



FRR Developer's Manual

Release latest

FRR

Oct 18, 2019

Contents

1	Process & Workflow	1
1.1	Mailing Lists	1
1.2	Development & Release Cycle	1
1.3	Submitting Patches and Enhancements	3
1.4	Programming Languages, Tools and Libraries	6
1.5	Code Reviews	7
1.6	Coding Practices & Style	7
1.7	Documentation	15
2	Building FRR	19
2.1	Alpine Linux 3.7+	19
2.2	CentOS 6	20
2.3	CentOS 7	25
2.4	Debian 8	28
2.5	Debian 9	31
2.6	Fedora 24+	34
2.7	FreeBSD 10	37
2.8	FreeBSD 11	40
2.9	FreeBSD 9	43
2.10	NetBSD 6	46
2.11	NetBSD 7	48
2.12	OmniOS (OpenSolaris)	51
2.13	OpenBSD 6	54
2.14	OpenWRT	57
2.15	Ubuntu 14.04 LTS	58
2.16	Ubuntu 16.04 LTS	62
2.17	Ubuntu 18.04 LTS	65
3	Packaging	71
3.1	Release Build Procedure for FRR Maintainers	71
3.2	Packaging Debian	72
3.3	Multi-Distribution builds	73
3.4	Packaging Red Hat	74
4	Process Architecture	77
4.1	Terminology	77
4.2	Event Architecture	77

4.3	Kernel Thread Architecture	80
4.4	Notes on Design and Documentation	83
5	Library Facilities (libfrr)	85
5.1	Memtypes	85
5.2	RCU	87
5.3	List implementations	91
5.4	Developer's Guide to Logging	100
5.5	Locking	103
5.6	Hooks	104
5.7	Command Line Interface	107
5.8	Modules	120
6	Testing	123
6.1	Topotests	123
7	BGPD	143
7.1	Next Hop Tracking	143
7.2	BGP-4[+] UPDATE Attribute Preprocessor Constants	148
8	OSPF	151
8.1	OSPF API Documentation	151
8.2	OSPF Segment Routing	159
9	Zebra	165
9.1	Overview of the Zebra Protocol	165
9.2	Zebra Protocol Definition	166
10	VTYSH	171
10.1	Architecture	171
10.2	Protocol	173
	Index	175

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 practices 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.frouting.org
Users & Operators	frog@lists.frouting.org
Announcements	announce@lists.frouting.org
<i>Security</i>	security@lists.frouting.org
<i>Technical Steering Committee</i>	tsc@lists.frouting.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://frouting.org/>.

1.2 Development & Release Cycle

1.2.1 Development

The master Git for FRR resides on [GitHub](#).

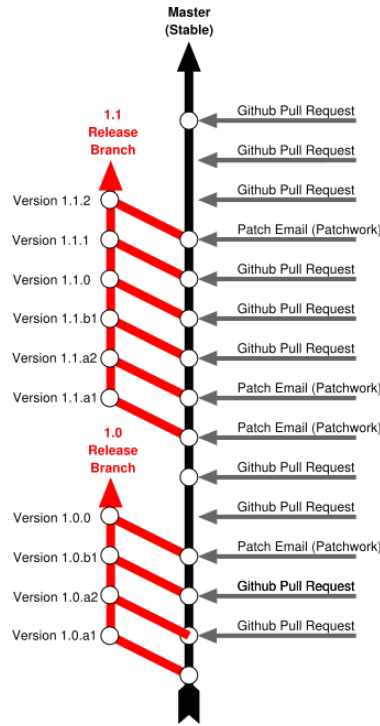


Fig. 1: Rough outline of FRR development workflow

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.

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.

We will pull a new development branch for the next release every 4 months. The current schedule is Feb/June/October 1. The decision for a MAJOR/MINOR release is made at the time of branch pull based on what has been received the previous 4 months. The branch name will be `dev/MAJOR.MINOR`. At this point in time the master branch and this new branch, `configure.ac`, documentation and packaging systems will be updated to reflect the next possible release name to allow for easy distinguishing.

After one month the development branch will be renamed to `stable/MAJOR.MINOR`. The branch is a stable branch. This process is not held up unless a crash or security issue has been found and needs to be addressed. Issues being fixed will not cause a delay.

Bugfix releases are made as needed at 1 month intervals until the next MAJOR.MINOR release branch is pulled. Depending on the severity of the bugs, bugfix releases may occur sooner.

Bugfixes are applied to the two most recent releases. However, backporting of bug fixes to older than the two most recent releases will not be prevented, if asked under the classical development workflow applying for a pull request.

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.

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 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 Submitting Patches and Enhancements

FRR accepts patches from two sources:

- Email (git format-patch)
- GitHub pull request

Contributors are highly encouraged to use GitHub's fork-and-PR workflow. It is easier for us to review it, test it, try it and discuss it on GitHub than it is via email, thus your patch will get more attention more quickly on GitHub.

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.

1.3.1 GitHub Pull Requests

The preferred method of submitting changes is a GitHub pull request. 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.

1.3.2 Patch Submission via Mailing List

As an alternative submission method, a patch can be mailed to the development mailing list. Patches received on the mailing list will be picked up by Patchwork and tested against the latest development branch.

The recommended way to send the patch (or series of NN patches) to the list is by using `git send-email` as follows (assuming they are the N most recent commit(s) in your git history):

```
git send-email -NN --annotate --to=dev@lists.frrouting.org
```

If your commits do not already contain a `Signed-off-by` line, then use the following command to add it (after making sure you agree to the Developer Certificate of Origin as outlined above):

```
git send-email -NN --annotate --signoff --to=dev@lists.frrouting.org
```

Submitting multi-commit patches as a GitHub pull request is **strongly encouraged** and increases the probability of your patch getting reviewed and merged in a timely manner.

1.3.3 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.

1.3.4 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

1.3.5 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.

`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.
```

1.3.6 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.)
 - 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.

1.4 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*.

1.4.1 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.

1.5 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.

1.5.1 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.

1.6 Coding Practices & Style

1.6.1 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.

You must also sign off on your commit.

See also:

Signing Off

1.6.2 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

1.6.3 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]
 * ...
 */
```

1.6.4 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 conflict¹ 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
```

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
<code>u_int8_t</code>	<code>uint8_t</code>
<code>u_int16_t</code>	<code>uint16_t</code>
<code>u_int32_t</code>	<code>uint32_t</code>
<code>u_int64_t</code>	<code>uint64_t</code>
<code>u_char</code>	<code>uint8_t</code> or <code>unsigned char</code>
<code>u_short</code>	<code>unsigned short</code>
<code>u_int</code>	<code>unsigned int</code>
<code>u_long</code>	<code>unsigned long</code>

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](#).

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/`
- `vttysh/`

BSD coding style applies to:

- `ldpd/`

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.

1.6.5 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_DEFINES`).

1.6.6 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.

1.6.7 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.)

1.6.8 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 instrumentation 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.

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.

1.6.9 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.)

1.6.10 CLI changes

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

1.6.11 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.

1.6.12 Miscellaneous

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

1.7 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
<code>doc/user</code>	User documentation; configuration guides; protocol overviews
<code>doc/developer</code>	Developer's documentation; API specs; datastructures; architecture overviews; project management procedure
<code>doc/manpages</code>	Source for manpages
<code>doc/figures</code>	Images and diagrams
<code>doc/extra</code>	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.

1.7.1 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:
 *     1 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.

1.7.2 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.

1.7.3 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 a combination of the `.. index::` and `.. clicmd::` directives. For example, the command `show pony` would be documented as follows:

```
.. index:: show pony
.. 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.

Configuration Snippets

When putting blocks of example configuration please use the `.. code-block:: frr` 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
!
```


2.1 Alpine Linux 3.7+

For building Alpine Linux dev packages, we use docker.

2.1.1 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>

2.1.2 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
```

2.1.3 Work with sources

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

2.1.4 Build apk packages

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

This will put the apk packages in:

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

2.1.5 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
cp /etc/frr/zebra.conf.sample /etc/frr/zebra.conf
vi /etc/frr/zebra.conf
/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
```

2.2 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

2.2.1 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.

2.2.2 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)
- fr_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)

2.2.3 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
```

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

Update rpm database & Install newer sphinx

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

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 binary libyang packages, which we offer for download [here](#).

Warning: libyang version 0.16.105 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 libpcre-dev or pcre-devel.

Note: For Debian-based systems, the official libyang package requires recent versions of swig (3.0.12) and debhelper (11) which are only available in Debian buster (10). However, libyang packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The python3-yang package will not work, but the other packages (libyang-dev is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version `0.16.x` it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.2.4 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 \
  --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-exampldir \
  --disable-ldpd \
  --enable-fpm \
  --with-pkg-git-version \
```

(continues on next page)

(continued from previous page)

```
--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 redhat/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 files

```
sudo install -p -m 755 redhat/frr.init /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
```

2.3 CentOS 7

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

2.3.1 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)

2.3.2 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 systemd-devel python-sphinx libcap-devel
```

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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.3.3 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 \
  --enable-systemd=yes \
  --disable-examplendir \
  --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 redhat/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 and redhat init files

```
sudo install -p -m 644 redhat/frr.service /usr/lib/systemd/system/frr.service
sudo install -p -m 755 redhat/frr.init /usr/lib/frr/frr
```

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
```

2.4 Debian 8

2.4.1 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)

2.4.2 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 libsystemd-dev \
  libsnmp-dev libcap-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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

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 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 \
  --enable-exampledir=/usr/share/doc/frr/examples/ \
  --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 frouting 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 frouting daemons you may see the below error due to the frouting 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
```

2.5 Debian 9

2.5.1 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 libsystemd-dev libcap-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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

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 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 \
  --enable-exampledir=/usr/share/doc/frr/examples/ \
  --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

2.5.3 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
```

2.6 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+.

2.6.1 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 systemd-devel libcap-devel
```

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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian

buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug in libyang](#).

2.6.2 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.

Note: If your platform uses `systemd`, please make sure to add `--enable-systemd=yes` to your `configure` options.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
  --prefix=/usr \
  --includedir=\${prefix}/include \
  --enable-exampledir=\${prefix}/share/doc/frr/examples \
  --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 \
```

(continues on next page)

(continued from previous page)

```
--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 individually 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 service files

```
sudo install -p -m 644 redhat/frr.service /usr/lib/systemd/system/frr.service
sudo install -p -m 755 redhat/frr.init /usr/lib/frr/frr
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
```

2.7 FreeBSD 10

2.7.1 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

2.7.2 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
```

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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

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

2.7.3 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
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
```

(continues on next page)

(continued from previous page)

```
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.

2.8 FreeBSD 11

2.8.1 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

2.8.2 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
```

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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

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

2.8.3 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
```

(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.

