode*` functions should pop all values that `lua_to*` pushed onto the Lua stack. For encoders that pushed a table, its decoder should pop the table at the end. The above is an example.

`int` is not a non const-qualified pointer, so for `int`:

```
void lua_decode_int_noop(lua_State *L, int idx, int i)
{ //noop
}
```

A `lua_to*` function provides identical functionality except that it first allocates memory for the new C type before decoding the value from the Lua stack, then returns a pointer to the newly allocated C type. You only need to implement this function to use with `frrscript_get_result` to retrieve a result of this type.

This function can and should be implemented using `lua_decode_*`:

```
void *lua_toprefix(lua_State *L, int idx)
{
        struct prefix *p = XCALLOC(MTYPE_SCRIPT_RES, sizeof(struct prefix));

        lua_decode_prefix(L, idx, p);
        return p;
}
```

The returned data must always be copied off the stack and the copy must be allocated with `MTYPE_SCRIPT_RES`. This way it is possible to unload the script (destroy the state) without invalidating any references to values stored in it. Note that it is the caller's responsibility to free the data.

Registering encoders and decoders for frrscript_call

To register a new type with its `lua_push*` and `lua_decode*` functions, add the mapping in the following macros in `frrscript.h`:

```
  #define ENCODE_ARGS_WITH_STATE(L, value) \
       _Generic((value), \
       ...
- struct peer * : lua_pushpeer \
+ struct peer * : lua_pushpeer, \
+ struct prefix * : lua_pushprefix \
  )((L), (value))

  #define DECODE_ARGS_WITH_STATE(L, value) \
       _Generic((value), \
       ...
- struct peer * : lua_decode_peer \
+ struct peer * : lua_decode_peer, \
+ struct prefix * : lua_decode_prefix \
  )((L), -1, (value))
```

At compile time, the compiler will search for encoders/decoders for the type of each value passed in via `frrscript_call`. If a encoder/decoder cannot be found, it will appear as a compile warning. Note that the types must match *exactly*. In the above example, we defined encoders/decoders for a value of `struct prefix *`, but not `struct prefix` or `const struct prefix *`.

`const` values are a special case. We want to use them in our Lua scripts but not modify them, so creating a decoder for them would be meaningless. But we still need a decoder for the type of value so that the compiler will be satisfied. For that, use `lua_decode_noop`:

```
  #define DECODE_ARGS_WITH_STATE(L, value) \
       _Generic((value), \
       ...
+ const struct prefix * : lua_decode_noop \
  )(L, -1, value)
```

Note: Encodable/decodable types are not restricted to simple values like integers, strings and tables. It is possible to encode a type such that the resultant object in Lua is an actual object-oriented object, complete with methods that call

back into defined C functions. See the Lua manual for how to do this; for a code example, look at how zlog is exported into the script environment.

5.10.4 Script Environment

Logging

For convenience, script environments are populated by default with a `log` object which contains methods corresponding to each of the `zlog` levels:

```
log.info("info")
log.warn("warn")
log.error("error")
log.notice("notice")
log.debug("debug")
```

The log messages will show up in the daemon's log output.

5.10.5 Examples

For a complete code example involving passing custom types, retrieving results, and doing complex calculations in Lua, look at the implementation of the `match script SCRIPT` command for BGP routemaps. This example calls into a script with a function named `route_match`, provides route prefix and attributes received from a peer and expects the function to return a match / no match / match and update result.

An example script to use with this follows. This function matches, does not match or updates a route depending on how many BGP UPDATE messages the peer has received when the script is called, simply as a demonstration of what can be accomplished with scripting.

```
-- Example route map matching
-- author: qlyoung
--
-- The following variables are available in the global environment:
--   log
--     logging library, with the usual functions
--
-- route_match arguments:
--   table prefix
--     the route under consideration
--   table attributes
--     the route's attributes
--   table peer
--     the peer which received this route
--   integer RM_FAILURE
--     status code in case of failure
--   integer RM_NOMATCH
--     status code for no match
--   integer RM_MATCH
--     status code for match
--   integer RM_MATCH_AND_CHANGE
--     status code for match-and-set
```

(continues on next page)

(continued from previous page)

```
--
-- route_match returns table with following keys:
--   integer action, required
--     resultant status code. Should be one of RM_*
--   table attributes, optional
--     updated route attributes
--

function route_match(prefix, attributes, peer,
        RM_FAILURE, RM_NOMATCH, RM_MATCH, RM_MATCH_AND_CHANGE)

        log.info("Evaluating route " .. prefix.network .. " from peer " .. peer.remote_
↪id.string)

        function on_match (prefix, attributes)
                log.info("Match")
                return {
                        attributes = RM_MATCH
                }
        end

        function on_nomatch (prefix, attributes)
                log.info("No match")
                return {
                        action = RM_NOMATCH
                }
        end

        function on_match_and_change (prefix, attributes)
                log.info("Match and change")
                attributes["metric"] = attributes["metric"] + 7
                return {
                        action = RM_MATCH_AND_CHANGE,
                        attributes = attributes
                }
        end

        special_routes = {
                ["172.16.10.4/24"] = on_match,
                ["172.16.13.1/8"] = on_nomatch,
                ["192.168.0.24/8"] = on_match_and_change,
        }


        if special_routes[prefix.network] then
                return special_routes[prefix.network](prefix, attributes)
        elseif peer.stats.update_in % 3 == 0 then
                return on_match(prefix, attributes)
        elseif peer.stats.update_in % 2 == 0 then
                return on_nomatch(prefix, attributes)
        else
                return on_match_and_change(prefix, attributes)
```

(continues on next page)

(continued from previous page)

```
        end
end
```

CHAPTER

SIX

FUZZING

This page describes the fuzzing targets and supported fuzzers available in FRR and how to use them. Familiarity with fuzzing techniques and tools is assumed.

6.1 Overview

It is well known that networked applications tend to be difficult to fuzz on their network-facing attack surfaces. Approaches involving actual network transmission tend to be slow and are subject to intermediate devices and networking stacks which tend to drop fuzzed packets, especially if the fuzzing surface covers IP itself. Some time was spent on fuzzing FRR this way with some mediocre results but attention quickly turned towards skipping the actual networking and instead adding fuzzing targets directly in the packet processing code for use by more traditional in- and out-of-process fuzzers. Results from this approach have been very fruitful.

The patches to add fuzzing targets are kept in a separate git branch. Typically it is better to keep them in the main branch so they are kept up to date and do not need to be constantly synchronized with the main codebase. Unfortunately, changes to FRR to support fuzzing necessarily extend far beyond the entrypoints. Checksums must be disarmed, interactions with the kernel must be skipped, sockets and files must be avoided, desired under/overflows must be marked, etc. There are the usual `LD_PRELOAD` libraries to emulate these things (preeny et al) but FRR is a very kernel-reliant program and these libraries tend to create annoying problems when used with FRR for whatever reason. Keeping this code in the main codebase is cluttering, difficult to work with / around, and runs the risk of accidentally introducing bugs even if `#ifdef`'d out. Consequently it's in a separate branch that is rebased on `master` from time to time.

6.2 Code

The git branch with fuzzing targets is located here:

https://github.com/FRRouting/frr/tree/fuzz

To build libFuzzer targets, pass `--enable-libfuzzer` to `configure`. To build AFL targets, compile with `afl-clang` as usual.

Fuzzing with sanitizers is strongly recommended, especially ASAN, which you can enable by passing `--enable-address-sanitizer` to `configure`.

Suggested UBSAN flags: `-fsanitize-recover=unsigned-integer-overflow,implicit-conversion -fsanitize=unsigned-integer-overflow,implicit-conversion,nullability-arg, nullability-assign,nullability-return` Recommended cflags: `-Wno-all -g3 -O3 -funroll-loops`

6.3 Design

All fuzzing targets have support for libFuzzer and AFL. This is done by writing the target as a libFuzzer entrypoint (`LLVMFuzzerTestOneInput()`) and calling it from the AFL entrypoint in `main()`. New targets should use this rule.

When adding AFL entrypoints, it's a good idea to use AFL persistent mode for better performance. Grep `bgpd/bgp_main.c` for `__AFL_INIT()` for an example of how to do this in FRR. Typically it involves moving all internal daemon setup into a setup function. Then this setup function is called exactly once for the lifetime of the process. In `LLVMFuzzerTestOneInput()` this means you need to call it at the start of the function protected by a static boolean that is set to true, since that function is your entrypoint. You also need to call it prior to `__AFL_INIT()` in `main()` because `main()` is your entrypoint in the AFL case.

6.3.1 Adding support to daemons

This section describes how to add entrypoints to daemons that do not have any yet.

Because libFuzzer has its own `main()` function, when adding fuzzing support to a daemon that doesn't have any targets already, `main()` needs to be `#ifdef`'d out like so:

```
#ifndef FUZZING_LIBFUZZER

int main(int argc, char **argv)
{
...
}

#endif /* FUZZING_LIBFUZZER */
```

The `FUZZING_LIBFUZZER` macro is set by `--enable-libfuzzer`.

Because libFuzzer can only be linked into daemons that have `LLVMFuzzerTestOneInput()` implemented, we can't pass `-fsanitize=fuzzer` to all daemons in `AM_CFLAGS`. It needs to go into a variable specific to each daemon. Since it can be thought of as a kind of sanitizer, for daemons that have libFuzzer support there are now individual flags variables for those daemons named `DAEMON_SAN_FLAGS` (e.g. `BGPD_SAN_FLAGS`, `ZEBRA_SAN_FLAGS`). This variable has the contents of the generic `SAN_FLAGS` plus any fuzzing-related flags. It is used in daemons' `subdir.am` in place of `SAN_FLAGS`. Daemons that don't support libFuzzer still use `SAN_FLAGS`. If you want to add fuzzing support to a daemon you need to do this flag variable conversion; look at `configure.ac` for examples, it is fairly straightforward. Remember to update `subdir.am` to use the new variable.

Do note that when fuzzing is enabled, `SAN_FLAGS` gains `-fsanitize=fuzzer-no-link`; the result is that all daemons are instrumented for fuzzing but only the ones with `LLVMFuzzerTestOneInput()` actually get linked with libFuzzer.

6.4 Targets

A given daemon can have lots of different paths that are interesting to fuzz. There's not really a great way to handle this, most fuzzers assume the program has one entrypoint. The approach taken in FRR for multiple entrypoints is to control which path is taken within `LLVMFuzzerTestOneInput()` using `#ifdef` and passing whatever controlling macro definition you want. Take a look at that function for the daemon you're interested in fuzzing, pick the target, add `#define MY_TARGET 1` somewhere before the `#ifdef` switch, recompile.

Table 1: Fuzzing Targets

Daemon	Target	Fuzzers
bgpd	packet parser	libfuzzer, afl
ospfd	packet parser	libfuzzer, afl
pimd	packet parser	libfuzzer, afl
vrrpd	packet parser	libfuzzer, afl
vrrpd	zapi parser	libfuzzer, afl
zebra	netlink	libfuzzer, afl
zebra	zserv / zapi	libfuzzer, afl

6.5 Fuzzer Notes

Some interesting seed corpuses for various daemons are available here.

For libFuzzer, you need to pass `-rss_limit_mb=0` if you are fuzzing with ASAN enabled, as you should.

For AFL, afl++ is strongly recommended; afl proper isn't really maintained anymore.

CHAPTER

SEVEN

TRACING

FRR has a small but growing number of static tracepoints available for use with various tracing systems. These tracepoints can assist with debugging, performance analysis and to help understand program flow. They can also be used for monitoring.

Developers are encouraged to write new static tracepoints where sensible. They are not compiled in by default, and even when they are, they have no overhead unless enabled by a tracer, so it is okay to be liberal with them.

7.1 Supported tracers

Presently two types of tracepoints are supported:

- LTTng tracepoints
- USDT probes

LTTng is a tracing framework for Linux only. It offers extremely low overhead and very rich tracing capabilities. FRR supports LTTng-UST, which is the userspace implementation. LTTng tracepoints are very rich in detail. No kernel modules are needed. Besides only being available for Linux, the primary downside of LTTng is the need to link to `lttng-ust`.

USDT probes originate from Solaris, where they were invented for use with dtrace. They are a kernel feature. At least Linux and FreeBSD support them. No library is needed; support is compiled in via a system header (`<sys/sdt.h>`). USDT probes are much slower than LTTng tracepoints and offer less flexibility in what information can be gleaned from them.

LTTng is capable of tracing USDT probes but has limited support for them. SystemTap and dtrace both work only with USDT probes.

7.2 Usage

To compile with tracepoints, use one of the following configure flags:

--enable-lttng=yes
: Generate LTTng tracepoints

--enable-usdt=yes
: Generate USDT probes

To trace with LTTng, compile with either one (prefer *--enable-lttng* run the target in non-forking mode (no `-d`) and use LTTng as usual (refer to LTTng user manual). When using USDT probes with LTTng, follow the example in this article. To trace with dtrace or SystemTap, compile with *–enable-usdt=yes* and use your tracer as usual.

To see available USDT probes:

```
readelf -n /usr/lib/frr/bgpd
```

Example:

```
root@host ~> readelf -n /usr/lib/frr/bgpd

Displaying notes found in: .note.ABI-tag
  Owner                 Data size    Description
  GNU                  0x00000010    NT_GNU_ABI_TAG (ABI version tag)
    OS: Linux, ABI: 3.2.0

Displaying notes found in: .note.gnu.build-id
  Owner                 Data size    Description
  GNU                  0x00000014    NT_GNU_BUILD_ID (unique build ID bitstring)
    Build ID: 4f42933a69dcb42a519bc459b2105177c8adf55d

Displaying notes found in: .note.stapsdt
  Owner                 Data size    Description
  stapsdt              0x00000045    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: packet_read
    Location: 0x000000000045ee48, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-96(%rbp) 8@-104(%rbp)
  stapsdt              0x00000047    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: open_process
    Location: 0x000000000047c43b, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-224(%rbp) 2@-226(%rbp)
  stapsdt              0x00000049    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: update_process
    Location: 0x000000000047c4bf, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-208(%rbp) 2@-210(%rbp)
  stapsdt              0x0000004f    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: notification_process
    Location: 0x000000000047c557, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-192(%rbp) 2@-194(%rbp)
  stapsdt              0x0000004c    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: keepalive_process
    Location: 0x000000000047c5db, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-176(%rbp) 2@-178(%rbp)
  stapsdt              0x0000004a    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: refresh_process
    Location: 0x000000000047c673, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-160(%rbp) 2@-162(%rbp)
  stapsdt              0x0000004d    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: capability_process
    Location: 0x000000000047c6f7, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-144(%rbp) 2@-146(%rbp)
```

(continues on next page)

(continued from previous page)

```
  stapsdt              0x0000006f    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: output_filter
    Location: 0x000000000048e33a, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-144(%rbp) 8@-152(%rbp) 4@-156(%rbp) 4@-160(%rbp) 8@-168(%rbp)
  stapsdt              0x0000007d    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: process_update
    Location: 0x000000000049 1f10, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-800(%rbp) 8@-808(%rbp) 4@-812(%rbp) 4@-816(%rbp) 4@-820(%rbp) 8@-832(
↪%rbp)
  stapsdt              0x0000006e    NT_STAPSDT (SystemTap probe descriptors)
    Provider: frr_bgp
    Name: input_filter
    Location: 0x00000000004940ed, Base: 0x00000000005a09d2, Semaphore: 0x0000000000000000
    Arguments: 8@-144(%rbp) 8@-152(%rbp) 4@-156(%rbp) 4@-160(%rbp) 8@-168(%rbp)
```

To see available LTTng probes, run the target, create a session and then:

```
lttng list --userspace | grep frr
```

Example:

```
root@host ~> lttng list --userspace | grep frr
PID: 11157 - Name: /usr/lib/frr/bgpd
      frr_libfrr:route_node_get (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:list_sort (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:list_delete_node (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:list_remove (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:list_add (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:memfree (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:memalloc (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:frr_pthread_stop (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:frr_pthread_run (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      frr_libfrr:thread_call (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:thread_cancel_async (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:thread_cancel (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:schedule_write (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:schedule_read (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:schedule_event (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:schedule_timer (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:hash_release (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:hash_insert (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_libfrr:hash_get (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:output_filter (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:input_filter (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:process_update (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:packet_read (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:refresh_process (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:capability_process (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:notification_process (loglevel: TRACE_INFO (6)) (type: tracepoint)
      frr_bgp:update_process (loglevel: TRACE_INFO (6)) (type: tracepoint)
```

(continues on next page)

(continued from previous page)

```
    frr_bgp:keepalive_process (loglevel: TRACE_INFO (6)) (type: tracepoint)
    frr_bgp:open_process (loglevel: TRACE_INFO (6)) (type: tracepoint)
```

When using LTTng, you can also get zlogs as trace events by enabling the `lttng_ust_tracelog:*` event class.

To see available SystemTap USDT probes, run:

```
stap -L 'process("/usr/lib/frr/bgpd").mark("*")'
```

Example:

```
root@host ~> stap -L 'process("/usr/lib/frr/bgpd").mark("*")'
process("/usr/lib/frr/bgpd").mark("capability_process") $arg1:long $arg2:long
process("/usr/lib/frr/bgpd").mark("input_filter") $arg1:long $arg2:long $arg3:long
↪$arg4:long $arg5:long
process("/usr/lib/frr/bgpd").mark("keepalive_process") $arg1:long $arg2:long
process("/usr/lib/frr/bgpd").mark("notification_process") $arg1:long $arg2:long
process("/usr/lib/frr/bgpd").mark("open_process") $arg1:long $arg2:long
process("/usr/lib/frr/bgpd").mark("output_filter") $arg1:long $arg2:long $arg3:long
↪$arg4:long $arg5:long
process("/usr/lib/frr/bgpd").mark("packet_read") $arg1:long $arg2:long
process("/usr/lib/frr/bgpd").mark("process_update") $arg1:long $arg2:long $arg3:long
↪$arg4:long $arg5:long $arg6:long
process("/usr/lib/frr/bgpd").mark("refresh_process") $arg1:long $arg2:long
process("/usr/lib/frr/bgpd").mark("update_process") $arg1:long $arg2:long
```

When using SystemTap, you can also easily attach to an existing function:

```
stap -L 'process("/usr/lib/frr/bgpd").function("bgp_update_receive")'
```

Example:

```
root@host ~> stap -L 'process("/usr/lib/frr/bgpd").function("bgp_update_receive")'
process("/usr/lib/frr/bgpd").function("bgp_update_receive@bgpd/bgp_packet.c:1531")
↪$peer:struct peer* $size:bgp_size_t $attr:struct attr $restart:_Bool $nlris:struct bgp_
↪nlri[] $__func__:char const[] const
```

Complete `bgp.stp` example using SystemTap to show BGP peer, prefix and aspath using `process_update` USDT:

```
global pkt_size;
probe begin
{
  ansi_clear_screen();
  println("Starting...");
}
probe process("/usr/lib/frr/bgpd").function("bgp_update_receive")
{
  pkt_size <<< $size;
}
probe process("/usr/lib/frr/bgpd").mark("process_update")
{
  aspath = @cast($arg6, "attr")->aspath;
  printf("> %s via %s (%s)\n",
```

(continues on next page)

(continued from previous page)

```
    user_string($arg2),
    user_string(@cast($arg1, "peer")->host),
    user_string(@cast(aspath, "aspath")->str));
}
probe end
{
  if (@count(pkt_size))
    print(@hist_linear(pkt_size, 0, 20, 2));
}
```

Output:

```
Starting...
> 192.168.0.0/24 via 192.168.0.1 (65534)
> 192.168.100.1/32 via 192.168.0.1 (65534)
> 172.16.16.1/32 via 192.168.0.1 (65534 65030)
^Cvalue |------------------------------------------------- count
    0 |                                                  0
    2 |                                                  0
    4 |@                                                 1
    6 |                                                  0
    8 |                                                  0
      ~
  18 |                                                  0
  20 |                                                  0
  >20 |@@@@@                                             5
```

7.3 Concepts

Tracepoints are statically defined points in code where a developer has determined that outside observers might gain something from knowing what is going on at that point. It's like logging but with the ability to dump large amounts of internal data with much higher performance. LTTng has a good summary here.

Each tracepoint has a "provider" and name. The provider is basically a namespace; for example, `bgpd` uses the provider name `frr_bgp`. The name is arbitrary, but because providers share a global namespace on the user's system, all providers from FRR should be prefixed by `frr_`. The tracepoint name is just the name of the event. Events are globally named by their provider and name. For example, the event when BGP reads a packet from a peer is `frr_bgp:packet_read`.

To do tracing, the tracing tool of choice is told which events to listen to. For example, to listen to all events from FRR's BGP implementation, you would enable the events `frr_bgp:*`. In the same tracing session you could also choose to record all memory allocations by enabling the `malloc` tracepoints in `libc` as well as all kernel skb operations using the various in-kernel tracepoints. This allows you to build as complete a view as desired of what the system is doing during the tracing window (subject to what tracepoints are available).

Of particular use are the tracepoints for FRR's internal event scheduler; tracing these allows you to see all events executed by all event loops for the target(s) in question. Here's a couple events selected from a trace of BGP during startup:

```
...

[18:41:35.750131763] (+0.000048901) host frr_libfrr:thread_call: { cpu_id =
```

(continues on next page)

(continued from previous page)

```
1 }, { threadmaster_name = "default", function_name = "zclient_connect",
scheduled_from = "lib/zclient.c", scheduled_on_line = 3877, thread_addr =
0x0, file_descriptor = 0, event_value = 0, argument_ptr = 0xA37F70, timer =
0 }

[18:41:35.750175124] (+0.000020001) host frr_libfrr:thread_call: { cpu_id =
1 }, { threadmaster_name = "default", function_name = "frr_config_read_in",
scheduled_from = "lib/libfrr.c", scheduled_on_line = 934, thread_addr = 0x0,
file_descriptor = 0, event_value = 0, argument_ptr = 0x0, timer = 0 }

[18:41:35.753341264] (+0.000010532) host frr_libfrr:thread_call: { cpu_id =
1 }, { threadmaster_name = "default", function_name = "bgp_event",
scheduled_from = "bgpd/bgpd.c", scheduled_on_line = 142, thread_addr = 0x0,
file_descriptor = 2, event_value = 2, argument_ptr = 0xE4D780, timer = 2 }

[18:41:35.753404186] (+0.000004910) host frr_libfrr:thread_call: { cpu_id =
1 }, { threadmaster_name = "default", function_name = "zclient_read",
scheduled_from = "lib/zclient.c", scheduled_on_line = 3891, thread_addr =
0x0, file_descriptor = 40, event_value = 40, argument_ptr = 0xA37F70, timer
= 40 }

...
```

Very useful for getting a time-ordered look into what the process is doing.

7.4 Adding Tracepoints

Adding new tracepoints is a two step process:

1. Define the tracepoint
2. Use the tracepoint

Tracepoint definitions state the "provider" and name of the tracepoint, along with any values it will produce, and how to format them. This is done with macros provided by LTTng. USDT probes do not use definitions and are inserted at the trace site with a single macro. However, to maintain support for both platforms, you must define an LTTng tracepoint when adding a new one. `frrtrace()` will expand to the appropriate `DTRACE_PROBEn` macro when USDT is in use.

If you are adding new tracepoints to a daemon that has no tracepoints, that daemon's `subdir.am` must be updated to conditionally link `lttng-ust`. Look at `bgpd/subdir.am` for an example of how to do this; grep for `UST_LIBS`. Create new files named `<daemon>_trace.[ch]`. Use `bgpd/bgp_trace.[h]` as boilerplate. If you are adding tracepoints to a daemon that already has them, look for the `<daemon>_trace.h` file; tracepoints are written here.

Refer to the LTTng developer docs for details on how to define tracepoints.

To use them, simply add a call to `frrtrace()` at the point you'd like the event to be emitted, like so:

```
...

switch (type) {
case BGP_MSG_OPEN:
        frrtrace(2, frr_bgp, open_process, peer, size); /* tracepoint */
        atomic_fetch_add_explicit(&peer->open_in, 1,
```

(continues on next page)

(continued from previous page)

```
                                    memory_order_relaxed);
        mprc = bgp_open_receive(peer, size);

...
```

After recompiling this tracepoint will now be available, either as a USDT probe or LTTng tracepoint, depending on your compilation choice.

7.4.1 trace.h

Because FRR supports multiple types of tracepoints, the code for creating them abstracts away the underlying system being used. This abstraction code is in `lib/trace.h`. There are 2 function-like macros that are used for working with tracepoints.

- `frrtrace()` defines tracepoints
- `frrtrace_enabled()` checks whether a tracepoint is enabled

There is also `frrtracelog()`, which is used in zlog core code to make zlog messages available as trace events to LTTng. This should not be used elsewhere.

There is additional documentation in the header. The key thing to note is that you should never include `trace.h` in source where you plan to put tracepoints; include the tracepoint definition header instead (e.g. `bgp_trace.h`).

7.5 Limitations

Tracers do not like `fork()` or `dlopen()`. LTTng has some workarounds for this involving interceptor libraries using `LD_PRELOAD`.

If you're running FRR in a typical daemonizing way (`-d` to the daemons) you'll need to run the daemons like so:

```
LD_PRELOAD=liblttng-ust-fork.so <daemon>
```

If you're using systemd this you can accomplish this for all daemons by modifying `frr.service` like so:

```
--- a/frr.service
+++ b/frr.service
@@ -7,6 +7,7 @@ Before=network.target
 OnFailure=heartbeat-failed@%n

 [Service]
+Environment="LD_PRELOAD=liblttng-ust-fork.so"
 Nice=-5
 Type=forking
 NotifyAccess=all
```

USDT tracepoints are relatively high overhead and probably shouldn't be used for "flight recorder" functionality, i.e. enabling and passively recording all events for monitoring purposes. It's generally okay to use LTTng like this, though.

CHAPTER

EIGHT

TESTING

8.1 Topotests

Topotests is a suite of topology tests for FRR built on top of micronet.

8.1.1 Installation and Setup

Topotests run under python3. Additionally, for ExaBGP (which is used in some of the BGP tests) an older python2 version (and the python2 version of `pip`) must be installed.

Tested with Ubuntu 20.04,Ubuntu 18.04, and Debian 11.

Instructions are the same for all setups (i.e. ExaBGP is only used for BGP tests).

Installing Topotest Requirements

```
apt-get install gdb
apt-get install iproute2
apt-get install net-tools
apt-get install python3-pip
python3 -m pip install wheel
python3 -m pip install 'pytest>=6.2.4'
python3 -m pip install 'pytest-xdist>=2.3.0'
python3 -m pip install 'scapy>=2.4.5'
python3 -m pip install xmltodict
# Use python2 pip to install older ExaBGP
python2 -m pip install 'exabgp<4.0.0'
useradd -d /var/run/exabgp/ -s /bin/false exabgp

# To enable the gRPC topotest install:
python3 -m pip install grpcio grpcio-tools

# Install Socat tool to run PIMv6 tests,
# Socat code can be taken from below url,
# which has latest changes done for PIMv6,
# join and traffic:
https://github.com/opensourcerouting/socat/
```

Enable Coredumps

Optional, will give better output.

```
disable apport (which move core files)
```

Set `enabled=0` in `/etc/default/apport`.

Next, update security limits by changing `/etc/security/limits.conf` to:

```
#<domain>      <type>  <item>         <value>
*               soft    core          unlimited
root            soft    core          unlimited
*               hard    core          unlimited
root            hard    core          unlimited
```

Reboot for options to take effect.

SNMP Utilities Installation

To run SNMP test you need to install SNMP utilities and MIBs. Unfortunately there are some errors in the upstream MIBS which need to be patched up. The following steps will get you there on Ubuntu 20.04.

```
apt install libsnmp-dev
apt install snmpd snmp
apt install snmp-mibs-downloader
download-mibs
wget http://www.iana.org/assignments/ianaippmmetricsregistry-mib/ianaippmmetricsregistry-
↪mib -O /usr/share/snmp/mibs/iana/IANA-IPPM-METRICS-REGISTRY-MIB
wget http://pastebin.com/raw.php?i=p3QyuXzZ -O /usr/share/snmp/mibs/ietf/SNMPv2-PDU
wget http://pastebin.com/raw.php?i=gG7j8nyk -O /usr/share/snmp/mibs/ietf/IPATM-IPMC-MIB
edit /etc/snmp/snmp.conf to look like this
# As the snmp packages come without MIB files due to license reasons, loading
# of MIBs is disabled by default. If you added the MIBs you can reenable
# loading them by commenting out the following line.
mibs +ALL
```

FRR Installation

FRR needs to be installed separately. It is assume to be configured like the standard Ubuntu Packages:

- Binaries in `/usr/lib/frr`
- State Directory `/var/run/frr`
- Running under user `frr`, group `frr`
- vtygroup: `frrvty`
- config directory: `/etc/frr`
- For FRR Packages, install the dbg package as well for coredump decoding

No FRR config needs to be done and no FRR daemons should be run ahead of the test. They are all started as part of the test.

Manual FRR build

If you prefer to manually build FRR, then use the following suggested config:

```
./configure \
    --prefix=/usr \
    --localstatedir=/var/run/frr \
    --sbindir=/usr/lib/frr \
    --sysconfdir=/etc/frr \
    --enable-vtysh \
    --enable-pimd \
    --enable-sharpd \
    --enable-multipath=64 \
    --enable-user=frr \
    --enable-group=frr \
    --enable-vty-group=frrvty \
    --enable-snmp=agentx \
    --with-pkg-extra-version=-my-manual-build
```

And create `frr` user and `frrvty` group as follows:

```
addgroup --system --gid 92 frr
addgroup --system --gid 85 frrvty
adduser --system --ingroup frr --home /var/run/frr/ \
   --gecos "FRRouting suite" --shell /bin/false frr
usermod -G frrvty frr
```

8.1.2 Executing Tests

Configure your sudo environment

Topotests must be run as root. Normally this will be accomplished through the use of the `sudo` command. In order for topotests to be able to open new windows (either XTerm or byobu/screen/tmux windows) certain environment variables must be passed through the sudo command. One way to do this is to specify the `-E` flag to `sudo`. This will carry over most if not all your environment variables include `PATH`. For example:

```
sudo -E python3 -m pytest -s -v
```

If you do not wish to use `-E` (e.g., to avoid `sudo` inheriting `PATH`) you can modify your */etc/sudoers* config file to specifically pass the environment variables required by topotests. Add the following commands to your `/etc/sudoers` config file.

```
Defaults env_keep="TMUX"
Defaults env_keep+="TMUX_PANE"
Defaults env_keep+="STY"
Defaults env_keep+="DISPLAY"
```

If there was already an `env_keep` configuration there be sure to use the += rather than = on the first line above as well.

