FRR Developer's Manual

Release latest

FRR

Jan 18, 2019

Contents

CHAPTER 1

Process & Workflow

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

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

1.1 Mailing Lists

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

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

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

1.2 Development & Release Cycle

1.2.1 Development

The master Git for FRR resides on GitHub.

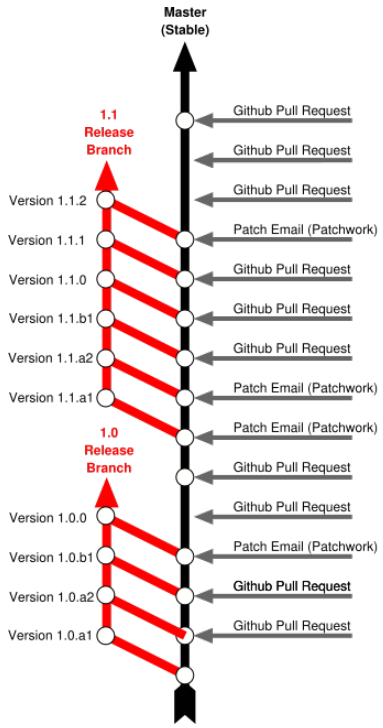


Fig. 1: Rough outline of FRR development workflow

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. The addition of a new routing protocol or daemon would fall under this class.

`MINOR` Small features, e.g. options for automatic BGP shutdown.

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

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. 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 noone 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*.

1.5 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.

- 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

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

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

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

```
git clang-format
```

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

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

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

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

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

One stylistic problem with the FRR codebase is the use of `DEFUN` macros for defining CLI commands. `clang-format` will happily format these macro invocations, but the result is often unsightly and difficult to read. Consequently, FRR takes a more relaxed position with how these are formatted. In general you should lean towards using the style exemplified in the section on *Command Line Interface*. Because `clang-format`

mangles this style, there is a Python script named `tools/indent.py` that wraps `clang-format` and handles `DEFUN` macros as well as some other edge cases specific to FRR. If you are submitting a new file, it is recommended to run that script over the new file, preferably after ensuring that the latest stable release of `clang-format` is in your `PATH`.

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

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

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

It is run like this:

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

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

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

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

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

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

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

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

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

Kernel and BSD styles are documented externally:

[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.

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

Exceptions

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

For `master`

BSD coding style applies to:

- `ldpd/`

`babeld` uses, approximately, the following style:

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

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

GNU coding style apply to the following parts:

- `lib/`
- `zebra/`
- `bgpd/`
- `ospfd/`
- `ospf6d/`

- `isisd/`
- `ripd/`
- `ripngd/`
- `vtysh/`

BSD coding style applies to:

- `ldpd/`

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_DEFINE`s).

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 Static Analysis and Sanitizers

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

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

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

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

to `configure`.

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

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

to `configure`.

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

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

to `configure`.

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

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.

1.6.8 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.

1.6.9 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.

1.6.10 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
`doc/user`	User documentation; configuration guides; protocol overviews
`doc/developer`	Developer's documentation; API specs; datastructures; architecture overviews; project management procedure
`doc/manpages`	Source for manpages
`doc/figures`	Images and diagrams
`doc/extra`	Miscellaneous Sphinx extensions, scripts, customizations, etc.

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

Indent and styling should follow existing conventions:

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

Characters for header levels should follow Python documentation guide:

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

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

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

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

1.7.1 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:
 *    7 if the text is cool, 0 otherwise
 */
int check_coolness(const char *text, bool is_clccfc);
```

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

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

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

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

1.7.2 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::` directive and specify `frr` as the highlighting language, as in the following example. This will tell Sphinx to use a custom Pygments lexer to highlight FRR configuration syntax.

```
.. code-block:: frr

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

CHAPTER 2

Building FRR

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

(As an alternative to this installation, you may prefer to create a FRR rpm package yourself and install that package instead. See instructions in redhat/README.rpm_build.md on how to build a rpm package)

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 associcated with (IFLA_INFO_SLAVE_KIND does not exist in the kernel headers, you can use a newer kernel + headers to get this functionality)
- frr_reload.py will not work, as this requires Python 2.7, and CentOS 6 only has 2.6. You can install Python 2.7 via IUS, but it won't work properly unless you compile and install the ipaddr package for it.
- Building the package requires Sphinx >= 1.1. Only a non-standard package provides a newer sphinx and requires manual installation (see below)

2.2.3 Install required packages

Add packages:

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

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
```

(continues on next page)

(continued from previous page)

```
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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-exampledir \
    --disable-ldpd \
    --enable-fpm \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion \
    SPHINXBUILD=sphinx-build2.7
make
make check PYTHON=/usr/bin/python2.7
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
```

(continues on next page)

(continued from previous page)

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

Load the modifed 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

(As an alternative to this installation, you may prefer to create a FRR rpm package yourself and install that package instead. See instructions in redhat/README.rpm_build.md on how to build a rpm package)

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 gawk \
  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
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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 \
```

(continues on next page)

(continued from previous page)

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

Create empty FRR configuration files

```
sudo mkdir /var/log/frr
sudo mkdir /etc/frr
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 modifed 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 gawk \
   libreadline-dev texinfo libjson-c-dev pkg-config bison flex \
   python-pip libc-ares-dev python3-dev python3-sphinx
```

Install newer pytest (>3.0) from pip

```
sudo pip install pytest
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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

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

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

Shared library error

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

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

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

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

2.5 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 \
  python-pip libc-ares-dev python3-dev python-pytest python3-sphinx
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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

(As an alternative to this installation, you may prefer to create a FRR rpm package yourself and install that package instead. See instructions in redhat/README.rpm_build.md on how to build a rpm package)

2.6.1 Install required packages

Add packages:

```
sudo dnf install git autoconf automake libtool make gawk \
  readline-devel texinfo net-snmp-devel groff pkgconfig \
  json-c-devel pam-devel pytest bison flex c-ares-devel \
  python3-devel python3-sphinx
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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

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

Add frr groups and user

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

Download Source, configure and compile it

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

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

Create empty FRR configuration files

```
sudo mkdir /var/log/frr
sudo mkdir /etc/frr
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/ldpd.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 (and MPLS)

Create a new file `/etc/sysctl.d/90-routing-sysctl.conf` with the following content: (Please make sure to list all interfaces with required MPLS similar to `net.mpls.conf.eth0.input=1`)

```
# Sysctl for routing
#
# Routing: We need to forward packets
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
```

Load the modifed sysctl's 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
```

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
```

Enable required frr at startup

```
sudo systemctl enable frr
```

Reboot or start FRR manually

```
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 managment tool if this is first package install and asked)

```
pkg install git autoconf automake libtool gmake gawk json-c pkgconf \
    bison flex py27-pytest c-ares python3 py-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)

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

```
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
```

(continues on next page)

(continued from previous page)

```
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.

2.8 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 managment tool if this is first package install and asked)

```
pkg install git autoconf automake libtool gmake gawk json-c pkgconf \
   bison flex py27-pytest c-ares python3 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)

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

```
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 managment tool if this is first package install and asked)

```
pkg install -y git autoconf automake libtool gmake gawk \
    pkgconf texinfo json-c bison flex py27-pytest c-ares \
    python3 py-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)

```
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
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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 \
    --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
```

(continues on next page)

(continued from previous page)

```
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 gawk openssl \
   pkg-config json-c python27 py27-test python35 py-sphinx
```

Install SSL Root Certificates (for git https access):

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

Select default Python and py.test

```
sudo ln -s /usr/pkg/bin/python2.7 /usr/bin/python
sudo ln -s /usr/pkg/bin/py.test-2.7 /usr/bin/py.test
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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 gawk openssl \
   pkg-config json-c python27 py27-test python35 py-sphinx
```

Install SSL Root Certificates (for git https access):

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

Select default Python and py.test

```
sudo ln -s /usr/pkg/bin/python2.7 /usr/bin/python
sudo ln -s /usr/pkg/bin/py.test-2.7 /usr/bin/py.test
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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-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 /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 \
```

(continues on next page)

(continued from previous page)

```
  developer/parser/bison \
  developer/object-file \
  developer/linker \
  developer/library/lint \
  developer/build/gnu-make \
  developer/gcc51 \
  library/idnkit \
  library/idnkit/header-idnkit \
  system/header \
  system/library/math/header-math \
  git libtool gawk 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
```

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
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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 gawk dejagnu openssl json-c py-test py-sphinx
```

Select Python2.7 as default (required for pytest)

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

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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/frrouting/frr.git frr
cd frr
export AUTOCONF_VERSION="2.69"
export AUTOMAKE_VERSION="1.15"
./bootstrap.sh
export LDFLAGS="-L/usr/local/lib"
export CPPFLAGS="-I/usr/local/include"
./configure \
    --sysconfdir=/etc/frr \
    --localstatedir=/var/frr \
    --enable-multipath=64 \
    --enable-user=_frr \
    --enable-group=_frr \
    --enable-vty-group=_frrvty \
    --enable-configfile-mask=0640 \
    --enable-logfile-mask=0640 \
    --enable-fpm \
```

(continues on next page)

(continued from previous page)

```
    --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 gawk 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
```

(continues on next page)

(continued from previous page)

```
./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 unneded packages For example: zebra bgpd ldpd isisd nhrpd ospfd ospf6d pimd ripd ripngd

Enable the serivce

- `service frr enable`

Start the service

- `service frr start`

2.15 Ubuntu 12.04LTS

- MPLS is not supported on `Ubuntu 12.04` with default kernel. MPLS requires Linux Kernel 4.5 or higher (LDP can be built, but may have limited use without MPLS) For an updated Ubuntu Kernel, see http://kernel.ubuntu.com/~kernel-ppa/mainline/