2.9 FreeBSD 9

2.9.1 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

2.9.2 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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.9.3 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 \
```

(continues on next page)

(continued from previous page)

```

--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.

2.10 NetBSD 6

2.10.1 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

2.10.2 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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version `0.16.x` it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.10.3 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-exampledir=/usr/pkg/share/examples/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 /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
```

2.11 NetBSD 7

2.11.1 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

2.11.2 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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.11.3 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-examplendir=/usr/pkg/share/examples/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
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
```

2.12 OmniOS (OpenSolaris)

2.12.1 OmniOS restrictions:

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

Enable IP & IPv6 forwarding

```
routeadm -e ipv4-forwarding
routeadm -e ipv6-forwarding
```

2.12.2 Install required packages

Add packages:

```
pkg install \
  developer/build/autoconf \
  developer/build/automake \
  developer/lexer/flex \
  developer/parser/bison \
  developer/object-file \
  developer/linker \
  developer/library/lint \
```

(continues on next page)

(continued from previous page)

```
developer/build/gnu-make \  
developer/gcc51 \  
library/idnkit \  
library/idnkit/header-idnkit \  
system/header \  
system/library/math/header-math \  
git libtool pkg-config
```

Add additional Solaris packages:

```
pkgadd -d http://get.opencsw.org/now  
/opt/csw/bin/pkgutil -U  
/opt/csw/bin/pkgutil -y -i texinfo  
/opt/csw/bin/pkgutil -y -i perl  
/opt/csw/bin/pkgutil -y -i libjson_c_dev  
/opt/csw/bin/pkgutil -y -i python27 py_pip python27_dev
```

Add libjson to Solaris equivalent of ld.so.conf

```
crle -l /opt/csw/lib -u
```

Add pytest:

```
pip install "pytest<5"
```

Install Sphinx::

```
pip install sphinx
```

Select Python 2.7 as default (required for pytest)

```
rm -f /usr/bin/python  
ln -s /opt/csw/bin/python2.7 /usr/bin/python
```

Fix PATH for all users and non-interactive sessions. Edit `/etc/default/login` and add the following default PATH:

```
PATH=/usr/gnu/bin:/usr/bin:/usr/sbin:/sbin:/opt/csw/bin
```

Edit `~/.profile` and add the following default PATH:

```
PATH=/usr/gnu/bin:/usr/bin:/usr/sbin:/sbin:/opt/csw/bin
```

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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

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

```
sudo groupadd -g 93 frr
sudo groupadd -g 94 frrvty
sudo useradd -g 93 -u 93 -G frrvty -c "FRR suite" \
  -d /nonexistent -s /bin/false frr
```

(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/opt/csw/lib"
export CPPFLAGS="-I/opt/csw/include"
export PKG_CONFIG_PATH=/opt/csw/lib/pkgconfig
./configure \
  --sysconfdir=/etc/frr \
  --enable-exampledir=/usr/share/doc/frr/examples/ \
  --localstatedir=/var/run/frr \
  --sbindir=/usr/lib/frr \
  --enable-multipath=64 \
```

(continues on next page)

(continued from previous page)

```
--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
```

Enable IP & IPv6 forwarding

```
routeadm -e ipv4-forwarding  
routeadm -e ipv6-forwarding
```

2.13 OpenBSD 6

2.13.1 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 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 binary `libyang` packages, which we offer for download [here](#).

```
Warning: libyang version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.13.2 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)

```
git clone https://github.com/frouting/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 \
```

(continues on next page)

(continued from previous page)

```

--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
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
```

2.14 OpenWRT

2.14.1 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](#).

2.14.2 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
cd feeds/routing
git fetch origin pull/319/head
git read-tree --prefix=frr/ -u FETCH_HEAD:frr
cd ../../package/feeds/routing/
ln -sv ../../../../feeds/routing/frr .
cd ../../..
```

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.

2.14.3 Work with sources

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

2.14.4 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`

2.15 Ubuntu 14.04 LTS

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

2.15.1 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
```

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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.15.2 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.

Note: If your platform uses `systemd`, please make sure to add `--enable-systemd=yes` to your configure options.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
  --prefix=/usr \
  --includedir=\${prefix}/include \
  --enable-exampledir=\${prefix}/share/doc/frr/examples \
  --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_ip_tunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-ip_tunnel
```

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.

2.16 Ubuntu 16.04 LTS

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

2.16.1 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 libsystemd-dev python-ipaddress python3-sphinx \
    install-info build-essential libsystemd-dev libsnpmp-dev perl libcap-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 binary `libyang` packages, which we offer for download [here](#).

Warning: `libyang` version 0.16.105 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 `libpcre-dev` or `pcre-devel`.

Note: For Debian-based systems, the official `libyang` package requires recent versions of `swig` (3.0.12) and `debhelper` (11) which are only available in Debian buster (10). However, `libyang` packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The `python3-yang` package will not work, but the other packages (`libyang-dev` is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing `cmake` and `libpcre-dev` or `pcre-devel`.

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

When building `libyang` version `0.16.x` it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to `cmake` to work around a [known bug](#) in `libyang`.

2.16.2 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.

Note: If your platform uses `systemd`, please make sure to add `--enable-systemd=yes` to your `configure` options.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
  --prefix=/usr \
  --includedir=${prefix}/include \
  --enable-exampledir=${prefix}/share/doc/frr/examples \
  --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 \
```

(continues on next page)

(continued from previous page)

```
--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 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_ip tunnel
```

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
```

2.17 Ubuntu 18.04 LTS

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

2.17.1 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 python3-pytest \
  libc-ares-dev python3-dev libsystemd-dev python-ipaddress python3-sphinx \
  install-info build-essential libsystemd-dev libsnmp-dev perl libcap-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 binary `libyang` packages, which we offer for download [here](#).

Warning: libyang version 0.16.105 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 libpcre-dev or pcre-devel.

Note: For Debian-based systems, the official libyang package requires recent versions of swig (3.0.12) and debhelper (11) which are only available in Debian buster (10). However, libyang packages built on Debian buster can be installed on both Debian jessie (8) and Debian stretch (9), as well as various Ubuntu systems. The python3-yang package will not work, but the other packages (libyang-dev is the one needed for FRR) will.

Option 2: Source Install

Note: Ensure that the [libyang build requirements](#) are met before continuing. Usually this entails installing cmake and libpcre-dev or pcre-devel.

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

When building libyang version 0.16.x it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake to work around a [known bug](#) in libyang.

Protobuf

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

ZeroMQ

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

2.17.2 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/ \
```

(continues on next page)

(continued from previous page)

```
--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.

Note: If your platform uses `systemd`, please make sure to add `--enable-systemd=yes` to your configure options.

```
git clone https://github.com/frrouting/frr.git frr
cd frr
./bootstrap.sh
./configure \
  --prefix=/usr \
  --includedir=\${prefix}/include \
  --enable-exampledir=\${prefix}/share/doc/frr/examples \
  --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_ip tunnel
```

And load the kernel modules on the running system:

```
sudo modprobe mpls-router mpls-ip tunnel
```

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
```


3.1 Release Build Procedure for FRR Maintainers

1. Rename branch (if needed)

```
git clone git@github.com:FRRouting/frr.git
cd frr
git checkout dev/5.0
git push origin :refs/heads/dev/5.0
git push origin dev/5.0:refs/heads/stable/5.0
```

2. Checkout the new stable branch:

```
git checkout stable/5.0
```

3. Update Changelog for RedHat Package:

Edit `redhat/frr.spec.in` and look for the `%changelog` section:

- Change last (top of list) entry from `%{version}` to previous fixed version number, i.e.:

```
* Tue Nov 7 2017 Martin Winter <mwinter@opensourcerouting.org> - %{version}
```

to:

```
* Tue Nov 7 2017 Martin Winter <mwinter@opensourcerouting.org> - 3.0.2
```

- Add new entry to the top of the list with `%{version}` tag and changelog for version. 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.

4. Update Changelog for Debian Packages:

Edit `debian/changelog-auto.in`:

- Change last (top of list) entry from `@VERSION@` to previous fixed version number, i.e.:

```
frr (@VERSION@) RELEASED; urgency=medium
```

to:

```
frr (3.0.2) RELEASED; urgency=medium
```

- Add a new entry to the top of the list with a @VERSION@ tag and changelog for version.
5. Change main version number:
 - Edit `configure.ac` and change version in the `AC_INIT` command
 - Create a new entry with the version as `%{version}` tag
 6. Test building at least a Red Hat and Ubuntu package (or create a PR to have the CI system test them)
 7. Commit the changes, adding the changelog to the commit message
 8. Create a git tag for the version:

```
git tag -a frr-5.0 -m "FRRouting Release 5.0"
```

9. Push the commit and tag(s) and watch for errors on CI:

```
git push  
git push --tags
```

10. Kick off the Release build plan on the CI system for the correct release
11. Send a Release Announcement with changes to `announce@lists.frrouting.org`
12. Kick off the Snapcraft build plan for the correct release
13. After CI plans succeed, release on GitHub by going to <https://github.com/FRRouting/frr/releases> and selecting “Draft a new release”.
14. Deploy Snapcraft release (after CI system finishes the tests for snapcraft testplan)

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 debian/control
```

Alternatively, you can manually install build dependencies for your platform as outlined in *Building FRR*.

4. Run `tools/tarsource.sh -V`:

```
./tools/tarsource.sh -V
```

This script sets up the `debian/changelog-auto` file with proper version information.

5. (optional) Append a distribution identifier if needed (see below under *Multi-Distribution builds*.)
6. Build Debian Package:

```
dpkg-buildpackage $options
```

Where *\$options* may contain any or all of the following items:

- build profiles specified with `-P`, e.g. `-Ppkg.frr.nortrlib, pkg.frr.nosystemd`. 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
<code>pkg.frr.rtrlib</code>	<code>pkg.frr.nortrlib</code>	builds <code>frr-rpki-rtrlib</code> package (or not)
<code>n/a</code>	<code>pkg.frr.nosystemd</code>	removes <code>libsystemd</code> dependency and disables unit file installation

Note: The `pkg.frr.nosystemd` option is only intended to support Ubuntu 14.04 (and should be enabled when building for that.)

- 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.)

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.

Furthermore, official Debian packages are built in `3.0 (quilt)` format with an “orig” tarball and a “debian” tarball. These tarballs are created by the `tarsource.sh` tool on any branch. The git repository however contains a `3.0 (git)` source format specifier to easily allow direct git builds.

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. Do the following after creating the changelog with *tarsource.sh*:

```
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 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
```

If your platform uses systemd:

```
yum install systemd-devel
```

If yum is not present on your system, use dnf instead.

3. Checkout FRR:

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

4. 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`.

5. 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/
```