Execute all tests in distributed test mode

```
sudo -E pytest -s -v -nauto --dist=loadfile
```

The above command must be executed from inside the topotests directory.

All test_* scripts in subdirectories are detected and executed (unless disabled in `pytest.ini` file). Pytest will execute up to N tests in parallel where N is based on the number of cores on the host.

Analyze Test Results (`analyze.py`)

By default router and execution logs are saved in `/tmp/topotests` and an XML results file is saved in `/tmp/topotests.xml`. An analysis tool `analyze.py` is provided to archive and analyze these results after the run completes.

After the test run completes one should pick an archive directory to store the results in and pass this value to `analyze.py`. On first execution the results are copied to that directory from `/tmp`, and subsequent runs use that directory for analyzing the results. Below is an example of this which also shows the default behavior which is to display all failed and errored tests in the run.

```
~/frr/tests/topotests# ./analyze.py -Ar run-save
bgp_multiview_topo1/test_bgp_multiview_topo1.py::test_bgp_converge
ospf_basic_functionality/test_ospf_lan.py::test_ospf_lan_tc1_p0
bgp_gr_functionality_topo2/test_bgp_gr_functionality_topo2.py::test_BGP_GR_10_p2
bgp_multiview_topo1/test_bgp_multiview_topo1.py::test_bgp_routingTable
```

Here we see that 4 tests have failed. We an dig deeper by displaying the captured logs and errors. First let's redisplay the results enumerated by adding the `-E` flag

```
~/frr/tests/topotests# ./analyze.py -Ar run-save -E
0 bgp_multiview_topo1/test_bgp_multiview_topo1.py::test_bgp_converge
1 ospf_basic_functionality/test_ospf_lan.py::test_ospf_lan_tc1_p0
2 bgp_gr_functionality_topo2/test_bgp_gr_functionality_topo2.py::test_BGP_GR_10_p2
3 bgp_multiview_topo1/test_bgp_multiview_topo1.py::test_bgp_routingTable
```

Now to look at the error message for a failed test we use `-T N` where N is the number of the test we are interested in along with `--errmsg` option.

```
~/frr/tests/topotests# ./analyze.py -Ar run-save -T0 --errmsg
bgp_multiview_topo1/test_bgp_multiview_topo1.py::test_bgp_converge: AssertionError: BGP␣
↪did not converge:

  IPv4 Unicast Summary (VIEW 1):
  BGP router identifier 172.30.1.1, local AS number 100 vrf-id -1
  BGP table version 1
  RIB entries 1, using 184 bytes of memory
  Peers 3, using 2169 KiB of memory

  Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/
↪PfxRcd   PfxSnt Desc
  172.16.1.1      4      65001         0         0        0    0    0    never     ␣
↪Connect        0 N/A
  172.16.1.2      4      65002         0         0        0    0    0    never     ␣
↪Connect        0 N/A
```

(continues on next page)

(continued from previous page)

```
 172.16.1.5      4      65005         0         0        0    0    0    never      ␣
↪Connect        0 N/A

 Total number of neighbors 3

assert False
```

Now to look at the full text of the error for a failed test we use `-T N` where N is the number of the test we are interested in along with `--errtext` option.

```
~/frr/tests/topotests# ./analyze.py -Ar run-save -T0 --errtext
bgp_multiview_topo1/test_bgp_multiview_topo1.py::test_bgp_converge: def test_bgp_
↪converge():
        "Check for BGP converged on all peers and BGP views"

        global fatal_error
        global net
        [...]
        else:
            # Bail out with error if a router fails to converge
            bgpStatus = net["r%s" % i].cmd('vtysh -c "show ip bgp view %s summary"' %␣
↪view)
>           assert False, "BGP did not converge:\n%s" % bgpStatus
E           AssertionError: BGP did not converge:
E
E             IPv4 Unicast Summary (VIEW 1):
E             BGP router identifier 172.30.1.1, local AS number 100 vrf-id -1
              [...]
E             Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/
↪Down State/PfxRcd   PfxSnt Desc
E             172.16.1.1      4      65001         0         0        0    0    0   ␣
↪never      Connect        0 N/A
E             172.16.1.2      4      65002         0         0        0    0    0   ␣
↪never      Connect        0 N/A
              [...]
```

To look at the full capture for a test including the stdout and stderr which includes full debug logs, just use the `-T N` option without the `--errmsg` or `--errtext` options.

```
~/frr/tests/topotests# ./analyze.py -Ar run-save -T0
@classname: bgp_multiview_topo1.test_bgp_multiview_topo1
@name: test_bgp_converge
@time: 141.401
@message: AssertionError: BGP did not converge:
[...]
system-out: ------------------------------- Captured Log ------------------------------
↪---
2021-08-09 02:55:06,581 DEBUG: lib.micronet_compat.topo: Topo(unnamed): Creating
2021-08-09 02:55:06,581 DEBUG: lib.micronet_compat.topo: Topo(unnamed): addHost r1
[...]
2021-08-09 02:57:16,932 DEBUG: topolog.r1: LinuxNamespace(r1): cmd_status("['/bin/bash',
↪'-c', 'vtysh -c "show ip bgp view 1 summary" 2> /dev/null | grep ^[0-9] | grep -vP "␣
↪11\\s+(\\d+)"']", kwargs: {'encoding': 'utf-8', 'stdout': -1, 'stderr': -2, 'shell':␣
↪False})
```

(continues on next page)

(continued from previous page)

```
2021-08-09 02:57:22,290 DEBUG: topolog.r1: LinuxNamespace(r1): cmd_status("['/bin/bash',
↪'-c', 'vtysh -c "show ip bgp view 1 summary" 2> /dev/null | grep ^[0-9] | grep -vP "␣
↪11\\s+(\\d+)"']", kwargs: {'encoding': 'utf-8', 'stdout': -1, 'stderr': -2, 'shell':␣
↪False})
2021-08-09 02:57:27,636 DEBUG: topolog.r1: LinuxNamespace(r1): cmd_status("['/bin/bash',
↪'-c', 'vtysh -c "show ip bgp view 1 summary"']", kwargs: {'encoding': 'utf-8', 'stdout
↪': -1, 'stderr': -2, 'shell': False})
-------------------------------- Captured Out --------------------------------
system-err: -------------------------------- Captured Err --------------------------------
↪---
```

Execute single test

```
cd test_to_be_run
./test_to_be_run.py
```

For example, and assuming you are inside the frr directory:

```
cd tests/topotests/bgp_l3vpn_to_bgp_vrf
./test_bgp_l3vpn_to_bgp_vrf.py
```

For further options, refer to pytest documentation.

Test will set exit code which can be used with `git bisect`.

For the simulated topology, see the description in the python file.

StdErr log from daemos after exit

To enable the reporting of any messages seen on StdErr after the daemons exit, the following env variable can be set:

```
export TOPOTESTS_CHECK_STDERR=Yes
```

(The value doesn't matter at this time. The check is whether the env variable exists or not.) There is no pass/fail on this reporting; the Output will be reported to the console.

Collect Memory Leak Information

FRR processes can report unfreed memory allocations upon exit. To enable the reporting of memory leaks, define an environment variable `TOPOTESTS_CHECK_MEMLEAK` with the file prefix, i.e.:

```
export TOPOTESTS_CHECK_MEMLEAK="/home/mydir/memleak_"
```

This will enable the check and output to console and the writing of the information to files with the given prefix (followed by testname), ie `/home/mydir/memcheck_test_bgp_multiview_topo1.txt` in case of a memory leak.

Running Topotests with AddressSanitizer

Topotests can be run with AddressSanitizer. It requires GCC 4.8 or newer. (Ubuntu 16.04 as suggested here is fine with GCC 5 as default). For more information on AddressSanitizer, see https://github.com/google/sanitizers/wiki/AddressSanitizer.

The checks are done automatically in the library call of `checkRouterRunning` (ie at beginning of tests when there is a check for all daemons running). No changes or extra configuration for topotests is required beside compiling the suite with AddressSanitizer enabled.

If a daemon crashed, then the errorlog is checked for AddressSanitizer output. If found, then this is added with context (calling test) to `/tmp/AddressSanitizer.txt` in Markdown compatible format.

Compiling for GCC AddressSanitizer requires to use `gcc` as a linker as well (instead of `ld`). Here is a suggest way to compile frr with AddressSanitizer for `master` branch:

```
git clone https://github.com/FRRouting/frr.git
cd frr
./bootstrap.sh
./configure \
    --enable-address-sanitizer \
    --prefix=/usr/lib/frr --sysconfdir=/etc/frr \
    --localstatedir=/var/run/frr \
    --sbindir=/usr/lib/frr --bindir=/usr/lib/frr \
    --with-moduledir=/usr/lib/frr/modules \
    --enable-multipath=0 --enable-rtadv \
    --enable-tcp-zebra --enable-fpm --enable-pimd \
    --enable-sharpd
make
sudo make install
# Create symlink for vtysh, so topotest finds it in /usr/lib/frr
sudo ln -s /usr/lib/frr/vtysh /usr/bin/
```

and create `frr` user and `frrvty` group as shown above.

Debugging Topotest Failures

Install and run tests inside `tmux` or `byobu` for best results.

`XTerm` is also fully supported. GNU `screen` can be used in most situations; however, it does not work as well with launching `vtysh` or shell on error.

For the below debugging options which launch programs or CLIs, topotest should be run within `tmux` (or `screen`)_, as `gdb`, the shell or `vtysh` will be launched using that windowing program, otherwise `xterm` will be attempted to launch the given programs.

NOTE: you must run the topotest (pytest) such that your DISPLAY, STY or TMUX environment variables are carried over. You can do this by passing the `-E` flag to `sudo` or you can modify your `/etc/sudoers` config to automatically pass that environment variable through to the `sudo` environment.

Spawning Debugging CLI, vtysh or Shells on Routers on Test Failure

One can have a debugging CLI invoked on test failures by specifying the `--cli-on-error` CLI option as shown in the example below.

```
sudo -E pytest --cli-on-error all-protocol-startup
```

The debugging CLI can run shell or vtysh commands on any combination of routers It can also open shells or vtysh in their own windows for any combination of routers. This is usually the most useful option when debugging failures. Here is the help command from within a CLI launched on error:

```
test_bgp_multiview_topo1/test_bgp_routingTable> help

Commands:
help                        :: this help
sh [hosts] <shell-command> :: execute <shell-command> on <host>
term [hosts]               :: open shell terminals for hosts
vtysh [hosts]              :: open vtysh terminals for hosts
[hosts] <vtysh-command>    :: execute vtysh-command on hosts

test_bgp_multiview_topo1/test_bgp_routingTable> r1 show int br
------ Host: r1 ------
Interface       Status  VRF             Addresses
---------       ------  ---             ---------
erspan0         down    default
gre0            down    default
gretap0         down    default
lo              up      default
r1-eth0         up      default         172.16.1.254/24
r1-stub         up      default         172.20.0.1/28

----------------------
test_bgp_multiview_topo1/test_bgp_routingTable>
```

Additionally, one can have `vtysh` or a shell launched on all routers when a test fails. To launch the given process on each router after a test failure specify one of `--shell-on-error` or `--vtysh-on-error`.

Spawning vtysh or Shells on Routers

Topotest can automatically launch a shell or `vtysh` for any or all routers in a test. This is enabled by specifying 1 of 2 CLI arguments `--shell` or `--vtysh`. Both of these options can be set to a single router value, multiple comma-seperated values, or `all`.

When either of these options are specified topotest will pause after setup and each test to allow for inspection of the router state.

Here's an example of launching `vtysh` on routers `rt1` and `rt2`.

```
sudo -E pytest --vtysh=rt1,rt2 all-protocol-startup
```

Debugging with GDB

Topotest can automatically launch any daemon with `gdb`, possibly setting breakpoints for any test run. This is enabled by specifying 1 or 2 CLI arguments `--gdb-routers` and `--gdb-daemons`. Additionally `--gdb-breakpoints` can be used to automatically set breakpoints in the launched `gdb` processes.

Each of these options can be set to a single value, multiple comma-seperated values, or `all`. If `--gdb-routers` is empty but `--gdb_daemons` is set then the given daemons will be launched in `gdb` on all routers in the test. Likewise if `--gdb_routers` is set, but `--gdb_daemons` is empty then all daemons on the given routers will be launched in `gdb`.

Here's an example of launching `zebra` and `bgpd` inside `gdb` on router `r1` with a breakpoint set on `nb_config_diff`

```
sudo -E pytest --gdb-routers=r1 \
       --gdb-daemons=bgpd,zebra \
       --gdb-breakpoints=nb_config_diff \
       all-protocol-startup
```

Detecting Memleaks with Valgrind

Topotest can automatically launch all daemons with `valgrind` to check for memleaks. This is enabled by specifying 1 or 2 CLI arguments. `--valgrind-memleaks` will enable general memleak detection, and `--valgrind-extra` enables extra functionality including generating a suppression file. The suppression file `tools/valgrind.supp` is used when memleak detection is enabled.

```
sudo -E pytest --valgrind-memleaks all-protocol-startup
```

8.1.3 Running Tests with Docker

There is a Docker image which allows to run topotests.

Quickstart

If you have Docker installed, you can run the topotests in Docker. The easiest way to do this, is to use the make targets from this repository.

Your current user needs to have access to the Docker daemon. Alternatively you can run these commands as root.

```
make topotests
```

This command will pull the most recent topotests image from Dockerhub, compile FRR inside of it, and run the topotests.

Advanced Usage

Internally, the topotests make target uses a shell script to pull the image and spawn the Docker container.

There are several environment variables which can be used to modify the behavior of the script, these can be listed by calling it with `-h`:

```
./tests/topotests/docker/frr-topotests.sh -h
```

For example, a volume is used to cache build artifacts between multiple runs of the image. If you need to force a complete recompile, you can set `TOPOTEST_CLEAN`:

```
TOPOTEST_CLEAN=1 ./tests/topotests/docker/frr-topotests.sh
```

By default, `frr-topotests.sh` will build frr and run pytest. If you append arguments and the first one starts with `/` or `./`, they will replace the call to pytest. If the appended arguments do not match this patttern, they will be provided to pytest as arguments. So, to run a specific test with more verbose logging:

```
./tests/topotests/docker/frr-topotests.sh -vv -s all-protocol-startup/test_all_protocol_
↪startup.py
```

And to compile FRR but drop into a shell instead of running pytest:

```
./tests/topotests/docker/frr-topotests.sh /bin/bash
```

Development

The Docker image just includes all the components to run the topotests, but not the topotests themselves. So if you just want to write tests and don't want to make changes to the environment provided by the Docker image. You don't need to build your own Docker image if you do not want to.

When developing new tests, there is one caveat though: The startup script of the container will run a `git-clean` on its copy of the FRR tree to avoid any pollution of the container with build artefacts from the host. This will also result in your newly written tests being unavailable in the container unless at least added to the index with `git-add`.

If you do want to test changes to the Docker image, you can locally build the image and run the tests without pulling from the registry using the following commands:

```
make topotests-build
TOPOTEST_PULL=0 make topotests
```

8.1.4 Guidelines

Executing Tests

To run the whole suite of tests the following commands must be executed at the top level directory of topotest:

```
$ # Change to the top level directory of topotests.
$ cd path/to/topotests
$ # Tests must be run as root, since micronet requires it.
$ sudo -E pytest
```

In order to run a specific test, you can use the following command:

```
$ # running a specific topology
$ sudo -E pytest ospf-topo1/
$ # or inside the test folder
$ cd ospf-topo1
$ sudo -E pytest # to run all tests inside the directory
$ sudo -E pytest test_ospf_topo1.py # to run a specific test
$ # or outside the test folder
$ cd ..
$ sudo -E pytest ospf-topo1/test_ospf_topo1.py # to run a specific one
```

The output of the tested daemons will be available at the temporary folder of your machine:

```
$ ls /tmp/topotest/ospf-topo1.test_ospf-topo1/r1
...
zebra.err # zebra stderr output
zebra.log # zebra log file
zebra.out # zebra stdout output
...
```

You can also run memory leak tests to get reports:

```
$ # Set the environment variable to apply to a specific test...
$ sudo -E env TOPOTESTS_CHECK_MEMLEAK="/tmp/memleak_report_" pytest ospf-topo1/test_ospf_
↪topo1.py
$ # ...or apply to all tests adding this line to the configuration file
$ echo 'memleak_path = /tmp/memleak_report_' >> pytest.ini
$ # You can also use your editor
$ $EDITOR pytest.ini
$ # After running tests you should see your files:
$ ls /tmp/memleak_report_*
memleak_report_test_ospf_topo1.txt
```

Writing a New Test

This section will guide you in all recommended steps to produce a standard topology test.

This is the recommended test writing routine:

- Write a topology (Graphviz recommended)
- Obtain configuration files
- Write the test itself
- Format the new code using black
- Create a Pull Request

Some things to keep in mind:

- BGP tests MUST use generous convergence timeouts - you must ensure that any test involving BGP uses a convergence timeout of at least 130 seconds.
- Topotests are run on a range of Linux versions: if your test requires some OS-specific capability (like mpls support, or vrf support), there are test functions available in the libraries that will help you determine whether your test should run or be skipped.

- Avoid including unstable data in your test: don't rely on link-local addresses or ifindex values, for example, because these can change from run to run.
- Using sleep is almost never appropriate. As an example: if the test resets the peers in BGP, the test should look for the peers re-converging instead of just sleeping an arbitrary amount of time and continuing on. See `verify_bgp_convergence` as a good example of this. In particular look at it's use of the `@retry` decorator. If you are having troubles figuring out what to look for, please do not be afraid to ask.
- Don't duplicate effort. There exists many protocol utility functions that can be found in their eponymous module under `tests/topotests/lib/` (e.g., `ospf.py`)

Topotest File Hierarchy

Before starting to write any tests one must know the file hierarchy. The repository hierarchy looks like this:

```
$ cd path/to/topotest
$ find ./*
...
./README.md # repository read me
./GUIDELINES.md # this file
./conftest.py # test hooks - pytest related functions
./example-test # example test folder
./example-test/__init__.py # python package marker - must always exist.
./example-test/test_template.jpg # generated topology picture - see next section
./example-test/test_template.dot # Graphviz dot file
./example-test/test_template.py # the topology plus the test
...
./ospf-topo1 # the ospf topology test
./ospf-topo1/r1 # router 1 configuration files
./ospf-topo1/r1/zebra.conf # zebra configuration file
./ospf-topo1/r1/ospfd.conf # ospf configuration file
./ospf-topo1/r1/ospfroute.txt # 'show ip ospf' output reference file
# removed other for shortness sake
...
./lib # shared test/topology functions
./lib/topogen.py # topogen implementation
./lib/topotest.py # topotest implementation
```

Guidelines for creating/editing topotest:

- New topologies that don't fit the existing directories should create its own
- Always remember to add the `__init__.py` to new folders, this makes auto complete engines and pylint happy
- Router (Quagga/FRR) specific code should go on topotest.py
- Generic/repeated router actions should have an abstraction in topogen.TopoRouter.
- Generic/repeated non-router code should go to topotest.py
- pytest related code should go to conftest.py (e.g. specialized asserts)

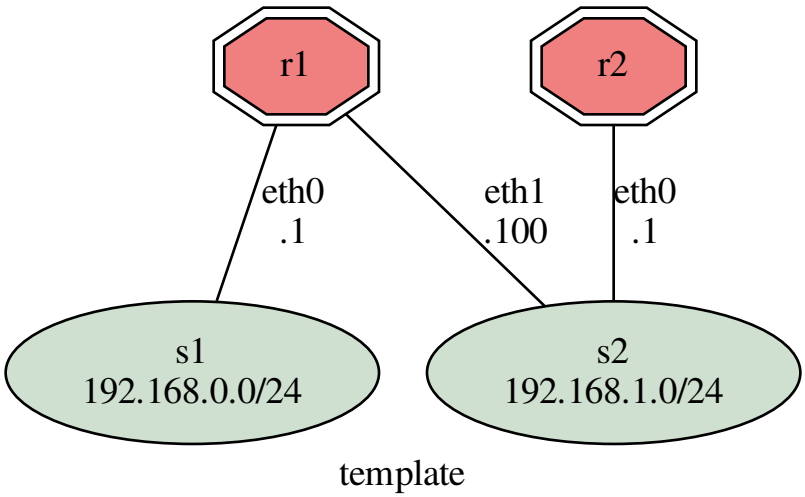
Defining the Topology

The first step to write a new test is to define the topology. This step can be done in many ways, but the recommended is to use Graphviz to generate a drawing of the topology. It allows us to see the topology graphically and to see the names of equipment, links and addresses.

Here is an example of Graphviz dot file that generates the template topology `tests/topotests/example-test/test_template.dot` (the inlined code might get outdated, please see the linked file):

```
graph template {
    label="template";

    # Routers
    r1 [
        shape=doubleoctagon,
        label="r1",
        fillcolor="#f08080",
        style=filled,
    ];
    r2 [
        shape=doubleoctagon,
        label="r2",
        fillcolor="#f08080",
        style=filled,
    ];

    # Switches
    s1 [
        shape=oval,
        label="s1\n192.168.0.0/24",
        fillcolor="#d0e0d0",
        style=filled,
    ];
    s2 [
        shape=oval,
        label="s2\n192.168.1.0/24",
        fillcolor="#d0e0d0",
        style=filled,
    ];

    # Connections
    r1 -- s1 [label="eth0\n.1"];

    r1 -- s2 [label="eth1\n.100"];
    r2 -- s2 [label="eth0\n.1"];
}
```

Here is the produced graph:



Generating / Obtaining Configuration Files

In order to get the configuration files or command output for each router, we need to run the topology and execute commands in `vtysh`. The quickest way to achieve that is writing the topology building code and running the topology.

To bootstrap your test topology, do the following steps:

- Copy the template test

```
$ mkdir new-topo/
$ touch new-topo/__init__.py
$ cp example-test/test_template.py new-topo/test_new_topo.py
```

- Modify the template according to your dot file

Here is the template topology described in the previous section in python code:

```
topodef = {
    "s1": "r1"
    "s2": ("r1", "r2")
}
```

If more specialized topology definitions, or router initialization arguments are required a build function can be used instead of a dictionary:

```
def build_topo(tgen):
    "Build function"

    # Create 2 routers
    for routern in range(1, 3):
        tgen.add_router("r{}".format(routern))
```

(continues on next page)

(continued from previous page)

```
    # Create a switch with just one router connected to it to simulate a
    # empty network.
    switch = tgen.add_switch("s1")
    switch.add_link(tgen.gears["r1"])

    # Create a connection between r1 and r2
    switch = tgen.add_switch("s2")
    switch.add_link(tgen.gears["r1"])
    switch.add_link(tgen.gears["r2"])
```

- Run the topology

Topogen allows us to run the topology without running any tests, you can do that using the following example commands:

```
$ # Running your bootstraped topology
$ sudo -E pytest -s --topology-only new-topo/test_new_topo.py
$ # Running the test_template.py topology
$ sudo -E pytest -s --topology-only example-test/test_template.py
$ # Running the ospf_topo1.py topology
$ sudo -E pytest -s --topology-only ospf-topo1/test_ospf_topo1.py
```

Parameters explanation:

-s
: Actives input/output capture. If this is not specified a new window will be opened for the interactive CLI, otherwise it will be activated inline.

--topology-only
: Don't run any tests, just build the topology.

After executing the commands above, you should get the following terminal output:

```
frr/tests/topotests# sudo -E pytest -s --topology-only ospf_topo1/test_ospf_topo1.py
============================= test session starts ==============================
platform linux -- Python 3.9.2, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/chopps/w/frr/tests/topotests, configfile: pytest.ini
plugins: forked-1.3.0, xdist-2.3.0
collected 11 items

[...]
unet>
```

The last line shows us that we are now using the CLI (Command Line Interface), from here you can call your router `vtysh` or even bash.

Here's the help text:

```
unet> help

Commands:
  help                       :: this help
  sh [hosts] <shell-command> :: execute <shell-command> on <host>
  term [hosts]               :: open shell terminals for hosts
```

(continues on next page)

(continued from previous page)

```
  vtysh [hosts]              :: open vtysh terminals for hosts
  [hosts] <vtysh-command>    :: execute vtysh-command on hosts
```

Here are some commands example:

```
unet> sh r1 ping 10.0.3.1
PING 10.0.3.1 (10.0.3.1) 56(84) bytes of data.
64 bytes from 10.0.3.1: icmp_seq=1 ttl=64 time=0.576 ms
64 bytes from 10.0.3.1: icmp_seq=2 ttl=64 time=0.083 ms
64 bytes from 10.0.3.1: icmp_seq=3 ttl=64 time=0.088 ms
^C
--- 10.0.3.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.083/0.249/0.576/0.231 ms

unet> r1 show run
Building configuration...

Current configuration:
!
frr version 8.1-dev-my-manual-build
frr defaults traditional
hostname r1
log file /tmp/topotests/ospf_topo1.test_ospf_topo1/r1/zebra.log
[...]
end

unet> show daemons
------ Host: r1 ------
 zebra ospfd ospf6d staticd
------- End: r1 ------
------ Host: r2 ------
 zebra ospfd ospf6d staticd
------- End: r2 ------
------ Host: r3 ------
 zebra ospfd ospf6d staticd
------- End: r3 ------
------ Host: r4 ------
 zebra ospfd ospf6d staticd
------- End: r4 ------
```

After you successfully configured your topology, you can obtain the configuration files (per-daemon) using the following commands:

```
unet> sh r3 vtysh -d ospfd

Hello, this is FRRouting (version 3.1-devrzalamena-build).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

r1# show running-config
Building configuration...
```

(continues on next page)

(continued from previous page)

```
Current configuration:
!
frr version 3.1-devrzalamena-build
frr defaults traditional
no service integrated-vtysh-config
!
log file ospfd.log
!
router ospf
 ospf router-id 10.0.255.3
 redistribute kernel
 redistribute connected
 redistribute static
 network 10.0.3.0/24 area 0
 network 10.0.10.0/24 area 0
 network 172.16.0.0/24 area 1
!
line vty
!
end
r1#
```

You can also login to the node specified by nsenter using bash, etc. A pid file for each node will be created in the relevant test dir. You can run scripts inside the node, or use vtysh's <tab> or <?> feature.

```
[unet shell]
# cd tests/topotests/srv6_locator
# ./test_srv6_locator.py --topology-only
unet> r1 show segment-routing srv6 locator
Locator:
Name                 ID      Prefix                   Status
-------------------- ------- ------------------------ -------
loc1                       1 2001:db8:1:1::/64        Up
loc2                       2 2001:db8:2:2::/64        Up

[Another shell]
# nsenter -a -t $(cat /tmp/topotests/srv6_locator.test_srv6_locator/r1.pid) bash --norc
# vtysh
r1# r1 show segment-routing srv6 locator
Locator:
Name                 ID      Prefix                   Status
-------------------- ------- ------------------------ -------
loc1                       1 2001:db8:1:1::/64        Up
loc2                       2 2001:db8:2:2::/64        Up
```

Writing Tests

Test topologies should always be bootstrapped from `tests/topotests/example_test/test_template.py` because it contains important boilerplate code that can't be avoided, like:

Example:

```
# For all routers arrange for:
# - starting zebra using config file from <rtrname>/zebra.conf
# - starting ospfd using an empty config file.
for rname, router in router_list.items():
    router.load_config(TopoRouter.RD_ZEBRA, "zebra.conf")
    router.load_config(TopoRouter.RD_OSPF)
```

- The topology definition or build function

```
topodef = {
    "s1": ("r1", "r2"),
    "s2": ("r2", "r3")
}

def build_topo(tgen):
    # topology build code
    ...
```

- pytest setup/teardown fixture to start the topology and supply `tgen` argument to tests.

```
@pytest.fixture(scope="module")
def tgen(request):
    "Setup/Teardown the environment and provide tgen argument to tests"

    tgen = Topogen(topodef, module.__name__)
    # or
    tgen = Topogen(build_topo, module.__name__)

    ...

    # Start and configure the router daemons
    tgen.start_router()

    # Provide tgen as argument to each test function
    yield tgen

    # Teardown after last test runs
    tgen.stop_topology()
```

Requirements:

- Directory name for a new topotest must not contain hyphen (-) characters. To separate words, use underscores (_). For example, `tests/topotests/bgp_new_example`.
- Test code should always be declared inside functions that begin with the `test_` prefix. Functions beginning with different prefixes will not be run by pytest.
- Configuration files and long output commands should go into separated files inside folders named after the equipment.

- Tests must be able to run without any interaction. To make sure your test conforms with this, run it without the `-s` parameter.
- Use black code formatter before creating a pull request. This ensures we have a unified code style.
- Mark test modules with pytest markers depending on the daemons used during the tests (see *Markers*)
- Always use IPv4 **RFC 5737** (`192.0.2.0/24`, `198.51.100.0/24`, `203.0.113.0/24`) and IPv6 **RFC 3849** (`2001:db8::/32`) ranges reserved for documentation.

Tips:

- Keep results in stack variables, so people inspecting code with `pdb` can easily print their values.

Don't do this:

```
assert foobar(router1, router2)
```

Do this instead:

```
result = foobar(router1, router2)
assert result
```

- Use `assert` messages to indicate where the test failed.

Example:

```
for router in router_list:
   # ...
   assert condition, 'Router "{}" condition failed'.format(router.name)
```

Debugging Execution

The most effective ways to inspect topology tests are:

- Run pytest with `--pdb` option. This option will cause a pdb shell to appear when an assertion fails

Example: `pytest -s --pdb ospf-topo1/test_ospf_topo1.py`

- Set a breakpoint in the test code with `pdb`

Example:

```
# Add the pdb import at the beginning of the file
import pdb
# ...

# Add a breakpoint where you think the problem is
def test_bla():
  # ...
  pdb.set_trace()
  # ...
```

The Python Debugger (pdb) shell allows us to run many useful operations like:

- Setting breaking point on file/function/conditions (e.g. `break`, `condition`)
- Inspecting variables (e.g. `p` (print), `pp` (pretty print))
- Running python code

Tip: The TopoGear (equipment abstraction class) implements the `__str__` method that allows the user to inspect equipment information.