2.15.1 Install required packages

Add packages:

```
apt-get install \
   git autoconf automake libtool make gawk libreadline-dev texinfo \
   dejagnu pkg-config libpam0g-dev libjson0-dev flex python-pip \
   libc-ares-dev python3-dev python3-sphinx install-info
```

Install newer bison from 14.04 package source (Ubuntu 12.04 package source is too old)

```
mkdir builddir
cd builddir
wget http://archive.ubuntu.com/ubuntu/pool/main/b/bison/bison_3.0.2.dfsg-2.dsc
wget http://archive.ubuntu.com/ubuntu/pool/main/b/bison/bison_3.0.2.dfsg.orig.tar.bz2
wget http://archive.ubuntu.com/ubuntu/pool/main/b/bison/bison_3.0.2.dfsg-2.debian.tar.
↪gz
tar -jxvf bison_3.0.2.dfsg.orig.tar.bz2
cd bison-3.0.2.dfsg/
tar xzf ../bison_3.0.2.dfsg-2.debian.tar.gz
sudo apt-get build-dep bison
debuild -b -uc -us
cd ..
sudo dpkg -i ./libbison-dev_3.0.2.dfsg-2_amd64.deb ./bison_3.0.2.dfsg-2_amd64.deb
cd ..
rm -rf builddir
```

Install newer version of autoconf and automake:

```
wget 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 ..

wget 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 pytest:

```
pip install pytest
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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

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

Add frr groups and user

```
sudo groupadd -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
```

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 \
    --prefix=/usr \
    --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 \
```

(continues on next page)

(continued from previous page)

```
    --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

Install the init.d service

```
sudo install -m 755 tools/frr /etc/init.d/frr
sudo install -m 644 tools/etc/frr/daemons /etc/frr/daemons
sudo install -m 644 -o frr -g frr tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
```

Enable daemons

Edit `/etc/frr/daemons` and change the value from "no" to "yes" for those daemons you want to start by systemd.
For example.

```
zebra=yes
bgpd=yes
ospfd=yes
ospf6d=yes
ripd=yes
ripngd=yes
isisd=yes
```

Start the init.d service

- /etc/init.d/frr start
- use `/etc/init.d/frr status` to check its status.

2.16 Ubuntu 14.04LTS

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

2.16.1 Install required packages

Add packages:

```
apt-get install \
   git autoconf automake libtool make gawk libreadline-dev texinfo dejagnu \
   pkg-config libpam0g-dev libjson-c-dev bison flex python-pytest \
   libc-ares-dev python3-dev python3-sphinx install-info
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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

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

Add frr groups and user

```
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
```

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 \
    --prefix=/usr \
    --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

Install the init.d service

```
sudo install -m 755 tools/frr /etc/init.d/frr
sudo install -m 644 tools/etc/frr/daemons /etc/frr/daemons
sudo install -m 644 -o frr -g frr tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
```

Enable daemons

Edit `/etc/frr/daemons` and change the value from "no" to "yes" for those daemons you want to start by systemd.
For example.

```
zebra=yes
bgpd=yes
ospfd=yes
ospf6d=yes
ripd=yes
ripngd=yes
isisd=yes
```

Start the init.d service

- /etc/init.d/frr start
- use `/etc/init.d/frr status` to check its status.

2.17 Ubuntu 16.04LTS

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

2.17.1 Install required packages

Add packages:

```
apt-get install \
   git autoconf automake libtool make gawk libreadline-dev texinfo dejagnu \
   pkg-config libpam0g-dev libjson-c-dev bison flex python-pytest \
   libc-ares-dev python3-dev libsystemd-dev python-ipaddress \
   python3-sphinx install-info
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

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

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

Add frr groups and user

```
sudo groupadd -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
```

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 \
    --prefix=/usr \
    --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 \
    --enable-systemd=yes \
    --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
```

(continues on next page)

(continued from previous page)

```
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 IPv4 & 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
```

Enable MPLS Forwarding (with Linux Kernel >= 4.5)

Edit `/etc/sysctl.conf` and the following lines. Make sure to add a line equal to `net.mpls.conf.eth0.input` or 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
```

Add MPLS kernel modules

Add the following lines to `/etc/modules-load.d/modules.conf`:

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

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

Install the systemd service (if rebooted from last step, change directory back to frr directory)

```
sudo install -m 644 tools/frr.service /etc/systemd/system/frr.service
sudo install -m 644 tools/etc/frr/daemons /etc/frr/daemons
sudo install -m 644 tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 644 -o frr -g frr tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
```

Enable daemons

Edit `/etc/frr/daemons` and change the value from "no" to "yes" for those daemons you want to start by systemd.
For example.

```
zebra=yes
bgpd=yes
ospfd=yes
ospf6d=yes
ripd=yes
ripngd=yes
isisd=yes
```

Enable the systemd service

- systemctl enable frr

Start the systemd service

- systemctl start frr
- use `systemctl status frr` to check its status.

2.18 Ubuntu 18.04 LTS

2.18.1 Install dependencies

Required packages

```
sudo apt-get install \
   git autoconf automake libtool make gawk libreadline-dev texinfo \
   pkg-config libpam0g-dev libjson-c-dev bison flex python-pytest \
   libc-ares-dev python3-dev libsystemd-dev python-ipaddress \
   python3-sphinx install-info
```

The libyang library can be installed from third-party packages available here.

Note: the libyang dev/devel packages need to be installed in addition to the libyang core package in order to build FRR successfully.

For example, for CentOS 7.x:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-0.16.46-0.x86_64.rpm
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/CentOS-7-x86_
↪64-Packages/libyang-devel-0.16.46-0.x86_64.rpm
sudo rpm -i libyang-0.16.46-0.x86_64.rpm libyang-devel-0.16.46-0.x86_64.rpm
```

or Ubuntu 18.04:

```
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang-dev_0.16.46_amd64.deb
wget https://ci1.netdef.org/artifact/LIBYANG-YANGRELEASE/shared/build-1/Ubuntu-18.04-
↪x86_64-Packages/libyang_0.16.46_amd64.deb
sudo apt install libpcre3-dev
sudo dpkg -i libyang-dev_0.16.46_amd64.deb libyang_0.16.46_amd64.deb
```

Alternatively, libyang can be built and installed manually by following the steps below:

```
git clone https://github.com/opensourcerouting/libyang
cd libyang
git checkout -b tmp origin/tmp
mkdir build; cd build
cmake -DENABLE_LYD_PRIV=ON ..
make
sudo make install
```

When building libyang on CentOS 6, it's also necessary to pass the `-DENABLE_CACHE=OFF` parameter to cmake.

Note: please check the libyang build requirements first.

Optional packages

Dependencies for additional functionality can be installed as-desired.

Protobuf

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

ZeroMQ

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

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

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

Add frr groups and user

```
sudo groupadd -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
```

Download source

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

Configure

Options below are provided as an example.

See also:

Installation section of user guide

```
cd frr
./bootstrap.sh
./configure \
    --prefix=/usr \
    --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 \
    --enable-systemd=yes \
    --with-pkg-git-version \
    --with-pkg-extra-version=-MyOwnFRRVersion
```

If optional packages were installed, the associated feature may now be enabled.

`--enable-protobuf`

Enable support for protobuf transport

`--enable-zeromq`

Enable support for ZeroMQ transport

Compile

```
make
make check
sudo make install
```

Create empty FRR configuration files

Although not strictly necessary, it's good practice to create empty configuration files _before_ starting FRR. This assures that the permissions are correct. If the files are not already present, FRR will create them.

It's also important to consider _which_ files to create. FRR supports writing configuration to a monolithic file, `/etc/frr/frr.conf`.

See also:

VTYSH section of user guide

The presence of `/etc/frr/frr.conf` on startup implicitly configures FRR to ignore daemon-specific configuration files.

Daemon-specific configuration

```
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
```

Monolithic configuration

```
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/frr.conf
```

Enable IPv4 & 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
```

Add MPLS kernel modules

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

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

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

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 systemd service

```
sudo install -m 644 tools/frr.service /etc/systemd/system/frr.service
sudo install -m 644 tools/etc/frr/daemons /etc/frr/daemons
sudo install -m 644 tools/etc/frr/frr.conf /etc/frr/frr.conf
sudo install -m 644 -o frr -g frr tools/etc/frr/vtysh.conf /etc/frr/vtysh.conf
```

Enable daemons

Edit `/etc/frr/daemons` and change the value from "no" to "yes" for those daemons you want to start by systemd. For example:

```
zebra=yes
bgpd=yes
ospfd=yes
ospf6d=yes
ripd=yes
ripngd=yes
isisd=yes
```

Enable the systemd service

Enabling the systemd service causes FRR to be started upon boot. To enable it, use the following command:

```
systemctl enable frr
```

Start the systemd service

```
systemctl start frr
```

After starting the service, you can use `systemctl status frr` to check its status.

CHAPTER 3

Packaging

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 `debianpkg/changelog.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 12.04, 14.04, 16.04, 17.10, 18.04, Debian 8 and 9)

Note: If you try to build for a different distro, then it will most likely fail because of the missing backport. See *Debian Backports* about adding a new backport.