6. 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_rpki:         %global   with_rpki             0 }
```

7. 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`.

Process Architecture

FRR inherited its overall design architecture from Quagga. The chosen model for Quagga is that of a suite of independent daemons that do IPC via Unix domain sockets. Within each daemon, the architecture follows the event-driven model. FRR has inherited this model as well. As FRR is deployed at larger scales and gains ever more features, each adding to the overall processing workload, we are approaching the saturation point for a single thread per daemon. In light of this, there are ongoing efforts to introduce multithreading to various components of FRR. This document aims to describe the current design choices and overall model for integrating the event-driven and multithreaded architectures into a cohesive whole.

4.1 Terminology

Because this document describes the architecture for true kernel threads as well as the event system, a digression on terminology is in order here.

Historically Quagga's event 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 context datastructure in this system is called a "threadmaster". 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 is POSIX threads.

4.2 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`.

4.2.1 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`

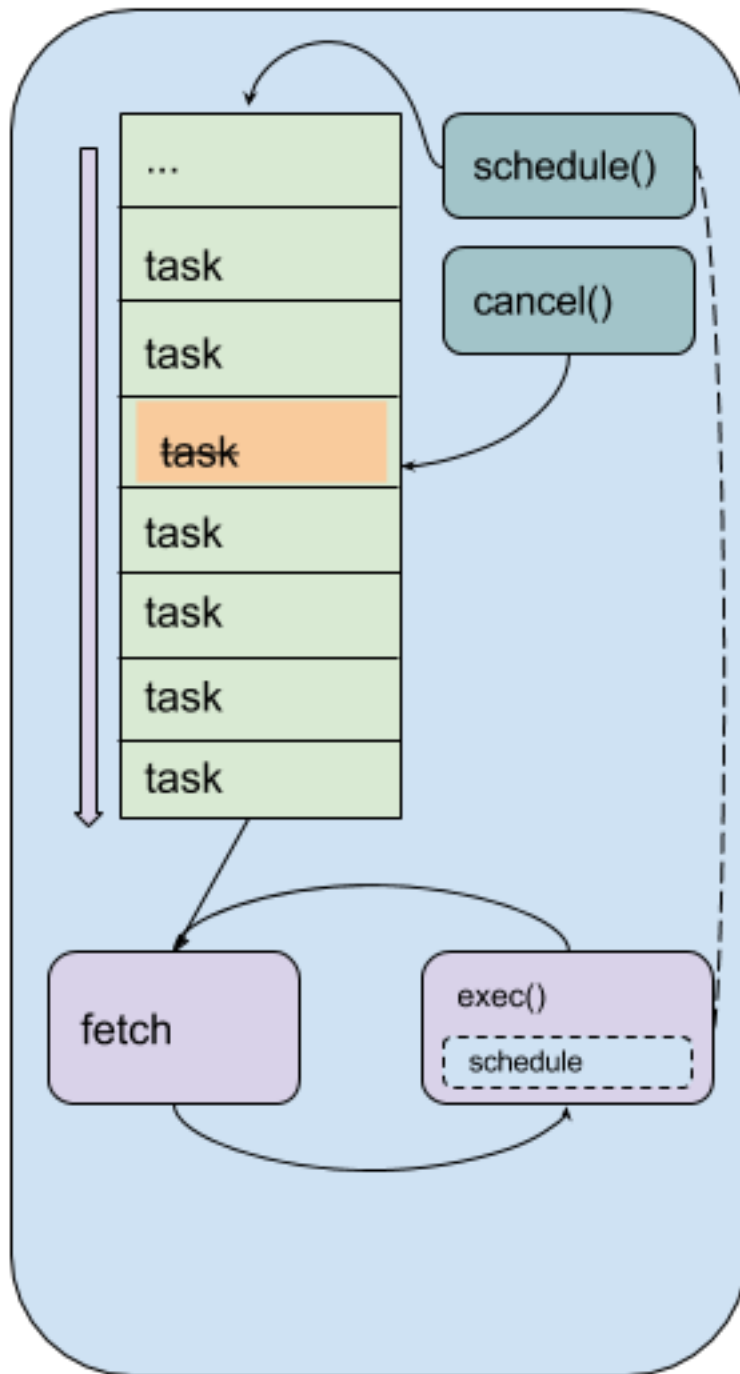


Fig. 1: Lifecycle of a program using a single threadmaster.

- `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 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.

4.3 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.

4.3.1 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.

4.3.2 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

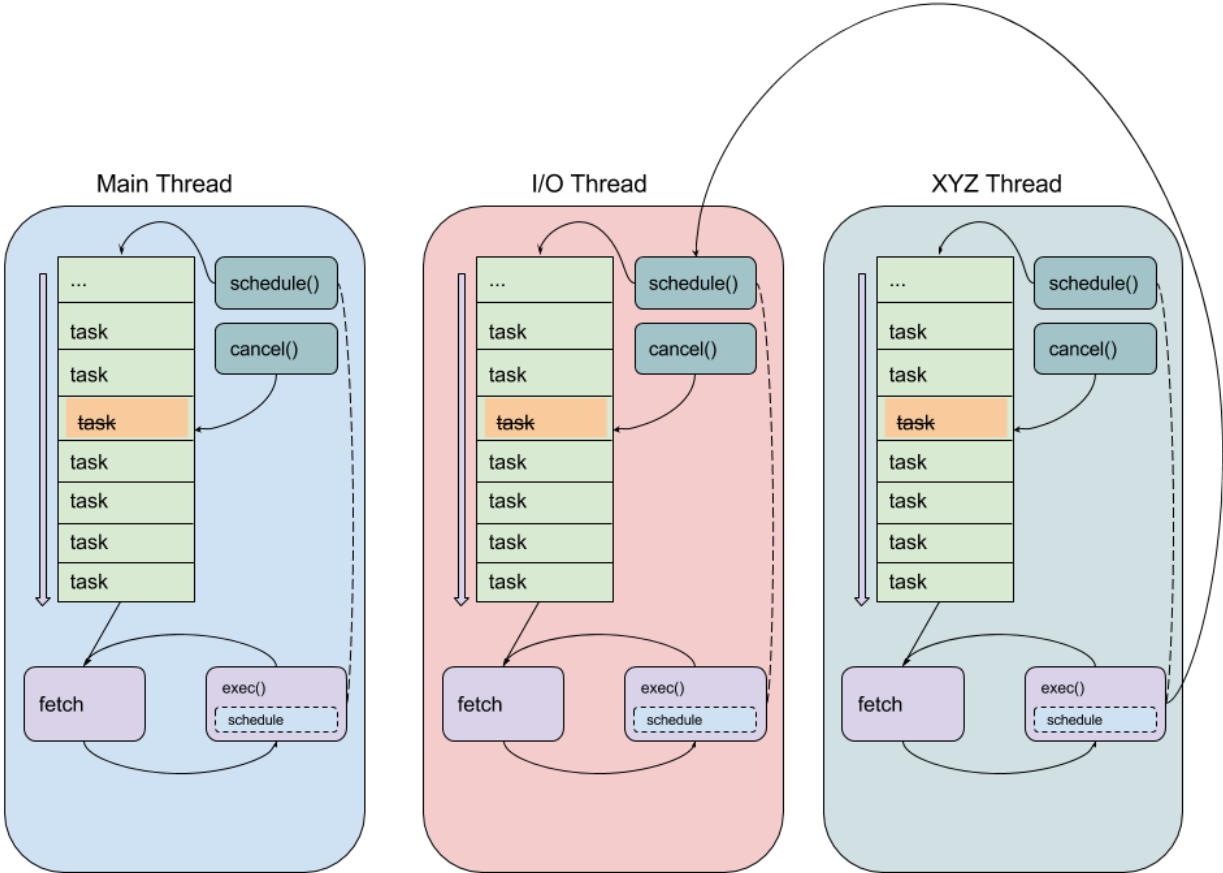


Fig. 2: Lifecycle of a program using multiple pthreads, each running their own threadmaster

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, 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
↔coalesce_timer
  0      1389.000      10  138900  248000  135549  255349  T  subgroup_
↔startup_timer_expire
  0         0.000         1     0         0         18         18  T  bgp_
↔run
  0         0.000         0     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
```

(continues on next page)

(continued from previous page)

```

    0          15.000          600          25          2000          15          153  T  (bgp_
↪routeadv_timer)
    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.

4.4 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.

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 MYPES. 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 MYPES 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.

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 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 queuing 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 List implementations

Note: The term *list* is used generically for lists, skiplists, trees and hash tables in this document.

5.3.1 Common list interface

FRR includes a set of list-like data structure 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 lists and a similar but not identical API for sorted lists - and heaps use a middle ground of both.

For unsorted lists, 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 lists, 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 list 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 list, 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
<code>_init, _fini</code>	yes	yes	yes	yes	yes
<code>_first, _next, _next_safe</code>	yes	yes	yes	yes	yes
<code>_add_head, _add_tail, _add_after</code>	yes	–	–	–	–
<code>_add</code>	–	yes	yes	yes	yes
<code>_del, _pop</code>	yes	yes	yes	yes	yes
<code>_find</code>	–	–	yes	yes	–
<code>_find_lt, _find_gteq</code>	–	–	–	yes	yes
use with <code>frr_each()</code> 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 list. 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)

```

(continues on next page)

(continued from previous page)

```

/* 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 list needs to be accessed from several C files) or it can be placed in a .c file (if the list 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 list 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 list, 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.

5.3.5 Common API

The following documentation assumes that a list has been defined using `Z` as the name, and `itemtype` being the type of the list items (e.g. `struct item`).

void **Z_init** (struct `Z_head` *)

Initializes the list 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 list must be empty when this function is called.

Warning: This function may `assert()` if the list is not empty.

size_t **Z_count** (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 lists with concurrent access, the value will already be outdated by the time this function returns and can therefore only be used as an estimate.

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 $O(\log n)$.

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 list 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.

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`.

itemtype ***Z_del** (struct Z_head *, itemtype *item)
Remove *item* from the list and return it.

Note: This function's behaviour is undefined if *item* is not actually on the list. Some structures return NULL in this case while others return *item*. The function may also call `assert()` (but most don't.)

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)`

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 lists. Also, sorted lists 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 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.
- **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 list is only used in one file.

DECLARE_XXX_NONUNIQ (*Z*, *type*, *field*, *compare_func*)

Same as above, but allow adding multiple items to the list that compare as equal in `compare_func`. Ordering between these items is undefined and depends on the list implementation.

`itemtype *Z_add` (`struct Z_head *`, `itemtype *item`)

Insert an item at the appropriate sorted position. If another item exists in the list that compares as equal (`compare_func() == 0`), `item` is not inserted into the list and the already-existing item in the list is returned. Otherwise, on successful insertion, `NULL` is returned.

For `_NONUNIQ` lists, this function always returns `NULL` since `item` can always be successfully added to the list.

`itemtype *Z_find` (`struct Z_head *`, `const itemtype *ref`)

Search the list 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 lists that contain unique items (i.e. `DECLARE_XXX_UNIQ`.) This is because on a list containing non-unique items, more than one item may compare as equal to the item that is searched for.

`itemtype *Z_find_gteq` (`struct Z_head *`, `const itemtype *ref`)

Search the list for an item that compares greater or equal to `ref`. See `Z_find()` above.

itemtype ***Z_find_lt** (struct Z_head *, const itemtype *ref)
 Search the list for an item that compares less than *ref*. See *Z_find()* above.

5.3.8 API for hash tables

DECLARE_XXX (Z, type, field, compare_func, hash_func)

Parameters

- **xxx** (*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 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.
- **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 list 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.

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 accesses (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);
```