Example of pdb usage:

```
> /media/sf_src/topotests/ospf-topo1/test_ospf_topo1.py(121)test_ospf_convergence()
-> for rnum in range(1, 5):
(Pdb) help
Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv

(Pdb) list
116                                     title2="Expected output")
117
118     def test_ospf_convergence():
119         "Test OSPF daemon convergence"
120         pdb.set_trace()
121  ->     for rnum in range(1, 5):
122             router = 'r{}'.format(rnum)
123
124             # Load expected results from the command
125             reffile = os.path.join(CWD, '{}/ospfroute.txt'.format(router))
126             expected = open(reffile).read()
(Pdb) step
> /media/sf_src/topotests/ospf-topo1/test_ospf_topo1.py(122)test_ospf_convergence()
-> router = 'r{}'.format(rnum)
(Pdb) step
> /media/sf_src/topotests/ospf-topo1/test_ospf_topo1.py(125)test_ospf_convergence()
-> reffile = os.path.join(CWD, '{}/ospfroute.txt'.format(router))
(Pdb) print rnum
1
(Pdb) print router
r1
(Pdb) tgen = get_topogen()
(Pdb) pp tgen.gears[router]
<lib.topogen.TopoRouter object at 0x7f74e06c9850>
(Pdb) pp str(tgen.gears[router])
'TopoGear<name="r1",links=["r1-eth0"<->"s1-eth0","r1-eth1"<->"s3-eth0"]> TopoRouter<>'
```

(continues on next page)

(continued from previous page)

```
(Pdb) l 125
120         pdb.set_trace()
121         for rnum in range(1, 5):
122             router = 'r{}'.format(rnum)
123
124             # Load expected results from the command
125  ->         reffile = os.path.join(CWD, '{}/ospfroute.txt'.format(router))
126             expected = open(reffile).read()
127
128             # Run test function until we get an result. Wait at most 60 seconds.
129             test_func = partial(compare_show_ip_ospf, router, expected)
130             result, diff = topotest.run_and_expect(test_func, '',
(Pdb) router1 = tgen.gears[router]
(Pdb) router1.vtysh_cmd('show ip ospf route')
'============ OSPF network routing table ============\r\nN    10.0.1.0/24           [10]␣
↪area: 0.0.0.0\r\n                           directly attached to r1-eth0\r\nN    10.0.
↪2.0/24           [20] area: 0.0.0.0\r\n                           via 10.0.3.3, r1-
↪eth1\r\nN    10.0.3.0/24           [10] area: 0.0.0.0\r\n                          ␣
↪directly attached to r1-eth1\r\nN    10.0.10.0/24          [20] area: 0.0.0.0\r\n    ␣
↪                    via 10.0.3.1, r1-eth1\r\nN IA 172.16.0.0/24         [20] area: 0.
↪0.0.0\r\n                           via 10.0.3.1, r1-eth1\r\nN IA 172.16.1.0/24       ␣
↪ [30] area: 0.0.0.0\r\n                           via 10.0.3.1, r1-eth1\r\n\r\
↪n============ OSPF router routing table ==============\r\nR    10.0.255.2            ␣
↪[10] area: 0.0.0.0, ASBR\r\n                           via 10.0.3.3, r1-eth1\r\nR   ␣
↪10.0.255.3           [10] area: 0.0.0.0, ABR, ASBR\r\n                          via␣
↪10.0.3.1, r1-eth1\r\nR    10.0.255.4         IA [20] area: 0.0.0.0, ASBR\r\n          ␣
↪                via 10.0.3.1, r1-eth1\r\n\r\n============ OSPF external routing table␣
↪===========\r\n\r\n\r\n'
 (Pdb) tgen.cli()
 unet>
```

To enable more debug messages in other Topogen subsystems, more logging messages can be displayed by modifying the test configuration file `pytest.ini`:

```
[topogen]
# Change the default verbosity line from 'info'...
#verbosity = info
# ...to 'debug'
verbosity = debug
```

Instructions for use, write or debug topologies can be found in *Guidelines*. To learn/remember common code snippets see *Snippets*.

Before creating a new topology, make sure that there isn't one already that does what you need. If nothing is similar, then you may create a new topology, preferably, using the newest template (`tests/topotests/example-test/test_template.py`).

8.1.5 Markers

To allow for automated selective testing on large scale continuous integration systems, all tests must be marked with at least one of the following markers:

- babeld
- bfdd
- bgpd
- eigrpd
- isisd
- ldpd
- nhrpd
- ospf6d
- ospfd
- pathd
- pbrd
- pimd
- ripd
- ripngd
- sharpd
- staticd
- vrrpd

The markers corespond to the daemon subdirectories in FRR's source code and have to be added to tests on a module level depending on which daemons are used during the test.

The goal is to have continuous integration systems scan code submissions, detect changes to files in a daemons subdirectory and select only tests using that daemon to run to shorten developers waiting times for test results and save test infrastructure resources.

Newly written modules and code changes on tests, which do not contain any or incorrect markers will be rejected by reviewers.

Registering markers

The Registration of new markers takes place in the file `tests/topotests/pytest.ini`:

```
# tests/topotests/pytest.ini
[pytest]
...
markers =
    babeld: Tests that run against BABELD
    bfdd: Tests that run against BFDD
    ...
    vrrpd: Tests that run against VRRPD
```

Adding markers to tests

Markers are added to a test by placing a global variable in the test module.

Adding a single marker:

```
import pytest
...

# add after imports, before defining classes or functions:
pytestmark = pytest.mark.bfdd

...

def test_using_bfdd():
```

Adding multiple markers:

```
import pytest
...

# add after imports, before defining classes or functions:
pytestmark = [
    pytest.mark.bgpd,
    pytest.mark.ospfd,
    pytest.mark.ospf6d
]

...

def test_using_bgpd_ospfd_ospf6d():
```

Selecting marked modules for testing

Selecting by a single marker:

```
pytest -v -m isisd
```

Selecting by multiple markers:

```
pytest -v -m "isisd or ldpd or nhrpd"
```

Further Information

The online pytest documentation provides further information and usage examples for pytest markers.

8.1.6 Snippets

This document will describe common snippets of code that are frequently needed to perform some test checks.

Checking for router / test failures

The following check uses the topogen API to check for software failure (e.g. zebra died) and/or for errors manually set by `Topogen.set_error()`.

```
# Get the topology reference
tgen = get_topogen()

# Check for errors in the topology
if tgen.routers_have_failure():
    # Skip the test with the topology errors as reason
    pytest.skip(tgen.errors)
```

Checking FRR routers version

This code snippet is usually run after the topology setup to make sure all routers instantiated in the topology have the correct software version.

```
# Get the topology reference
tgen = get_topogen()

# Get the router list
router_list = tgen.routers()

# Run the check for all routers
for router in router_list.values():
    if router.has_version('<', '3'):
        # Set topology error, so the next tests are skipped
        tgen.set_error('unsupported version')
```

A sample of this snippet in a test can be found here.

Interacting with equipment

You might want to interact with the topology equipment during the tests and there are different ways to do so.

Notes:

1. When using the Topogen API, all the equipment code derives from `Topogear` (lib/topogen.py). If you feel brave you can look by yourself how the abstractions that will be mentioned here work.
2. When not using the `Topogen` API there is only one way to interact with the equipment, which is by calling the `mininet` API functions directly to spawn commands.

Interacting with the Linux sandbox

Without Topogen:

```
global net
output = net['r1'].cmd('echo "foobar"')
print 'output is: {}'.format(output)
```

With Topogen:

```
tgen = get_topogen()
output = tgen.gears['r1'].run('echo "foobar"')
print 'output is: {}'.format(output)
```

Interacting with VTYSH

Without Topogen:

```
global net
output = net['r1'].cmd('vtysh "show ip route" 2>/dev/null')
print 'output is: {}'.format(output)
```

With Topogen:

```
tgen = get_topogen()
output = tgen.gears['r1'].vtysh_cmd("show ip route")
print 'output is: {}'.format(output)
```

Topogen also supports sending multiple lines of command:

```
tgen = get_topogen()
output = tgen.gears['r1'].vtysh_cmd("""
configure terminal
router bgp 10
  bgp router-id 10.0.255.1
  neighbor 1.2.3.4 remote-as 10
  !
router bgp 11
  bgp router-id 10.0.255.2
  !
""")
print 'output is: {}'.format(output)
```

You might also want to run multiple commands and get only the commands that failed:

```
tgen = get_topogen()
output = tgen.gears['r1'].vtysh_multicmd("""
configure terminal
router bgp 10
  bgp router-id 10.0.255.1
  neighbor 1.2.3.4 remote-as 10
  !
router bgp 11
```

(continues on next page)

(continued from previous page)

```
  bgp router-id 10.0.255.2
  !
""", pretty_output=false)
print 'output is: {}'.format(output)
```

Translating vtysh JSON output into Python structures:

```
tgen = get_topogen()
json_output = tgen.gears['r1'].vtysh_cmd("show ip route json", isjson=True)
output = json.dumps(json_output, indent=4)
print 'output is: {}'.format(output)

# You can also access the data structure as normal. For example:
# protocol = json_output['1.1.1.1/32']['protocol']
# assert protocol == "ospf", "wrong protocol"
```

Note: `vtysh_(multi)cmd` is only available for router types of equipment.

Invoking mininet CLI

Without Topogen:

```
CLI(net)
```

With Topogen:

```
tgen = get_topogen()
tgen.mininet_cli()
```

Reading files

Loading a normal text file content in the current directory:

```
# If you are using Topogen
# CURDIR = CWD
#
# Otherwise find the directory manually:
CURDIR = os.path.dirname(os.path.realpath(__file__))

file_name = '{}/r1/show_ip_route.txt'.format(CURDIR)
file_content = open(file_name).read()
```

Loading JSON from a file:

```
import json

file_name = '{}/r1/show_ip_route.json'.format(CURDIR)
file_content = json.loads(open(file_name).read())
```

Comparing JSON output

After obtaining JSON output formatted with Python data structures, you may use it to assert a minimalist schema:

```
tgen = get_topogen()
json_output = tgen.gears['r1'].vtysh_cmd("show ip route json", isjson=True)

expect = {
  '1.1.1.1/32': {
    'protocol': 'ospf'
  }
}

assertmsg = "route 1.1.1.1/32 was not learned through OSPF"
assert json_cmp(json_output, expect) is None, assertmsg
```

`json_cmp` function description (it might be outdated, you can find the latest description in the source code at `tests/topotests/lib/topotest.py`

```
JSON compare function. Receives two parameters:
* `d1`: json value
* `d2`: json subset which we expect

Returns `None` when all keys that `d1` has matches `d2`,
otherwise a string containing what failed.

Note: key absence can be tested by adding a key with value `None`.
```

Pausing execution

Preferably, choose the `sleep` function that `topotest` provides, as it prints a notice during the test execution to help debug topology test execution time.

```
# Using the topotest sleep
from lib import topotest

topotest.sleep(10, 'waiting 10 seconds for bla')
# or just tell it the time:
# topotest.sleep(10)
# It will print 'Sleeping for 10 seconds'.

# Or you can also use the Python sleep, but it won't show anything
from time import sleep
sleep(5)
```

iproute2 Linux commands as JSON

`topotest` has two helpers implemented that parses the output of `ip route` commands to JSON. It might simplify your comparison needs by only needing to provide a Python dictionary.

```
from lib import topotest

tgen = get_topogen()
routes = topotest.ip4_route(tgen.gears['r1'])
expected = {
  '10.0.1.0/24': {},
  '10.0.2.0/24': {
    'dev': 'r1-eth0'
  }
}

assertmsg = "failed to find 10.0.1.0/24 and/or 10.0.2.0/24"
assert json_cmp(routes, expected) is None, assertmsg
```

8.1.7 License

8.2 Topotests with JSON

8.2.1 Overview

On top of current topotests framework following enhancements are done:

- Creating the topology and assigning IPs to router' interfaces dynamically. It is achieved by using json file, in which user specify the number of routers, links to each router, interfaces for the routers and protocol configurations for all routers.
- Creating the configurations dynamically. It is achieved by using `/usr/lib/frr/frr-reload.py` utility, which takes running configuration and the newly created configuration for any particular router and creates a delta file(diff file) and loads it to router.

8.2.2 Logging of test case executions

- The execution log for each test is saved in the test specific directory create under */tmp/topotests* (e.g., */tmp/topotests/<testdirname.testfilename>/exec.log*)
- Additionally all test logs are captured in the *topotest.xml* results file. This file will be saved in */tmp/topotests/topotests.xml*. In order to extract the logs for a particular test one can use the *analyze.py* utility found in the topotests base directory.
- Router's current configuration, as it is changed during the test, can be displayed on console or sent to logs by adding `show_router_config = True` in `pytest.ini`.

Note: directory "/tmp/topotests/" is created by topotests by default, making use of same directory to save execution logs.

8.2.3 Guidelines

Writing New Tests

This section will guide you in all recommended steps to produce a standard topology test.

This is the recommended test writing routine:

- Create a json file which will have routers and protocol configurations
- Write and debug the tests
- Format the new code using black
- Create a Pull Request

Note: BGP tests MUST use generous convergence timeouts - you must ensure that any test involving BGP uses a convergence timeout that is proportional to the configured BGP timers. If the timers are not reduced from their defaults this means 130 seconds; however, it is highly recommended that timers be reduced from the default values unless the test requires they not be.

File Hierarchy

Before starting to write any tests one must know the file hierarchy. The repository hierarchy looks like this:

```
$ cd frr/tests/topotests
$ find ./*
...
./example_test/
./example_test/test_template_json.json # input json file, having topology, interfaces,␣
↪bgp and other configuration
./example_test/test_template_json.py # test script to write and execute testcases
...
./lib # shared test/topology functions
./lib/topojson.py # library to create topology and configurations dynamically from json␣
↪file
./lib/common_config.py # library to create protocol's common configurations ex- static_
↪routes, prefix_lists, route_maps etc.
./lib/bgp.py # library to create and test bgp configurations
```

Defining the Topology and initial configuration in JSON file

The first step to write a new test is to define the topology and initial configuration. User has to define topology and initial configuration in JSON file. Here is an example of JSON file:

```
BGP neighborship with single phy-link, sample JSON file:
{
"ipv4base": "192.168.0.0",
"ipv4mask": 30,
"ipv6base": "fd00::",
"ipv6mask": 64,
"link_ip_start": {"ipv4": "192.168.0.0", "v4mask": 30, "ipv6": "fd00::", "v6mask": 64},
```

(continues on next page)

(continued from previous page)

```
"lo_prefix": {"ipv4": "1.0.", "v4mask": 32, "ipv6": "2001:DB8:F::", "v6mask": 128},
"routers": {
    "r1": {
        "links": {
            "lo": {"ipv4": "auto", "ipv6": "auto", "type": "loopback"},
            "r2": {"ipv4": "auto", "ipv6": "auto"},
            "r3": {"ipv4": "auto", "ipv6": "auto"}
        },
        "bgp": {
            "local_as": "64512",
            "address_family": {
                "ipv4": {
                    "unicast": {
                        "neighbor": {
                            "r2": {
                                "dest_link": {
                                    "r1": {}
                                }
                            },
                            "r3": {
                                "dest_link": {
                                    "r1": {}
                                }
                            }
                        }
                    }
                }
            }
        }
    },
    "r2": {
        "links": {
            "lo": {"ipv4": "auto", "ipv6": "auto", "type": "loopback"},
            "r1": {"ipv4": "auto", "ipv6": "auto"},
            "r3": {"ipv4": "auto", "ipv6": "auto"}
        },
        "bgp": {
            "local_as": "64512",
            "address_family": {
                "ipv4": {
                    "unicast": {
                        "redistribute": [
                            {
                                "redist_type": "static"
                            }
                        ],
                        "neighbor": {
                            "r1": {
                                "dest_link": {
                                    "r2": {}
                                }
                            },
```

(continues on next page)

(continued from previous page)

```
                            "r3": {
                                "dest_link": {
                                    "r2": {}
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    ...
```

BGP neighboship with loopback interface, sample JSON file:

```
{
"ipv4base": "192.168.0.0",
"ipv4mask": 30,
"ipv6base": "fd00::",
"ipv6mask": 64,
"link_ip_start": {"ipv4": "192.168.0.0", "v4mask": 30, "ipv6": "fd00::", "v6mask": 64},
"lo_prefix": {"ipv4": "1.0.", "v4mask": 32, "ipv6": "2001:DB8:F::", "v6mask": 128},
"routers": {
    "r1": {
        "links": {
            "lo": {"ipv4": "auto", "ipv6": "auto", "type": "loopback",
                   "add_static_route":"yes"},
            "r2": {"ipv4": "auto", "ipv6": "auto"}
        },
        "bgp": {
            "local_as": "64512",
            "address_family": {
                "ipv4": {
                    "unicast": {
                        "neighbor": {
                            "r2": {
                                "dest_link": {
                                    "lo": {
                                        "source_link": "lo"
                                    }
                                }
                            }
                        }
                    }
                }
            }
        },
        "static_routes": [
            {
                "network": "1.0.2.17/32",
                "next_hop": "192.168.0.1
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    },
    "r2": {
        "links": {
            "lo": {"ipv4": "auto", "ipv6": "auto", "type": "loopback",
                   "add_static_route":"yes"},
            "r1": {"ipv4": "auto", "ipv6": "auto"},
            "r3": {"ipv4": "auto", "ipv6": "auto"}
        },
        "bgp": {
            "local_as": "64512",
            "address_family": {
                "ipv4": {
                    "unicast": {
                        "redistribute": [
                            {
                                "redist_type": "static"
                            }
                        ],
                        "neighbor": {
                            "r1": {
                                "dest_link": {
                                    "lo": {
                                        "source_link": "lo"
                                    }
                                }
                            },
                            "r3": {
                                "dest_link": {
                                    "lo": {
                                        "source_link": "lo"
                                    }
                                }
                            }
                        }
                    }
                }
            }
        },
        "static_routes": [
            {
                "network": "192.0.20.1/32",
                "no_of_ip": 9,
                "admin_distance": 100,
                "next_hop": "192.168.0.1",
                "tag": 4001
            }
        ],
    }
    ...
```

BGP neighborship with Multiple phy-links, sample JSON file:

```
{
"ipv4base": "192.168.0.0",
"ipv4mask": 30,
"ipv6base": "fd00::",
"ipv6mask": 64,
"link_ip_start": {"ipv4": "192.168.0.0", "v4mask": 30, "ipv6": "fd00::", "v6mask": 64},
"lo_prefix": {"ipv4": "1.0.", "v4mask": 32, "ipv6": "2001:DB8:F::", "v6mask": 128},
"routers": {
    "r1": {
        "links": {
            "lo": {"ipv4": "auto", "ipv6": "auto", "type": "loopback"},
            "r2-link1": {"ipv4": "auto", "ipv6": "auto"},
            "r2-link2": {"ipv4": "auto", "ipv6": "auto"}
        },
        "bgp": {
            "local_as": "64512",
            "address_family": {
                "ipv4": {
                    "unicast": {
                        "neighbor": {
                            "r2": {
                                "dest_link": {
                                    "r1-link1": {}
                                }
                            }
                        }
                    }
                }
            }
        }
    },
    "r2": {
        "links": {
            "lo": {"ipv4": "auto", "ipv6": "auto", "type": "loopback"},
            "r1-link1": {"ipv4": "auto", "ipv6": "auto"},
            "r1-link2": {"ipv4": "auto", "ipv6": "auto"},
            "r3-link1": {"ipv4": "auto", "ipv6": "auto"},
            "r3-link2": {"ipv4": "auto", "ipv6": "auto"}
        },
        "bgp": {
            "local_as": "64512",
            "address_family": {
                "ipv4": {
                    "unicast": {
                        "redistribute": [
                            {
                                "redist_type": "static"
                            }
                        ],
                        "neighbor": {
                            "r1": {
                                "dest_link": {
                                    "r2-link1": {}
```

(continues on next page)

(continued from previous page)

```
                            }
                        },
                        "r3": {
                            "dest_link": {
                                "r2-link1": {}
                            }
                        }
                    }
                }
            }
        }
    }
    ...
```

JSON File Explained

Mandatory keywords/options in JSON:

- `ipv4base` : base ipv4 address to generate ips, ex - 192.168.0.0
- `ipv4mask` : mask for ipv4 address, ex - 30
- `ipv6base` : base ipv6 address to generate ips, ex - fd00:
- `ipv6mask` : mask for ipv6 address, ex - 64
- `link_ip_start` : physical interface base ipv4 and ipv6 address
- `lo_prefix` : loopback interface base ipv4 and ipv6 address
- `routers` : user can add number of routers as per topology, router's name can be any logical name, ex- r1 or a0.
- `r1` : name of the router
- `lo` : loopback interface dict, ipv4 and/or ipv6 addresses generated automatically
- `type` : type of interface, to identify loopback interface
- `links` : physical interfaces dict, ipv4 and/or ipv6 addresses generated automatically
- `r2-link1` : it will be used when routers have multiple links. 'r2' is router name, 'link' is any logical name, '1' is to identify link number, router name and link must be seperated by hyphen (-), ex- a0-peer1

Optional keywords/options in JSON:

- `bgp` : bgp configuration
- `local_as` : Local AS number
- `unicast` : All SAFI configuration
- `neighbor`: All neighbor details
- `dest_link` : Destination link to which router will connect
- `router_id` : bgp router-id
- `source_link` : if user wants to establish bgp neighborship with loopback interface, add `source_link`: `lo`
- `keepalivetimer` : Keep alive timer for BGP neighbor

- `holddowntimer` : Hold down timer for BGP neighbor
- `static_routes` : create static routes for routers
- `redistribute` : redistribute static and/or connected routes
- `prefix_lists` : create Prefix-lists for routers

Building topology and configurations

Topology and initial configuration as well as teardown are invoked through the use of a pytest fixture:

```
from lib import fixtures

tgen = pytest.fixture(fixtures.tgen_json, scope="module")


# tgen is defined above
# topo is a fixture defined in ../conftest.py and automatically available
def test_bgp_convergence(tgen, topo):
    bgp_convergence = bgp.verify_bgp_convergence(tgen, topo)
    assert bgp_convergence
```

The *fixtures.topo_json* function calls *topojson.setup_module_from_json()* to create and return a new *topogen.Topogen()* object using the JSON config file with the same base filename as the test (i.e., *test_file.py* -> *test_file.json*). Additionally, the fixture calls *tgen.stop_topology()* after all the tests have run to cleanup. The function is only invoked once per file/module (scope="module"), but the resulting object is passed to each function that has *tgen* as an argument.

For more info on the powerful pytest fixtures feature please see FIXTURES.

Creating configuration files

Router's configuration would be saved in config file frr_json.conf. Common configurations are like, static routes, prefixlists and route maps etc configs, these configs can be used by any other protocols as it is. BGP config will be specific to BGP protocol testing.

- json file is passed to API Topogen() which saves the JSON object in *self.json_topo*
- The Topogen object is then passed to API build_config_from_json(), which looks for configuration tags in new JSON object.
- If tag is found in the JSON object, configuration is created as per input and written to file frr_json.conf
- Once JSON parsing is over, frr_json.conf is loaded onto respective router. Config loading is done using 'vtysh -f <file>'. Initial config at this point is also saved frr_json_initial.conf. This file can be used to reset configuration on router, during the course of execution.
- Reset of configuration is done using frr "reload.py" utility, which calculates the difference between router's running config and user's config and loads delta file to router. API used - reset_config_on_router()

Writing Tests

Test topologies should always be bootstrapped from the *example_test/test_template_json.py* when possible in order to take advantage of the most recent infrastructure support code.

Example:

- Define a module scoped fixture to setup/teardown and supply the tests with the *Topogen* object.

```
import pytest
from lib import fixtures

tgen = pytest.fixture(fixtures.tgen_json, scope="module")
```

- Define test functions using pytest fixtures

```
from lib import bgp

# tgen is defined above
# topo is a global available fixture defined in ../conftest.py
def test_bgp_convergence(tgen, topo):
    "Test for BGP convergence."

    # Don't run this test if we have any failure.
    if tgen.routers_have_failure():
        pytest.skip(tgen.errors)

    bgp_convergence = bgp.verify_bgp_convergence(tgen, topo)
    assert bgp_convergence
```

CHAPTER

NINE

BGPD

9.1 Next Hop Tracking

Next hop tracking is an optimization feature that reduces the processing time involved in the BGP bestpath algorithm by monitoring changes to the routing table.

9.1.1 Background

Recursive routes are of the form:

```
p/m --> n
[Ex: 1.1.0.0/16 --> 2.2.2.2]
```

where 'n' itself is resolved through another route as follows:

```
p2/m --> h, interface
[Ex: 2.2.2.0/24 --> 3.3.3.3, eth0]
```

Usually, BGP routes are recursive in nature and BGP nexthops get resolved through an IGP route. IGP usually adds its routes pointing to an interface (these are called non-recursive routes).

When BGP receives a recursive route from a peer, it needs to validate the nexthop. The path is marked valid or invalid based on the reachability status of the nexthop. Nexthop validation is also important for BGP decision process as the metric to reach the nexthop is a parameter to best path selection process.

As it goes with routing, this is a dynamic process. Route to the nexthop can change. The nexthop can become unreachable or reachable. In the current BGP implementation, the nexthop validation is done periodically in the scanner run. The default scanner run interval is one minute. Every minute, the scanner task walks the entire BGP table. It checks the validity of each nexthop with Zebra (the routing table manager) through a request and response message exchange between BGP and Zebra process. BGP process is blocked for that duration. The mechanism has two major drawbacks:

- The scanner task runs to completion. That can potentially starve the other tasks for long periods of time, based on the BGP table size and number of nexthops.
- Convergence around routing changes that affect the nexthops can be long (around a minute with the default intervals). The interval can be shortened to achieve faster reaction time, but it makes the first problem worse, with the scanner task consuming most of the CPU resources.

The next-hop tracking feature makes this process event-driven. It eliminates periodic nexthop validation and introduces an asynchronous communication path between BGP and Zebra for route change notifications that can then be acted upon.

9.1.2 Goal

Stating the obvious, the main goal is to remove the two limitations we discussed in the previous section. The goals, in a constructive tone, are the following:

- **Fairness**: the scanner run should not consume an unjustly high amount of CPU time. This should give an overall good performance and response time to other events (route changes, session events, IO/user interface).
- **Convergence**: BGP must react to nexthop changes instantly and provide sub-second convergence. This may involve diverting the routes from one nexthop to another.

9.1.3 Overview of changes

The changes are in both BGP and Zebra modules. The short summary is the following:

- Zebra implements a registration mechanism by which clients can register for next hop notification. Consequently, it maintains a separate table, per (VRF, AF) pair, of next hops and interested client-list per next hop.
- When the main routing table changes in Zebra, it evaluates the next hop table: for each next hop, it checks if the route table modifications have changed its state. If so, it notifies the interested clients.
- BGP is one such client. It registers the next hops corresponding to all of its received routes/paths. It also threads the paths against each nexthop structure.
- When BGP receives a next hop notification from Zebra, it walks the corresponding path list. It makes them valid or invalid depending on the next hop notification. It then re-computes best path for the corresponding destination. This may result in re-announcing those destinations to peers.

9.1.4 Design

Modules

The core design introduces an "nht" (next hop tracking) module in BGP and "rnh" (recursive nexthop) module in Zebra. The "nht" module provides the following APIs:

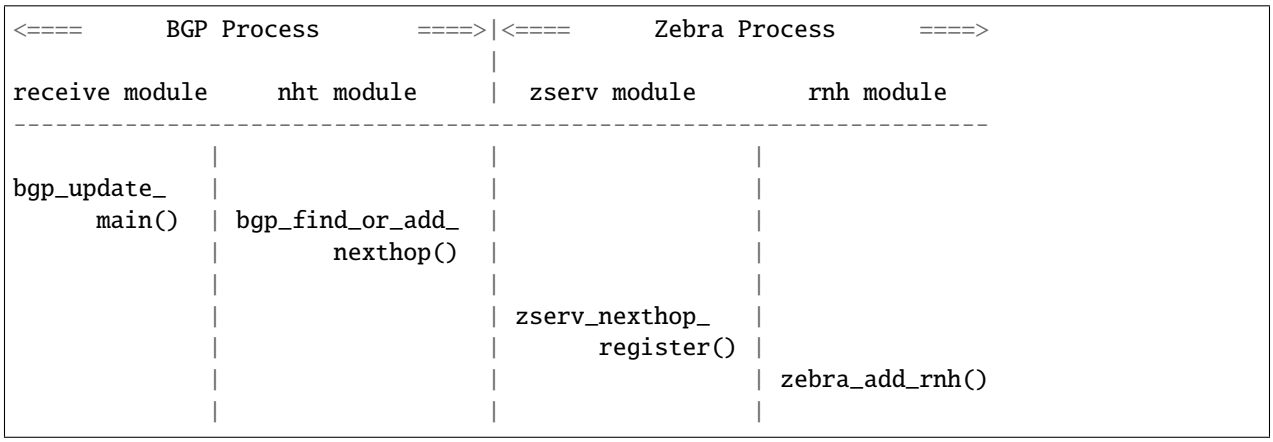
Function	Action
bgp_find_or_add_nexthop()	find or add a nexthop in BGP nexthop table
bgp_parse_nexthop_update()	parse a nexthop update message coming from zebra

The "rnh" module provides the following APIs:

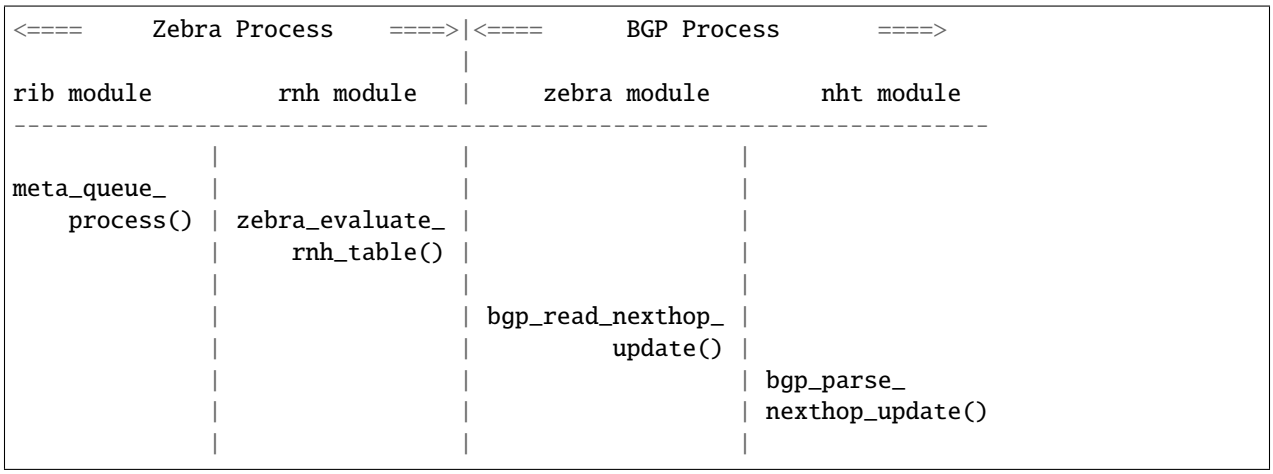
Function	Action
zebra_add_rnh()	add a recursive nexthop
zebra_delete_rnh()	delete a recursive nexthop
zebra_lookup_rnh()	lookup a recursive nexthop
zebra_add_rnh_client()	register a client for nexthop notifications against a recursive nexthop
zebra_remove_rnh_client()	remove the client registration for a recursive nexthop
zebra_evaluate_rnh_table()	(re)evaluate the recursive nexthop table (most probably because the main routing table has changed).
zebra_cleanup_rnh_client()	Cleanup a client from the "rnh" module data structures (most probably because the client is going away).