---

1. Install build dependencies for your platform as outlined in *Building FRR*.
2. Install the following additional packages:
   - on Ubuntu 12.04, 14.04, 16.04, 17.10, Debian 8 and 9:

```
apt-get install realpath equivs groff fakeroot debhelper devscripts
```

   - on Ubuntu 18.04: (realpath is now part of preinstalled by coreutils)

```
apt-get install equivs groff fakeroot debhelper devscripts
```

3. Checkout FRR under a **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>
```

4. Run `bootstrap.sh` and make a dist tarball:

```
./bootstrap.sh
./configure --with-pkg-extra-version=-MyDebPkgVersion
make dist
```

Note: Configure parameters are not important for the Debian Package building - except the *with-pkg-extra-version* if you want to give the Debian package a specific name to mark your own unoffical build.

5. Edit `debianpkg/rules` and set the configuration as needed.

   Look for section `dh_auto_configure` to modify the configure options as needed. Options might be different between the top-level `rules` and `backports/XXXX/debian/rules`. Please adjust as needed on all files.

6. Create backports debian sources

   Rename the `debianpkg` directory to `debian` and create the backports (Debian requires to not ship a `debian` directory inside the source directory to avoid build conflicts with the reserved `debian` subdirectory name during the build):

```
mv debianpkg debian
make -f debian/rules backports
```

   This will create a `frr_*.orig.tar.gz` with the source (same as the dist tarball), as well as multiple `frr_*.debian.tar.xz` and `frr_*.dsc` corresponding to each distribution for which a backport is available.

7. Create a new directory to build the package and populate with package source.

```
mkdir frrpkg
cd frrpkg
tar xf ~/frr/frr_*.orig.tar.gz
cd frr*
. /etc/os-release
tar xf ~/frr/frr_*${ID}${VERSION_ID}*.debian.tar.xz
```

8. Build Debian package dependencies and install them as needed.

```
sudo mk-build-deps --install debian/control
```

9. Build Debian Package

   Building with standard options:

```
debuild -b -uc -us
```

   Or change some options (see *rules* file for available options):

```
debuild --set-envvar=WANT_BGP_VNC=1 --set-envvar=WANT_CUMULUS_MODE=1 -b -uc -us
```

To build with RPKI:

- Download the librtr packages from https://ci1.netdef.org/browse/RPKI-RTRLIB/latestSuccessful/artifact
- install librtr-dev on the build server

Then build with:

```
debuild --set-envvar=WANT_RPKI=1 -b -uc -us
```

RPKI packages have an additonal dependency of `librtr0` which can be found at the same URL.

10. Done!

If all worked correctly, then you should end up with the Debian packages under `frrpkg`. If distributed, please make sure you distribute it together with the sources (`frr_*.orig.tar.gz`, `frr_*.debian.tar.xz` and `frr_*.dsc`)

The build procedure can also be executed automatically using the `tools/build-debian-package.sh` script. For example:

```
EXTRA_VERSION="-myversion" WANT_SNMP=1 WANT_CUMULUS_MODE=1 tools/build-debian-package.
↪sh
```

3.2.1 Debian Backports

The `debianpkg/backports` directory contains the Debian directories for backports to other Debian platforms. These are built via the `3.0 (custom)` source format, which allows one to build a source package directly out of tarballs (e.g. an orig.tar.gz tarball and a debian.tar.gz file), at which point the format can be changed to a real format (e.g. `3.0 (quilt)`).

Source packages are assembled via targets of the same name as the system to which the backport is done (e.g. `precise`), included in `debian/rules`.

To create a new Debian backport:

- Add its name to `KNOWN_BACKPORTS`, defined in `debian/rules`.
- Create a directory of the same name in `debian/backports`.
- Add the files `exclude`, `versionext`, and `debian/source/format` under this directory.

For the last point, these files should contain the following:

`exclude` Contains whitespace-separated paths (relative to the root of the source dir) that should be excluded from the source package (e.g. `debian/patches`).

`versionext` Contains the suffix added to the version number for this backport's build. Distributions often have guidelines for what this should be. If left empty, no new `debian/changelog` entry is created.

`debian/source/format` Contains the source format of the resulting source package. As of of the writing of this document the only supported format is `3.0 (quilt)`.

- Add appropriate files under the `debian/` subdirectory. These will be included in the source package, overriding any top-level `debian/` files with equivalent paths.

CHAPTER 4

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`

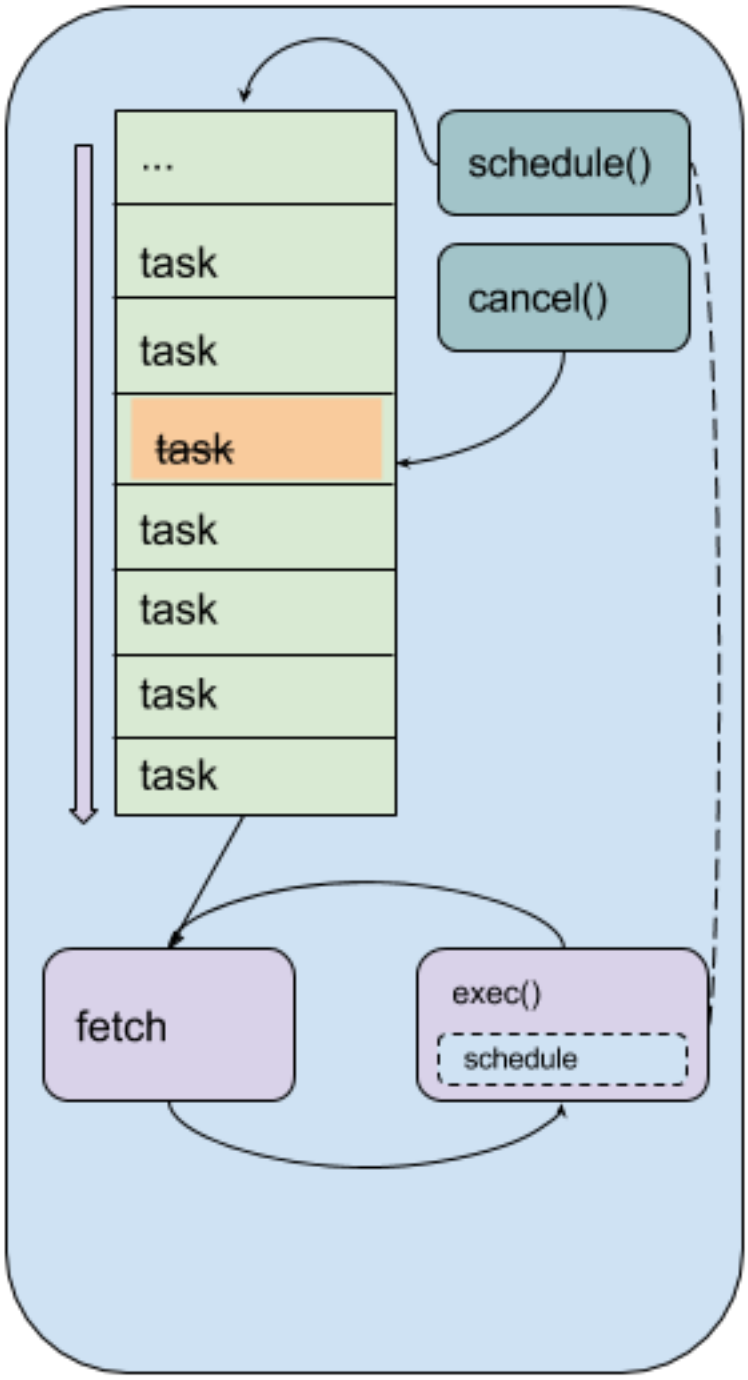


Fig. 1: Lifecycle of a program using a single threadmaster.

- `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

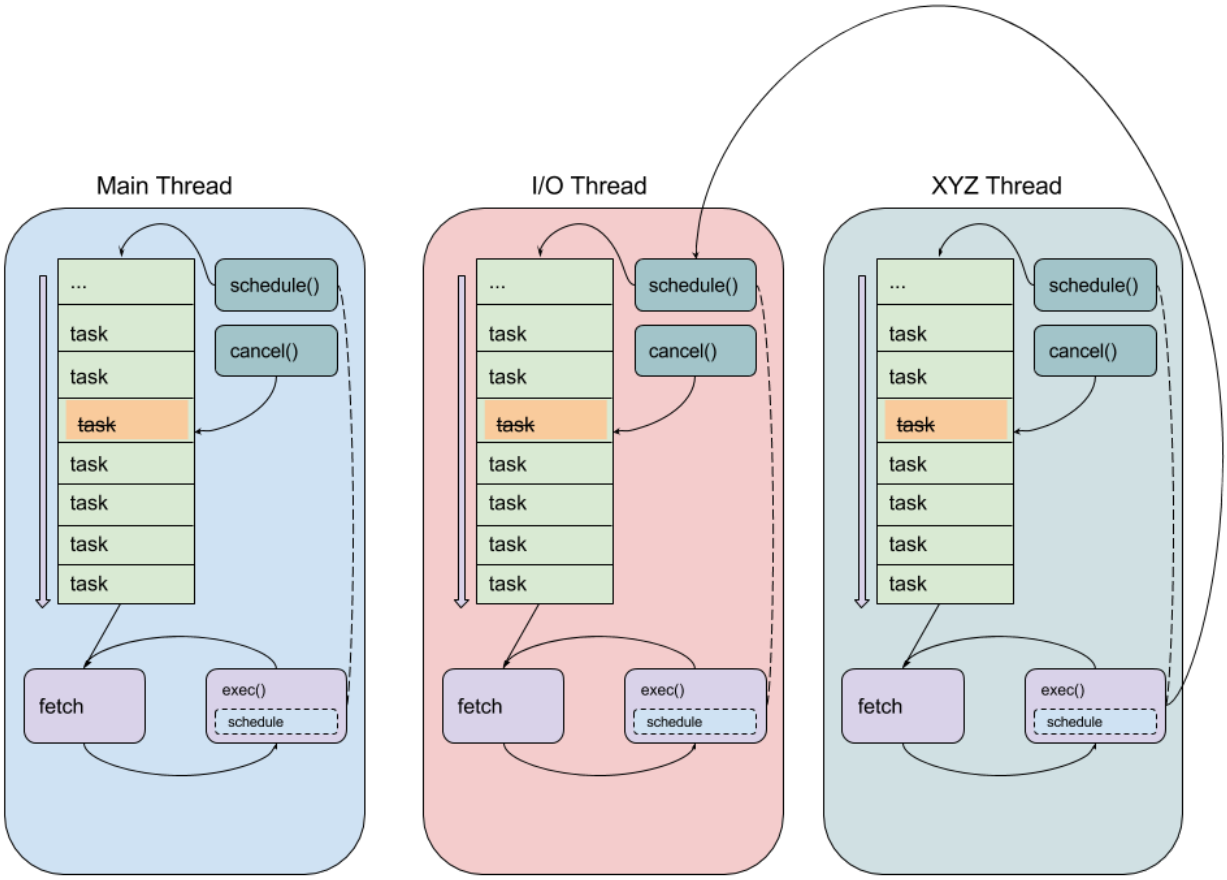


Fig. 2: Lifecycle of a program using multiple pthreads, each running their own `threadmaster`

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
    0       1389.000        10   138900    248000   135549    255349   T   subgroup_
↪coalesce_timer
    0          0.000         1        0         0       18        18   T   bgp_
↪startup_timer_expire
    0        850.000        18    47222    222000    47795    233814   T   work_queue_
↪run
    0          0.000        10        0         0        6        14   T   update_
↪subgroup_merge_check_thread_cb
    0          0.000         8        0         0      117       160  W    zclient_
↪flush_data
    2          2.000         1     2000      2000      831       831 R     bgp_accept
    0          1.000         1     1000      1000     2832      2832    E  zclient_
↪connect
    1      42082.000    240574      174     37000      178     72810 R     vtysh_read
    1        152.000      1885       80      2000       96      6292 R     zclient_
↪read
    0     549346.000   2997298      183      7000      153     20242    E  bgp_event
    0       2120.000       300     7066     14000     6813     22046   T   (bgp_
↪holdtime_timer)
    0          0.000         2        0         0       57        59   T   update_
↪group_refresh_default_originate_route_map
    0         90.000         1    90000     90000    73729     73729   T   bgp_route_
↪map_update_timer
    0       1417.000      9147      154     48000      132     61998   T   bgp_
↪process_packet
  300      71807.000   2995200       23      3000       24     11066   T   (bgp_
↪connect_timer)
    0       1894.000     12713      148     45000      112     33606   T   (bgp_
↪generate_updgrp_packets)
    0          0.000         1        0         0      105       105  W    vtysh_write
    0         52.000       599       86      2000      138      6992   T   (bgp_start_
↪timer)
    1          1.000         8      125      1000      164       593 R     vtysh_
↪accept
```