(continues on next page)

(continued from previous page)

```
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);
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_writelock(&itemhead_rwlock);
pthread_rwlock_unlock(&itemhead_rwlock);
/* i now guaranteed to be gone from the list.
 * note nothing between writelock() and unlock() */
XFREE(MTYPE_ITEM, i);
```

5.3.11 FAQ

Why is the list head not `const` in the list APIs? The semantics that a `const` list head would imply are not obvious. It could mean any of the following:

- the list just shouldn't be allocated/deallocated, but may be modified. This doesn't actually work since the list head needs to be modified for inserting or deleting items.
- the list shouldn't be modified, but items can. This may make sense for iterating, but it's not exactly consistent - an item might be on more than one list, does it apply to all of them? If not, which one?
- neither the list nor the items should be modified. This is consistent, but hard to do without creating a `const` copy of every single list function. Ease of use trumps this.

Why is there no "is this item on a/the list" test? It's slow for several of the data structures, and the work of adding it just hasn't been done. It can certainly be added if it's needed.

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 Developer's Guide to 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 data-structures. 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);
```

Extensions

`printfrr()` format strings can be extended with suffixes after `%p` or `%d`. The following extended format specifiers are available:

Specifier	Argument	Output
%Lu	uint64_t	12345
%Ld	int64_t	-12345
%pI4	struct in_addr * in_addr_t *	1.2.3.4
%pI6	struct in6_addr *	fe80::1234
%pFX	struct prefix *	fe80::1234/64
%pSG4	struct prefix_sg *	(* , 1.2.3.4)
%pRN	struct route_node *	192.168.1.0/24 (dst-only node) 2001:db8::/32 from fe80::/64 (SADR node)
%pNHv	struct nexthop *	1.2.3.4, via eth0
%pNHs	struct nexthop *	1.2.3.4 if 15

Printf features like field lengths can be used normally with these extensions, e.g. `%-15pI4` works correctly.

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 above may become outdated.

`%Ld` is not an “extension” for `printfrr`; it’s wired directly into the main `printf` logic.

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.

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 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*

5.4.3 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.

5.4.4 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

5.5 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 (`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)
            /* 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 */
    }
}
```

(continues on next page)

(continued from previous page)

```

    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 (`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 be 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()`.

5.6 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 implementation 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.

5.6.1 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.

5.6.2 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.)

5.6.3 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.

5.6.4 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.

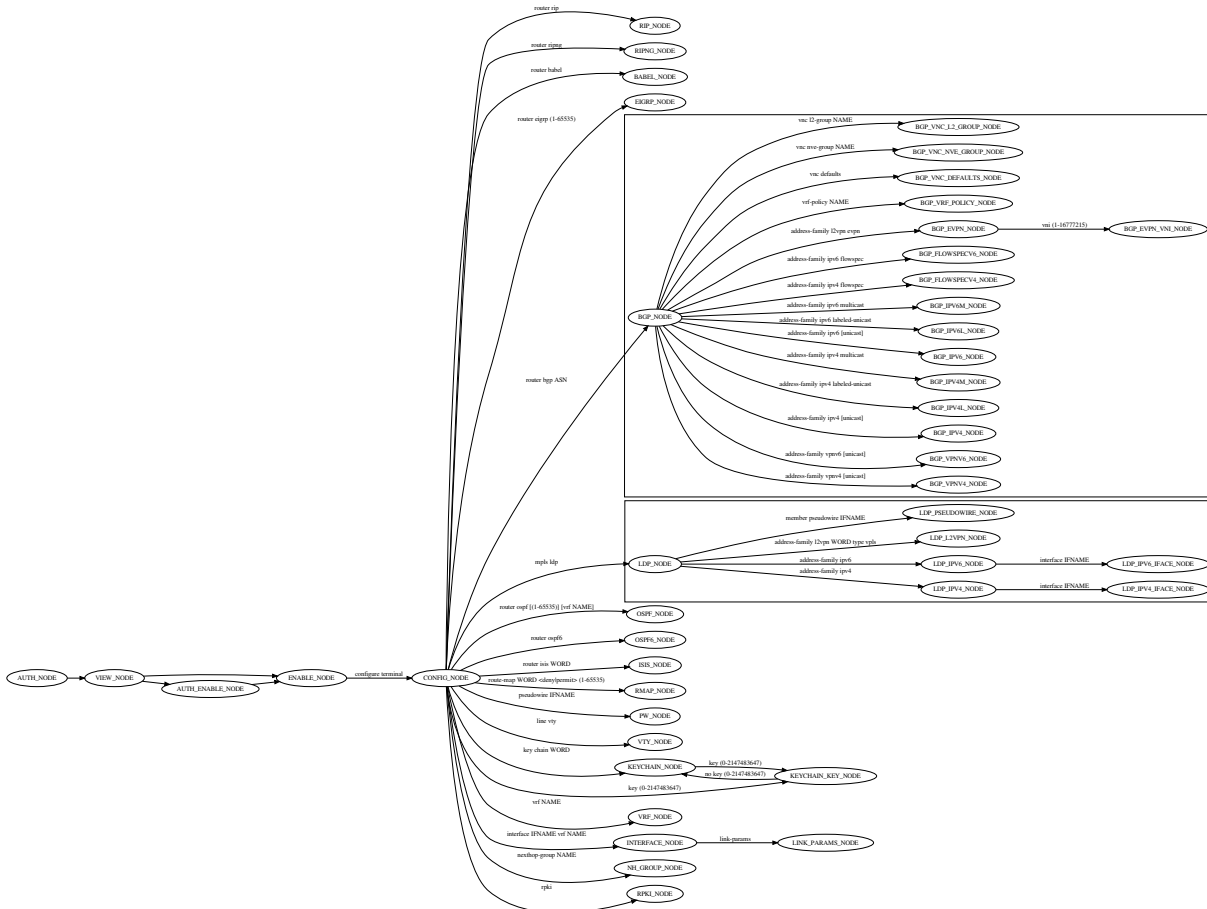
5.7 Command Line Interface

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.

5.7.1 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.

5.7.2 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.

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_lex.l` and `lib/command_parse.y`, 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      ::=  selector_seq_seq "|" selector_token_seq
selector_token_seq    ::=  selector_token_seq 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]`.

- `{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 *	NULL
X:X::X:X/M	const struct prefix_ipv6 *	NULL
A.B.C.D/M + X:X::X:X/M	const struct prefix *	NULL
(0-9)	long	0
VARIABLE	const char *	NULL
word	const char *	NULL
<i>all other</i>	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`.
- Pointers will be `NULL` when the argument is optional and the user did not use it.
- 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`, `vtty`, `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 ...
...
#ifdef VTYSH_EXTRACT_PL
#include "daemon/filename_clippy.c"
#endif

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
```

5.7.3 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.

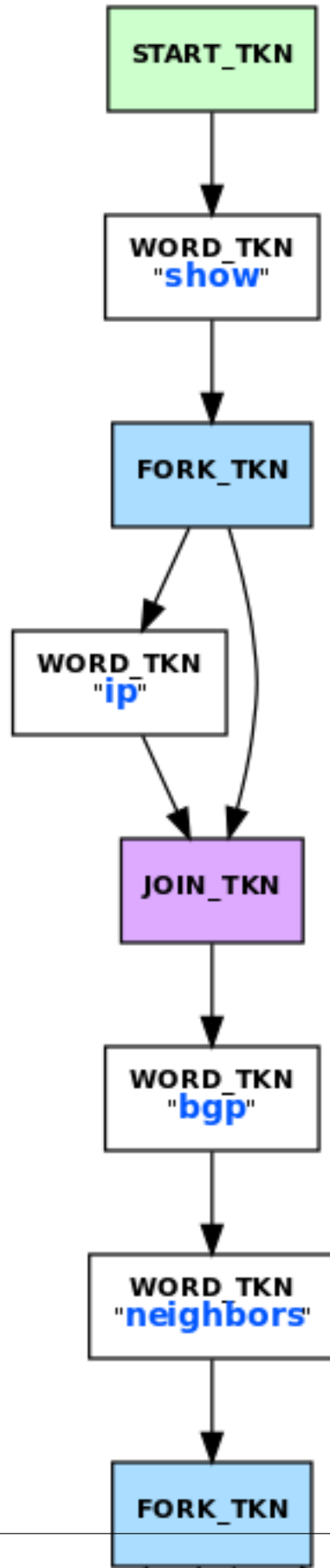
Keywords follow the same form, except that there is an edge from `JOIN` to `FORK`. Because of this the CLI graph cannot be called acyclic. There is special logic in the input matching code that keeps a stack of paths already taken through the node in order to disallow following the same path more than 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 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 */

```

(continues on next page)

(continued from previous page)

```

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.

5.7.4 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.

5.7.5 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.

5.8 Modules

FRR has facilities to load DSOs at startup via `dlopen()`. These are used to implement modules, such as SNMP and FPM.

5.8.1 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.

5.8.2 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 `snappcraft` (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`.

5.8.3 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
```

(continues on next page)

(continued from previous page)

```

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.

5.8.4 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.

5.8.5 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 `zfpd_init()` function inside `zebra_fpm.c`.

5.8.6 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.

5.8.7 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.

6.1 Topotests

Topotests is a suite of topology tests for FRR built on top of Mininet.

6.1.1 Installation and Setup

Only tested with Ubuntu 16.04 and Ubuntu 18.04 (which uses Mininet 2.2.x).

Instructions are the same for all setups (i.e. ExaBGP is only used for BGP tests).