4.2. Control flow

The next hop registration control flow is the following:

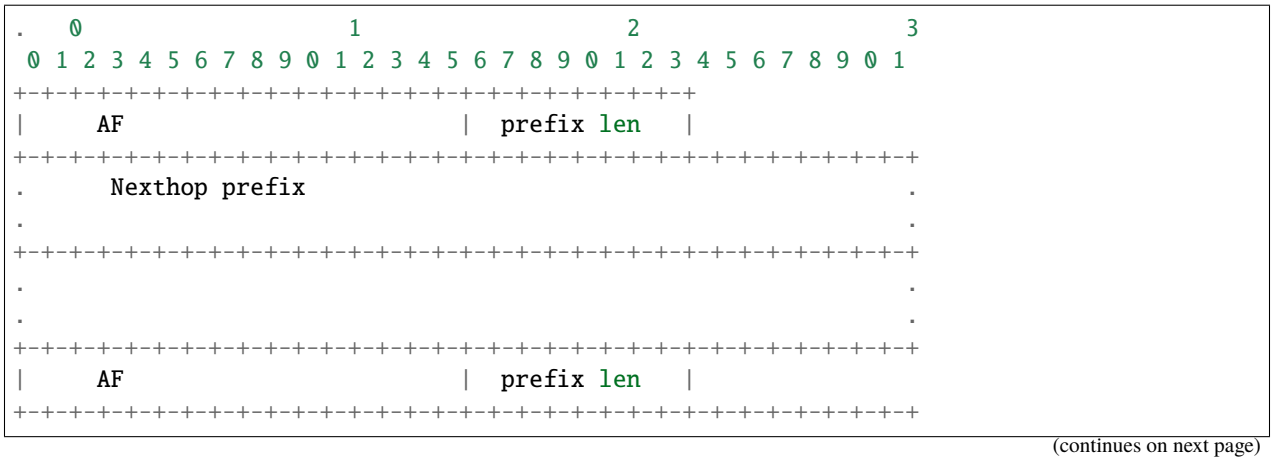


The next hop notification control flow is the following:



zclient message format

ZEBRA_NEXTHOP_REGISTER and ZEBRA_NEXTHOP_UNREGISTER messages are encoded in the following way:



(continues on next page)

(continued from previous page)

```
.      Nexthop prefix                                           .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

ZEBRA_NEXTHOP_UPDATE message is encoded as follows:

```
.   0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     AF                        |  prefix len   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.      Nexthop prefix getting resolved                          .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        metric                                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  #nexthops    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| nexthop type  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.      resolving Nexthop details                                .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| nexthop type  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.      resolving Nexthop details                                .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

BGP data structure

Legend:

```
/\   struct bgp_node: a BGP destination/route/prefix
\/

[ ]  struct bgp_path_info: a BGP path (e.g. route received from a peer)


 _
(_)  struct bgp_nexthop_cache: a BGP nexthop

/\          NULL
\/--+         ^
    |         :
    +--[ ]--[ ]--[ ]--> NULL
/\            :
\/--+         :
    |         :
    +--[ ]--[ ]--> NULL
              :
```

(continues on next page)

(continued from previous page)

```
 _           :
(_)..........
```

Zebra data structure

RNH table:

```
.  O
  / \
 O   O
    / \
   O   O

struct rnh
{
  uint8_t flags;
  struct route_entry *state;
  struct list *client_list;
  struct route_node *node;
};
```

User interface changes

```
frr# show ip nht
3.3.3.3
 resolved via kernel
 via 11.0.0.6, swp1
 Client list: bgp(fd 12)
11.0.0.10
 resolved via connected
 is directly connected, swp2
 Client list: bgp(fd 12)
11.0.0.18
 resolved via connected
 is directly connected, swp4
 Client list: bgp(fd 12)
11.11.11.11
 resolved via kernel
 via 10.0.1.2, eth0
 Client list: bgp(fd 12)

frr# show ip bgp nexthop
Current BGP nexthop cache:
 3.3.3.3 valid [IGP metric 0], #paths 3
  Last update: Wed Oct 16 04:43:49 2013

 11.0.0.10 valid [IGP metric 1], #paths 1
  Last update: Wed Oct 16 04:43:51 2013
```

(continues on next page)

(continued from previous page)

```
 11.0.0.18 valid [IGP metric 1], #paths 2
  Last update: Wed Oct 16 04:43:47 2013

 11.11.11.11 valid [IGP metric 0], #paths 1
  Last update: Wed Oct 16 04:43:47 2013

frr# show ipv6 nht
frr# show ip bgp nexthop detail

frr# debug bgp nht
frr# debug zebra nht

6. Sample test cases

     r2----r3
    /  \  /
  r1----r4

- Verify that a change in IGP cost triggers NHT
  + shutdown the r1-r4 and r2-r4 links
  + no shut the r1-r4 and r2-r4 links and wait for OSPF to come back
    up
  + We should be back to the original nexthop via r4 now
- Verify that a NH becoming unreachable triggers NHT
  + Shutdown all links to r4
- Verify that a NH becoming reachable triggers NHT
  + no shut all links to r4
```

Future work

- route-policy for next hop validation (e.g. ignore default route)
- damping for rapid next hop changes
- prioritized handling of nexthop changes ((un)reachability vs. metric changes)
- handling recursion loop, e.g:

```
11.11.11.11/32 -> 12.12.12.12
12.12.12.12/32 -> 11.11.11.11
11.0.0.0/8 -> <interface>
```

- better statistics

9.2 BGP-4[+] UPDATE Attribute Preprocessor Constants

This is a list of preprocessor constants that map to BGP attributes defined by various BGP RFCs. In the code these are defined as BGP_ATTR_<ATTR>.

Value	Attribute	References
1 2 3 4	ORIGIN AS_PATH NEXT_HOP MULTI_EXIT_DISC	[RFC 4271] [RFC 4271] [RFC 4271]
5 6 7 8	LOCAL_PREF ATOMIC_AGGREGATE AGGRE-	[RFC 4271] [RFC 4271] [RFC 4271]
9 10 14	GATOR COMMUNITIES ORIGINATOR_ID CLUS-	[RFC 4271] [RFC 1997] [RFC 4456]
15 16	TER_LIST MP_REACH_NLRI MP_UNREACH_NLRI	[RFC 4456] [RFC 4760] [RFC 4760]
17 18	EXT_COMMUNITIES AS4_PATH AS4_AGGREGATOR	[RFC 4360] [RFC 4893] [RFC 4893]

CHAPTER

TEN

FPM

FPM stands for Forwarding Plane Manager and it's a module for use with Zebra.

The encapsulation header for the messages exchanged with the FPM is defined by the file `fpm/fpm.h` in the frr tree. The routes themselves are encoded in Netlink or protobuf format, with Netlink being the default.

Netlink is standard format for encoding messages to talk with kernel space in Linux and it is also the name of the socket type used by it. The FPM netlink usage differs from Linux's in:

- Linux netlink sockets use datagrams in a multicast fashion, FPM uses as a stream and it is unicast.
- FPM netlink messages might have more or less information than a normal Linux netlink socket message (example: RTM_NEWROUTE might add an extra route attribute to signalize VxLAN encapsulation).

Protobuf is one of a number of new serialization formats wherein the message schema is expressed in a purpose-built language. Code for encoding/decoding to/from the wire format is generated from the schema. Protobuf messages can be extended easily while maintaining backward-compatibility with older code. Protobuf has the following advantages over Netlink:

- Code for serialization/deserialization is generated automatically. This reduces the likelihood of bugs, allows third-party programs to be integrated quickly, and makes it easy to add fields.
- The message format is not tied to an OS (Linux), and can be evolved independently.

Note: Currently there are two FPM modules in `zebra`:

- `fpm`
- `dplane_fpm_nl`

10.1 fpm

The first FPM implementation that was built using hooks in `zebra` route handling functions. It uses its own netlink/protobuf encoding functions to translate `zebra` route data structures into formatted binary data.

10.2 dplane_fpm_nl

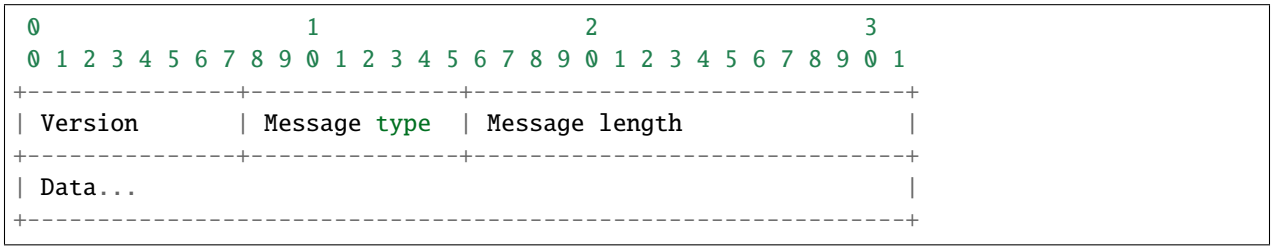
The newer FPM implementation that was built using `zebra`'s data plane framework as a plugin. It only supports netlink and it shares `zebra`'s netlink functions to translate route event snapshots into formatted binary data.

10.2.1 Protocol Specification

FPM (in any mode) uses a TCP connection to talk with external applications. It operates as TCP client and uses the CLI configured address/port to connect to the FPM server (defaults to port `2620`).

FPM frames all data with a header to help the external reader figure how many bytes it has to read in order to read the full message (this helps simulates datagrams like in the original netlink Linux kernel usage).

Frame header:



10.3 Version

Currently there is only one version, so it should be always `1`.

10.4 Message Type

Defines what underlining protocol we are using: netlink (`1`) or protobuf (`2`).

10.5 Message Length

Amount of data in this frame in network byte order.

10.6 Data

The netlink or protobuf message payload.

10.7 Route Status Notification from ASIC

The dplane_fpm_nl has the ability to read route netlink messages from the underlying fpm implementation that can tell zebra whether or not the route has been Offloaded/Failed or Trapped. The end developer must send the data up the same socket that has been created to listen for FPM messages from Zebra. The data sent must have a Frame Header with Version set to 1, Message Type set to 1 and an appropriate message Length. The message data must contain a RTM_NEWROUTE netlink message that sends the prefix and nexthops associated with the route. Finally rtm_flags must contain RTM_F_OFFLOAD, RTM_F_TRAP and or RTM_F_OFFLOAD_FAILED to signify what has happened to the route in the ASIC.

CHAPTER

ELEVEN

NORTHBOUND GRPC

To enable gRPC support one needs to add *–enable-grpc* when running *configure*. Additionally, when launching each daemon one needs to request the gRPC module be loaded and which port to bind to. This can be done by adding *-M grpc:<port>* to the daemon's CLI arguments.

Currently there is no gRPC "routing" so you will need to bind your gRPC *channel* to the particular daemon's gRPC port to interact with that daemon's gRPC northbound interface.

The minimum version of gRPC known to work is 1.16.1.

11.1 Programming Language Bindings

The gRPC supported programming language bindings can be found here: https://grpc.io/docs/languages/

After picking a programming language that supports gRPC bindings, the next step is to generate the FRR northbound bindings. To generate the northbound bindings you'll need the programming language binding generator tools and those are language specific.

11.1.1 C++ Example

The next sections will use C++ as an example for accessing FRR northbound through gRPC.

Generating C++ FRR Bindings

Generating FRR northbound bindings for C++ example:

```
# Install gRPC (e.g., on Ubuntu 20.04)
sudo apt-get install libgrpc++-dev libgrpc-dev

mkdir /tmp/frr-cpp
cd grpc

protoc --cpp_out=/tmp/frr-cpp \
       --grpc_out=/tmp/frr-cpp \
       -I $(pwd) \
       --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` \
        frr-northbound.proto
```

Using C++ To Get Version and Interfaces State

Below is a sample program to print all interfaces discovered.

```
# test.cpp
#include <string>
#include <sstream>
#include <grpc/grpc.h>
#include <grpcpp/create_channel.h>
#include "frr-northbound.pb.h"
#include "frr-northbound.grpc.pb.h"

int main() {
    frr::GetRequest request;
    frr::GetResponse reply;
    grpc::ClientContext context;
    grpc::Status status;

    auto channel = grpc::CreateChannel("localhost:50051",
                                       grpc::InsecureChannelCredentials());
    auto stub = frr::Northbound::NewStub(channel);

    request.set_type(frr::GetRequest::ALL);
    request.set_encoding(frr::JSON);
    request.set_with_defaults(true);
    request.add_path("/frr-interface:lib");
    auto stream = stub->Get(&context, request);

    std::ostringstream ss;
    while (stream->Read(&reply))
      ss << reply.data().data() << std::endl;

    status = stream->Finish();
    assert(status.ok());
    std::cout << "Interface Info:\n" << ss.str() << std::endl;
}
```

Below is how to compile and run the program, with the example output:

```
$ g++ -o test test.cpp frr-northbound.grpc.pb.cc frr-northbound.pb.cc -lgrpc++ -lprotobuf
$ ./test
Interface Info:
{
  "frr-interface:lib": {
    "interface": [
      {
        "name": "lo",
        "vrf": "default",
        "state": {
          "if-index": 1,
          "mtu": 0,
          "mtu6": 65536,
          "speed": 0,
          "metric": 0,
```

(continues on next page)

(continued from previous page)


```
          "phy-address": "00:00:00:00:00:00"
        },
        "frr-zebra:zebra": {
          "state": {
            "up-count": 0,
            "down-count": 0,
            "ptm-status": "disabled"
          }
        }
      },
      {
        "name": "r1-eth0",
        "vrf": "default",
        "state": {
          "if-index": 2,
          "mtu": 1500,
          "mtu6": 1500,
          "speed": 10000,
          "metric": 0,
          "phy-address": "02:37:ac:63:59:b9"
        },
        "frr-zebra:zebra": {
          "state": {
            "up-count": 0,
            "down-count": 0,
            "ptm-status": "disabled"
          }
        }
      }
    ]
  },
  "frr-zebra:zebra": {
    "mcast-rpf-lookup": "mrib-then-urib",
    "workqueue-hold-timer": 10,
    "zapi-packets": 1000,
    "import-kernel-table": {
      "distance": 15
    },
    "dplane-queue-limit": 200
  }
}
```

11.1.2 Python Example

The next sections will use Python as an example for writing scripts to use the northbound.

Generating Python FRR Bindings

Generating FRR northbound bindings for Python example:

```
# Install python3 virtual environment capability e.g.,
sudo apt-get install python3-venv

# Create a virtual environment for python grpc and activate
python3 -m venv venv-grpc
source venv-grpc/bin/activate

# Install grpc requirements
pip install grpcio grpcio-tools

mkdir /tmp/frr-python
cd grpc

python3 -m grpc_tools.protoc  \
        --python_out=/tmp/frr-python \
        --grpc_python_out=/tmp/frr-python \
        -I $(pwd) \
        frr-northbound.proto
```

Using Python To Get Capabilities and Interfaces State

Below is a sample script to print capabilities and all interfaces Python discovered. This demostrates the 2 different RPC results one gets from gRPC, Unary (*GetCapabilities*) and Streaming (*Get*) for the interface state.

```
import grpc
import frr_northbound_pb2
import frr_northbound_pb2_grpc

channel = grpc.insecure_channel('localhost:50051')
stub = frr_northbound_pb2_grpc.NorthboundStub(channel)

# Print Capabilities
request = frr_northbound_pb2.GetCapabilitiesRequest()
response = stub.GetCapabilities(request)
print(response)

# Print Interface State and Config
request = frr_northbound_pb2.GetRequest()
request.path.append("/frr-interface:lib")
request.type=frr_northbound_pb2.GetRequest.ALL
request.encoding=frr_northbound_pb2.XML

for r in stub.Get(request):
    print(r.data.data)
```

The previous script will output something like:

```
frr_version: "7.7-dev-my-manual-build"
rollback_support: true
supported_modules {
  name: "frr-filter"
  organization: "FRRouting"
  revision: "2019-07-04"
}
supported_modules {
  name: "frr-interface"
  organization: "FRRouting"
  revision: "2020-02-05"
}
[...]
supported_encodings: JSON
supported_encodings: XML

<lib xmlns="http://frrouting.org/yang/interface">
  <interface>
    <name>lo</name>
    <vrf>default</vrf>
    <state>
      <if-index>1</if-index>
      <mtu>0</mtu>
      <mtu6>65536</mtu6>
      <speed>0</speed>
      <metric>0</metric>
      <phy-address>00:00:00:00:00:00</phy-address>
    </state>
    <zebra xmlns="http://frrouting.org/yang/zebra">
      <state>
        <up-count>0</up-count>
        <down-count>0</down-count>
      </state>
    </zebra>
  </interface>
  <interface>
    <name>r1-eth0</name>
    <vrf>default</vrf>
    <state>
      <if-index>2</if-index>
      <mtu>1500</mtu>
      <mtu6>1500</mtu6>
      <speed>10000</speed>
      <metric>0</metric>
      <phy-address>f2:62:2e:f3:4c:e4</phy-address>
    </state>
    <zebra xmlns="http://frrouting.org/yang/zebra">
      <state>
        <up-count>0</up-count>
        <down-count>0</down-count>
      </state>
```

(continues on next page)

(continued from previous page)

```
    </zebra>
  </interface>
</lib>
```

11.1.3 Ruby Example

Next sections will use Ruby as an example for writing scripts to use the northbound.

Generating Ruby FRR Bindings

Generating FRR northbound bindings for Ruby example:

```
# Install the required gems:
# - grpc: the gem that will talk with FRR's gRPC plugin.
# - grpc-tools: the gem that provides the code generator.
gem install grpc
gem install grpc-tools

# Create your project/scripts directory:
mkdir /tmp/frr-ruby

# Go to FRR's grpc directory:
cd grpc

# Generate the ruby bindings:
grpc_tools_ruby_protoc \
  --ruby_out=/tmp/frr-ruby \
  --grpc_out=/tmp/frr-ruby \
  frr-northbound.proto
```

Using Ruby To Get Interfaces State

Here is a sample script to print all interfaces FRR discovered:

```
require 'frr-northbound_services_pb'

# Create the connection with FRR's gRPC:
stub = Frr::Northbound::Stub.new('localhost:50051', :this_channel_is_insecure)

# Create a new state request to get interface state:
request = Frr::GetRequest.new
request.type = :STATE
request.path.push('/frr-interface:lib')

# Ask FRR.
response = stub.get(request)

# Print the response.
response.each do |result|
```

(continues on next page)

(continued from previous page)

```
  result.data.data.each_line do |line|
    puts line
  end
end
```

Note: The generated files will assume that they are in the search path (e.g. inside gem) so you'll need to either edit it to use `require_relative` or tell Ruby where to look for them. For simplicity we'll use `-I .` to tell it is in the current directory.

The previous script will output something like this:

```
$ cd /tmp/frr-ruby
# Add `-I.` so ruby finds the FRR generated file locally.
$ ruby -I. interface.rb
{
  "frr-interface:lib": {
    "interface": [
      {
        "name": "eth0",
        "vrf": "default",
        "state": {
          "if-index": 2,
          "mtu": 1500,
          "mtu6": 1500,
          "speed": 1000,
          "metric": 0,
          "phy-address": "11:22:33:44:55:66"
        },
        "frr-zebra:zebra": {
          "state": {
            "up-count": 0,
            "down-count": 0
          }
        }
      },
      {
        "name": "lo",
        "vrf": "default",
        "state": {
          "if-index": 1,
          "mtu": 0,
          "mtu6": 65536,
          "speed": 0,
          "metric": 0,
          "phy-address": "00:00:00:00:00:00"
        },
        "frr-zebra:zebra": {
          "state": {
            "up-count": 0,
            "down-count": 0
          }
```

(continues on next page)

(continued from previous page)

```
        }
      }
    ]
  }
}
```

Using Ruby To Create BFD Profiles

In this example you'll learn how to edit configuration using JSON and programmatic (XPath) format.

```
require 'frr-northbound_services_pb'

# Create the connection with FRR's gRPC:
stub = Frr::Northbound::Stub.new('localhost:50051', :this_channel_is_insecure)

# Create a new candidate configuration change.
new_candidate = stub.create_candidate(Frr::CreateCandidateRequest.new)

# Use JSON to configure.
request = Frr::LoadToCandidateRequest.new
request.candidate_id = new_candidate.candidate_id
request.type = :MERGE
request.config = Frr::DataTree.new
request.config.encoding = :JSON
request.config.data = <<-EOJ
{
  "frr-bfdd:bfdd": {
    "bfd": {
      "profile": [
        {
          "name": "test-prof",
          "detection-multiplier": 4,
          "required-receive-interval": 800000
        }
      ]
    }
  }
}
EOJ

# Load configuration to candidate.
stub.load_to_candidate(request)

# Commit candidate.
stub.commit(
  Frr::CommitRequest.new(
    candidate_id: new_candidate.candidate_id,
    phase: :ALL,
    comment: 'create test-prof'
  )
)
```

(continues on next page)

(continued from previous page)

```
#
# Now lets delete the previous profile and create a new one.
#

# Create a new candidate configuration change.
new_candidate = stub.create_candidate(Frr::CreateCandidateRequest.new)

# Edit the configuration candidate.
request = Frr::EditCandidateRequest.new
request.candidate_id = new_candidate.candidate_id

# Delete previously created profile.
request.delete.push(
  Frr::PathValue.new(
    path: "/frr-bfdd:bfdd/bfd/profile[name='test-prof']",
  )
)

# Add new profile with two configurations.
request.update.push(
  Frr::PathValue.new(
    path: "/frr-bfdd:bfdd/bfd/profile[name='test-prof-2']/detection-multiplier",
    value: 5.to_s
  )
)
request.update.push(
  Frr::PathValue.new(
    path: "/frr-bfdd:bfdd/bfd/profile[name='test-prof-2']/desired-transmission-interval",
    value: 900_000.to_s
  )
)

# Modify the candidate.
stub.edit_candidate(request)

# Commit the candidate configuration.
stub.commit(
  Frr::CommitRequest.new(
    candidate_id: new_candidate.candidate_id,
    phase: :ALL,
    comment: 'replace test-prof with test-prof-2'
  )
)
```

And here is the new FRR configuration:

```
$ sudo vtysh -c 'show running-config'
...
bfd
 profile test-prof-2
  detect-multiplier 5
```

(continues on next page)

(continued from previous page)

```
  transmit-interval 900
 !
!
```

CHAPTER

TWELVE

OSPFD

12.1 OSPF API Documentation

12.1.1 Disclaimer

The OSPF daemon contains an API for application access to the LSA database. This API and documentation was created by Ralph Keller, originally as patch for Zebra. Unfortunately, the page containing documentation for the API is no longer online. This page is an attempt to recreate documentation for the API (with lots of help from the WayBack-Machine).

Ralph has kindly licensed this documentation under GPLv2+. Please preserve the acknowledgements at the bottom of this document.

12.1.2 Introduction

This page describes an API that allows external applications to access the link-state database (LSDB) of the OSPF daemon. The implementation is based on the OSPF code from FRRouting (forked from Quagga and formerly Zebra) routing protocol suite and is subject to the GNU General Public License. The OSPF API provides you with the following functionality:

- Retrieval of the full or partial link-state database of the OSPF daemon. This allows applications to obtain an exact copy of the LSDB including router LSAs, network LSAs and so on. Whenever a new LSA arrives at the OSPF daemon, the API module immediately informs the application by sending a message. This way, the application is always synchronized with the LSDB of the OSPF daemon.
- Origination of own opaque LSAs (of type 9, 10, or 11) which are then distributed transparently to other routers within the flooding scope and received by other applications through the OSPF API.

Opaque LSAs, which are described in **RFC 2370**, allow you to distribute application-specific information within a network using the OSPF protocol. The information contained in opaque LSAs is transparent for the routing process but it can be processed by other modules such as traffic engineering (e.g., MPLS-TE).

12.1.3 Architecture

The following picture depicts the architecture of the Quagga/Zebra protocol suite. The OSPF daemon is extended with opaque LSA capabilities and an API for external applications. The OSPF core module executes the OSPF protocol by discovering neighbors and exchanging neighbor state. The opaque module, implemented by Masahiko Endo, provides functions to exchange opaque LSAs between routers. Opaque LSAs can be generated by several modules such as the MPLS-TE module or the API server module. These modules then invoke the opaque module to flood their data to neighbors within the flooding scope.

The client, which is an application potentially running on a different node than the OSPF daemon, links against the OSPF API client library. This client library establishes a socket connection with the API server module of the OSPF daemon and uses this connection to retrieve LSAs and originate opaque LSAs.

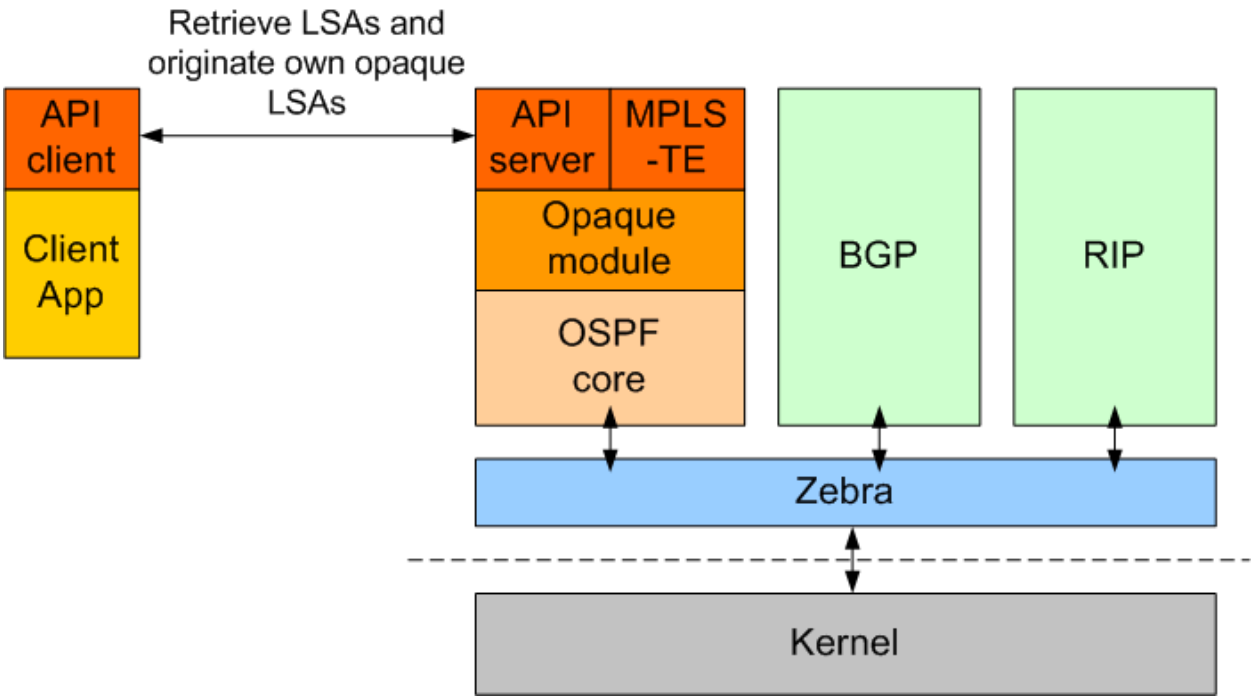


Fig. 1: image

The OSPF API server module works like any other internal opaque module (such as the MPLS-TE module), but listens to connections from external applications that want to communicate with the OSPF daemon. The API server module can handle multiple clients concurrently.

One of the main objectives of the implementation is to make as little changes to the existing Zebra code as possible.

12.1.4 Installation & Configuration

Download FRRouting and unpack it.

Configure and build FRR (note that `--enable-opaque-lsa` also enables the ospfapi server and ospfclient).

```
% sh ./configure --enable-opaque-lsa
% make
```

This should also compile the client library and sample application in ospfclient.

Make sure that you have enabled opaque LSAs in your configuration. Add the `ospf opaque-lsa` statement to your `ospfd.conf`:

```
! -*- ospf -*-
!
! OSPFd sample configuration file
!
!
hostname xxxxx
password xxxxx

router ospf
  router-id 10.0.0.1
  network 10.0.0.1/24 area 1
  neighbor 10.0.0.2
  network 10.0.1.2/24 area 1
  neighbor 10.0.1.1
  ospf opaque-lsa      <============ add this statement!
```

12.1.5 Usage

In the following we describe how you can use the sample application to originate opaque LSAs. The sample application first registers with the OSPF daemon the opaque type it wants to inject and then waits until the OSPF daemon is ready to accept opaque LSAs of that type. Then the client application originates an opaque LSA, waits 10 seconds and then updates the opaque LSA with new opaque data. After another 20 seconds, the client application deletes the opaque LSA from the LSDB. If the clients terminates unexpectedly, the OSPF API module will remove all the opaque LSAs that the application registered. Since the opaque LSAs are flooded to other routers, we will see the opaque LSAs in all routers according to the flooding scope of the opaque LSA.