(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.

CHAPTER 5

Library Facilities (libfrr)

5.1 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.

5.1.1 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 neccessarily 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*

5.1.2 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.

5.1.3 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

5.2 Memtypes

FRR includes wrappers arround `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);
}
```

5.2.1 Definition

DECLARE_MGROUP (name)
: This macro forward-declares a memory group and should be placed in a `.h` file. It expands to an `extern struct memgroup` statement.

DEFINE_MGROUP (mname, description)
Defines/implements a memory group. Must be placed into exactly one `.c` file (multiple inclusion will result in a link-time symbol conflict).

Contains additional logic (constructor and destructor) to register the memory group in a global list.

DECLARE_MTYPE (name)
Forward-declares a memory type and makes `MTYPE_name` available for use. Note that the `MTYPE_` prefix must not be included in the name, it is automatically prefixed.

`MTYPE_name` is created as a *static const* symbol, i.e. a compile-time constant. It refers to an `extern struct memtype _mt_name`, where *name* is replaced with the actual name.

DEFINE_MTYPE (group, name, description)
Define/implement a memory type, must be placed into exactly one `.c` file (multiple inclusion will result in a link-time symbol conflict).

Like `DEFINE_MGROUP`, this contains actual code to register the MTYPE under its group.

DEFINE_MTYPE_STATIC (group, name, description)
Same as `DEFINE_MTYPE`, but the `DEFINE_MTYPE_STATIC` variant places the C `static` keyword on the definition, restricting the MTYPE's availability to the current source file. This should be appropriate in >80% of cases.

Todo: Daemons currently have `daemon_memory.[ch]` files listing all of their MTYPEs. This is not how it should be, most of these types should be moved into the appropriate files where they are used. Only a few MTYPEs should remain non-static after that.

5.2.2 Usage

void ***XMALLOC** (struct memtype **mtype*, size_t *size*)

void ***XCALLOC** (struct memtype **mtype*, size_t *size*)

void ***XSTRDUP** (struct memtype **mtype*, size_t *size*)
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.

5.3 Hooks

Libfrr provides type-safe subscribable hook points where other pieces of code can add one or more callback functions. "type-safe" in this case applies to the function pointers used for subscriptions. The implementations checks (at compile-time) wheter 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.

5.3.1 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.

5.3.2 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.)

5.3.3 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.

5.3.4 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.

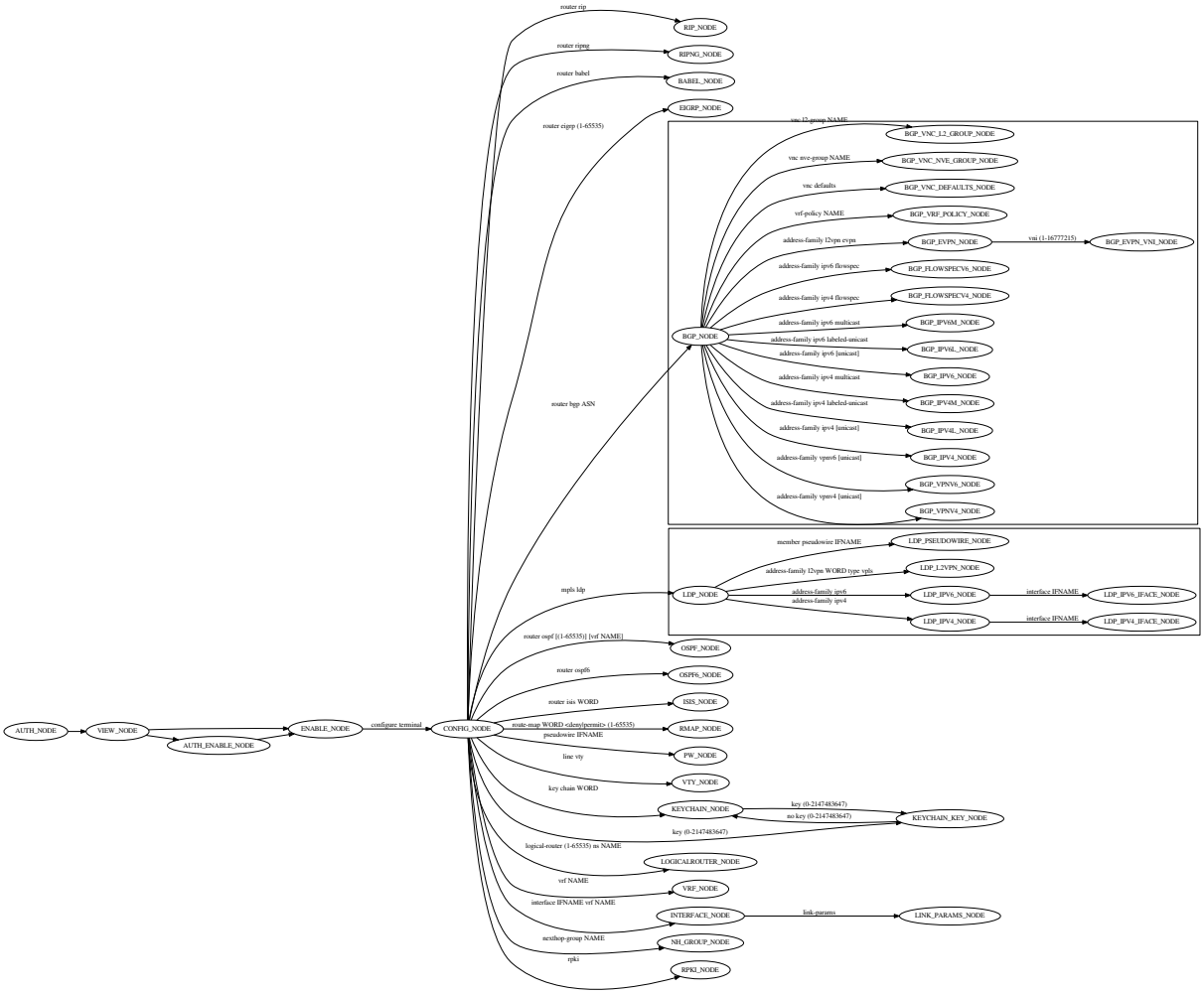
5.4 Command Line Interface

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.

5.4.1 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.

5.4.2 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_seq selector_token
                              selector_token
selector_token          ::=   selector
                              simple_token
```

Tokens

The various capitalized tokens in the BNF above are in fact themselves placeholders, but not defined as such in the formal grammar; the grammar provides the structure, and the tokens are actually more like a type system for the strings you write in your CLI definitions. A CLI definition string is broken apart and each piece is assigned a type by the lexer based on a set of regular expressions. The parser uses the type information to verify the string and determine the structure of the CLI graph; additional metadata (such as the raw text of each token) is encoded into the graph as it is constructed by the parser, but this is merely a dumb copy job.