Installing Mininet Infrastructure

```
apt-get install mininet
apt-get install python-pip
apt-get install iproute
pip install ipaddr
pip install "pytest<5"
pip install exabgp==3.4.17 (Newer 4.0 version of exabgp is not yet
supported)
useradd -d /var/run/exabgp/ -s /bin/false exabgp
```

Enable Coredumps

Optional, will give better output.

```
apt-get install gdb
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.

FRR Installation

FRR needs to be installed separately. It is assumed 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 core dump 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-multipath=64 \
  --enable-user=frr \
  --enable-group=frr \
  --enable-vty-group=frrvty \
  --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
```


6.1.2 Executing Tests

Execute all tests with output to console

```
py.test -s -v --tb=no
```

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).

`--tb=no` disables the python traceback which might be irrelevant unless the test script itself is debugged.

Execute single test

```
cd test_to_be_run
./test_to_be_run.py
```

For example, and assuming you are inside the `fr` 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.

If you need to clear the mininet setup between tests (if it isn't cleanly shutdown), then use the `mn -c` command to clean up the environment.

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 \
  --enable-exampledir=/usr/lib/frr/examples \
  --with-module-dir=/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.

6.1.3 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 pattern, 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
```

6.1.4 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 Mininet requires it.
$ sudo pytest
```

In order to run a specific test, you can use the following command:

```
$ # running a specific topology
$ sudo pytest ospf-topol/
$ # or inside the test folder
$ cd ospf-topol
```

(continues on next page)

(continued from previous page)

```
$ sudo pytest # to run all tests inside the directory
$ sudo pytest test_ospf_topol.py # to run a specific test
$ # or outside the test folder
$ cd ..
$ sudo pytest ospf-topol/test_ospf_topol.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-topol.test_ospf-topol/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 env TOPOTESTS_CHECK_MEMLEAK="/tmp/memleak_report_" pytest ospf-topol/test_ospf_
↳topol.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_topol.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
- Create a Pull Request

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
```

(continues on next page)

(continued from previous page)

```

./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 confest.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,

```

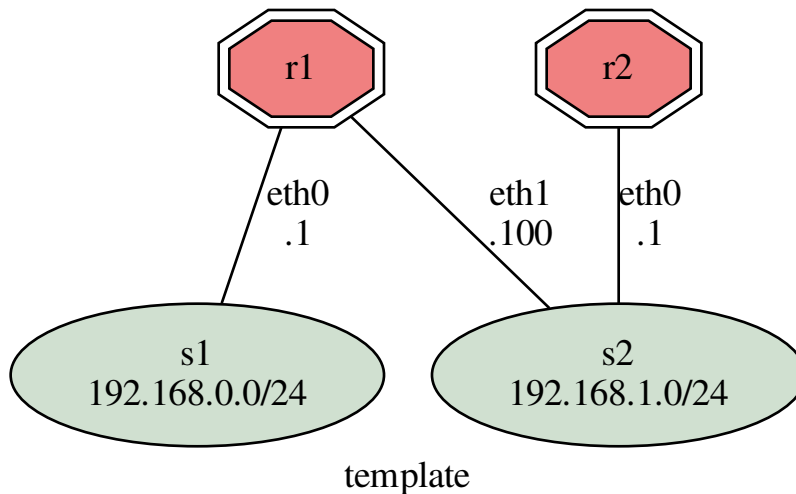
(continues on next page)

(continued from previous page)

```
    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 `vt ysh`. 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:

```
class TemplateTopo(Topo):
    "Test topology builder"
    def build(self, *_args, **_opts):
        "Build function"
        tgen = get_topogen(self)

        # Create 2 routers
        for routern in range(1, 3):
            tgen.add_router('r{}'.format(routern))

        # 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 bootstrapped topology
$ sudo pytest -s --topology-only new-topo/test_new_topo.py
$ # Running the test_template.py topology
$ sudo pytest -s --topology-only example-test/test_template.py
$ # Running the ospf_topo1.py topology
$ sudo pytest -s --topology-only ospf-topo1/test_ospf_topo1.py
```

Parameters explanation:

-s

Activates input/output capture. This is required by mininet in order to show the interactive shell.

--topology-only

Don't run any tests, just build the topology.

After executing the commands above, you should get the following terminal output:

```
=== test session starts ===
platform linux2 -- Python 2.7.12, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
rootdir: /media/sf_src/topotests, inifile: pytest.ini
collected 3 items

ospf-topo1/test_ospf_topo1.py *** Starting controller

*** Starting 6 switches
switch1 switch2 switch3 switch4 switch5 switch6 ...
```

(continues on next page)

(continued from previous page)

```
r2: frr zebra started
r2: frr ospfd started
r3: frr zebra started
r3: frr ospfd started
r1: frr zebra started
r1: frr ospfd started
r4: frr zebra started
r4: frr ospfd started
*** Starting CLI:
mininet>
```

The last line shows us that we are now using the Mininet CLI (Command Line Interface), from here you can call your router vtysh or even bash.

Here are some commands example:

```
mininet> 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

mininet> r1 ping 10.0.3.3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=64 time=2.87 ms
64 bytes from 10.0.3.3: icmp_seq=2 ttl=64 time=0.080 ms
64 bytes from 10.0.3.3: icmp_seq=3 ttl=64 time=0.091 ms
^C
--- 10.0.3.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.080/1.014/2.872/1.313 ms

mininet> r3 vtysh

Hello, this is FRRouting (version 3.1-devrzalamena-build).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

frr-1# show running-config
Building configuration...

Current configuration:
!
frr version 3.1-devrzalamena-build
frr defaults traditional
hostname r3
no service integrated-vtysh-config
!
log file zebra.log
!
```

(continues on next page)

(continued from previous page)

```
log file ospfd.log
!
interface r3-eth0
 ip address 10.0.3.1/24
!
interface r3-eth1
 ip address 10.0.10.1/24
!
interface r3-eth2
 ip address 172.16.0.2/24
!
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
frr-1#
```

After you successfully configured your topology, you can obtain the configuration files (per-daemon) using the following commands:

```
mininet> r3 vtysh -d ospfd

Hello, this is FRRouting (version 3.1-devrzalamena-build).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

frr-1# show running-config
Building configuration...

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
frr-1#
```

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:

- imports: `os`, `sys`, `pytest`, `topotest/topogen` and `mininet` topology class
- The global variable `CWD` (Current Working directory): which is most likely going to be used to reference the routers configuration file location

Example:

```
# For all registered routers, load the zebra configuration file
for rname, router in router_list.iteritems():
    router.load_config(
        TopoRouter.RD_ZEBRA,
        os.path.join(CWD, '{}zebra.conf'.format(rname))
    )
# os.path.join() joins the CWD string with arguments adding the necessary
# slashes ('/'). Arguments must not begin with '/'.
```

- The topology class that inherits from `Mininet Topo` class:

```
class TemplateTopo(Topo):
    def build(self, *_args, **_opts):
        tgen = get_topogen(self)
        # topology build code
```

- `pytest setup_module()` and `teardown_module()` to start the topology

```
def setup_module(_m):
    tgen = Topogen(TemplateTopo)
    tgen.start_topology('debug')

def teardown_module(_m):
    tgen = get_topogen()
    tgen.stop_topology()
```

- `__main__` initialization code (to support running the script directly)

```
if __name__ == '__main__':
    sys.exit(pytest.main(["-s"]))
```

Requirements:

- 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.

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-topol/test_ospf_topol.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-topol/test_ospf_topol.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
```

(continues on next page)

(continued from previous page)

```

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-topol/test_ospf_topol.py(122)test_ospf_convergence()
-> router = 'r{}'.format(rnum)
(Pdb) step
> /media/sf_src/topotests/ospf-topol/test_ospf_topol.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<>'
(Pdb) 1 125
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\n\r\n      10.0.1.0/24
-> [10] area: 0.0.0.0\r\n\r\n      directly attached to r1-eth0\r\n\r\n
-> 10.0.2.0/24      [20] area: 0.0.0.0\r\n\r\n      via 10.0.3.
-> 3, r1-eth1\r\n\r\n      10.0.3.0/24      [10] area: 0.0.0.0\r\n\r\n      (continues on next page)
->      directly attached to r1-eth1\r\n\r\n      10.0.10.0/24      [20] area: 0.0.0.
-> 0\r\n\r\n      via 10.0.3.1, r1-eth1\r\n\r\n IA 172.16.0.0/24
136 [20] area: 0.0.0.0\r\n\r\n      via 10.0.3.1, r1-eth1\r\n\r\n IA 172.16.
-> 16.1.0/24      [30] area: 0.0.0.0\r\n\r\n      via 10.0.3.1, r1-
-> eth1\r\n\r\n\r\n=====OSPF router routing table=====\r\n\r\n      10.0.255.2
->      [10] area: 0.0.0.0, ASBR\r\n\r\n      via 10.0.3.3, r1-

```

(continued from previous page)

```
(Pdb) tgen.mininet_cli()
*** Starting CLI:
mininet>
```

To enable more debug messages in other Topogen subsystems (like Mininet), 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`).

6.1.5 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
!
""")
```

(continues on next page)

(continued from previous page)

```
"""
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
  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__))
```

(continues on next page)

(continued from previous page)

```
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
```

6.1.6 License

All the configs and scripts are licensed under a ISC-style license. See Python scripts for details.

7.1 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.

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 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
<code>bgp_find_or_add_nexthop()</code>	find or add a nexthop in BGP nexthop table
<code>bgp_find_nexthop()</code>	find a nexthop in BGP nexthop table
<code>bgp_parse_nexthop_update()</code>	parse a nexthop update message coming from zebra

The “rnh” module provides the following APIs:


```

.  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                               .
.
+-----+-----+-----+-----+
.
.
+-----+-----+-----+-----+
|   AF                               | prefix len |
+-----+-----+-----+-----+
.   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

```

(continues on next page)

(continued from previous page)

```

\/--+      ^
 |         :
 +--[ ]--[ ]--[ ]--> NULL
/\         :
\/--+      :
 |         :
 +--[ ]--[ ]--> NULL
           :
           :
_         :
( ).....

```

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