We have a very simple demo setup, just two routers connected with an ATM point-to-point link. Start the modified OSPF daemons on two adjacent routers. First run on msr2:

```
# ./ospfd --apiserver -f /usr/local/etc/ospfd.conf
```

And on the neighboring router msr3:

```
# ./ospfd --apiserver -f /usr/local/etc/ospfd.conf
```

Now the two routers form adjacency and start exchanging their databases. Looking at the OSPF daemon of msr2 (or msr3), you see this:

```
ospfd> show ip ospf database

       OSPF Router with ID (10.0.0.1)

                Router Link States (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum  Link count
10.0.0.1        10.0.0.1          55 0x80000003 0xc62f 2
10.0.0.2        10.0.0.2          55 0x80000003 0xe3e4 3

                Net Link States (Area 0.0.0.1)
```

(continues on next page)

(continued from previous page)

```
Link ID         ADV Router      Age  Seq#       CkSum
10.0.0.2        10.0.0.2          60 0x80000001 0x5fcb
```

Now we start the sample main application that originates an opaque LSA.

```
# cd ospfapi/apiclient
# ./main msr2 10 250 20 0.0.0.0 0.0.0.1
```

This originates an opaque LSA of type 10 (area local), with opaque type 250 (experimental), opaque id of 20 (chosen arbitrarily), interface address 0.0.0.0 (which is used only for opaque LSAs type 9), and area 0.0.0.1

Again looking at the OSPF database you see:

```
ospfd> show ip ospf database

       OSPF Router with ID (10.0.0.1)

                Router Link States (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum  Link count
10.0.0.1        10.0.0.1         437 0x80000003 0xc62f 2
10.0.0.2        10.0.0.2         437 0x80000003 0xe3e4 3

                Net Link States (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum
10.0.0.2        10.0.0.2         442 0x80000001 0x5fcb

                Area-Local Opaque-LSA (Area 0.0.0.1)

Opaque-Type/Id  ADV Router      Age  Seq#       CkSum
250.0.0.20      10.0.0.1           0 0x80000001 0x58a6  <=== opaque LSA
```

You can take a closer look at this opaque LSA:

```
ospfd> show ip ospf database opaque-area

       OSPF Router with ID (10.0.0.1)


                Area-Local Opaque-LSA (Area 0.0.0.1)

LS age: 4
Options: 66
LS Type: Area-Local Opaque-LSA
Link State ID: 250.0.0.20 (Area-Local Opaque-Type/ID)
Advertising Router: 10.0.0.1
LS Seq Number: 80000001
Checksum: 0x58a6
Length: 24
Opaque-Type 250 (Private/Experimental)
Opaque-ID 0x14
Opaque-Info: 4 octets of data
```

(continues on next page)

(continued from previous page)

```
Added using OSPF API: 4 octets of opaque data
Opaque data: 1 0 0 0 <==== counter is 1
```

Note that the main application updates the opaque LSA after 10 seconds, then it looks as follows:

```
ospfd> show ip ospf database opaque-area

       OSPF Router with ID (10.0.0.1)


                Area-Local Opaque-LSA (Area 0.0.0.1)

LS age: 1
Options: 66
LS Type: Area-Local Opaque-LSA
Link State ID: 250.0.0.20 (Area-Local Opaque-Type/ID)
Advertising Router: 10.0.0.1
LS Seq Number: 80000002
Checksum: 0x59a3
Length: 24
Opaque-Type 250 (Private/Experimental)
Opaque-ID   0x14
Opaque-Info: 4 octets of data
Added using OSPF API: 4 octets of opaque data
Opaque data: 2 0 0 0  <==== counter is now 2
```

Note that the payload of the opaque LSA has changed as you can see above.

Then, again after another 20 seconds, the opaque LSA is flushed from the LSDB.

Important note:

In order to originate an opaque LSA, there must be at least one active opaque-capable neighbor. Thus, you cannot originate opaque LSAs if no neighbors are present. If you try to originate when no neighbors are ready, you will receive a not ready error message. The reason for this restriction is that it might be possible that some routers have an identical opaque LSA from a previous origination in their LSDB that unfortunately could not be flushed due to a crash, and now if the router comes up again and starts originating a new opaque LSA, the new opaque LSA is considered older since it has a lower sequence number and is ignored by other routers (that consider the stalled opaque LSA as more recent). However, if the originating router first synchronizes the database before originating opaque LSAs, it will detect the older opaque LSA and can flush it first.

12.1.6 Protocol and Message Formats

If you are developing your own client application and you don't want to make use of the client library (due to the GNU license restriction or whatever reason), you can implement your own client-side message handling. The OSPF API uses two connections between the client and the OSPF API server: One connection is used for a synchronous request /reply protocol and another connection is used for asynchronous notifications (e.g., LSA update, neighbor status change).

Each message begins with the following header:

The message type field can take one of the following values:

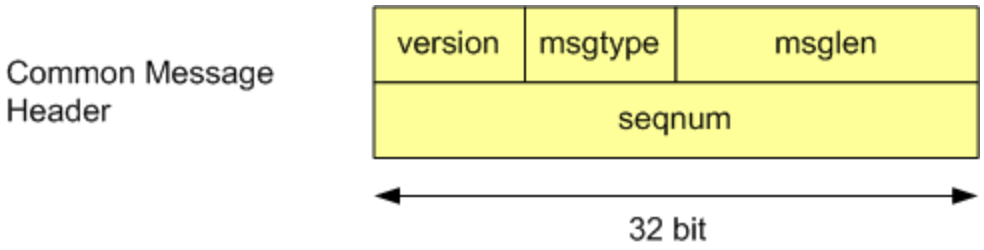


Fig. 2: image

Messages to OSPF daemon	Value
MSG_REGISTER_OPAQUETYPE	1
MSG_UNREGISTER_OPAQUETYPE	2
MSG_REGISTER_EVENT	3
MSG_SYNC_LSDB	4
MSG_ORIGINATE_REQUEST	5
MSG_DELETE_REQUEST	6

Messages from OSPF daemon	Value
MSG_REPLY	10
MSG_READY_NOTIFY	11
MSG_LSA_UPDATE_NOTIFY	12
MSG_LSA_DELETE_NOTIFY	13
MSG_NEW_IF	14
MSG_DEL_IF	15
MSG_ISM_CHANGE	16
MSG_NSM_CHANGE	17

The synchronous requests and replies have the following message formats:

The origin field allows origin-based filtering using the following origin types:

Origin	Value
NON_SELF_ORIGINATED	0
SELF_ORIGINATED	1
ANY_ORIGIN	2

The reply message has one of the following error codes:

Error code	Value
API_OK	0
API_NOSUCHINTERFACE	-1
API_NOSUCHAREA	-2
API_NOSUCHLSA	-3
API_ILLEGALSATYPE	-4
API_ILLEGALOPAQUETYPE	-5
API_OPAQUETYPEINUSE	-6
API_NOMEMORY	-7
API_ERROR	-99
API_UNDEF	-100

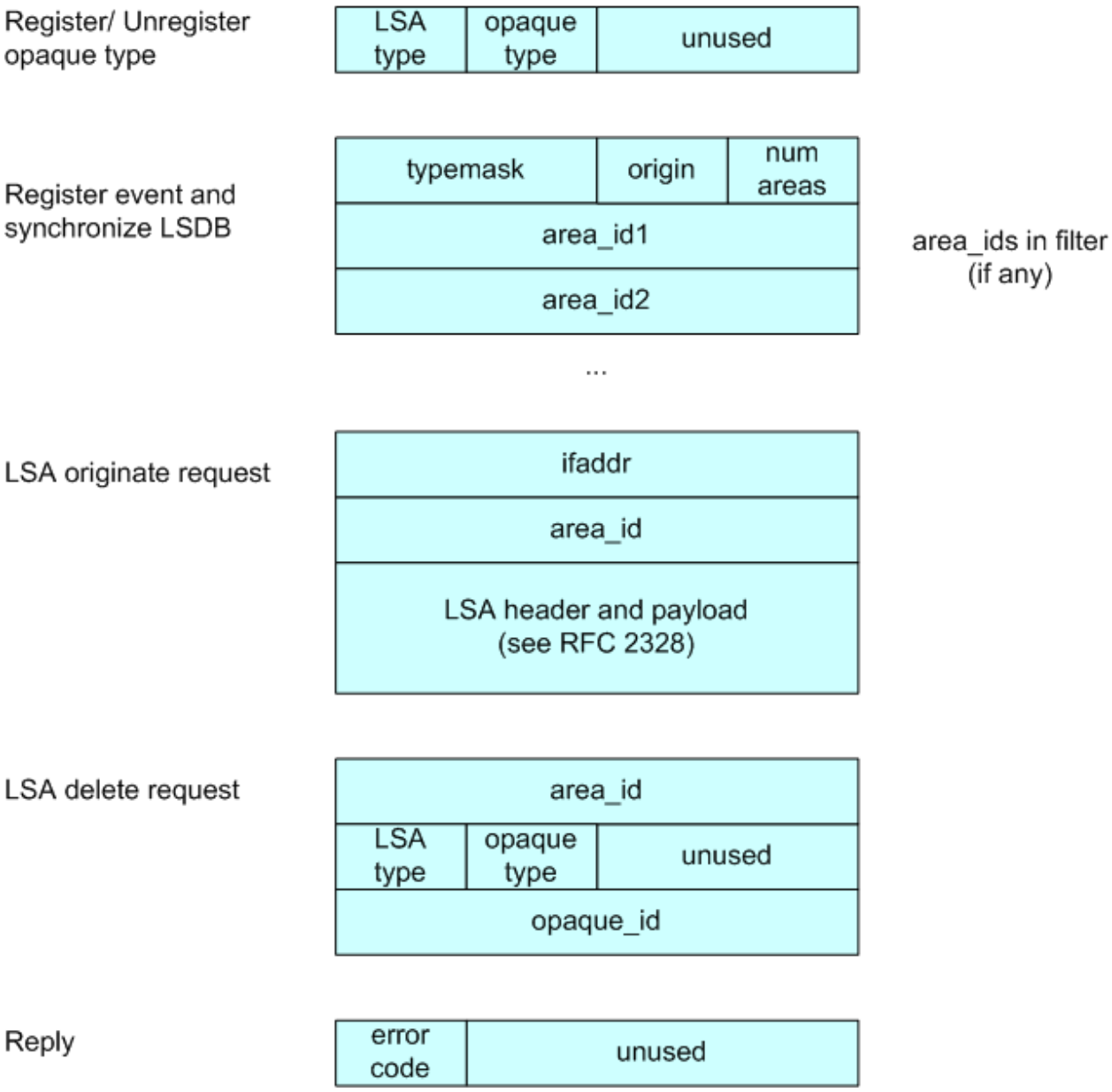

Register/ Unregister opaque type
LSA type
opaque type
unused
Register event and synchronize LSDB
typemask
origin
num areas
area_id1
area_id2
...
area_ids in filter (if any)
LSA originate request
ifaddr
area_id
LSA header and payload (see RFC 2328)
LSA delete request
area_id
LSA type
opaque type
unused
opaque_id
Reply
error code
unused

Fig. 3: image

The asynchronous notifications have the following message formats:

12.1.7 Original Acknowledgments from Ralph Keller

I would like to thank Masahiko Endo, the author of the opaque LSA extension module, for his great support. His wonderful ASCII graphs explaining the internal workings of this code, and his invaluable input proved to be crucial in designing a useful API for accessing the link state database of the OSPF daemon. Once, he even decided to take the plane from Tokyo to Zurich so that we could actually meet and have face-to-face discussions, which was a lot of fun. Clearly, without Masahiko no API would ever be completed. I also would like to thank Daniel Bauer who wrote an opaque LSA implementation too and was willing to test the OSPF API code in one of his projects.

12.2 OSPF Segment Routing

This is an EXPERIMENTAL support of *RFC 8665*. DON'T use it for production network.

12.2.1 Supported Features

- Automatic computation of Primary and Backup Adjacency SID with Cisco experimental remote IP address
- SRGB & SRLB configuration
- Prefix configuration for Node SID with optional NO-PHP flag (Linux kernel support both mode)
- Node MSD configuration (with Linux Kernel >= 4.10 a maximum of 32 labels could be stack)
- Automatic provisioning of MPLS table
- Equal Cost Multi-Path (ECMP)
- Static route configuration with label stack up to 32 labels
- TI-LFA (for P2P interfaces only)

12.2.2 Interoperability

- Tested on various topology including point-to-point and LAN interfaces in a mix of FRRouting instance and Cisco IOS-XR 6.0.x
- Check OSPF LSA conformity with latest wireshark release 2.5.0-rc

12.2.3 Implementation details

Concepts

Segment Routing used 3 different OPAQUE LSA in OSPF to carry the various information:

- **Router Information:** flood the Segment Routing capabilities of the node. This include the supported algorithms, the Segment Routing Global Block (SRGB) and the Maximum Stack Depth (MSD).
- **Extended Link:** flood the Adjaceny and Lan Adjacency Segment Identifier
- **Extended Prefix:** flood the Prefix Segment Identifier

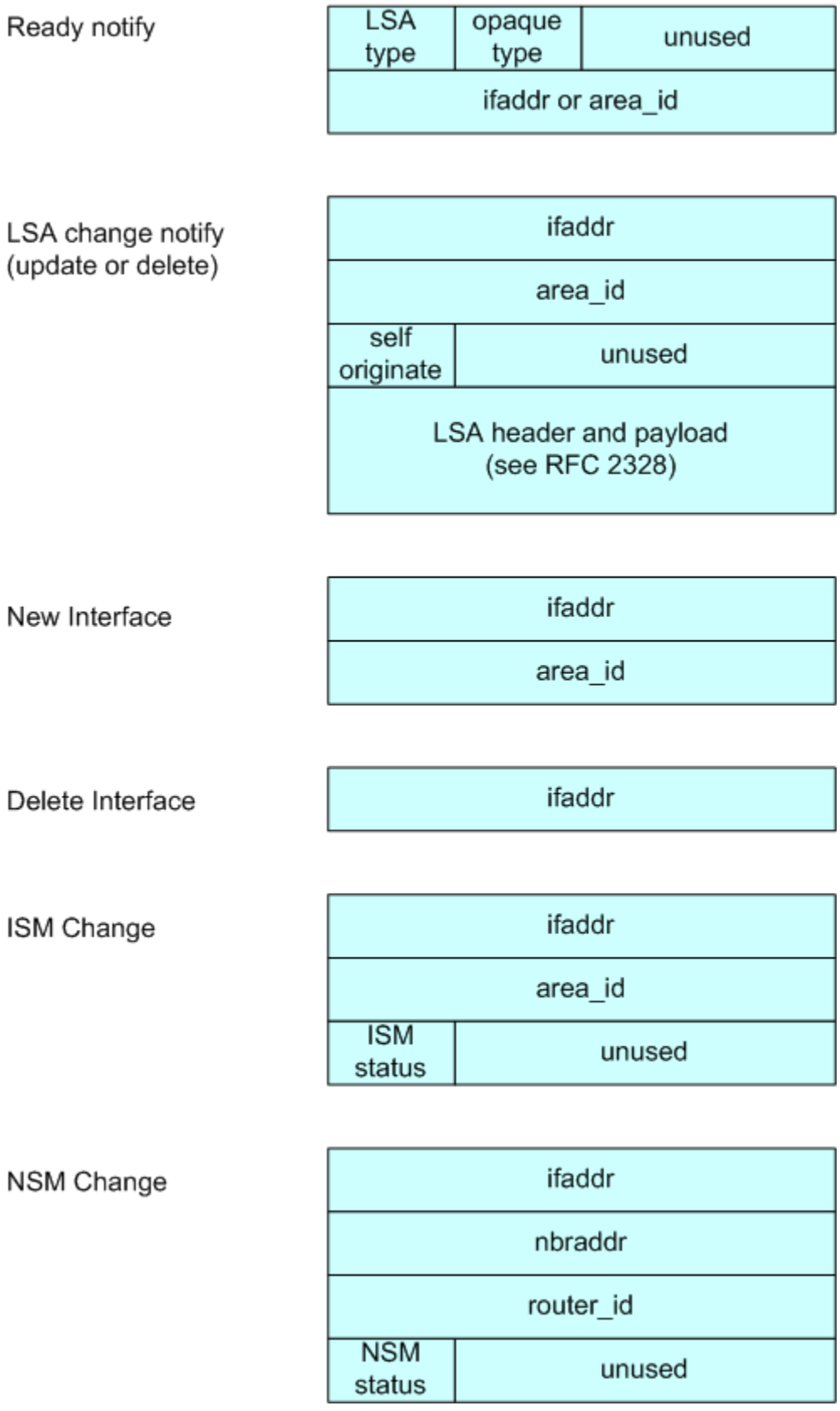

Ready notify
LSA type
opaque type
unused
ifaddr or area_id
LSA change notify (update or delete)
ifaddr
area_id
self originate
unused
LSA header and payload (see RFC 2328)
New Interface
ifaddr
area_id
Delete Interface
ifaddr
ISM Change
ifaddr
area_id
ISM status
unused
NSM Change
ifaddr
nbraddr
router_id
NSM status
unused

Fig. 4: image

The implementation follows previous TE and Router Information codes. It used the OPAQUE LSA functions defined in ospf_opaque.[c,h] as well as the OSPF API. This latter is mandatory for the implementation as it provides the Callback to Segment Routing functions (see below) when an Extended Link / Prefix or Router Information LSA s are received.

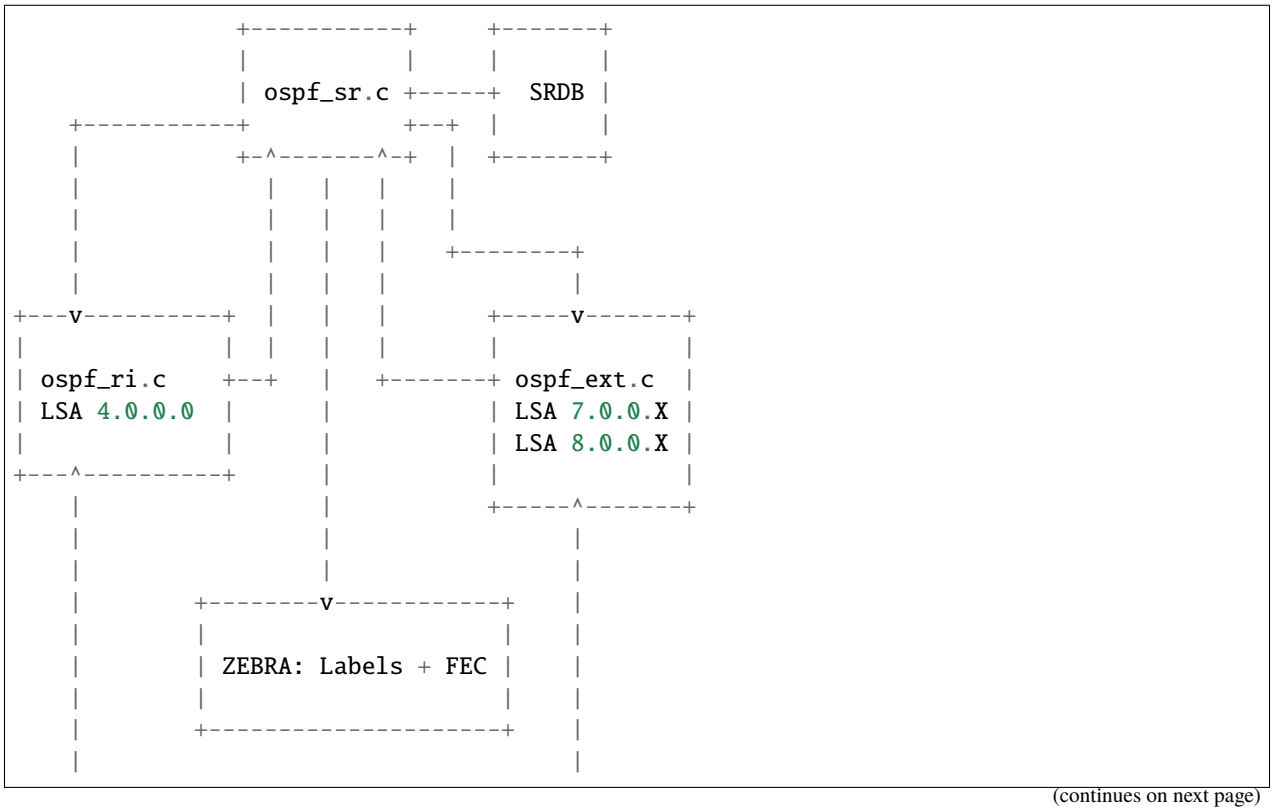
Overview

Following files where modified or added:

- ospd_ri.[c,h] have been modified to add the new TLVs for Segment Routing.
- ospf_ext.[c,h] implement RFC7684 as base support of Extended Link and Prefix Opaque LSA.
- ospf_sr.[c,h] implement the earth of Segment Routing. It adds a new Segment Routing database to manage Segment Identifiers per Link and Prefix and Segment Routing enable node, Callback functions to process incoming LSA and install MPLS FIB entry through Zebra.

The figure below shows the relation between the various files:

- ospf_sr.c centralized all the Segment Routing processing. It receives Opaque LSA Router Information (4.0.0.0) from ospf_ri.c and Extended Prefix (7.0.0.X) Link (8.0.0.X) from ospf_ext.c. Once received, it parse TLVs and SubTLVs and store information in SRDB (which is defined in ospf_sr.h). For each received LSA, NHLFE is computed and send to Zebra to add/remove new MPLS labels entries and FEC. New CLI configurations are also centralized in ospf_sr.c. This CLI will trigger the flooding of new LSA Router Information (4.0.0.0), Extended Prefix (7.0.0.X) and Link (8.0.0.X) by ospf_ri.c, respectively ospf_ext.c.
- ospf_ri.c send back to ospf_sr.c received Router Information LSA and update Self Router Information LSA with parameters provided by ospf_sr.c i.e. SRGB and MSD. It use ospf_opaque.c functions to send/received these Opaque LSAs.
- ospf_ext.c send back to ospf_sr.c received Extended Prefix and Link Opaque LSA and send self Extended Prefix and Link Opaque LSA through ospf_opaque.c functions.



(continues on next page)

(continued from previous page)

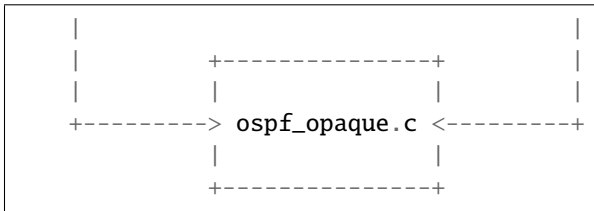


Figure 1: Overview of Segment Routing interaction

Module interactions

To process incoming LSA, the code is based on the capability to call *hook()* functions when LSA are inserted or delete to / from the LSDB and the possibility to register particular treatment for Opaque LSA. The first point is provided by the OSPF API feature and the second by the Opaque implementation itself. Indeed, it is possible to register callback function for a given Opaque LSA ID (see *ospf_register_opaque_functab()* function defined in *ospf_opaque.c*). Each time a new LSA is added to the LSDB, the *new_lsa_hook()* function previously register for this LSA type is called. For Opaque LSA it is the *ospf_opaque_lsa_install_hook()*. For deletion, it is *ospf_opaque_lsa_delete_hook()*.

Note that incoming LSA which is already present in the LSDB will be inserted after the old instance of this LSA remove from the LSDB. Thus, after the first time, each incoming LSA will trigger a *delete* following by an *install*. This is not very helpful to handle real LSA deletion. In fact, LSA deletion is done by Flushing LSA i.e. flood LSA after setting its age to MAX_AGE. Then, a garbage function has the role to remove all LSA with *age == MAX_AGE* in the LSDB. So, to handle LSA Flush, the best is to look to the LSA age to determine if it is an installation or a future deletion i.e. the flushed LSA is first store in the LSDB with MAX_AGE waiting for the garbage collector function.

Router Information LSAs

To activate Segment Routing, new CLI command *segment-routing on* has been introduced. When this command is activated, function *ospf_router_info_update_sr()* is called to indicate to Router Information process that Segment Routing TLVs must be flood. Same function is called to modify the Segment Routing Global Block (SRGB) and Maximum Stack Depth (MSD) TLV. Only Shortest Path First (SPF) Algorithm is supported, so no possibility to modify this TLV is offer by the code.

When Opaque LSA Type 4 i.e. Router Information are stored in LSDB, function *ospf_opaque_lsa_install_hook()* will call the previously registered function *ospf_router_info_lsa_update()*. In turn, the function will simply trigger *ospf_sr_ri_lsa_update()* or *ospf_sr_ri_lsa_delete* in function of the LSA age. Before, it verifies that the LSA Opaque Type is 4 (Router Information). Self Opaque LSA are not send back to the Segment Routing functions as information are already stored.

Extended Link Prefix LSAs

Like for Router Information, Segment Routing is activate at the Extended Link/Prefix level with new *segment-routing on* command. This triggers automatically the flooding of Extended Link LSA for all ospf interfaces where adjacency is full. For Extended Prefix LSA, the new CLI command *segment-routing prefix ...* will trigger the flooding of Prefix SID TLV/SubTLVs.

When Opaque LSA Type 7 i.e. Extended Prefix and Type 8 i.e. Extended Link are store in the LSDB, *ospf_ext_pref_update_lsa()* respectively *ospf_ext_link_update_lsa()* are called like for Router Information LSA. In turn, they respectively trigger *ospf_sr_ext_prefix_lsa_update()* / *ospf_sr_ext_link_lsa_update()* or *ospf_sr_ext_prefix_lsa_delete()* / *ospf_sr_ext_link_lsa_delete()* if the LSA age is equal to MAX_AGE.

Zebra

When a new MPLS entry or new Forwarding Equivalent Class (FEC) must be added or deleted in the data plane, *add_sid_nhlfe()* respectively *del_sid_nhlfe()* are called. Once check the validity of labels, they are send to ZEBRA layer through *ZEBRA_MPLS_LABELS_ADD* command, respectively *ZEBRA_MPLS_LABELS_DELETE* command for deletion. This is completed by a new labelled route through *ZEBRA_ROUTE_ADD* command, respectively *ZEBRA_ROUTE_DELETE* command.

TI-LFA

Experimental support for Topology Independent LFA (Loop-Free Alternate), see for example 'draft-bashandy-rtgwg-segment-routing-ti-lfa-05'. The related files are *ospf_ti_lfa.c/h*.

The current implementation is rather naive and does not support the advanced optimizations suggested in e.g. RFC7490 or RFC8102. It focuses on providing the essential infrastructure which can also later be used to enhance the algorithmic aspects.

Supported features:

- Link and node protection
- Intra-area support
- Proper use of Prefix- and Adjacency-SIDs in label stacks
- Asymmetric weights (using reverse SPF)
- Non-adjacent P/Q spaces
- Protection of Prefix-SIDs

If configured for every SPF run the routing table is enriched with additional backup paths for every prefix. The corresponding Prefix-SIDs are updated with backup paths too within the OSPF SR update task.

Informal High-Level Algorithm Description:

```
p_spaces = empty_list()

for every protected_resource (link or node):
  p_space = generate_p_space(protected_resource)
  p_space.q_spaces = empty_list()

  for every destination that is affected by the protected_resource:
    q_space = generate_q_space(destination)

    # The label stack is stored in q_space
    generate_label_stack(p_space, q_space)

    # The p_space collects all its q_spaces
    p_spaces.q_spaces.add(q_space)

  p_spaces.add(p_space)

adjust_routing_table(p_spaces)
```

Possible Performance Improvements:

- Improve overall datastructures, get away from linked lists for vertices

- Don't calculate a Q space for every destination, but for a minimum set of backup paths that cover all destinations in the post-convergence SPF. The thinking here is that once a backup path is known that it is also a backup path for all nodes on the path themselves. This can be done by using the leafs of a trimmed minimum spanning tree generated out of the post- convergence SPF tree for that particular P space.
- For an alternative (maybe better) optimization look at https://tools.ietf.org/html/rfc7490#section-5.2.1.3 which describes using the Q space of the node which is affected by e.g. a link failure. Note that this optimization is topology dependent.

It is highly recommended to read e.g. *Segment Routing I/II* by Filsfils to understand the basics of Ti-LFA.

12.2.4 Configuration

Linux Kernel

In order to use OSPF Segment Routing, you must setup MPLS data plane. Up to know, only Linux Kernel version >= 4.5 is supported.

First, the MPLS modules aren't loaded by default, so you'll need to load them yourself:

```
modprobe mpls_router
modprobe mpls_gso
modprobe mpls_iptunnel
```

Then, you must activate MPLS on the interface you would used:

```
sysctl -w net.mpls.conf.enp0s9.input=1
sysctl -w net.mpls.conf.lo.input=1
sysctl -w net.mpls.platform_labels=1048575
```

The last line fix the maximum MPLS label value.

Once OSPFd start with Segment Routing, you could check that MPLS routes are enable with:

```
ip -M route
ip route
```

The first command show the MPLS LFIB table while the second show the FIB table which contains route with MPLS label encapsulation.

If you disable Penultimate Hop Popping with the *no-php-flag* (see below), you MUST check that RP filter is not enable for the interface you intend to use, especially the *lo* one. For that purpose, disable RP filtering with:

```
systcl -w net.ipv4.conf.all.rp_filter=0
sysctl -w net.ipv4.conf.lo.rp_filter=0
```

OSPFd

Here it is a simple example of configuration to enable Segment Routing. Note that *opaque capability* and *router information* must be set to activate Opaque LSA prior to Segment Routing.

```
router ospf
 ospf router-id 192.168.1.11
 capability opaque
 segment-routing on
 segment-routing global-block 10000 19999 local-block 5000 5999
 segment-routing node-msd 8
 segment-routing prefix 192.168.1.11/32 index 1100
```

The first segment-routing statement enables it. The second and third one set the SRGB and SRLB respectively, fourth line the MSD and finally, set the Prefix SID index for a given prefix.

Note that only prefix of Loopback interface could be configured with a Prefix SID. It is possible to add *no-php-flag* at the end of the prefix command to disable Penultimate Hop Popping. This advertises to peers that they MUST NOT pop the MPLS label prior to sending the packet.

12.2.5 Known limitations

- Runs only within default VRF
- Only single Area is supported. ABR is not yet supported
- Only SPF algorithm is supported
- Extended Prefix Range is not supported
- With NO Penultimate Hop Popping, it is not possible to express a Segment Path with an Adjacency SID due to the impossibility for the Linux Kernel to perform double POP instruction.

12.2.6 Credits

- Author: Anselme Sawadogo <anselmesawadogo@gmail.com>
- Author: Olivier Dugeon <olivier.dugeon@orange.com>
- Copyright (C) 2016 - 2018 Orange Labs http://www.orange.com

This work has been performed in the framework of the H2020-ICT-2014 project 5GEx (Grant Agreement no. 671636), which is partially funded by the European Commission.

CHAPTER

THIRTEEN

ZEBRA

13.1 Overview of the Zebra Protocol

The Zebra protocol (or `ZAPI`) is used by protocol daemons to communicate with the **zebra** daemon.

Each protocol daemon may request and send information to and from the **zebra** daemon such as interface states, routing state, nexthop-validation, and so on. Protocol daemons may also install routes with **zebra**. The **zebra** daemon manages which routes are installed into the forwarding table with the kernel. Some daemons use more than one ZAPI connection. This is supported: each ZAPI session is identified by a tuple of: `{protocol, instance, session_id}`. LDPD is an example: it uses a second, synchronous ZAPI session to manage label blocks. The default value for `session_id` is zero; daemons who use multiple ZAPI sessions must assign unique values to the sessions' ids.

The Zebra protocol is a streaming protocol, with a common header. Version 0 lacks a version field and is implicitly versioned. Version 1 and all subsequent versions have a version field. Version 0 can be distinguished from all other versions by examining the 3rd byte of the header, which contains a marker value of 255 (in Quagga) or 254 (in FRR) for all versions except version 0. The marker byte corresponds to the command field in version 0, and the marker value is a reserved command in version 0.

13.1.1 Version History

- Version 0

  Used by all versions of GNU Zebra and all version of Quagga up to and including Quagga 0.98. This version has no `version` field, and so is implicitly versioned as version 0.