Here is a brief summary of the various token types along with examples.

Token type	Syntax	Description
WORD	show ip bgp	Matches itself. In the given example every token is a WORD.
IPV4	A.B.C.D	Matches an IPv4 address.
IPV6	X:X::X:X	Matches an IPv6 address.
IPV4_PREFIX	A.B.C.D/M	Matches an IPv4 prefix in CIDR notation.
IPV6_PREFIX	X:X::X:X/M	Matches an IPv6 prefix in CIDR notation.
MAC	M:A:C	Matches a 48-bit mac address.
MAC_PREFIX	M:A:C/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`
all other	`const char *`	`NULL`

Note the following details:

- Not all parameters are pointers, some are passed as values.
- When the type is not `const char *`, there will be an extra `_str` argument with type `const char *`.
- You can give a variable name not only to `VARIABLE` tokens but also to `word` tokens (e.g. constant words). This is useful if some parts of a command are optional. The type will be `const char *`.
- `[no]` will be passed as `const char *no`.
- 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, vty, argc, argv`) are still present and can be used. A DEFUN can simply be **edited into a DEFPY without further changes and it will still work**; this allows easy forward migration.
- A file may contain both `DEFUN` and `DEFPY` statements.

Getting a parameter dump

The clidef.py script can be called to get a list of DEFUNs/DEFPYs with the parameter name/type list:

```
lib/clippy python/clidef.py --all-defun --show lib/plist.c > /dev/null
```

The generated code is printed to stdout, the info dump to stderr. The `--all-defun` argument will make it process DEFUN blocks as well as DEFPYs, which is useful prior to converting some DEFUNs. **The dump does not list the "_str" arguments** to keep the output shorter.

Note that the `clidef.py` script cannot be run with python directly, it needs to be run with *clippy* since the latter makes the CLI parser available.

Include & Makefile requirements

A source file that uses DEFPY needs to include the `*_clippy.c` file **before all DEFPY statements**:

```
/* GPL header */
#include ...
...
#ifndef 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
```

5.4.3 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.

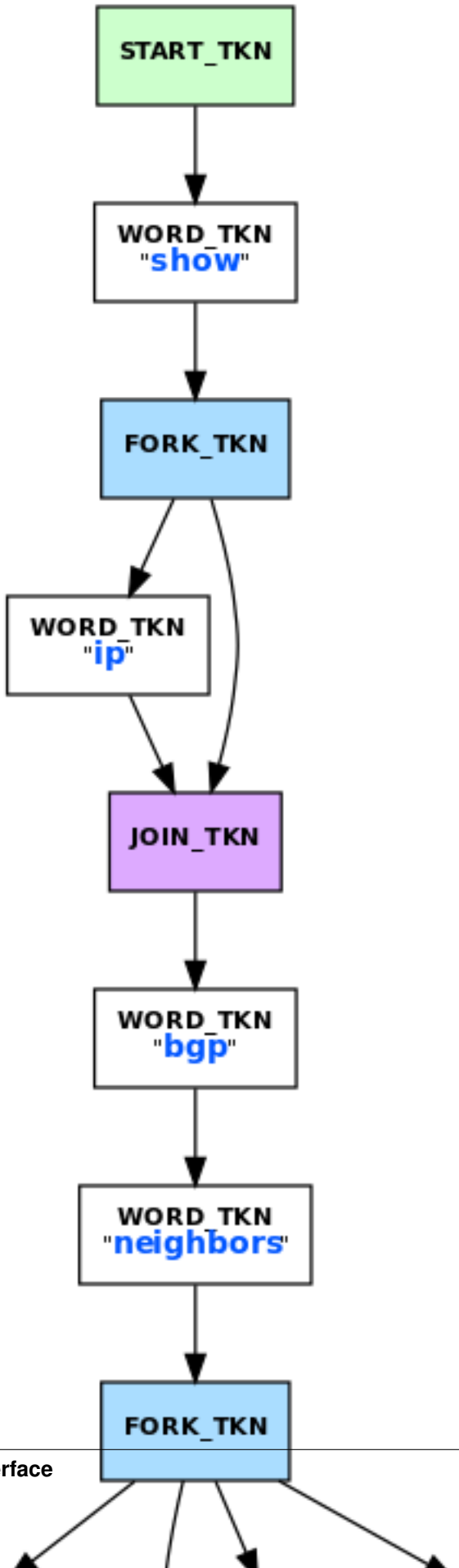
Keywords follow the same form, except that there is an edge from `JOIN` to `FORK`. Because of this the CLI graph cannot be called acyclic. There is special logic in the input matching code that keeps a stack of paths already taken through the node in order to disallow following the same path more than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


START_TKN
WORD_TKN
"show"
FORK_TKN
WORD_TKN
"ip"
JOIN_TKN
WORD_TKN
"bgp"
WORD_TKN
"neighbors"
FORK_TKN

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.

5.4.4 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.

5.4.5 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.

5.5 Modules

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

5.5.1 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.

5.5.2 Installation

Look for `moduledir` in `configure.ac`, default is normally `/usr/lib64/frr/modules` but depends on `--libdir` / `--prefix`.

The daemon's name is prepended when looking for a module, e.g. "snmp" tries to find "zebra_snmp" first when used in zebra. This is just to make it nicer for the user, with the snmp module having the same name everywhere.

Modules can be packaged separately from FRR. The SNMP and FPM modules are good candidates for this because they have dependencies (net-snmp / protobuf) that are not FRR dependencies. However, any distro packages should have an "exact-match" dependency onto the FRR package. Using a module from a different FRR version will probably blow up nicely.

For snapcraft (and during development), modules can be loaded with full path (e.g. -M `$SNAP/lib/frr/modules/zebra_snmp.so`). Note that libtool puts output files in the .libs directory, so during development you have to use `./zebra -M .libs/zebra_snmp.so`.

5.5.3 Creating a module

... best to look at the existing SNMP or FPM modules.

Basic boilerplate:

```
#include "hook.h"
#include "module.h"

static int
module_init (void)
{
  hook_register(frr_late_init, module_late_init);
  return 0;
}

FRR_MODULE_SETUP(
    .name = "my module",
```

(continues on next page)