```

(continues on next page)

```

11.0.0.10 valid [IGP metric 1], #paths 1
  Last update: Wed Oct 16 04:43:51 2013

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

7.2 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] [RFC
5 6 7 8	LOCAL_PREF ATOMIC_AGGREGATE AG-	4271] [RFC 4271] [RFC 4271] [RFC 4271]
9 10 11	GREGATOR COMMUNITIES ORIGINA-	[RFC 1997] [RFC 4456] [RFC 4456] [draft-
12 13	TOR_ID CLUSTER_LIST DPA ADVER-	ietf-idr-bgp-dpa-05.txt(expired)] [RFC 1863]
14 15	TISER RCID_PATH MP_REACH_NLRI	[RFC 1863] [RFC 4760] [RFC 4760] [RFC
16 17	MP_UNREACH_NLRI EXT_COMMUNITIES	4360] [RFC 4893] [RFC 4893]
18	AS4_PATH AS4_AGGREGATOR	

8.1 OSPF API Documentation

8.1.1 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 WayBackMachine).

Ralph has kindly licensed this documentation under GPLv2+. Please preserve the acknowledgements at the bottom of this document.

8.1.2 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).

8.1.3 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.

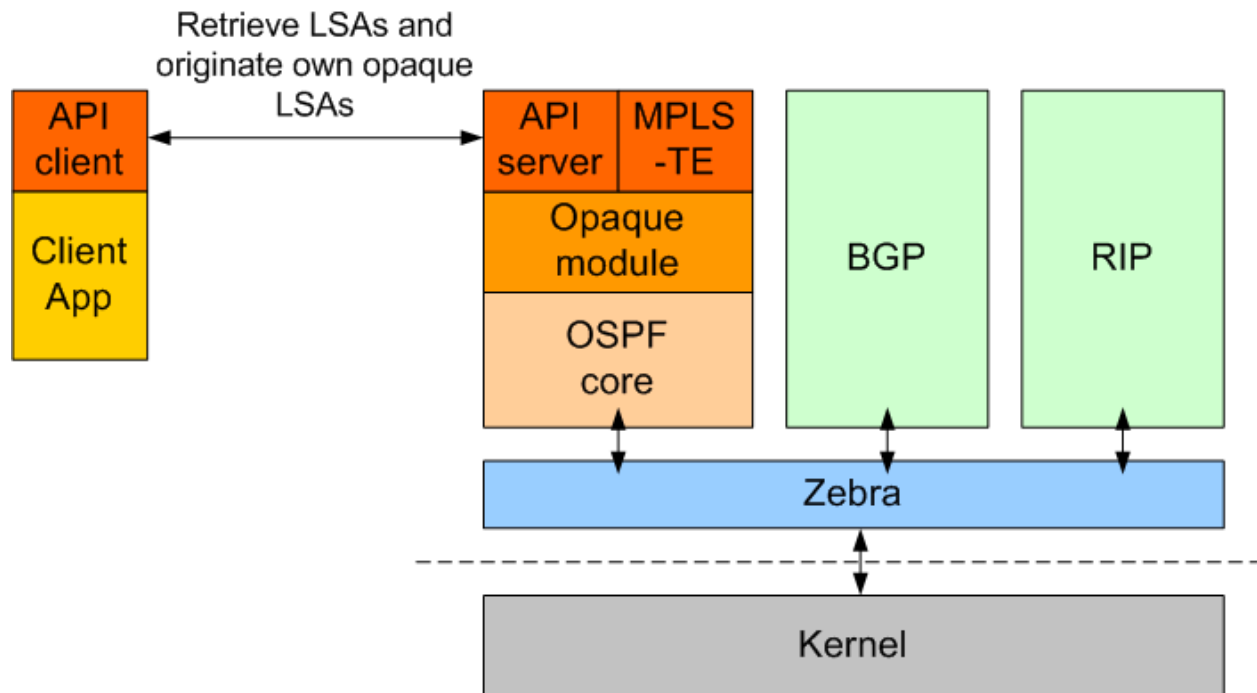


Fig. 1: image

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.

8.1.4 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!
```

8.1.5 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)

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
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)
```

(continues on next page)

(continued from previous page)

```

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.

8.1.6 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:

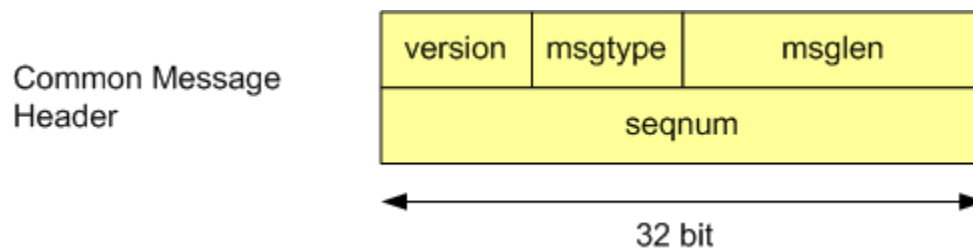


Fig. 2: image

The message type field can take one of the following values:

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

The asynchronous notifications have the following message formats:

8.1.7 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.

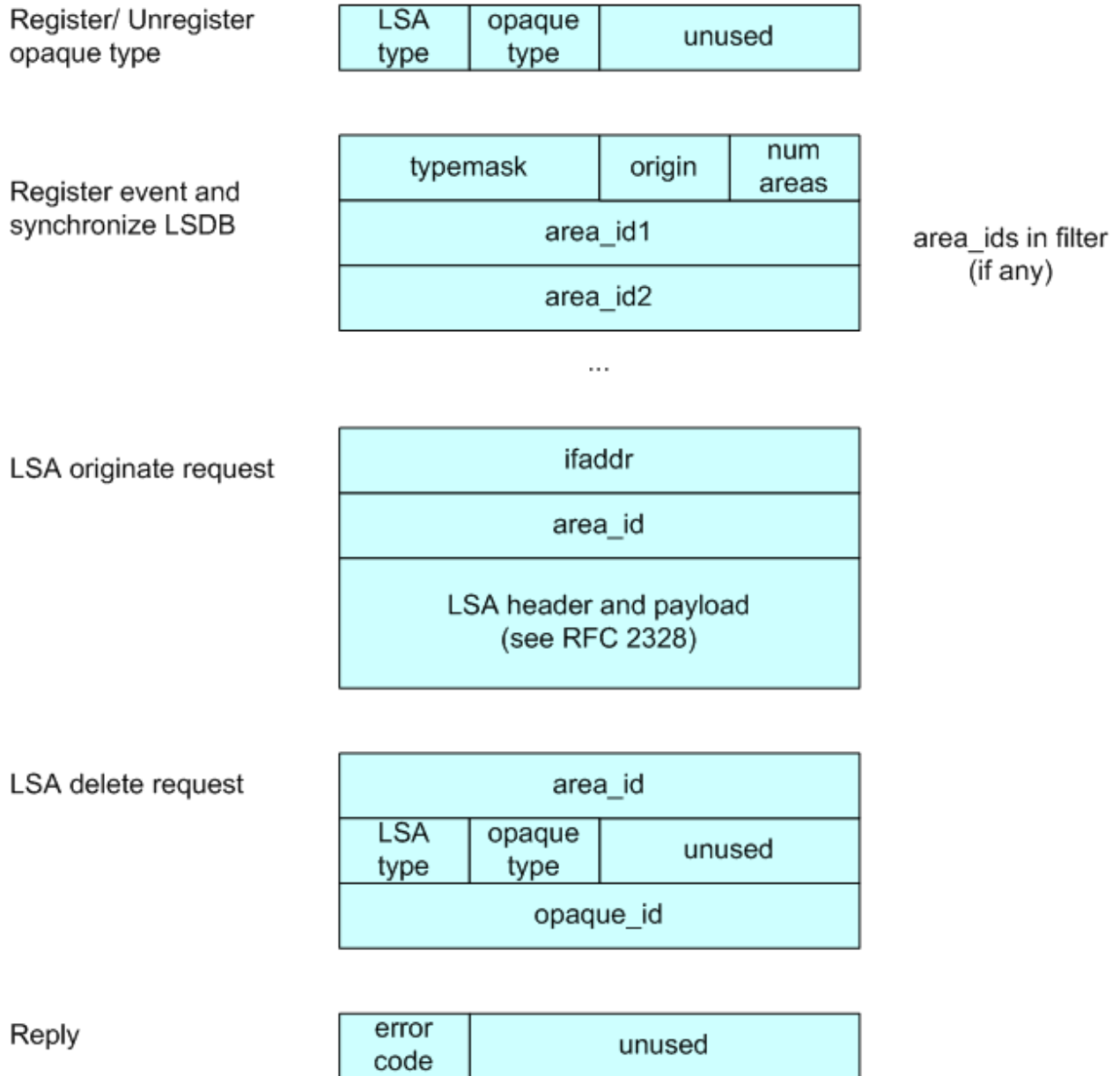


Fig. 3: image

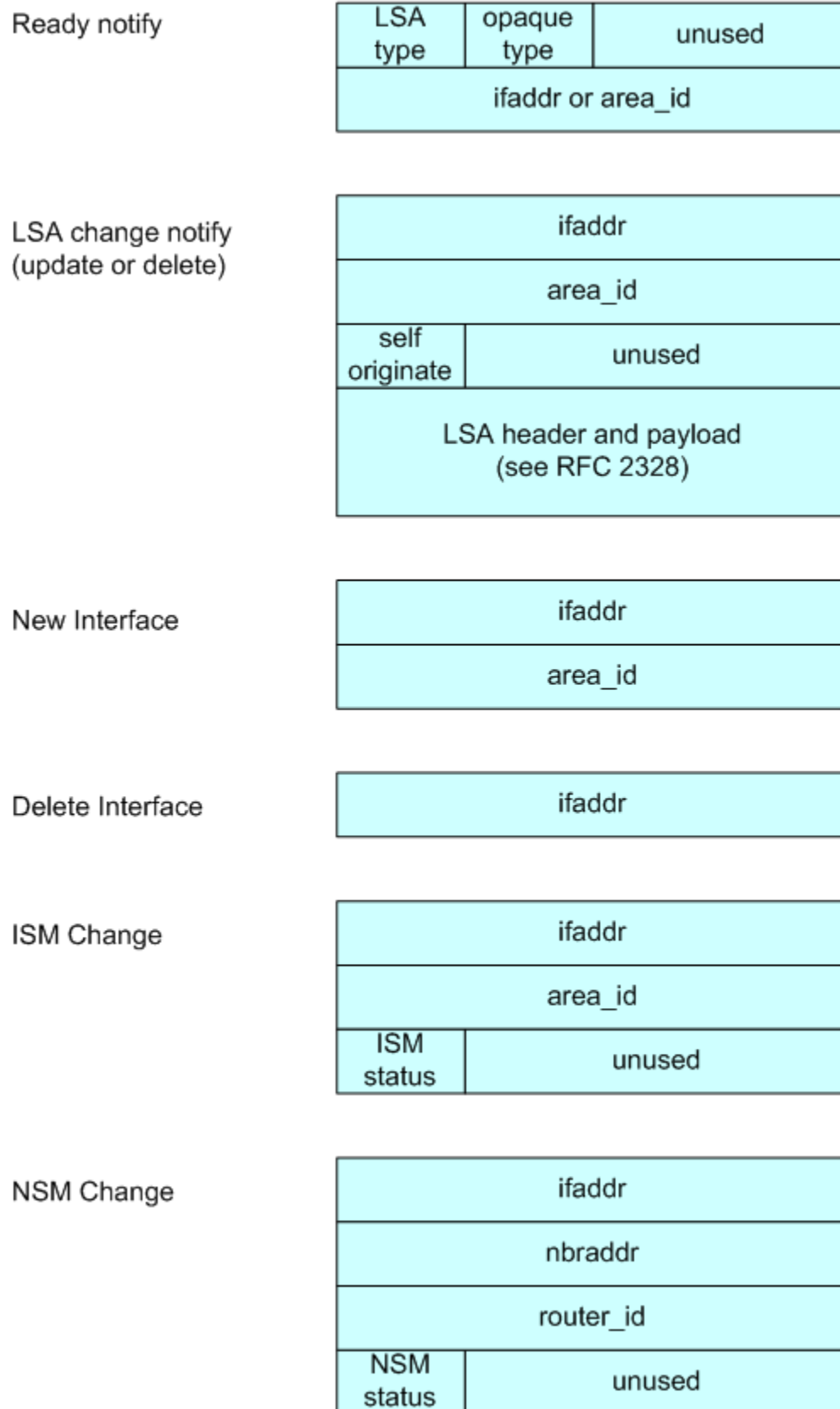


Fig. 4: image

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.