- Version 1

  Added `marker` and `version` fields, increased `command` field to 16 bits. Used by Quagga versions 0.99.3 through 0.99.20.

- Version 2

  Used by Quagga versions 0.99.21 through 0.99.23.

- Version 3

  Added `vrf_id` field. Used by Quagga versions 0.99.23 until FRR fork.

- Version 4

  Change marker value to 254 to prevent people mixing and matching Quagga and FRR daemon binaries. Used by FRR versions 2.0 through 3.0.3.

- Version 5

  Increased VRF identifier field from 16 to 32 bits. Used by FRR versions 4.0 through 5.0.1.

- Version 6

  Removed the following commands:

  - ZEBRA_IPV4_ROUTE_ADD
  - ZEBRA_IPV4_ROUTE_DELETE
  - ZEBRA_IPV6_ROUTE_ADD
  - ZEBRA_IPV6_ROUTE_DELETE

  Used since FRR version 6.0.

13.2 Zebra Protocol Definition

13.2.1 Zebra Protocol Header Field Definitions

Length Total packet length including this header.

Marker Static marker. The marker value, when it exists, is 255 in all versions of Quagga. It is 254 in all versions of FRR. This is to allow version 0 headers (which do not include version explicitly) to be distinguished from versioned headers.

Version Zebra protocol version number. Clients should not continue processing messages past the version field for versions they do not recognise.

Command The Zebra protocol command.

Current Version

```
Version 5, 6

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |     Marker    |    Version    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             VRF ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Command            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Past Versions

```
Version 0

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |    Command    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Version 1, 2

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |     Marker    |    Version    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Command            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Version 3, 4

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |     Marker    |    Version    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             VRF ID            |            Command            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

13.2.2 Zebra Protocol Commands

Command	Value
ZEBRA_INTERFACE_ADD	0
ZEBRA_INTERFACE_DELETE	1
ZEBRA_INTERFACE_ADDRESS_ADD	2
ZEBRA_INTERFACE_ADDRESS_DELETE	3
ZEBRA_INTERFACE_UP	4
ZEBRA_INTERFACE_DOWN	5
ZEBRA_INTERFACE_SET_MASTER	6
ZEBRA_INTERFACE_SET_PROTODOWN	7
ZEBRA_ROUTE_ADD	8
ZEBRA_ROUTE_DELETE	9
ZEBRA_ROUTE_NOTIFY_OWNER	10
ZEBRA_REDISTRIBUTE_ADD	11
ZEBRA_REDISTRIBUTE_DELETE	12
ZEBRA_REDISTRIBUTE_DEFAULT_ADD	13
ZEBRA_REDISTRIBUTE_DEFAULT_DELETE	14
ZEBRA_ROUTER_ID_ADD	15
ZEBRA_ROUTER_ID_DELETE	16
ZEBRA_ROUTER_ID_UPDATE	17
ZEBRA_HELLO	18
ZEBRA_CAPABILITIES	19
ZEBRA_NEXTHOP_REGISTER	20
ZEBRA_NEXTHOP_UNREGISTER	21
ZEBRA_NEXTHOP_UPDATE	22
ZEBRA_INTERFACE_NBR_ADDRESS_ADD	23
ZEBRA_INTERFACE_NBR_ADDRESS_DELETE	24
ZEBRA_INTERFACE_BFD_DEST_UPDATE	25

continues on next page

Table 1 – continued from previous page

Command	Value
ZEBRA_IMPORT_ROUTE_REGISTER	26
ZEBRA_IMPORT_ROUTE_UNREGISTER	27
ZEBRA_IMPORT_CHECK_UPDATE	28
ZEBRA_BFD_DEST_REGISTER	29
ZEBRA_BFD_DEST_DEREGISTER	30
ZEBRA_BFD_DEST_UPDATE	31
ZEBRA_BFD_DEST_REPLAY	32
ZEBRA_REDISTRIBUTE_ROUTE_ADD	33
ZEBRA_REDISTRIBUTE_ROUTE_DEL	34
ZEBRA_VRF_UNREGISTER	35
ZEBRA_VRF_ADD	36
ZEBRA_VRF_DELETE	37
ZEBRA_VRF_LABEL	38
ZEBRA_INTERFACE_VRF_UPDATE	39
ZEBRA_BFD_CLIENT_REGISTER	40
ZEBRA_BFD_CLIENT_DEREGISTER	41
ZEBRA_INTERFACE_ENABLE_RADV	42
ZEBRA_INTERFACE_DISABLE_RADV	43
ZEBRA_NEXTHOP_LOOKUP_MRIB	44
ZEBRA_INTERFACE_LINK_PARAMS	45
ZEBRA_MPLS_LABELS_ADD	46
ZEBRA_MPLS_LABELS_DELETE	47
ZEBRA_MPLS_LABELS_REPLACE	48
ZEBRA_IPMR_ROUTE_STATS	49
ZEBRA_LABEL_MANAGER_CONNECT	50
ZEBRA_LABEL_MANAGER_CONNECT_ASYNC	51
ZEBRA_GET_LABEL_CHUNK	52
ZEBRA_RELEASE_LABEL_CHUNK	53
ZEBRA_FEC_REGISTER	54
ZEBRA_FEC_UNREGISTER	55
ZEBRA_FEC_UPDATE	56
ZEBRA_ADVERTISE_DEFAULT_GW	57
ZEBRA_ADVERTISE_SVI_MACIP	58
ZEBRA_ADVERTISE_SUBNET	59
ZEBRA_ADVERTISE_ALL_VNI	60
ZEBRA_LOCAL_ES_ADD	61
ZEBRA_LOCAL_ES_DEL	62
ZEBRA_VNI_ADD	63
ZEBRA_VNI_DEL	64
ZEBRA_L3VNI_ADD	65
ZEBRA_L3VNI_DEL	66
ZEBRA_REMOTE_VTEP_ADD	67
ZEBRA_REMOTE_VTEP_DEL	68
ZEBRA_MACIP_ADD	69
ZEBRA_MACIP_DEL	70
ZEBRA_IP_PREFIX_ROUTE_ADD	71
ZEBRA_IP_PREFIX_ROUTE_DEL	72
ZEBRA_REMOTE_MACIP_ADD	73
ZEBRA_REMOTE_MACIP_DEL	74

continues on next page

Table 1 – continued from previous page

Command	Value
ZEBRA_DUPLICATE_ADDR_DETECTION	75
ZEBRA_PW_ADD	76
ZEBRA_PW_DELETE	77
ZEBRA_PW_SET	78
ZEBRA_PW_UNSET	79
ZEBRA_PW_STATUS_UPDATE	80
ZEBRA_RULE_ADD	81
ZEBRA_RULE_DELETE	82
ZEBRA_RULE_NOTIFY_OWNER	83
ZEBRA_TABLE_MANAGER_CONNECT	84
ZEBRA_GET_TABLE_CHUNK	85
ZEBRA_RELEASE_TABLE_CHUNK	86
ZEBRA_IPSET_CREATE	87
ZEBRA_IPSET_DESTROY	88
ZEBRA_IPSET_ENTRY_ADD	89
ZEBRA_IPSET_ENTRY_DELETE	90
ZEBRA_IPSET_NOTIFY_OWNER	91
ZEBRA_IPSET_ENTRY_NOTIFY_OWNER	92
ZEBRA_IPTABLE_ADD	93
ZEBRA_IPTABLE_DELETE	94
ZEBRA_IPTABLE_NOTIFY_OWNER	95
ZEBRA_VXLAN_FLOOD_CONTROL	96
ZEBRA_VXLAN_SG_ADD	97
ZEBRA_VXLAN_SG_DEL	98
ZEBRA_VXLAN_SG_REPLAY	99
ZEBRA_MLAG_PROCESS_UP	100
ZEBRA_MLAG_PROCESS_DOWN	101
ZEBRA_MLAG_CLIENT_REGISTER	102
ZEBRA_MLAG_CLIENT_UNREGISTER	103
ZEBRA_MLAG_FORWARD_MSG	104
ZEBRA_ERROR	105
ZEBRA_CLIENT_CAPABILITIES	106
ZEBRA_OPAQUE_MESSAGE	107
ZEBRA_OPAQUE_REGISTER	108
ZEBRA_OPAQUE_UNREGISTER	109
ZEBRA_NEIGH_DISCOVER	110

13.3 Dataplane batching

Dataplane batching is an optimization feature that reduces the processing time involved in the user space to kernel space transition for every message we want to send.

13.3.1 Design

With our dataplane abstraction, we create a queue of dataplane context objects for the messages we want to send to the kernel. In a separate pthread, we loop over this queue and send the context objects to the appropriate dataplane. A batching enhancement tightly integrates with the dataplane context objects so they are able to be batch sent to dataplanes that support it.

There is one main change in the dataplane code. It does not call kernel-dependent functions one-by-one, but instead it hands a list of work down to the kernel level for processing.

Netlink

At the moment, this is the only dataplane that allows for batch sending messages to it.

When messages must be sent to the kernel, they are consecutively added to the batch represented by the *struct nl_batch*. Context objects are firstly encoded to their binary representation. All the encoding functions use the same interface: take a context object, a buffer and a size of the buffer as an argument. It is important that they should handle a situation in which a message wouldn't fit in the buffer and return a proper error. To achieve a zero-copy (in the user space only) messages are encoded to the same buffer which will be passed to the kernel. Hence, we can theoretically hit the boundary of the buffer.

Messages stored in the batch are sent if one of the conditions occurs:

- When an encoding function returns the buffer overflow error. The context object that caused this error is re-added to the new, empty batch.
- When the size of the batch hits certain limit.
- When the namespace of a currently being processed context object is different from all the previous ones. They have to be sent through distinct sockets, so the messages cannot share the same buffer.
- After the last message from the list is processed.

As mentioned earlier, there is a special threshold which is smaller than the size of the underlying buffer. It prevents the overflow error and thus eliminates the case, in which a message is encoded twice.

The buffer used in the batching is global, since allocating that big amount of memory every time wouldn't be most effective. However, its size can be changed dynamically, using hidden vtysh command: `zebra kernel netlink batch-tx-buf (1-1048576) (1-1048576)`. This feature is only used in tests and shouldn't be utilized in any other place.

For every failed message in the batch, the kernel responds with an error message. Error messages are kept in the same order as they were sent, so parsing the response is straightforward. We use the two pointer technique to match requests with responses and then set appropriate status of dataplane context objects. There is also a global receive buffer and it is assumed that whatever the kernel sends it will fit in this buffer. The payload of netlink error messages consists of a error code and the original netlink message of the request, so the batch response won't be bigger than the batch request increased by some space for the headers.

CHAPTER

FOURTEEN

VTYSH

See also:

Command Line Interface

14.1 Architecture

VTYSH is a shell for FRR daemons. It amalgamates all the CLI commands defined in each of the daemons and presents them to the user in a single shell, which saves the user from having to telnet to each of the daemons and use their individual shells. The amalgamation is achieved by *extracting* commands from daemons and injecting them into VTYSH at build time.

At runtime, VTYSH maintains an instance of a CLI mode tree just like each daemon. However, the mode tree in VTYSH contains (almost) all commands from every daemon in the same tree, whereas individual daemons have trees that only contain commands relevant to themselves. VTYSH also uses the library CLI facilities to maintain the user's current position in the tree (the current node). Note that this position must be synchronized with all daemons; if a daemon receives a command that causes it to change its current node, VTYSH must also change its node. Since the extraction script does not understand the handler code of commands, but only their definitions, this and other behaviors must be manually programmed into VTYSH for every case where the internal state of VTYSH must change in response to a command. Details on how this is done are discussed in the *Special DEFUNs* section.

VTYSH also handles writing and applying the integrated configuration file, `/etc/frr/frr.conf`. Since it has knowledge of the entire command space of FRR, it can intelligently distribute configuration commands only to the daemons that understand them. Similarly, when writing the configuration file it takes care of combining multiple instances of configuration blocks and simplifying the output. This is discussed in *Configuration Management*.

14.1.1 Command Extraction

To build `vtysh`, the `python/xref2vtysh.py` script scans through the `frr.xref` file created earlier in the build process. This file contains a list of all `DEFUN` and `install_element` sites in the code, generated directly from the binaries (and therefore matching exactly what is really available.)

This list is collated and transformed into `DEFSH` (and `install_element`) statements, output to `vtysh_cmd.c`. Each `DEFSH` contains the name of the command plus `_vtysh`, as well as a flag that indicates which daemons the command was found in. When the command is executed in VTYSH, this flag is inspected to determine which daemons to send the command to. This way, commands are only sent to the daemons that know about them, avoiding spurious errors from daemons that don't have the command defined.

The extraction script contains lots of hardcoded knowledge about what sources to look at and what flags to use for certain commands.

Note: The `vtysh_scan` Makefile variable and `#ifndef VTYSH_EXTRACT_PL` checks in source files are no longer used. Remove them when rebasing older changes.

14.1.2 Special DEFUNs

In addition to the vanilla `DEFUN` macro for defining CLI commands, there are several VTYSH-specific `DEFUN` variants that each serve different purposes.

`DEFSH` Used almost exclusively by generated VTYSH code. This macro defines a `cmd_element` with no handler function; the command, when executed, is simply forwarded to the daemons indicated in the daemon flag.

`DEFUN_NOSH` Used by daemons. Has the same expansion as a `DEFUN`, but `xref2vtysh.py` will skip these definitions when extracting commands. This is typically used when VTYSH must take some special action upon receiving the command, and the programmer therefore needs to write VTYSH's copy of the command manually instead of using the generated version.

`DEFUNSH` The same as `DEFUN`, but with an argument that allows specifying the `->daemon` field of the generated `cmd_element`. This is used by VTYSH to determine which daemons to send the command to.

`DEFUNSH_ATTR` A version of `DEFUNSH` that allows setting the `->attr` field of the generated `cmd_element`. Not used in practice.

14.1.3 Configuration Management

When integrated configuration is used, VTYSH manages writing, reading and applying the FRR configuration file. VTYSH can be made to read and apply an integrated configuration to all running daemons by launching it with `-f <file>`. It sends the appropriate configuration lines to the relevant daemons in the same way that commands entered by the user on VTYSH's shell prompt are processed.

Configuration writing is more complicated. VTYSH makes a best-effort attempt to combine and simplify the configuration as much as possible. A working example is best to explain this behavior.

Example

Suppose we have just *staticd* and *zebra* running on the system, and use VTYSH to apply the following configuration snippet:

```
!
vrf blue
 ip protocol static route-map ExampleRoutemap
 ip route 192.168.0.0/24 192.168.0.1
 exit-vrf
!
```

Note that *staticd* defines static route commands and *zebra* defines `ip protocol` commands. Therefore if we ask only *zebra* for its configuration, we get the following:

```
(config)# do sh running-config zebra
Building configuration...

...
!
```

(continues on next page)

(continued from previous page)

```
vrf blue
 ip protocol static route-map ExampleRoutemap
 exit-vrf
!
...
```

Note that the static route doesn't show up there. Similarly, if we ask *staticd* for its configuration, we get:

```
(config)# do sh running-config staticd

...
!
vrf blue
 ip route 192.168.0.0/24 192.168.0.1
 exit-vrf
!
...
```

But when we display the configuration with VTYSH, we see:

```
ubuntu-bionic(config)# do sh running-config

...
!
vrf blue
 ip protocol static route-map ExampleRoutemap
 ip route 192.168.0.0/24 192.168.0.1
 exit-vrf
!
...
```

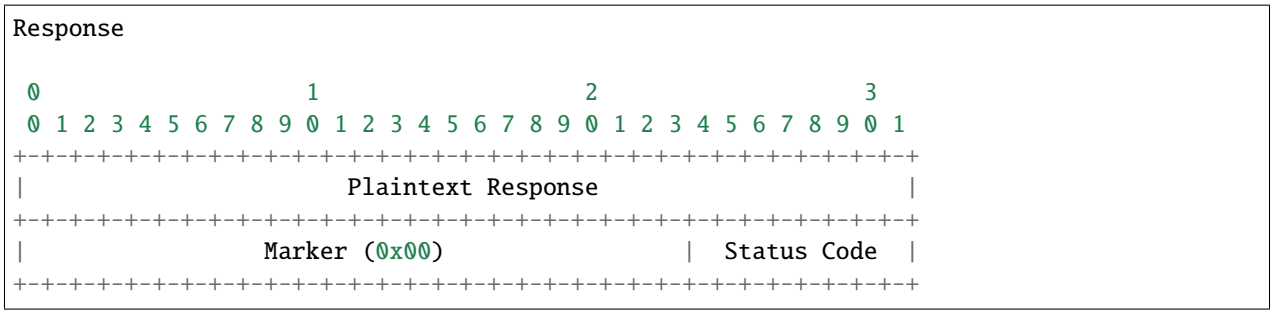
This is because VTYSH asks each daemon for its currently running configuration, and combines equivalent blocks together. In the above example, it combined the `vrf blue` blocks from both *zebra* and *staticd* together into one. This is done in `vtysh_config.c`.

14.2 Protocol

VTYSH communicates with FRR daemons by way of domain socket. Each daemon creates its own socket, typically in `/var/run/frr/<daemon>.vty`. The protocol is very simple. In the VTYSH to daemon direction, messages are simply NUL-terminated strings, whose content are CLI commands. Here is a typical message from VTYSH to a daemon:

```
Request

00000000: 646f 2077 7269 7465 2074 6572 6d69 6e61  do write termina
00000010: 6c0a 00                                  l..
```

The response format has some more data in it. First is a NUL-terminated string containing the plaintext response, which is just the output of the command that was sent in the request. This is displayed to the user. The plaintext response is followed by 3 null marker bytes, followed by a 1-byte status code that indicates whether the command was successful or not.



The first `0x00` byte in the marker also serves to terminate the plaintext response.

CHAPTER

FIFTEEN

PATHD

15.1 Internals

15.1.1 PATHD Internals

Architecture

Overview

The pathd deamon manages the segment routing policies, it owns the data structures representing them and can load modules that manipulate them like the PCEP module. Its responsibility is to select a candidate path for each configured policy and to install it into Zebra.

Zebra

Zebra manages policies that are active or pending to be activated due to the next hop not being available yet. In zebra, policy data structures and APIs are defined in *zebra_srte.[hc]*.

The responsibilities of Zebra are:

- Store the policies' segment list.
- Install the policies when their next-hop is available.
- Notify other daemons of the status of the policies.

Adding and removing policies is done using the commands *ZEBRA_SR_POLICY_SET* and *ZEBRA_SR_POLICY_DELETE* as parameter of the function *zebra_send_sr_policy* all defined in *zclient.[hc]*.

If the first segment of the policy is an unknown label, it is kept until notified by the mpls hooks *zebra_mpls_label_created*, and then it is installed.

To get notified when a policy status changes, a client can implement the *sr_policy_notify_status* callback defined in *zclient.[hc]*.

For encoding/decoding the various data structures used to comunicate with zebra, the following functions are available from *zclient.[hc]*: *zapi_sr_policy_encode*, *zapi_sr_policy_decode* and *zapi_sr_policy_notify_status_decode*.

Pathd

The pathd daemon manages all the possible candidate paths for the segment routing policies and selects the best one following the segment routing policy draft. It also supports loadable modules for handling dynamic candidate paths and the creation of new policies and candidate paths at runtime.

The responsibilities of the pathd base daemon, not including any optional modules, are:

- Store the policies and all the possible candidate paths for them.
- Select the best candidate path for each policy and send it to Zebra.
- Provide VTYSH configuration to set up policies and candidate paths.
- Provide a Northbound API to manipulate **configured** policies and candidate paths.
- Handle loadable modules for extending the functionality.
- Provide an API to the loadable module to manipulate policies and candidate paths.

Threading Model

The daemon runs completely inside the main thread using FRR event model, there is no threading involved.

Source Code

Internal Data Structures

The main data structures for policies and candidate paths are defined in *pathd.h* and implemented in *pathd.c*.

When modifying these structures, either directly or through the functions exported by *pathd.h*, nothing should be deleted/freed right away. The deletion or modification flags must be set and when all the changes are done, the function *srte_apply_changes* must be called. When called, a new candidate path may be elected and sent to Zebra, and all the structures flagged as deleted will be freed. In addition, a hook will be called so dynamic modules can perform any required action when the elected candidate path changes.

Northbound API

The northbound API is defined in *path_nb.[ch]* and implemented in *path_nb_config.c* for configuration data and *path_nb_state.c* for operational data.

Command Line Client

The command-line client (VTYSH) is implemented in *path_cli.c*.

Interface with Zebra

All the functions interfacing with Zebra are defined and implemented in *path_zebra.[hc]*.

Loadable Module API

For the time being, the API the loadable module uses is defined by *pathd.h*, but in the future, it should be moved to a dedicated include file.

15.1.2 PCEP Module Internals

Introduction

The PCEP module for the pathd daemon implements the PCEP protocol described in **RFC 5440** to update the policies and candidate paths.

The protocol encoding/decoding and the basic session management is handled by the pceplib external library 1.2.

Together with pceplib, this module supports at least partially:

- **RFC 5440**

  Most of the protocol defined in the RFC is implemented. All the messages can be parsed, but this was only tested in the context of segment routing. Only a very small subset of metric types can be configured, and there is a known issue with some Cisco routers not following the IANA numbers for metrics.

- **RFC 8231**

  Support delegation of candidate path after performing the initial computation request. If the PCE does not respond or cannot compute a path, an empty candidate path is delegated to the PCE. Only tested in the context of segment routing.

- **RFC 8408**

  Only used to comunicate the support for segment routing to the PCE.

- **RFC 8664**

  All the NAI types are implemented, but only the MPLS NAI are supported. If the PCE provide segments that are not MPLS labels, the PCC will return an error.

Note that pceplib supports more RFCs and drafts, see pceplib README for more details.

Architecture

Overview

The module is separated into multiple layers:

- pathd interface
- command-line console
- controller
- PCC
- pceplib interface

The pathd interface handles all the interactions with the daemon API.

The command-line console handles all the VTYSH configuration commands.

The controller manages the multiple PCC connections and the interaction between them and the daemon interface.

The PCC handles a single connection to a PCE through a pceplib session.

The pceplib interface abstracts the API of the pceplib.

Threading Model

The module requires multiple threads to cooperate:

- The main thread used by the pathd daemon.
- The controller pthread used to isolate the PCC from the main thread.
- The possible threads started in the pceplib library.

To ensure thread safety, all the controller and PCC state data structures can only be read and modified in the controller thread, and all the global data structures can only be read and modified in the main thread. Most of the interactions between these threads are done through FRR timers and events.

The controller is the bridge between the two threads, all the functions that **MUST** be called from the main thread start with the prefix *pcep_ctrl_* and all the functions that **MUST** be called from the controller thread start with the prefix *pcep_thread_*. When an asynchronous action must be taken in a different thread, an FRR event is sent to the thread. If some synchronous operation is needed, the calling thread will block and run a callback in the other thread, there the result is **COPIED** and returned to the calling thread.

No function other than the controller functions defined for it should be called from the main thread. The only exception being some utility functions from *path_pcep_lib.[hc]*.

All the calls to pathd API functions **MUST** be performed in the main thread, for that, the controller sends FRR events handled in function *path_pcep.c:pcep_main_event_handler*.

For the same reason, the console client only runs in the main thread. It can freely use the global variable, but **MUST** use controller's *pcep_ctrl_* functions to interact with the PCCs.

Source Code

Generic Data Structures

The data structures are defined in multiple places, and where they are defined dictates where they can be used.

The data structures defined in *path_pcep.h* can be used anywhere in the module.

Internally, throughout the module, the *struct path* data structure is used to describe PCEP messages. It is a simplified flattened structure that can represent multiple complex PCEP message types. The conversion from this structure to the PCEP data structures used by pceplib is done in the pceplib interface layer.

The data structures defined in *path_pcep_controller.h* should only be used in *path_pcep_controller.c*. Even if a structure pointer is passed as a parameter to functions defined in *path_pcep_pcc.h*, these should consider it as an opaque data structure only used to call back controller functions.

The same applies to the structures defined in *path_pcep_pcc.h*, even if the controller owns a reference to this data structure, it should never read or modify it directly, it should be considered an opaque structure.

The global data structure can be accessed from the pathd interface layer *path_pcep.c* and the command line client code *path_pcep_cli.c*.

Interface With Pathd

All the functions calling or called by the pathd daemon are implemented in *path_pcep.c*. These functions **MUST** run in the main FRR thread, and all the interactions with the controller and the PCCs **MUST** pass through the controller's *pcep_ctrl_* prefixed functions.

To handle asynchronous events from the PCCs, a callback is passed to *pcep_ctrl_initialize* that is called in the FRR main thread context.

Command Line Client

All the command line configuration commands (VTYSH) are implemented in *path_pcep_cli.c*. All the functions there run in the main FRR thread and can freely access the global variables. All the interaction with the controller's and the PCCs **MUST** pass through the controller *pcep_ctrl_* prefixed functions.

Debugging Helpers

All the functions formating data structures for debugging and logging purposes are implemented in *path_pcep_debug.[hc]*.

Interface with pceplib

All the functions calling the pceplib external library are defined in *path_pcep_lib.[hc]*. Some functions are called from the main FRR thread, like *pcep_lib_initialize*, *pcep_lib_finalize*; some can be called from either thread, like *pcep_lib_free_counters*; some function must be called from the controller thread, like *pcep_lib_connect*. This will probably be formalized later on with function prefix like done in the controller.

Controller

The controller is defined and implemented in *path_pcep_controller.[hc]*. Part of the controller code runs in FRR main thread and part runs in its own FRR pthread started to isolate the main thread from the PCCs' event loop. To communicate between the threads it uses FRR events, timers and *thread_execute* calls.

PCC

Each PCC instance owns its state and runs in the controller thread. They are defined and implemented in *path_pcep_pcc.[hc]*. All the interactions with the daemon must pass through some controller's *pcep_thread_* prefixed function.

CHAPTER

SIXTEEN

PCEPLIB

16.1 Overview

The PCEPlib is a PCEP implementation library that can be used by either a PCE or PCC.

Currently, only the FRR pathd has been implemented as a PCC with the PCEPlib. The PCEPlib is able to simultaneously connect to multiple PCEP peers and can maintain persistent PCEP connections.

16.2 PCEPlib compliance

The PCEPlib implements version 1 of the PCEP protocol, according to RFC 5440.

Additionally, the PCEPlib implements the following PCEP extensions:

- RFC 8281 PCE initiated for PCE-Initiated LSP Setup
- RFC 8231 Extensions for Stateful PCE
- RFC 8232 Optimizations of Label Switched Path State Synchronization Procedures for a Stateful PCE
- RFC 8282 Extensions to PCEP for Inter-Layer MPLS and GMPLS Traffic Engineering
- RFC 8408 Conveying Path Setup Type in PCE Communication Protocol (PCEP) Messages
- draft-ietf-pce-segment-routing-07, draft-ietf-pce-segment-routing-16, RFC 8664 Segment routing protocol extensions
- RFC 7470 Conveying Vendor-Specific Constraints
- Draft-ietf-pce-association-group-10 Establishing Relationships Between Sets of Label Switched Paths
- Draft-barth-pce-segment-routing-policy-cp-04 Segment Routing Policy Candidate Paths

16.3 PCEPlib Architecture

The PCEPlib is comprised of the following modules, each of which will be detailed in the following sections.

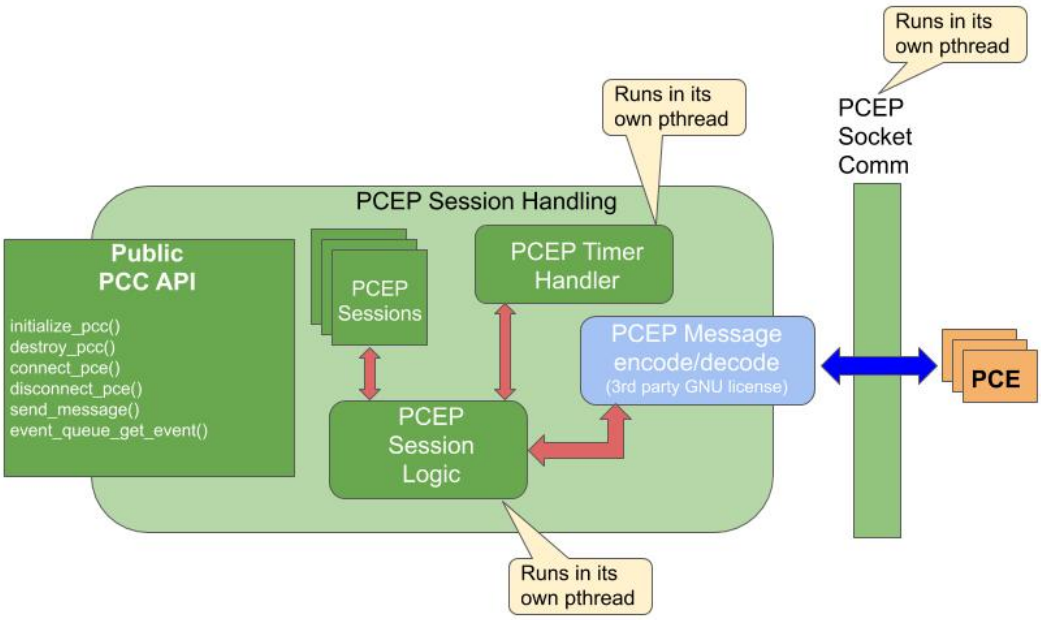
- **pcep_messages**
  - PCEP messages, objects, and TLVs implementations
- **pcep_pcc**
  - PCEPlib public PCC API with a sample PCC binary
- **pcep_session_logic**

  - PCEP Session handling

- **pcep_socket_comm**
  - Socket communications
- **pcep_timers**
  - PCEP timers
- **pcep_utils**
  - Internal utilities used by the PCEPlib modules.

The interaction of these modules can be seen in the following diagram.

PCEPlib Architecture:



16.3.1 PCEP Session Logic library

The PCEP Session Logic library orchestrates calls to the rest of the PCC libraries.