(continued from previous page)

```
    .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.

5.5.4 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.

5.5.5 Command line parameters

Command line parameters can be passed directly to a module by appending a colon to the module name when loading it, e.g. `-M mymodule:myparameter`. The text after the colon will be accessible in the module's code through `THIS_MODULE->load_args`. For example, see how the format parameter is configured in the `zfpm_init()` function inside `zebra_fpm.c`.

5.5.6 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.

5.5.7 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.

CHAPTER 6

BGPD

6.1 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.

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 Design

Modules

The core design introduces an "nht" (next hop tracking) module in BGP and "rnh" (recursive nexthop) module in Zebra. The "nht" module provides the following APIs:

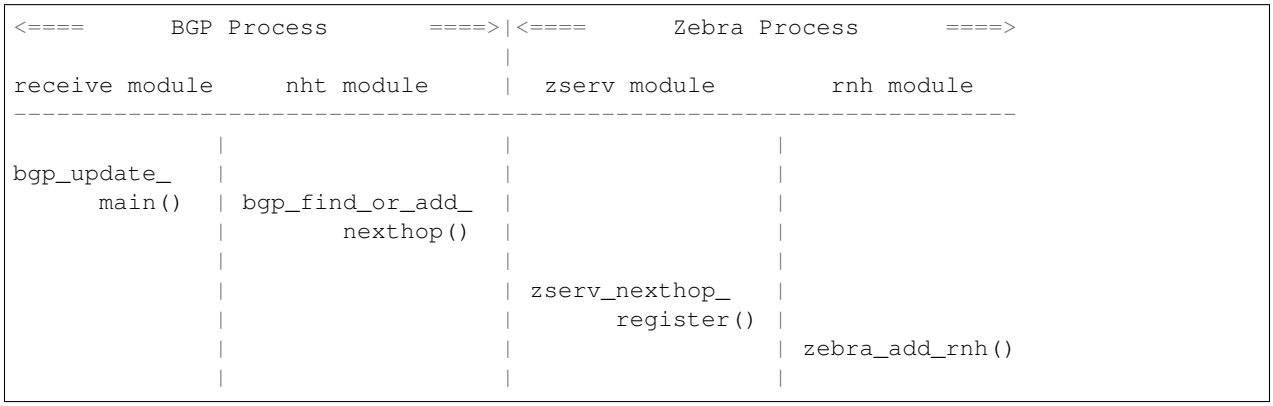
Function	Action
bgp_find_or_add_nexthop()	find or add a nexthop in BGP nexthop table
bgp_find_nexthop()	find a nexthop in BGP nexthop table
bgp_parse_nexthop_update()	parse a nexthop update message coming from zebra

The "rnh" module provides the following APIs:

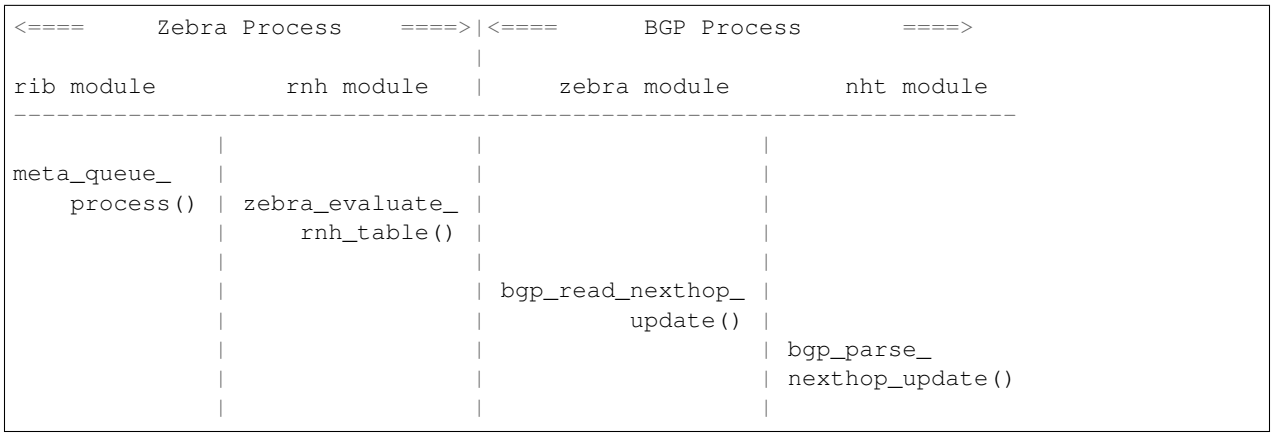
Function	Action
zebra_add_rnh()	add a recursive nexthop
zebra_delete_rnh()	delete a recursive nexthop
zebra_lookup_rnh()	lookup a recursive nexthop
zebra_add_rnh_client()	register a client for nexthop notifications against a recursive nexthop
zebra_remove_rnh_client()	remove the client registration for a recursive nexthop
zebra_evaluate_rnh_table()	(re)evaluate the recursive nexthop table (most probably because the main routing table has changed).
zebra_cleanup_rnh_client()	Cleanup a client from the "rnh" module data structures (most probably because the client is going away).

4.2. Control flow

The next hop registration control flow is the following:

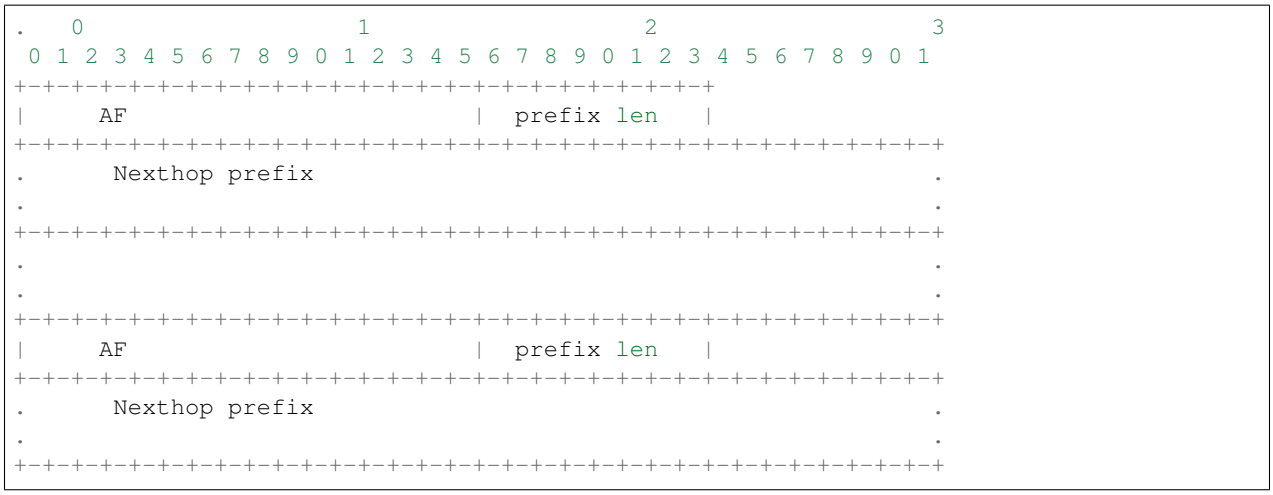


The next hop notification control flow is the following:

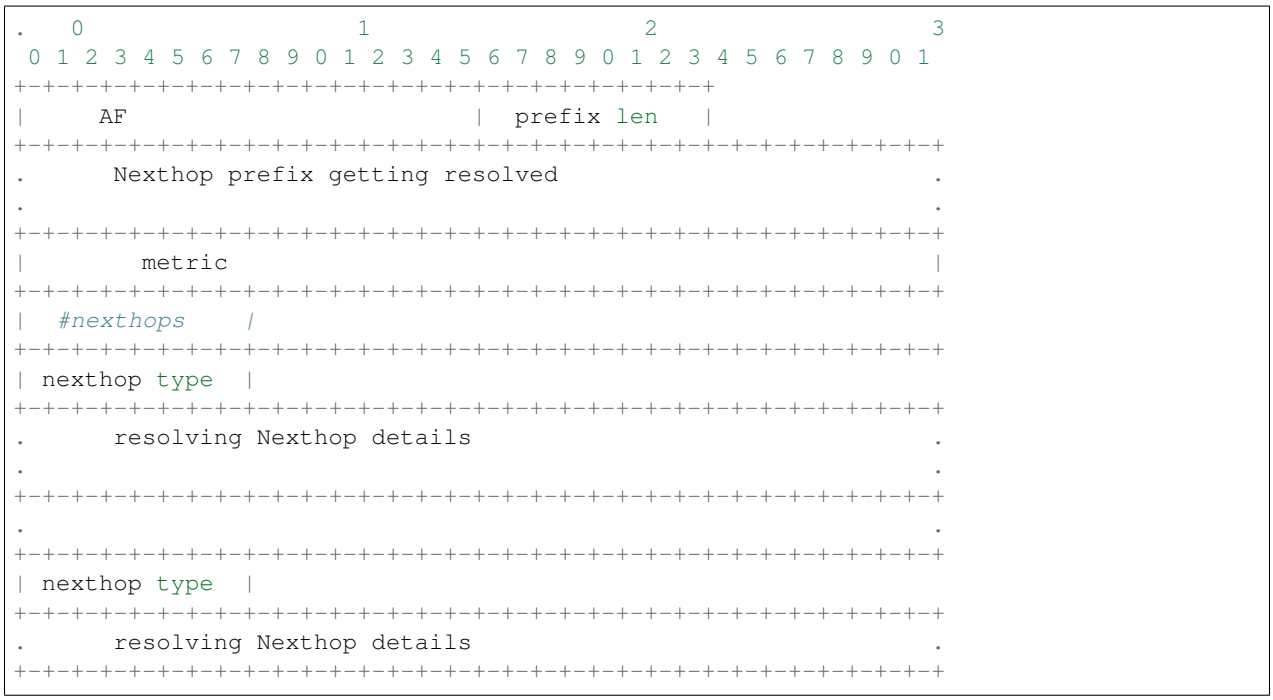


zclient message format

ZEBRA_NEXTHOP_REGISTER and ZEBRA_NEXTHOP_UNREGISTER messages are encoded in the following way:



ZEBRA_NEXTHOP_UPDATE message is encoded as follows:



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)

(continued from previous 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

6.2 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 5 6 7 8 9 10 11 12 13 14 15 16 17 18	ORIGIN AS_PATH NEXT_HOP MULTI_EXIT_DISC LOCAL_PREF ATOMIC_AGGREGATE AGGREGATOR COMMUNITIES ORIGINATOR_ID CLUSTER_LIST DPA ADVERTISER RCID_PATH MP_REACH_NLRI MP_UNREACH_NLRI EXT_COMMUNITIES AS4_PATH AS4_AGGREGATOR	[RFC 4271] [RFC 4271] [RFC 4271] [RFC 4271] [RFC 4271] [RFC 4271] [RFC 4271] [RFC 1997] [RFC 4456] [RFC 4456] [draft-ietf-idr-bgp-dpa-05.txt(expired)] [RFC 1863] [RFC 1863] [RFC 4760] [RFC 4760] [RFC 4360] [RFC 4893] [RFC 4893]

CHAPTER 7

OSPFD

7.1 OSPF API Documentation

7.1.1 Disclaimer

The OSPF daemon contains an API for application access to the LSA database. This API was created by Ralph Keller, originally as patch for Zebra. Unfortunately, the page containing documentation of the API is no longer online. This page is an attempt to recreate documentation for the API (with lots of help of the WayBackMachine)

7.1.2 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).

7.1.3 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.

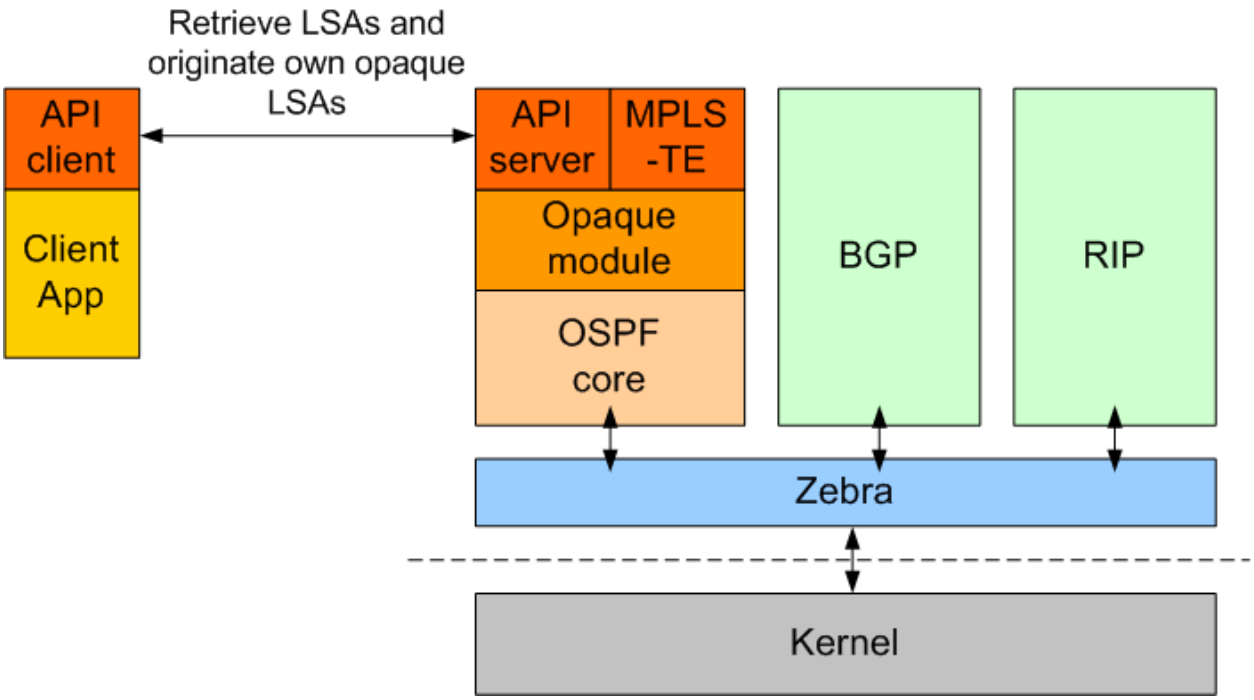


Fig. 1: image

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.

7.1.4 Installation & Configuration

Download FRRouting and unpack

Configure your frr version (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!
```