8.2 OSPF Segment Routing

This is an EXPERIMENTAL support of draft *draft-ietf-ospf-segment-routing-extensions-24*. DON'T use it for production network.

8.2.1 Supported Features

- Automatic computation of Primary and Backup Adjacency SID with Cisco experimental remote IP address
- SRGB 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
- Static route configuration with label stack up to 32 labels

8.2.2 Interoperability

- Tested on various topology including point-to-point and LAN interfaces in a mix of Free Range Routing instance and Cisco IOS-XR 6.0.x
- Check OSPF LSA conformity with latest wireshark release 2.5.0-rc

8.2.3 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 Adjacency and Lan Adjacency Segment Identifier
- **Extended Prefix:** flood the Prefix Segment Identifier

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.

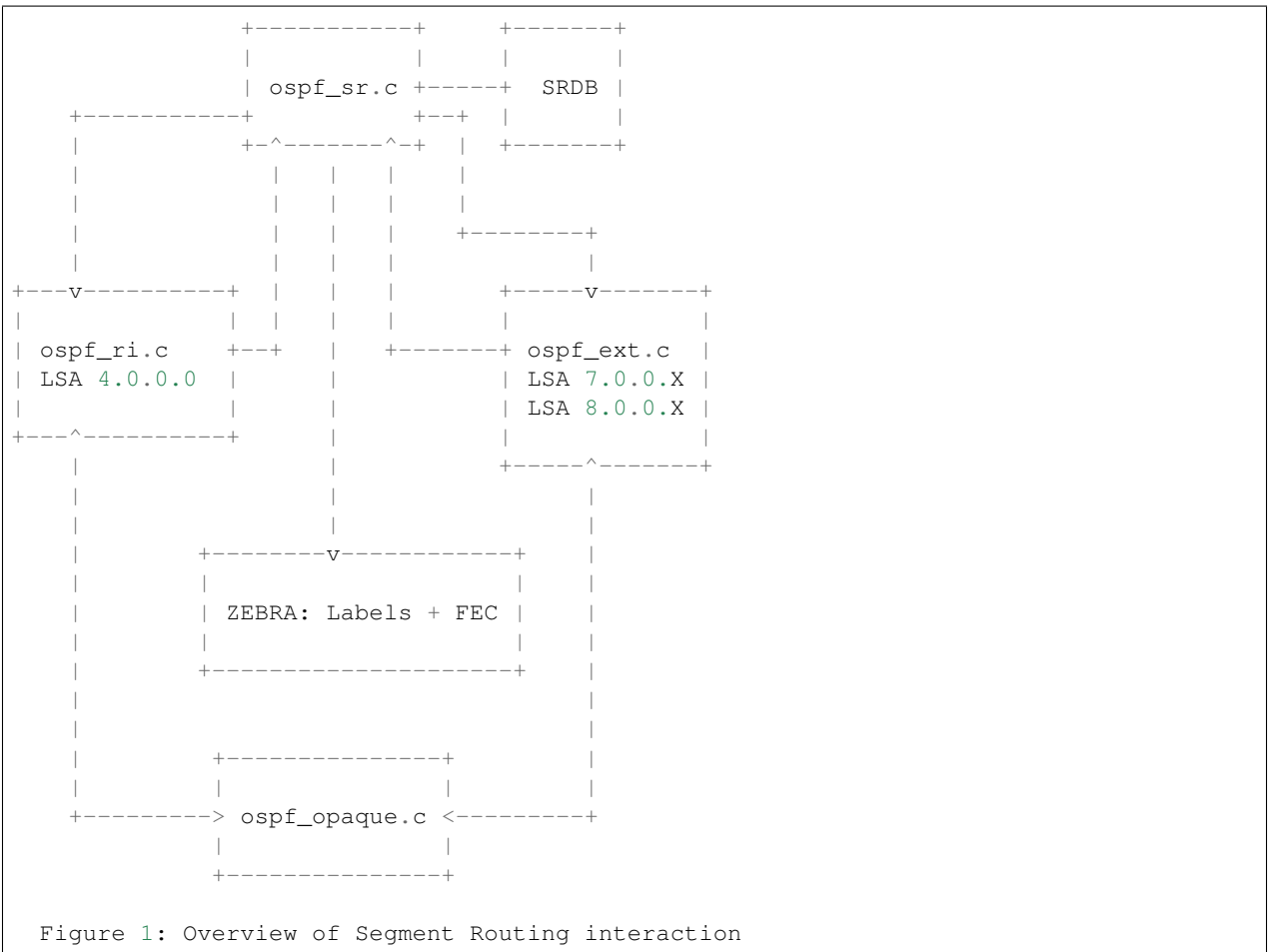


Figure 1: Overview of Segment Routing interaction

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.

8.2.4 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_ip tunnel
```

Then, you must activate MPLS on the interface you would use:

```
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:

```
sysctl -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
mpls-te on
mpls-te router-address 192.168.1.11
router-info area 0.0.0.0
segment-routing on
segment-routing global-block 10000 19999
segment-routing node-msd 8
segment-routing prefix 192.168.1.11/32 index 1100
```

The first segment-routing statement enable it. The Second one set the SRGB, third 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.

8.2.5 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

- MPLS table are not flush at startup. Thus, restarting zebra process is mandatory to remove old MPLS entries in the data plane after a crash of ospfd daemon
- 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.

8.2.6 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.

9.1 Overview of the Zebra Protocol

The Zebra protocol 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.

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.

9.1.1 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.

9.2 Zebra Protocol Definition

9.2.1 Zebra Protocol Header Field Definitions

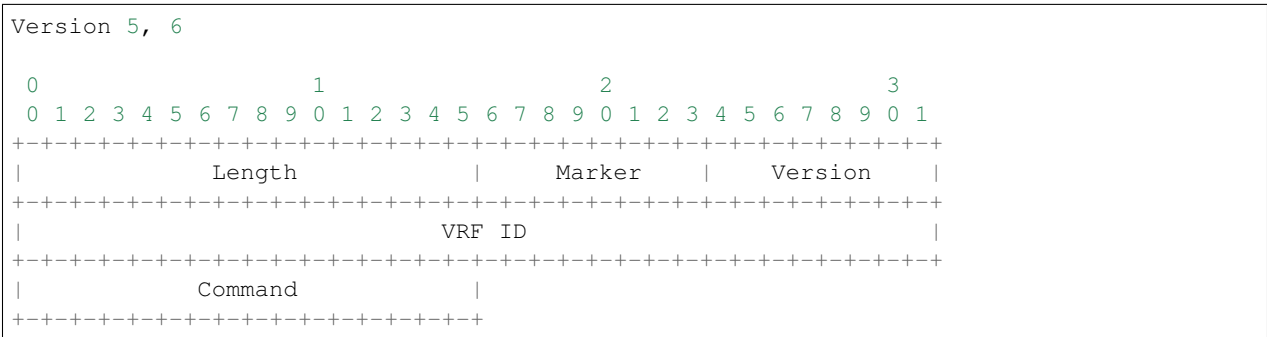
Length Total packet length including this header.

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.

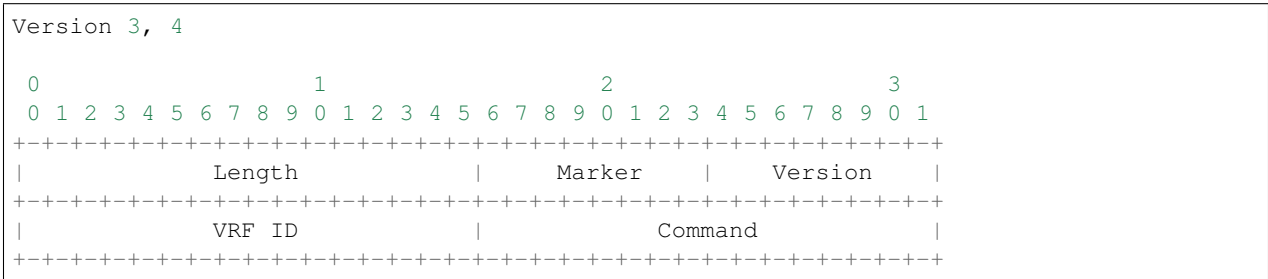
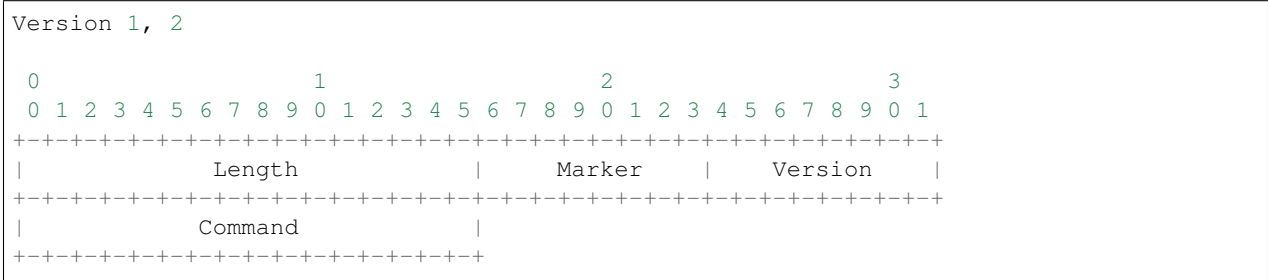
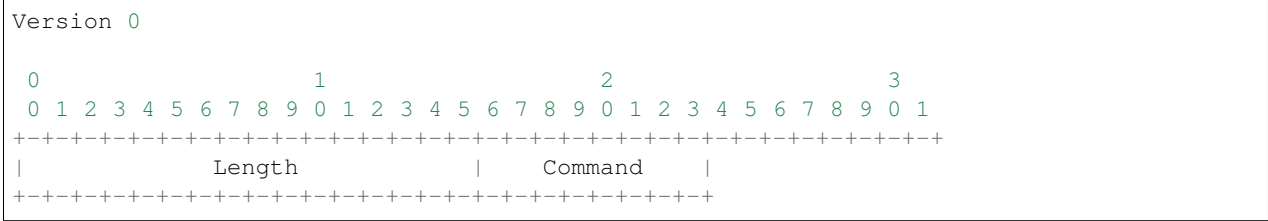
Version Zebra protocol version number. Clients should not continue processing messages past the version field for versions they do not recognise.