PCEP Session Logic library responsibilities:

- Handle messages received from "PCEP Socket Comm"
- Create and manage "PCEP Session" objects
- Set timers and react to timer expirations
- Manage counters

The PCEP Session Logic library will have 2 main triggers controlled by a pthread condition variable:

- Timer expirations - `on_timer_expire()` callback
- Messages received from PCEP SocketComm - `message_received()` callback

The counters are created and managed using the `pcep_utils/pcep_utils_counters.h` counters library. The following are the different counter groups managed:

- **COUNTER_SUBGROUP_ID_RX_MSG**
- **COUNTER_SUBGROUP_ID_TX_MSG**
- **COUNTER_SUBGROUP_ID_RX_OBJ**
- **COUNTER_SUBGROUP_ID_TX_OBJ**
- **COUNTER_SUBGROUP_ID_RX_SUBOBJ**
- **COUNTER_SUBGROUP_ID_TX_SUBOBJ**
- **COUNTER_SUBGROUP_ID_RX_RO_SR_SUBOBJ**
- **COUNTER_SUBGROUP_ID_TX_RO_SR_SUBOBJ**
- **COUNTER_SUBGROUP_ID_RX_TLV**
- **COUNTER_SUBGROUP_ID_TX_TLV**
- **COUNTER_SUBGROUP_ID_EVENT**

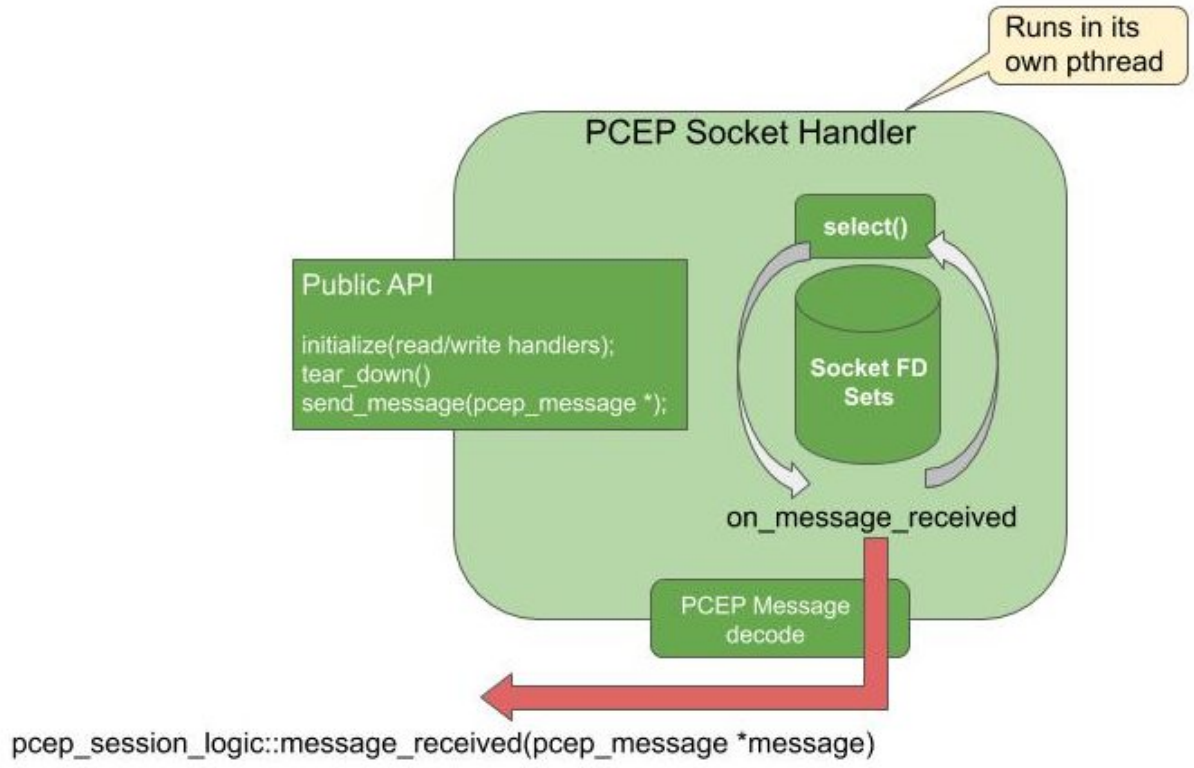
The counters can be obtained and reset as explained later in the PCEPlib PCC API.

16.3.2 PCEP Socket Comm library

PCEP communication can be configured to be handled internally in this simple library. When this library is instantiated by the PCEP Session Logic, callbacks are provided to handle received messages and error conditions.

The following diagram illustrates how the library works.

PCEPlib Socket Comm:

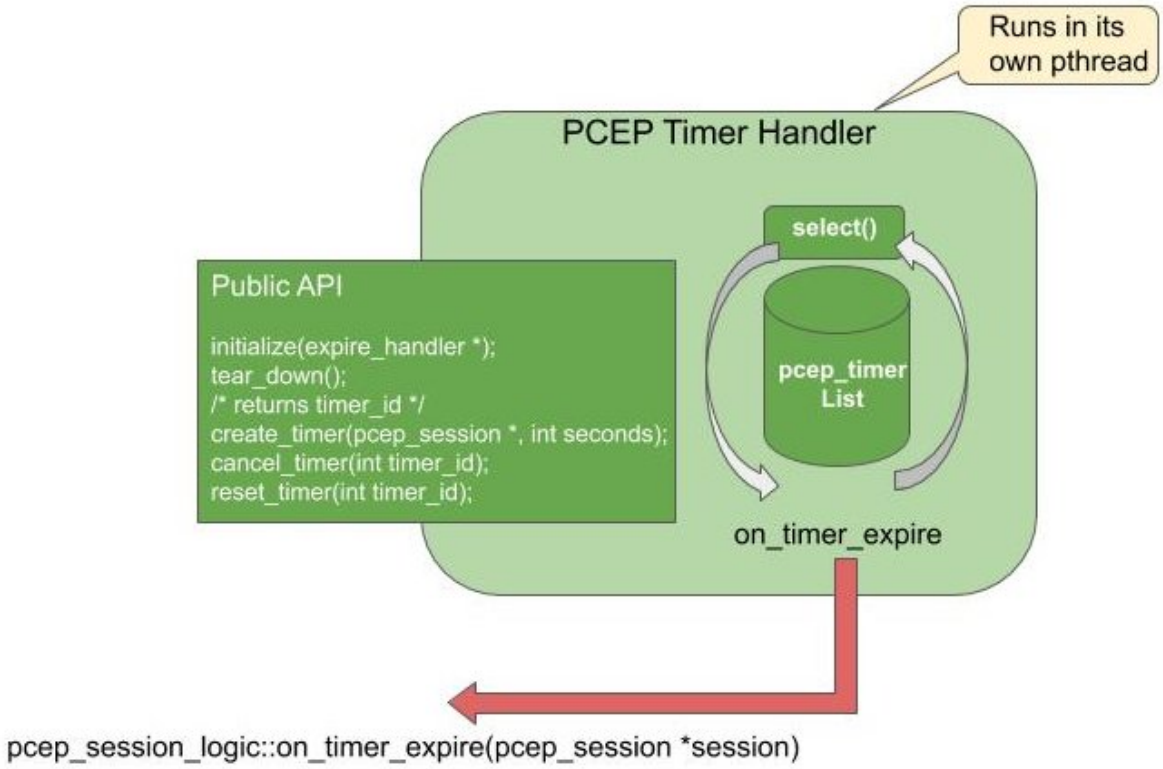


16.3.3 PCEP Timers library

Timers can be configured to be handled internally by this library. When this library is instantiated by the PCEP Session Logic, callbacks are provided to ha:0 ndle timer expirations. The following timers are implemented and handled, according to RFC 5440.

- **Open KeepWait (fixed at 60 seconds)**
  - Set once the PCC sends an Open, and if it expires before receiving a KeepAlive or PCErr, then the PCC should send a PCErr and close the TCP connection
- **Keepalive timer**
  - How often the PCC should send Keepalive messages to the PCE (and vice-versa)
  - The timer will be reset after any message is sent: any message serves as a Keepalive
- **DeadTimer**
  - If no messages are received before expiration, the session is declared as down
  - Reset everytime any message is received
- **PCReq request timer**
  - How long the PCC waits for the PCE to reply to PCReq messages.

PCEPlib Timers:



16.3.4 PCEP Messages library

The PCEP Messages library has all of the implemented PCEP messages, objects, TLVs, and related functionality.

The following header files can be used for creating and handling received PCEP entities.

- pcep-messages.h
- pcep-objects.h
- pcep-tlvs.h

PCEP Messages

The following PCEP messages can be created and received:

- `struct pcep_message* pcep_msg_create_open(...);`
- `struct pcep_message* pcep_msg_create_open_with_tlvs(...);`
- `struct pcep_message* pcep_msg_create_request(...);`
- `struct pcep_message* pcep_msg_create_request_ipv6(...);`
- `struct pcep_message* pcep_msg_create_reply(...);`
- `struct pcep_message* pcep_msg_create_close(...);`
- `struct pcep_message* pcep_msg_create_error(...);`
- `struct pcep_message* pcep_msg_create_error_with_objects(...);`
- `struct pcep_message* pcep_msg_create_keepalive(...);`
- `struct pcep_message* pcep_msg_create_report(...);`
- `struct pcep_message* pcep_msg_create_update(...);`
- `struct pcep_message* pcep_msg_create_initiate(...);`

Refer to `pcep_messages/include/pcep-messages.h` and the API section below for more details.

PCEP Objects

The following PCEP objects can be created and received:

- `struct pcep_object_open* pcep_obj_create_open(...);`
- `struct pcep_object_rp* pcep_obj_create_rp(...);`
- `struct pcep_object_notify* pcep_obj_create_notify(...);`
- `struct pcep_object_nopath* pcep_obj_create_nopath(...);`
- `struct pcep_object_association_ipv4* pcep_obj_create_association_ipv4(...);`
- `struct pcep_object_association_ipv6* pcep_obj_create_association_ipv6(...);`
- `struct pcep_object_endpoints_ipv4* pcep_obj_create_endpoint_ipv4(...);`
- `struct pcep_object_endpoints_ipv6* pcep_obj_create_endpoint_ipv6(...);`
- `struct pcep_object_bandwidth* pcep_obj_create_bandwidth(...);`
- `struct pcep_object_metric* pcep_obj_create_metric(...);`

- `struct pcep_object_lspa* pcep_obj_create_lspa(...);`
- `struct pcep_object_svec* pcep_obj_create_svec(...);`
- `struct pcep_object_error* pcep_obj_create_error(...);`
- `struct pcep_object_close* pcep_obj_create_close(...);`
- `struct pcep_object_srp* pcep_obj_create_srp(...);`
- `struct pcep_object_lsp* pcep_obj_create_lsp(...);`
- `struct pcep_object_vendor_info* pcep_obj_create_vendor_info(...);`
- `struct pcep_object_ro* pcep_obj_create_ero(...);`
- `struct pcep_object_ro* pcep_obj_create_rro(...);`
- `struct pcep_object_ro* pcep_obj_create_iro(...);`
- `struct pcep_ro_subobj_ipv4* pcep_obj_create_ro_subobj_ipv4(...);`
- `struct pcep_ro_subobj_ipv6* pcep_obj_create_ro_subobj_ipv6(...);`
- `struct pcep_ro_subobj_unnum* pcep_obj_create_ro_subobj_unnum(...);`
- `struct pcep_ro_subobj_32label* pcep_obj_create_ro_subobj_32label(...);`
- `struct pcep_ro_subobj_asn* pcep_obj_create_ro_subobj_asn(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_nonai(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_ipv4_node(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_ipv6_node(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_ipv4_adj(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_ipv6_adj(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_unnumbered_ipv4_adj(...);`
- `struct pcep_ro_subobj_sr* pcep_obj_create_ro_subobj_sr_linklocal_ipv6_adj(...);`

Refer to `pcep_messages/include/pcep-objects.h` and the API section below for more details.

PCEP TLVs

The following PCEP TLVs (Tag, Length, Value) can be created and received:

- **Open Object TLVs**
  - `struct pcep_object_tlv_stateful_pce_capability* pcep_tlv_create_stateful_pce_capability(...);`
  - `struct pcep_object_tlv_lsp_db_version* pcep_tlv_create_lsp_db_version(...);`
  - `struct pcep_object_tlv_speaker_entity_identifier* pcep_tlv_create_speaker_entity_id(...);`
  - `struct pcep_object_tlv_path_setup_type* pcep_tlv_create_path_setup_type(...);`
  - `struct pcep_object_tlv_path_setup_type_capability* pcep_tlv_create_path_setup_type_capabili...);`
  - `struct pcep_object_tlv_sr_pce_capability* pcep_tlv_create_sr_pce_capability(...);`

- **LSP Object TLVs**
  - `struct pcep_object_tlv_ipv4_lsp_identifier* pcep_tlv_create_ipv4_lsp_identifiers(...);`
  - `struct pcep_object_tlv_ipv6_lsp_identifier* pcep_tlv_create_ipv6_lsp_identifiers(...);`
  - `struct pcep_object_tlv_symbolic_path_name* pcep_tlv_create_symbolic_path_name(...);`
  - `struct pcep_object_tlv_lsp_error_code* pcep_tlv_create_lsp_error_code(...);`
  - `struct pcep_object_tlv_rsvp_error_spec* pcep_tlv_create_rsvp_ipv4_error_spec(...);`
  - `struct pcep_object_tlv_rsvp_error_spec* pcep_tlv_create_rsvp_ipv6_error_spec(...);`
  - `struct pcep_object_tlv_nopath_vector* pcep_tlv_create_nopath_vector(...);`
  - `struct pcep_object_tlv_vendor_info* pcep_tlv_create_vendor_info(...);`
  - `struct pcep_object_tlv_arbitrary* pcep_tlv_create_tlv_arbitrary(...);`
- **SRPAG (SR Association Group) TLVs**
  - `struct pcep_object_tlv_srpag_pol_id *pcep_tlv_create_srpag_pol_id_ipv4(...);`
  - `struct pcep_object_tlv_srpag_pol_id *pcep_tlv_create_srpag_pol_id_ipv6(...);`
  - `struct pcep_object_tlv_srpag_pol_name *pcep_tlv_create_srpag_pol_name(...);`
  - `struct pcep_object_tlv_srpag_cp_id *pcep_tlv_create_srpag_cp_id(...);`
  - `struct pcep_object_tlv_srpag_cp_pref *pcep_tlv_create_srpag_cp_pref(...);`

Refer to `pcep_messages/include/pcep-tlvs.h` and the API section below for more details.

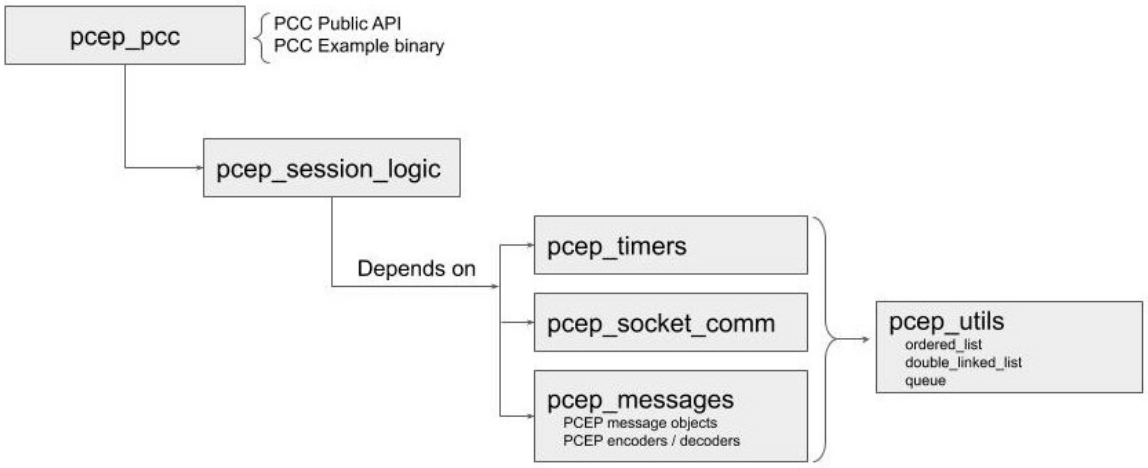
16.3.5 PCEP PCC

This module has a Public PCC API library (explained in detail later) and a sample PCC binary. The APIs in this library encapsulate other PCEPlib libraries for simplicity. With this API, the PCEPlib PCC can be started and stopped, and the PCEPlib event queue can be accessed. The PCEP Messages library is not encapsulated, and should be used directly.

16.3.6 Internal Dependencies

The following diagram illustrates the internal PCEPlib library dependencies.

PCEPlib internal dependencies:



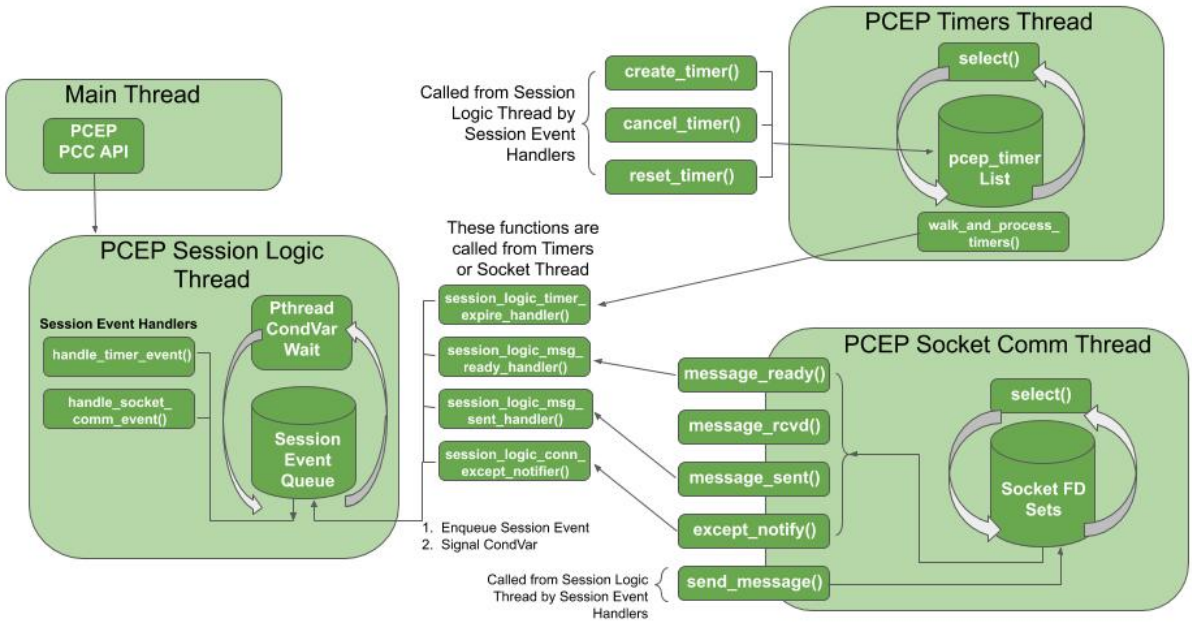
16.3.7 External Dependencies

Originally the PCEPlib was based on the open source libpcep project, but that dependency has been reduced to just one source file (pcep-tools.[ch]).

16.3.8 PCEPlib Threading model

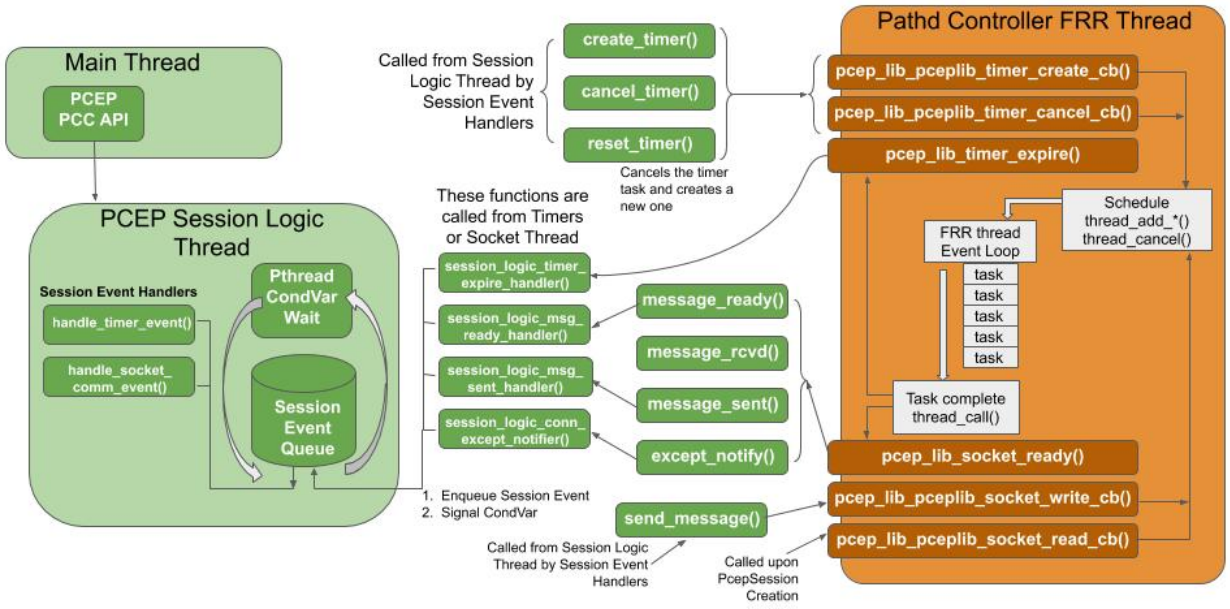
The PCEPlib can be run in stand-alone mode whereby a thread is launched for timers and socket comm, as is illustrated in the following diagram.

PCEPlib Threading model:



The PCEPlib can also be configured to use an external timers and socket infrastructure like the FRR threads and tasks. In this case, no internal threads are launched for timers and socket comm, as is illustrated in the following diagram.

PCEPlib Threading model with external infra:



16.3.9 Building

The autotools build system is used and integrated with the frr build system.

16.3.10 Testing

The Unit Tests for an individual library are executed with the `make check` command. The Unit Test binary will be written to the project `build` directory. All Unit Tests are executed with Valgrind, and any memory issues reported by Valgrind will cause the Unit Test to fail.

16.4 PCEPlib PCC API

The following sections describe the PCEPlib PCC API.

16.4.1 PCEPlib PCC Initialization and Destruction

The PCEPlib can be initialized to handle memory, timers, and socket comm internally in what is called stand-alone mode, or with an external infrastructure, like FRR.

PCEPlib PCC Initialization and Destruction in stand-alone mode

PCEPlib PCC initialization and destruction functions:

- `bool initialize_pcc();`
- `bool initialize_pcc_wait_for_completion();`
- `bool destroy_pcc();`

The PCC can be initialized with either `initialize_pcc()` or `initialize_pcc_wait_for_completion()`.

- **`initialize_pcc_wait_for_completion()` blocks until `destroy_pcc()`** is called from a separate pthread.
- `initialize_pcc()` is non-blocking and will be stopped when `destroy_pcc()` is called.

Both initialize functions will launch 3 pthreads:

- 1 Timer pthread
- 1 SocketComm pthread
- 1 SessionLogic pthread

When `destroy_pcc()` is called, all pthreads will be stopped and all resources will be released.

All 3 functions return true upon success, and false otherwise.

PCEPlib PCC Initialization and Destruction with FRR infrastructure

PCEPlib PCC initialization and destruction functions:

- `bool initialize_pcc_infra(struct pceplib_infra_config *infra_config);`
- `bool destroy_pcc();`

The `pceplib_infra_config` struct has the following fields:

- **void *pceplib_infra_mt**
  - FRR Memory type pointer for infra related memory management
- **void *pceplib_messages_mt**
  - FRR Memory type pointer for PCEP messages related memory management
- **pceplib_malloc_func mfunc**
  - FRR malloc function pointer
- **pceplib_calloc_func cfunc**
  - FRR calloc function pointer
- **pceplib_realloc_func rfunc**
  - FRR realloc function pointer
- **pceplib_strdup_func sfunc**

  - FRR strdup function pointer
- **pceplib_free_func ffunc**
  - FRR free function pointer
- **void *external_infra_data**
  - FRR data used by FRR timers and sockets infrastructure
- **ext_timer_create timer_create_func**
  - FRR timer create function pointer
- **ext_timer_cancel timer_cancel_func**
  - FRR timer cancel function pointer
- **ext_socket_write socket_write_func**
  - FRR socket write function pointer, indicating fd is ready to be written to
- **ext_socket_read socket_read_func**
  - FRR socket write function pointer, indicating fd is ready to be read from

16.4.2 PCEPlib PCC configuration

PCEPlib PCC configuratoin functions:

- `pcep_configuration *create_default_pcep_configuration();`
- `void destroy_pcep_configuration(pcep_configuration *config);`

A `pcep_configuration` object with default values is created with `create_default_pcep_configuration()`. These values can be tailored to specific use cases.

Created `pcep_configuration` objects are destroyed with `destroy_pcep_configuration()`.

PCEPlib PCC configuration paramaters

The `pcep_configuration` object is defined in `pcep_session_logic/include/pcep_session_logic.h` The attributes in the `pcep_configuration` object are detailed as follows.

PCEP Connection parameters:

- **dst_pcep_port**
  - Defaults to 0, in which case the default PCEP TCP destination port 4189 will be used.
  - Set to use a specific PCEP TCP destination port.
- **src_pcep_port**
  - Defaults to 0, in which case the default PCEP TCP source port 4189 will be used.
  - Set to use a specific PCEP TCP source port.
- **Source IP**
  - Defaults to IPv4 INADDR_ANY
  - Set **src_ip.src_ipv4** and **is_src_ipv6=false** to set the source IPv4.
  - Set **src_ip.src_ipv6** and **is_src_ipv6=true** to set the source IPv6.

- **socket_connect_timeout_millis**
  - Maximum amount of time to wait to connect to the PCE TCP socket before failing, in milliseconds.

PCEP Versioning:

- **pcep_msg_versioning->draft_ietf_pce_segment_routing_07**
  - Defaults to false, in which case draft 16 versioning will be used.
  - Set to true to use draft 07 versioning.

PCEP Open Message Parameters:

- **keep_alive_seconds**
  - Sent to PCE in PCEP Open Msg
  - Recommended value = 30, Minimum value = 1
  - Disabled by setting value = 0
- **dead_timer_seconds**
  - Sent to PCE in PCEP Open Msg
  - Recommended value = 4 * keepalive timer value
- **Supported value ranges for PCEP Open Message received from the PCE**
  - **min_keep_alive_seconds**, **max_keep_alive_seconds**
  - **min_dead_timer_seconds**, **max_dead_timer_seconds**
- **request_time_seconds**
  - When a PCC sends a PcReq to a PCE, the amount of time a PCC will wait for a PcRep reply from the PCE.
- **max_unknown_requests**
  - If a PCC/PCE receives PCRep/PCReq messages with unknown requests at a rate equal or greater than MAX-UNKNOWN-REQUESTS per minute, the PCC/PCE MUST send a PCEP CLOSE message.
  - Recommended value = 5
- **max_unknown_messages**
  - If a PCC/PCE receives unrecognized messages at a rate equal or greater than MAX-UNKNOWN-MESSAGES per minute, the PCC/PCE MUST send a PCEP CLOSE message
  - Recommended value = 5

Stateful PCE Capability TLV configuration parameters (RFC 8231, 8232, 8281, and draft-ietf-pce-segment-routing-16):

- **support_stateful_pce_lsp_update**
  - If this flag is true, then a Stateful PCE Capability TLV will be added to the PCEP Open object, with the LSP Update Capability U-flag set true.
  - The rest of these parameters are used to configure the Stateful PCE Capability TLV
- **support_pce_lsp_instantiation**
  - Sets the I-flag true, indicating the PCC allows instantiation of an LSP by a PCE.
- **support_include_db_version**

  - Sets the S-bit true, indicating the PCC will include the LSP-DB-VERSION TLV in each LSP object. See lsp_db_version below.

- **support_lsp_triggered_resync**
  - Sets the T-bit true, indicating the PCE can trigger resynchronization of LSPs at any point in the life of the session.
- **support_lsp_delta_sync**
  - Sets the D-bit true, indicating the PCEP speaker allows incremental (delta) State Synchronization.
- **support_pce_triggered_initial_sync**
  - Sets the F-bit true, indicating the PCE SHOULD trigger initial (first) State Synchronization

LSP DB Version TLV configuration parameters:

- **lsp_db_version**
  - If this parameter has a value other than 0, and the above support_include_db_version flag is true, then an LSP DB Version TLV will be added to the PCEP Open object.
  - This parameter should only be set if LSP-DB survived a restart and is available.
  - This value will be copied over to the pcep_session upon initialization.

SR PCE Capability sub-TLV configuration parameters (draft-ietf-pce-segment-routing-16):

- **support_sr_te_pst**
  - If this flag is true, then an SR PCE Capability sub-TLV will be added to a Path Setup type Capability TLV, which will be added to the PCEP Open object.
  - The PST used in the Path Setup type Capability will be 1, indicating the Path is setup using Segment Routing Traffic Engineering.

Only set the following fields if the **support_sr_te_pst** flag is true.

- **pcc_can_resolve_nai_to_sid**
  - Sets the N-flag true, indicating that the PCC is capable of resolving a Node or Adjacency Identifier to a SID
- **max_sid_depth**
  - If set other than 0, then the PCC imposes a limit on the Maximum SID depth.
  - If this parameter is other than 0, then the X bit will be true, and the parameter value will be set in the MSD field.

16.4.3 PCEPlib PCC connections

PCEPlib PCC connect and disconnect functions:

- `pcep_session *connect_pce(pcep_configuration *config, struct in_addr *pce_ip);`
- `pcep_session *connect_pce_ipv6(pcep_configuration *config, struct in6_addr *pce_ip);`
- `void disconnect_pce(pcep_session *session);`

When connecting to a PCE, a `pcep_session` will be returned on success, NULL otherwise.

Refer to the above PCC configuration parameters section for setting the source and destination PCEP TCP ports, and the source IP address and version.

16.4.4 PCEP Messages, Objects, and TLVs

The PCEP messages, objects, and TLVs created in the PCEPlib are high-level API structures, meaning they need to be encoded before being sent on-the-wire, and the raw data received needs to be decoded into these structures. This makes using these objects much easier for the library consumer, since they do not need to know the detailed raw format of the PCEP entities.

PCEP Messages

Received messages (in the `pcep_event` explained below) are of type `pcep_message`, which have the following fields:

- **`struct pcep_message_header *msg_header;`**
  - Defines the PCEP version and message type
- **`double_linked_list *obj_list;`**
  - A double linked list of the message objects
  - Each entry is a pointer to a `struct pcep_object_header`, and using the `object_class` and `object_type` fields, the pointer can be cast to the appropriate object structure to access the rest of the object fields
- **`uint8_t *encoded_message;`**
  - This field is only populated for received messages or once the `pcep_encode_message()` function has been called on the message.
  - This field is a pointer to the raw PCEP data for the entire message, including all objects and TLVs.
- **`uint16_t encoded_message_length;`**
  - This field is only populated for received messages or once the `pcep_encode_message()` function has been called on the message.
  - This field is the length of the entire raw message, including all objects and TLVs.
  - This field is in host byte order.

PCEP Objects

A PCEP message has a double linked list of pointers to `struct pcep_object_header` structures, which have the following fields:

- `enum pcep_object_classes object_class;`
- `enum pcep_object_types object_type;`
- **`bool flag_p;`**
  - PCC Processing rule bit: When set, the object MUST be taken into account, when cleared the object is optional
- **`bool flag_i;`**
  - PCE Ignore bit: indicates to a PCC whether or not an optional object was processed
- **`double_linked_list *tlv_list;`**
  - A double linked list of the object TLVs
  - Each entry is a pointer to a `struct pcep_object_tlv_header`, and using the TLV type field, the pointer can be cast to the appropriate TLV structure to access the rest of the TLV fields

- `uint8_t *encoded_object;`
  - This field is only populated for received objects or once the `pcep_encode_object()` (called by `pcep_encode_message()`) function has been called on the object.
  - Pointer into the encoded_message field (from the pcep_message) where the raw object PCEP data starts.
- `uint16_t encoded_object_length;`
  - This field is only populated for received objects or once the `pcep_encode_object()` (called by `pcep_encode_message()`) function has been called on the object.
  - This field is the length of the entire raw TLV
  - This field is in host byte order.

The object class and type can be used to cast the `struct pcep_object_header` pointer to the appropriate object structure so the specific object fields can be accessed.

PCEP TLVs

A PCEP object has a double linked list of pointers to `struct pcep_object_tlv_header` structures, which have the following fields:

- `enum pcep_object_tlv_types type;`
- `uint8_t *encoded_tlv;`
  - This field is only populated for received TLVs or once the `pcep_encode_tlv()` (called by `pcep_encode_message()`) function has been called on the TLV.
  - Pointer into the encoded_message field (from the pcep_message) where the raw TLV PCEP data starts.
- `uint16_t encoded_tlv_length;`
  - This field is only populated for received TLVs or once the `pcep_encode_tlv()` (called by `pcep_encode_message()`) function has been called on the TLV.
  - This field is the length of the entire raw TLV
  - This field is in host byte order.

Memory management

Any of the PCEPlib Message Library functions that receive a pointer to a `double_linked_list`, `pcep_object_header`, or `pcep_object_tlv_header`, transfer the ownership of the entity to the PCEPlib. The memory will be freed internally when the encapsulating structure is freed. If the memory for any of these is freed by the caller, then there will be a double memory free error when the memory is freed internally in the PCEPlib.

Any of the PCEPlib Message Library functions that receive either a pointer to a `struct in_addr` or `struct in6_addr` will allocate memory for the IP address internally and copy the IP address. It is the responsibility of the caller to manage the memory for the IP address passed into the PCEPlib Message Library functions.

For messages received via the event queue (explained below), the message will be freed when the event is freed by calling `destroy_pcep_event()`.

When sending messages, the message will be freed internally in the PCEPlib when the `send_message()` `pcep_pcc` API function when the `free_after_send` flag is set true.

To manually delete a message, call the `pcep_msg_free_message()` function. Internally, this will call `pcep_obj_free_object()` and `pcep_obj_free_tlv()` appropriately.

16.4.5 Sending a PCEP Report message

This section shows how to send a PCEP Report messages from the PCC to the PCE, and serves as an example of how to send other messages. Refer to the sample PCC binary located in `pcep_pcc/src/pcep_pcc.c` for code examples os sending a PCEP Report message.

The Report message must have at least an SRP, LSP, and ERO object.

The PCEP Report message objects are created with the following APIs:

- `struct pcep_object_srp *pcep_obj_create_srp(...);`
- `struct pcep_object_lsp *pcep_obj_create_lsp(...);`
- **`struct pcep_object_ro *pcep_obj_create_ero(...);`**
  - Create ero subobjects with the `pcep_obj_create_ro_subobj_*(...);` functions

PCEP Report message is created with the following API:

- `struct pcep_header *pcep_msg_create_report(double_linked_list *report_object_list);`

A PCEP report messages is sent with the following API:

- `void send_message(pcep_session *session, pcep_message *message, bool free_after_send);`

16.4.6 PCEPlib Received event queue

PCEP events and messages of interest to the PCEPlib consumer will be stored internally in a message queue for retrieval.

The following are the event types:

- **MESSAGE_RECEIVED**
- **PCE_CLOSED_SOCKET**
- **PCE_SENT_PCEP_CLOSE**
- **PCE_DEAD_TIMER_EXPIRED**
- **PCE_OPEN_KEEP_WAIT_TIMER_EXPIRED**
- **PCC_CONNECTED_TO_PCE**
- **PCC_CONNECTION_FAILURE**
- **PCC_PCEP_SESSION_CLOSED**
- **PCC_RCVD_INVALID_OPEN**
- **PCC_SENT_INVALID_OPEN**
- **PCC_RCVD_MAX_INVALID_MSGS**
- **PCC_RCVD_MAX_UNKOWN_MSGS**

The following PCEP messages will not be posted on the message queue, as they are handled internally in the library:

- **Open**
- **Keep Alive**
- **Close**

Received event queue API:

- **bool event_queue_is_empty();**
  - Returns true if the queue is empty, false otherwise
- **uint32_t event_queue_num_events_available();**
  - Return the number of events on the queue, 0 if empty
- **struct pcep_event *event_queue_get_event();**
  - Return the next event on the queue, NULL if empty
  - The `message` pointer will only be non-NULL if `event_type` is `MESSAGE_RECEIVED`
- **void destroy_pcep_event(struct pcep_event *event);**
  - Free the PCEP Event resources, including the PCEP message if present

16.4.7 PCEPlib Counters

The PCEPlib counters are managed in the `pcep_session_logic` library, and can be accessed in the `pcep_session_counters` field of the `pcep_session` structure. There are 2 API functions to manage the counters:

- **void dump_pcep_session_counters(pcep_session *session);**
  - Dump all of the counters to the logs
- `void reset_pcep_session_counters(pcep_session *session);`

CHAPTER

SEVENTEEN

LINK STATE API DOCUMENTATION

17.1 Introduction

The Link State (LS) API aims to provide a set of structures and functions to build and manage a Traffic Engineering Database for the various FRR daemons. This API has been designed for several use cases:

- BGP Link State (BGP-LS): where BGP protocol need to collect the link state information from the routing daemons (IS-IS and/or OSPF) to implement RFC7572
- Path Computation Element (PCE): where path computation algorithms are based on Traffic Engineering Database
- ReSerVation Protocol (RSVP): where signaling need to know the Traffic Engineering topology of the network in order to determine the path of RSVP tunnels

17.2 Architecture

The main requirements from the various uses cases are as follow:

- Provides a set of data model and function to ease Link State information manipulation (storage, serialize, parse ...)
- Ease and normalize Link State information exchange between FRR daemons
- Provides database structure for Traffic Engineering Database (TED)

To ease Link State understanding, FRR daemons have been classified into two categories:

- **Consumer**: Daemons that consume Link State information e.g. BGPd
- **Producer**: Daemons that are able to collect Link State information and send them to consumer daemons e.g. OSPFd IS-ISd

Zebra daemon, and more precisely, the ZAPI message is used to convey the Link State information between *producer* and *consumer*, but, Zebra acts as a simple pass through and does not store any Link State information. A new ZAPI **Opaque** message has been design for that purpose.

Each consumer and producer daemons are free to store or not Link State data and organise the information following the Traffic Engineering Database model provided by the API or any other data structure e.g. Hash, RB-tree ...

17.3 Link State API

This is the low level API that allows any daemons manipulate the Link State elements that are stored in the Link State Database.

17.3.1 Data structures

3 types of Link State structure have been defined:

struct **ls_node**
: that groups all information related to a node

struct **ls_attributes**
: that groups all information related to a link

struct **ls_prefix**
: that groups all information related to a prefix

These 3 types of structures are those handled by BGP-LS (see RFC7752) and suitable to describe a Traffic Engineering topology.

Each structure, in addition to the specific parameters, embed the node identifier which advertises the Link State and a bit mask as flags to indicates which parameters are valid i.e. for which the value is valid and corresponds to a Link State information conveyed by the routing protocol.

struct **ls_node_id**
: defines the Node identifier as router ID IPv4 address plus the area ID for OSPF or the ISO System ID plus the IS-IS level for IS-IS.

17.3.2 Functions

A set of functions is provided to create, delete and compare Link State Node, Atribute and Prefix:

struct *ls_node* ***ls_node_new**(struct *ls_node_id* adv, struct in_addr router_id, struct in6_addr router6_id)

struct *ls_attributes* ***ls_attributes_new**(struct *ls_node_id* adv, struct in_addr local, struct in6_addr local6, uint32_t local_id)

struct *ls_prefix* ***ls_prefix_new**(struct *ls_node_id* adv, struct prefix p)
: Create respectively a new Link State Node, Attribute or Prefix. Structure is dynamically allocated. Link State Node ID (adv) is mandatory and:

  - at least one of IPv4 or IPv6 must be provided for the router ID (router_id or router6_id) for Node
  - at least one of local, local6 or local_id must be provided for Attribute
  - prefix is mandatory for Link State Prefix.

void **ls_node_del**(struct *ls_node* *node)

void **ls_attributes_del**(struct *ls_attributes* *attr)

void **ls_prefix_del**(struct *ls_prefix* *pref)
: Remove, respectively Link State Node, Attributes or Prefix. Data structure is freed.

void **ls_attributes_srlg_del**(struct *ls_attributes* *attr)
Remove SRLGs attribute if defined. Data structure is freed.

int **ls_node_same**(struct *ls_node* *n1, struct *ls_node* *n2)

int **ls_attributes_same**(struct *ls_attributes* *a1, struct *ls_attributes* *a2)

int **ls_prefix_same**(struct *ls_prefix* *p1, struct *ls_prefix* *p2)
Check, respectively if two Link State Nodes, Attributes or Prefix are equal. Note that these routines have the same return value sense as '==' (which is different from a comparison).

17.4 Link State TED

This is the high level API that provides functions to create, update, delete a Link State Database to build a Traffic Engineering Database (TED).

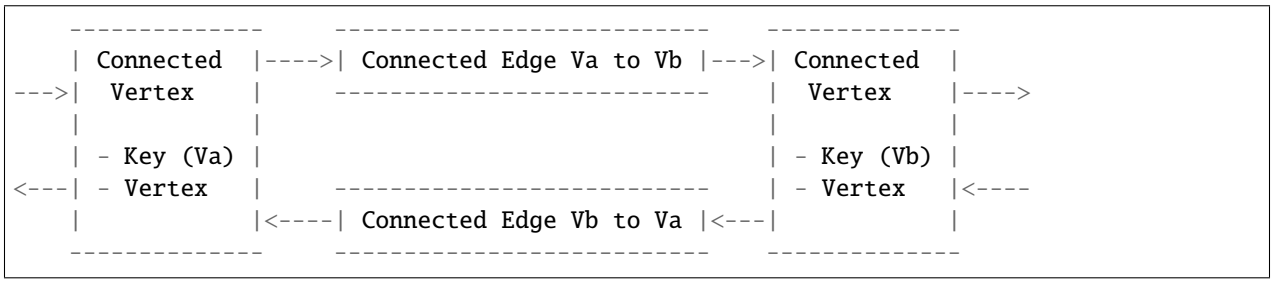
17.4.1 Data Structures

The Traffic Engineering is modeled as a Graph in order to ease Path Computation algorithm implementation. Denoted **G(V, E)**, a graph is composed by a list of **Vertices (V)** which represents the network Node and a list of **Edges (E)** which represents Link. An additional list of **prefixes (P)** is also added and also attached to the *Vertex (V)* which advertise it.

Vertex (V) contains the list of outgoing *Edges (E)* that connect this Vertex with its direct neighbors and the list of incoming *Edges (E)* that connect the direct neighbors to this Vertex. Indeed, the *Edge (E)* is unidirectional, thus, it is necessary to add 2 Edges to model a bidirectional relation between 2 Vertices. Finally, the *Vertex (V)* contains a pointer to the corresponding Link State Node.

Edge (E) contains the source and destination Vertex that this Edge is connecting and a pointer to the corresponding Link State Attributes.

A unique Key is used to identify both Vertices and Edges within the Graph.



4 data structures have been defined to implement the Graph model:

struct **ls_vertex**

struct **ls_edge**

struct **ls_ted**

- *ls_prefix*

TED stores Vertex, Edge and Subnet elements with a RB Tree structure. The Vertex key corresponds to the Router ID for OSPF and ISO System ID for IS-IS. The Edge key corresponds to the IPv4 address, the lowest 64 bits of the IPv6 address or the combination of the local & remote ID of the interface. The Subnet key corresponds to the Prefix address (v4 or v6).

An additional status for Vertex, Edge and Subnet allows to determine the state of the element in the TED: UNSET, NEW, UPDATE, DELETE, SYNC, ORPHAN. Normal state is SYNC. NEW, UPDATE and DELETE are temporary state when element is processed. UNSET is normally never used and ORPHAN serves to identify elements that must be remove when TED is cleaning.

17.4.2 Vertex, Edges and Subnets management functions

struct *ls_vertex* ***ls_vertex_add**(struct *ls_ted* *ted, struct *ls_node* *node)

struct *ls_edge* ***ls_edge_add**(struct *ls_ted* *ted, struct *ls_attributes* *attributes)

struct ls_subnet ***ls_subnet_add**(struct *ls_ted* *ted, struct *ls_prefix* *pref)
Add, respectively new Vertex, Edge or Subnet to the Link State Datebase. Vertex, Edge or Subnet are created from, respectively the Link State Node, Attribute or Prefix structure. Data structure are dynamically allocated.

struct *ls_vertex* ***ls_vertex_update**(struct *ls_ted* *ted, struct *ls_node* *node)

struct *ls_edge* ***ls_edge_update**(struct *ls_ted* *ted, struct *ls_attributes* *attributes)

struct ls_subnet ***ls_subnet_update**(struct *ls_ted* *ted, struct *ls_prefix* *pref)
Update, respectively Vertex, Edge or Subnet with, respectively the Link State Node, Attribute or Prefix. A new data structure is created if no one corresponds to the Link State Node, Attribute or Prefix. If element already exists in the TED, its associated Link State information is replaced by the new one if there are different and the old associated Link State information is deleted and memory freed.

void **ls_vertex_del**(struct *ls_ted* *ted, struct *ls_vertex* *vertex)

void **ls_vertex_del_all**(struct *ls_ted* *ted, struct *ls_vertex* *vertex)

void **ls_edge_del**(struct *ls_ted* *ted, struct *ls_edge* *edge)

void **ls_edge_del_all**(struct *ls_ted* *ted, struct *ls_edge* *edge)

void **ls_subnet_del**(struct *ls_ted* *ted, struct ls_subnet *subnet)

void **ls_subnet_del_all**(struct *ls_ted* *ted, struct ls_subnet *subnet)
Delete, respectively Link State Vertex, Edge or Subnet. Data structure are freed but not the associated Link State information with the simple *_del()* form of the function while the *_del_all()* version freed also associated Link State information. TED is not modified if Vertex, Edge or Subnet is NULL or not found in the Data Base. Note that references between Vertices, Edges and Subnets are removed first.

struct *ls_vertex* ***ls_find_vertex_by_key**(struct *ls_ted* *ted, const uint64_t key)

struct *ls_vertex* ***ls_find_vertex_by_id**(struct *ls_ted* *ted, struct *ls_node_id* id)
Find Vertex in the TED by its unique key or its Link State Node ID. Return Vertex if found, NULL otherwise.

struct *ls_edge* ***ls_find_edge_by_key**(struct *ls_ted* *ted, const uint64_t key)

struct *ls_edge* ***ls_find_edge_by_source**(struct *ls_ted* *ted, struct *ls_attributes* *attributes);

struct *ls_edge* ***ls_find_edge_by_destination**(struct *ls_ted* *ted, struct *ls_attributes* *attributes);
Find Edge in the Link State Data Base by its key, source or distination (local IPv4 or IPv6 address or local ID) informations of the Link State Attributes. Return Edge if found, NULL otherwise.

struct ls_subnet ***ls_find_subnet**(struct *ls_ted* *ted, const struct *prefix* prefix)
Find Subnet in the Link State Data Base by its key, i.e. the associated prefix. Return Subnet if found, NULL otherwise.

int **ls_vertex_same**(struct *ls_vertex* *v1, struct *ls_vertex* *v2)

int **ls_edge_same**(struct *ls_edge* *e1, struct *ls_edge* *e2)

int **ls_subnet_same**(struct ls_subnet *s1, struct ls_subnet *s2)
Check, respectively if two Vertices, Edges or Subnets are equal. Note that these routines has the same return value sense as '==' (which is different from a comparison).

17.4.3 TED management functions

Some helpers functions have been also provided to ease TED management:

struct *ls_ted* ***ls_ted_new**(const uint32_t key, char *name, uint32_t asn)
Create a new Link State Data Base. Key must be different from 0. Name could be NULL and AS number equal to 0 if unknown.

void **ls_ted_del**(struct *ls_ted* *ted)

void **ls_ted_del_all**(struct *ls_ted* *ted)
Delete existing Link State Data Base. Vertices, Edges, and Subnets are not removed with ls_ted_del() function while they are with ls_ted_del_all().

void **ls_connect_vertices**(struct *ls_vertex* *src, struct *ls_vertex* *dst, struct *ls_edge* *edge)
Connect Source and Destination Vertices by given Edge. Only non NULL source and destination vertices are connected.

void **ls_connect**(struct *ls_vertex* *vertex, struct *ls_edge* *edge, bool source)

void **ls_disconnect**(struct *ls_vertex* *vertex, struct *ls_edge* *edge, bool source)
Connect / Disconnect Link State Edge to the Link State Vertex which could be a Source (source = true) or a Destination (source = false) Vertex.

void **ls_disconnect_edge**(struct *ls_edge* *edge)
Disconnect Link State Edge from both Source and Destination Vertex. Note that Edge is not removed but its status is marked as ORPHAN.

void **ls_vertex_clean**(struct *ls_ted* *ted, struct *ls_vertex* *vertex, struct *zclient* *zclient)
Clean Vertex structure by removing all Edges and Subnets marked as ORPHAN from this vertex. Corresponding Link State Update message is sent if zclient parameter is not NULL. Note that associated Link State Attribute and Prefix are also removed and memory freed.

void **ls_ted_clean**(struct *ls_ted* *ted)
Clean Link State Data Base by removing all Vertices, Edges and SubNets marked as ORPHAN. Note that associated Link State Node, Attributes and Prefix are removed too.

void **ls_show_vertex**(struct *ls_vertex* *vertex, struct *vty* *vty, struct json_object *json, bool verbose)

void **ls_show_edge**(struct ls_edeg *edge, struct *vty* *vty, struct json_object *json, bool verbose)

void **ls_show_subnet**(struct ls_subnet *subnet, struct *vty* *vty, struct json_object *json, bool verbose)

void **ls_show_vertices**(struct *ls_ted* *ted, struct *vty* *vty, struct json_object *json, bool verbose)

void **ls_show_edges**(struct *ls_ted* *ted, struct *vty* *vty, struct json_object *json, bool verbose)

void **ls_show_subnets**(struct *ls_ted* *ted, struct *vty* *vty, struct json_object *json, bool verbose)

void **ls_show_ted**(struct *ls_ted* *ted, struct *vty* *vty, struct json_object *json, bool verbose)
Respectively, show Vertex, Edge, Subnet provided as parameter, all Vertices, all Edges, all Subnets and the whole TED if not specified. Output could be more detailed with verbose parameter for VTY output. If both JSON and VTY output are specified, JSON takes precedence over VTY.

void **ls_dump_ted**(struct *ls_ted* *ted)
Dump TED information to the current logging output.

17.5 Link State Messages

This part of the API provides functions and data structure to ease the communication between the *Producer* and *Consumer* daemons.

17.5.1 Communications principles

Recent ZAPI Opaque Message is used to exchange Link State data between daemons. For that purpose, Link State API provides new functions to serialize and parse Link State information through the ZAPI Opaque message. A dedicated flag, named ZAPI_OPAQUE_FLAG_UNICAST, allows daemons to send a unicast or a multicast Opaque message and is used as follow for the Link State exchange:

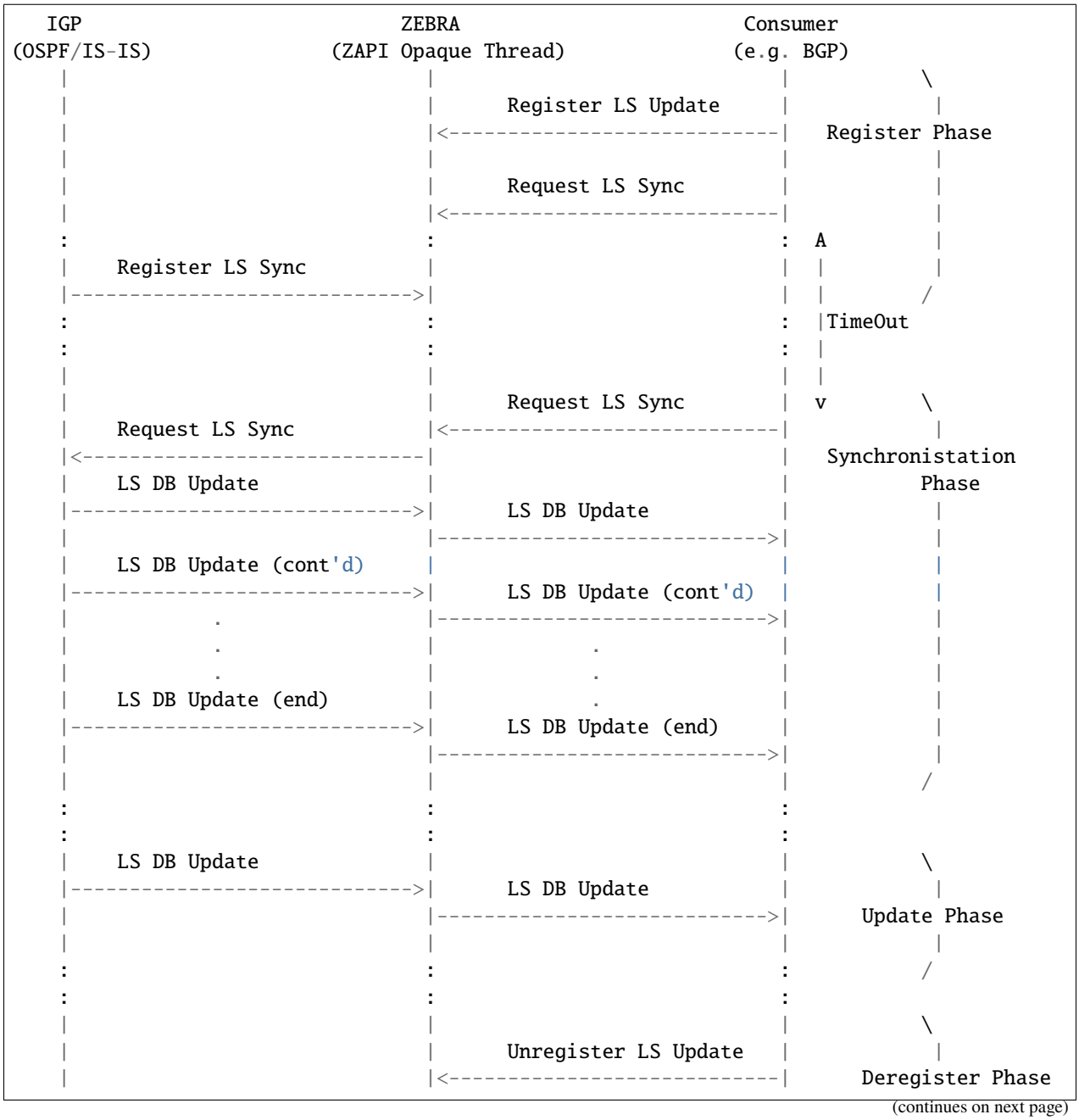
- Multicast: To send data update to all daemons that have subscribed to the Link State Update message
- Unicast: To send initial Link State information from a particular daemon. All data are send only to the daemon that request Link State Synchronisatio

Figure 1 below, illustrates the ZAPI Opaque message exchange between a *Producer* (an IGP like OSPF or IS-IS) and a *Consumer* (e.g. BGP). The message sequences are as follows:

- First, both *Producer* and *Consumer* must register to their respective ZAPI Opaque Message: **Link State Sync** for the *Producer* in order to receive Database synchronisation request from a *Consumer*, **Link State Update** for the *Consumer* in order to received any Link State update from a *Producer*. These register messages are stored by Zebra to determine to which daemon it should redistribute the ZAPI messages it receives.
- Then, the *Consumer* sends a **Link State Synchronistation** request with the Multicast method in order to receive the complete Link State Database from a *Producer*. ZEBRA daemon forwards this message to any *Producer* daemons that previously registered to this message. If no *Producer* has yet registered, the request is lost. Thus, if the *Consumer* receives no response whithin a given timer, it means that no *Producer* are available right now. So, the *Consumer* must send the same request until it receives a Link State Database Synchronistation message.

This behaviour is necessary as we can't control in which order daemons are started. It is up to the *Consumer* daemon to fix the timeout and the number of retry.

- When a *Producer* receives a **Link State Synchronisation** request, it starts sending all elements of its own Link State Database through the **Link State Database Synchronisation** message. These messages are send with the Unicast method to avoid flooding other daemons with these elements. ZEBRA layer ensures to forward the message to the right daemon.
- When a *Producer* update its Link State Database, it automatically sends a **Link State Update** message with the Multicast method. In turn, ZEBRA daemon forwards the message to all *Consumer* daemons that previously registered to this message. if no daemon is registered, the message is lost.
- A daemon could unregister from the ZAPI Opaque message registry at any time. In this case, the ZEBRA daemon stops to forward any messages it receives to this daemon, even if it was previously converns.



(continues on next page)

(continued from previous page)

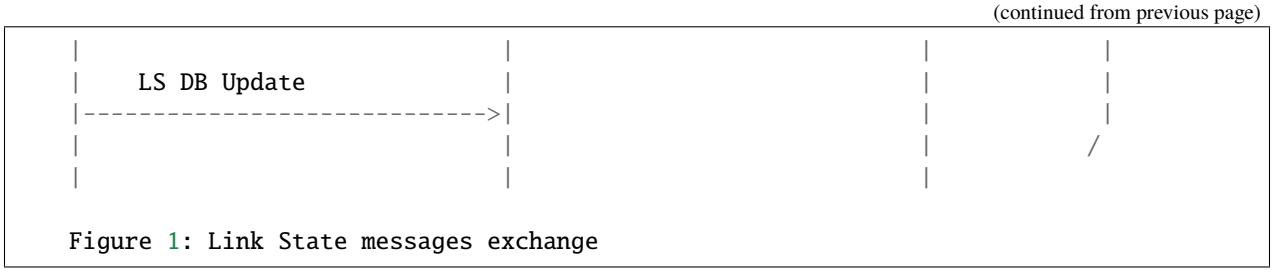


Figure 1: Link State messages exchange

17.5.2 Data Structures

The Link State Message is defined to convey Link State parameters from the routing protocol (OSPF or IS-IS) to other daemons e.g. BGP.

struct **ls_message**

The structure is composed of:

- Event of the message:
  - Sync: Send the whole LS DB following a request
  - Add: Send the a new Link State element
  - Update: Send an update of an existing Link State element
  - Delete: Indicate that the given Link State element is removed
- Type of Link State element: Node, Attribute or Prefix
- Remote node id when known
- Data: Node, Attributes or Prefix

A Link State Message can carry only one Link State Element (Node, Attributes of Prefix) at once, and only one Link State Message is sent through ZAPI Opaque Link State type at once.

17.5.3 Functions

int **ls_register**(struct *zclient* *zclient, bool server)

int **ls_unregister**(struct *zclient* *zclient, bool server)
Register / Unregister daemon to received ZAPI Link State Opaque messages. Server must be set to true for *Producer* and to false for *Consumer*.

int **ls_request_sync**(struct *zclient* *zclient)
Request initial Synchronisation to collect the whole Link State Database.

struct *ls_message* ***ls_parse_msg**(struct stream *s)
Parse Link State Message from stream. Used this function once receiving a new ZAPI Opaque message of type Link State.

void **ls_delete_msg**(struct *ls_message* *msg)
Delete existing message. Data structure is freed.

int **ls_send_msg**(struct *zclient* *zclient, struct *ls_message* *msg, struct zapi_opaque_reg_info *dst)
Send Link State Message as new ZAPI Opaque message of type Link State. If destination is not NULL, message is sent as Unicast otherwise it is broadcast to all registered daemon.

struct *ls_message* ***ls_vertex2msg**(struct *ls_message* *msg, struct *ls_vertex* *vertex)

struct *ls_message* ***ls_edge2msg**(struct *ls_message* *msg, struct *ls_edge* *edge)

struct *ls_message* ***ls_subnet2msg**(struct *ls_message* *msg, struct ls_subnet *subnet)
Create respectively a new Link State Message from a Link State Vertex, Edge or Subnet. If Link State Message is NULL, a new data structure is dynamically allocated. Note that the Vertex, Edge and Subnet status is used to determine the corresponding Link State Message event: ADD, UPDATE, DELETE, SYNC.

int **ls_msg2vertex**(struct *ls_ted* *ted, struct *ls_message* *msg)

int **ls_msg2edge**(struct *ls_ted* *ted, struct *ls_message* *msg)

int **ls_msg2subnet**(struct *ls_ted* *ted, struct *ls_message* *msg)
Convert Link State Message respectively in Vertex, Edge or Subnet and update the Link State Database accordingly to the message event: SYNC, ADD, UPDATE or DELETE.

struct ls_element ***ls_msg2ted**(struct *ls_ted* *ted, struct *ls_message* *msg, bool delete)

struct ls_element ***ls_stream2ted**(struct *ls_ted* *ted, struct *ls_message* *msg, bool delete)
Convert Link State Message or Stream Buffer in a Link State element (Vertex, Edge or Subnet) and update the Link State Database accordingly to the message event: SYNC, ADD, UPDATE or DELETE. The function return the generic structure ls_element that point to the Vertex, Edge or Subnet which has been added, updated or synchronous in the database. Note that the delete boolean parameter governs the action for the DELETE action: true, Link State Element is removed from the database and NULL is return. If set to false, database is not updated and the function sets the Link State Element status to Delete and return the element for futur deletion by the calling function.

int **ls_sync_ted**(struct *ls_ted* *ted, struct *zclient* *zclient, struct zapi_opaque_reg_info *dst)
Send all the content of the Link State Data Base to the given destination. Link State content is sent is this order: Vertices, Edges then Subnet. This function must be used when a daemon request a Link State Data Base Synchronization.

INDEX

Symbols

A

B

C

D

F

H

L

M

P

R

S

V

X

Z