Command The Zebra protocol command.

Current Version



Past Versions



9.2.2 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_ROUTE_ADD	7
ZEBRA_ROUTE_DELETE	8
ZEBRA_ROUTE_NOTIFY_OWNER	9
ZEBRA_REDISTRIBUTE_ADD	10
ZEBRA_REDISTRIBUTE_DELETE	11
ZEBRA_REDISTRIBUTE_DEFAULT_ADD	12
ZEBRA_REDISTRIBUTE_DEFAULT_DELETE	13
ZEBRA_ROUTER_ID_ADD	14
ZEBRA_ROUTER_ID_DELETE	15
ZEBRA_ROUTER_ID_UPDATE	16
ZEBRA_HELLO	17
ZEBRA_CAPABILITIES	18
ZEBRA_NEXTHOP_REGISTER	19

Continued on next page

Table 1 – continued from previous page

Command	Value
ZEBRA_NEXTHOP_UNREGISTER	20
ZEBRA_NEXTHOP_UPDATE	21
ZEBRA_INTERFACE_NBR_ADDRESS_ADD	22
ZEBRA_INTERFACE_NBR_ADDRESS_DELETE	23
ZEBRA_INTERFACE_BFD_DEST_UPDATE	24
ZEBRA_IMPORT_ROUTE_REGISTER	25
ZEBRA_IMPORT_ROUTE_UNREGISTER	26
ZEBRA_IMPORT_CHECK_UPDATE	27
ZEBRA_BFD_DEST_REGISTER	28
ZEBRA_BFD_DEST_DEREGISTER	29
ZEBRA_BFD_DEST_UPDATE	30
ZEBRA_BFD_DEST_REPLAY	31
ZEBRA_REDISTRIBUTE_ROUTE_ADD	32
ZEBRA_REDISTRIBUTE_ROUTE_DEL	33
ZEBRA_VRF_UNREGISTER	34
ZEBRA_VRF_ADD	35
ZEBRA_VRF_DELETE	36
ZEBRA_VRF_LABEL	37
ZEBRA_INTERFACE_VRF_UPDATE	38
ZEBRA_BFD_CLIENT_REGISTER	39
ZEBRA_BFD_CLIENT_DEREGISTER	40
ZEBRA_INTERFACE_ENABLE_RADV	41
ZEBRA_INTERFACE_DISABLE_RADV	42
ZEBRA_IPV3_NEXTHOP_LOOKUP_MRIB	43
ZEBRA_INTERFACE_LINK_PARAMS	44
ZEBRA_MPLS_LABELS_ADD	45
ZEBRA_MPLS_LABELS_DELETE	46
ZEBRA_IPMR_ROUTE_STATS	47
ZEBRA_LABEL_MANAGER_CONNECT	48
ZEBRA_LABEL_MANAGER_CONNECT_ASYNC	49
ZEBRA_GET_LABEL_CHUNK	50
ZEBRA_RELEASE_LABEL_CHUNK	51
ZEBRA_FEC_REGISTER	52
ZEBRA_FEC_UNREGISTER	53
ZEBRA_FEC_UPDATE	54
ZEBRA_ADVERTISE_DEFAULT_GW	55
ZEBRA_ADVERTISE_SUBNET	56
ZEBRA_ADVERTISE_ALL_VNI	57
ZEBRA_LOCAL_ES_ADD	58
ZEBRA_LOCAL_ES_DEL	59
ZEBRA_VNI_ADD	60
ZEBRA_VNI_DEL	61
ZEBRA_L3VNI_ADD	62
ZEBRA_L3VNI_DEL	63
ZEBRA_REMOTE_VTEP_ADD	64
ZEBRA_REMOTE_VTEP_DEL	65
ZEBRA_MACIP_ADD	66
ZEBRA_MACIP_DEL	67
ZEBRA_IP_PREFIX_ROUTE_ADD	68

Continued on next page

Table 1 – continued from previous page

Command	Value
ZEBRA_IP_PREFIX_ROUTE_DEL	69
ZEBRA_REMOTE_MACIP_ADD	70
ZEBRA_REMOTE_MACIP_DEL	71
ZEBRA_PW_ADD	72
ZEBRA_PW_DELETE	73
ZEBRA_PW_SET	74
ZEBRA_PW_UNSET	75
ZEBRA_PW_STATUS_UPDATE	76
ZEBRA_RULE_ADD	77
ZEBRA_RULE_DELETE	78
ZEBRA_RULE_NOTIFY_OWNER	79
ZEBRA_TABLE_MANAGER_CONNECT	80
ZEBRA_GET_TABLE_CHUNK	81
ZEBRA_RELEASE_TABLE_CHUNK	82
ZEBRA_IPSET_CREATE	83
ZEBRA_IPSET_DESTROY	84
ZEBRA_IPSET_ENTRY_ADD	85
ZEBRA_IPSET_ENTRY_DELETE	86
ZEBRA_IPSET_NOTIFY_OWNER	87
ZEBRA_IPSET_ENTRY_NOTIFY_OWNER	88
ZEBRA_IPTABLE_ADD	89
ZEBRA_IPTABLE_DELETE	90
ZEBRA_IPTABLE_NOTIFY_OWNER	91
ZEBRA_VXLAN_FLOOD_CONTROL	92

See also:

Command Line Interface

10.1 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*.

10.1.1 Command Extraction

When VTYSH is built, a Perl script named `extract.pl` searches the FRR codebase looking for DEFUN's. It extracts these DEFUN's, transforms them into DEFUSH's and appends them to `vtysh_cmd.c`. Each DEFUSH 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.

10.1.2 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.

DEFSSH 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 `extract.pl` 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.

10.1.3 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`.

10.2 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

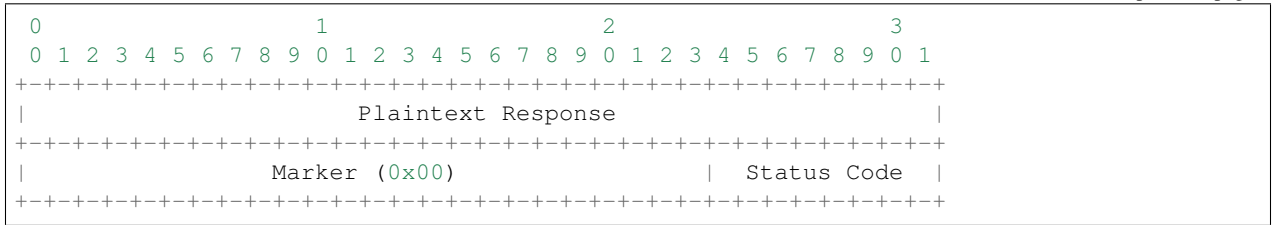
00000000: 646f 2077 7269 7465 2074 6572 6d69 6e61  do write termina
00000010: 6c0a 00                                     l..
```

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.

```
Response
```

(continues on next page)

(continued from previous page)



The first 0x00 byte in the marker also serves to terminate the plaintext response.

Symbols

-topology-only
 pytest command line option, 131
 -s
 pytest command line option, 131

A

asnprintfrr (*C function*), 100
 asprintfrr (*C function*), 100

C

csnprintfrr (*C function*), 100

D

DECLARE_HOOK (*C macro*), 105
 DECLARE_KOOH (*C macro*), 106
 DECLARE_MGROUP (*C macro*), 86
 DECLARE_MTYPE (*C macro*), 86
 DECLARE_XXX (*C function*), 95, 97
 DECLARE_XXX_NONUNIQ (*C function*), 96
 DECLARE_XXX_UNIQ (*C function*), 96
 DEFINE_HOOK (*C macro*), 106
 DEFINE_KOOH (*C macro*), 106
 DEFINE_MGROUP (*C macro*), 86
 DEFINE_MTYPE (*C macro*), 86
 DEFINE_MTYPE_STATIC (*C macro*), 86

F

frr_each (*C function*), 93
 frr_each_from (*C function*), 93
 frr_each_safe (*C function*), 93
 frr_mutex_lock_autounlock (*C function*), 104
 frr_with_mutex (*C function*), 103

H

hook_call (*C function*), 106
 hook_register (*C function*), 106
 hook_register_arg (*C function*), 106
 hook_register_arg_prio (*C function*), 106

hook_register_prio (*C function*), 106
 hook_unregister (*C function*), 106
 hook_unregister_arg (*C function*), 107

M

memtype (*C type*), 86

P

pytest command line option
 -topology-only, 131
 -s, 131

R

rcu_action (*C type*), 89
 rcu_close (*C function*), 90
 rcu_free (*C function*), 89
 rcu_head (*C type*), 89
 rcu_head_close (*C type*), 89
 rcu_read_lock (*C function*), 88
 rcu_read_unlock (*C function*), 88
 rcu_shutdown (*C function*), 90
 rcu_thread (*C type*), 90
 rcu_thread_prepare (*C function*), 90
 rcu_thread_start (*C function*), 90
 rcu_thread_unprepare (*C function*), 90
 RFC
 RFC 2370, 151

S

snprintfrr (*C function*), 100

V

vasnprintfrr (*C function*), 100
 vasprintfrr (*C function*), 100
 vcsnprintfrr (*C function*), 100
 vsnprintfrr (*C function*), 100

X

XALLOC (*C function*), 86

XFREE (*C function*), 86
XMALLOC (*C function*), 86
XREALLOC (*C function*), 86
XSTRDUP (*C function*), 86

Z

Z_add (*C function*), 96
Z_add_after (*C function*), 95
Z_add_head (*C function*), 95
Z_add_tail (*C function*), 95
Z_count (*C function*), 94
Z_del (*C function*), 95
Z_find (*C function*), 96
Z_find_gteq (*C function*), 96
Z_find_lt (*C function*), 96
Z_fini (*C function*), 94
Z_first (*C function*), 94
Z_init (*C function*), 94
Z_init_size (*C function*), 97
Z_next (*C function*), 94
Z_next_safe (*C function*), 94
Z_pop (*C function*), 94