7.1.5 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:

```
> msr2:/home/keller/ospfapi/zebra/ospfd# ./ospfd -f /usr/local/etc/ospfd.conf
```

And on the neighboring router msr3:

```
> msr3:/home/keller/ospfapi/zebra/ospfd# ./ospfd -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 of no neighbors are present. If you try to originate even so no neighbor is 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.

7.1.6 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:

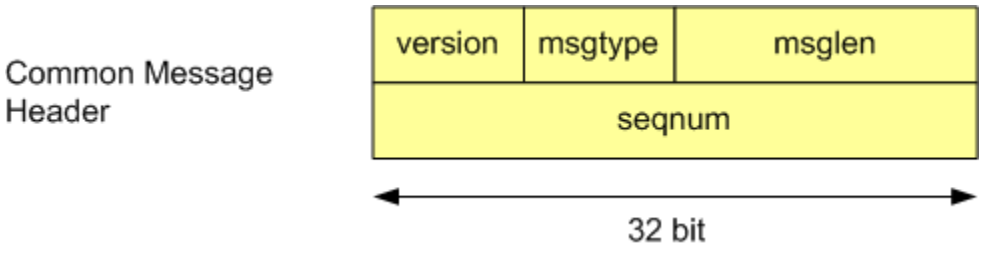


Fig. 2: image

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

Messages to OSPF deamon	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 deamon	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 to select according to the following types of origins:

Origin	Value
NON_SELF_ORIGINATED	0
SELF_ORIGINATED	1
ANY_ORIGIN	2

The reply message has on 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:

7.1.7 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.

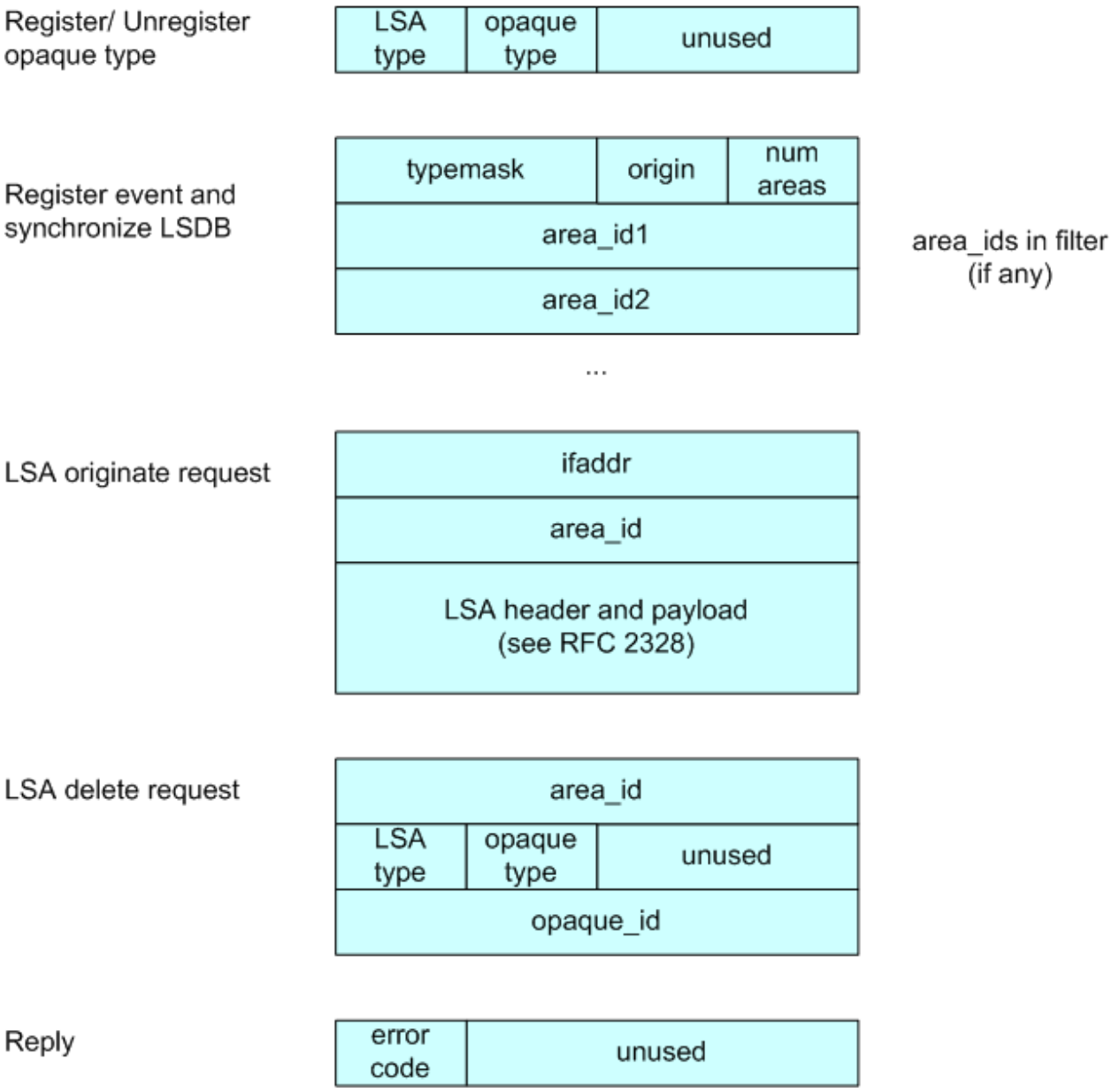

Register/ Unregister opaque type
LSA type
opaque type
unused
Register event and synchronize LSDB
typemask
origin
num areas
area_id1
area_id2
...
area_ids in filter (if any)
LSA originate request
ifaddr
area_id
LSA header and payload (see RFC 2328)
LSA delete request
area_id
LSA type
opaque type
unused
opaque_id
Reply
error code
unused

Fig. 3: image

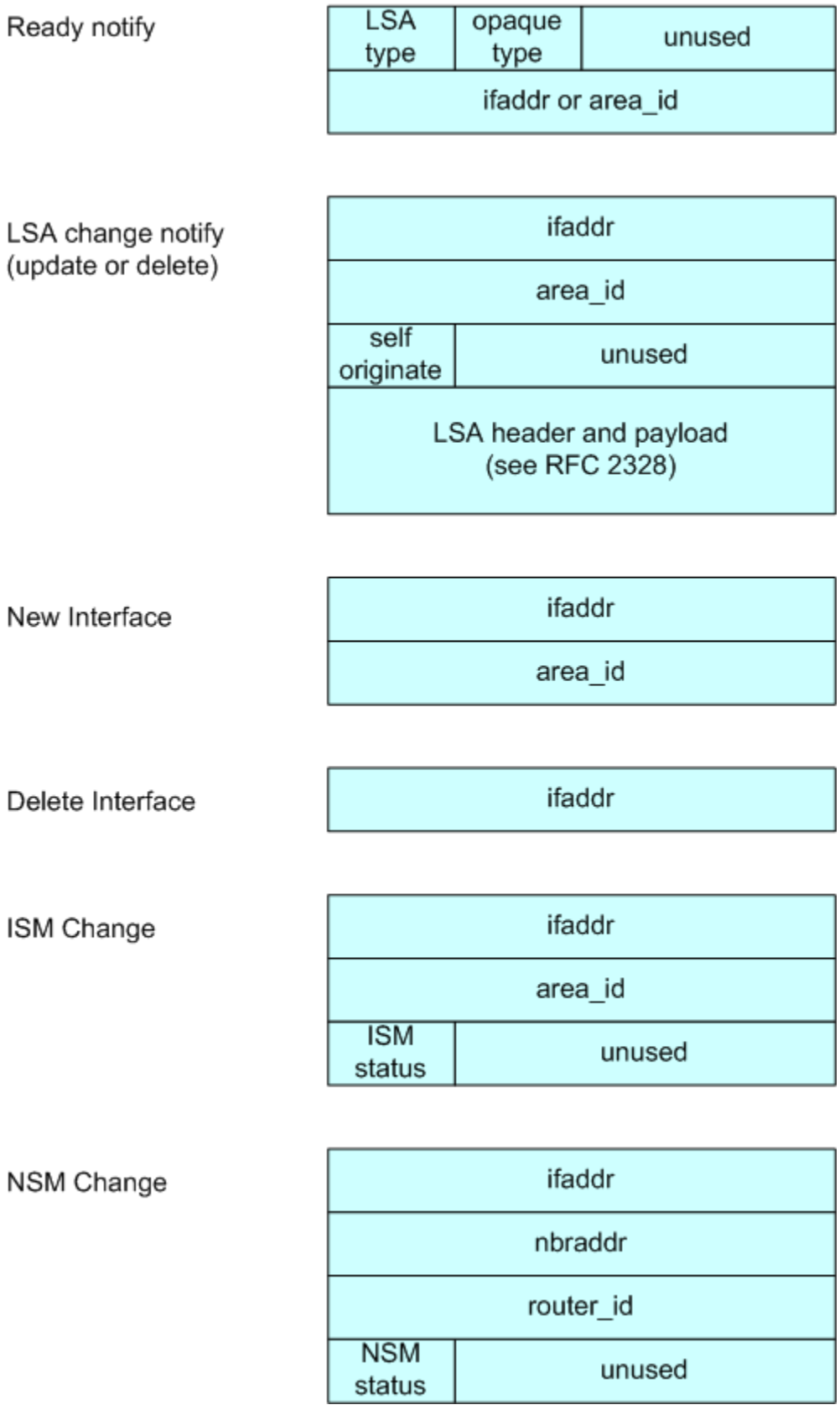

Ready notify
LSA type
opaque type
unused
ifaddr or area_id
LSA change notify (update or delete)
ifaddr
area_id
self originate
unused
LSA header and payload (see RFC 2328)
New Interface
ifaddr
area_id
Delete Interface
ifaddr
ISM Change
ifaddr
area_id
ISM status
unused
NSM Change
ifaddr
nbraddr
router_id
NSM status
unused

Fig. 4: image

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.

7.2 OSPF Segment Routing

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

7.2.1 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

7.2.2 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

7.2.3 Implementation details

Concepts

Segment Routing used 3 differents OPAQUE LSA in OSPF to carry the various information:

- **Router Information:** flood the Segment Routing capabilities of the node. This include the supported algorithms, the Segment Routing Global Block (SRGB) and the Maximum Stack Depth (MSD).
- **Extended Link:** flood the Adjaceny and Lan Adjacency Segment Identifier
- **Extended Prefix:** flood the Prefix Segment Identifier

The implementation follow 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 paramters 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.

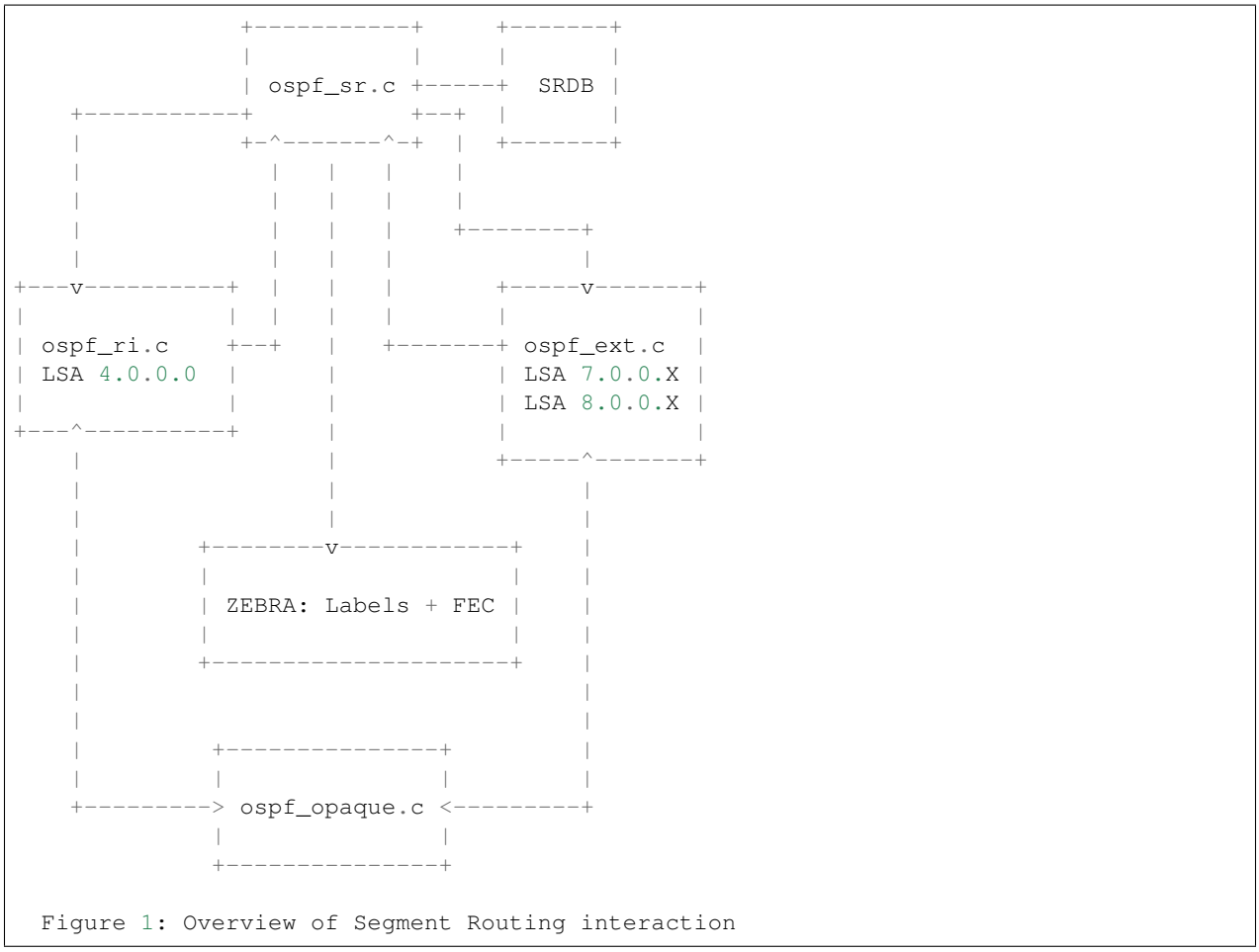


Figure 1: Overview of Segment Routing interaction

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 helpfull to handle real LSA deletion. In fact, LSA deletion is done by Flushing LSA i.e. flood LSA after seting 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 possiblity to modify this TLV is offer by the code.

When Opaque LSA Tyep 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 trigger automtically the flooding of Extended Link LSA for all ospf interface 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.

7.2.4 Configuration

Linux Kernel

In order to use OSPF Segment Routing, you must setup MPLS data plane. Up to know, only Linux Kernel version >= 4.5 is supported.

First, the MPLS modules aren't loaded by default, so you'll need to load them yourself:

```
modprobe mpls_router
modprobe mpls_gso
modprobe mpls_iptunnel
```

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

```
sysctl -w net.mpls.conf.enp0s9.input=1
sysctl -w net.mpls.conf.lo.input=1
sysctl -w net.mpls.platform_labels=1048575
```

The last line fix the maximum MPLS label value.

Once OSPFd start with Segment Routing, you could check that MPLS routes are enable with:

```
ip -M route
ip route
```

The first command show the MPLS LFIB table while the second show the FIB table which contains route with MPLS label encapsulation.

If you disable Penultimate Hop Popping with the *no-php-flag* (see below), you MUST check that RP filter is not enable for the interface you intend to use, especially the *lo* one. For that purpose, disable RP filtering with:

```
systcl -w net.ipv4.conf.all.rp_filter=0
sysctl -w net.ipv4.conf.lo.rp_filter=0
```

OSPFd

Here it is a simple example of configuration to enable Segment Routing. Note that *opaque capability* and *router information* must be set to activate Opaque LSA prior to Segment Routing.

```
router ospf
 ospf router-id 192.168.1.11
 capability opaque
  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 disbale Penultimate Hop Popping. This advertises peers that they MUST NOT pop the MPLS label prior to sending the packet.

7.2.5 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.

7.2.6 Credits

- Author: Anselme Sawadogo <anselmesawadogo@gmail.com>
- Author: Olivier Dugeon <olivier.dugeon@orange.com>
- Copyright (C) 2016 - 2018 Orange Labs http://www.orange.com

This work has been performed in the framework of the H2020-ICT-2014 project 5GEx (Grant Agreement no. 671636), which is partially funded by the European Commission.

CHAPTER 8

Zebra

8.1 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.

8.1.1 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.

8.2 Zebra Protocol Definition

8.2.1 Zebra Protocol Header Field Definitions

Length Total packet length including this header.

Marker Static marker. The marker value, when it exists, is 255 in all versions of Quagga. It is 254 in all versions of FRR. This is to allow version 0 headers (which do not include version explicitly) to be distinguished from versioned headers.

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

Command The Zebra protocol command.

Current Version

```
Version 5, 6

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |     Marker    |    Version    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             VRF ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Command           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Past Versions

```
Version 0

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |    Command    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Version 1, 2

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |     Marker    |    Version    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Command            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Version 3, 4

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |     Marker    |    Version    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             VRF ID            |            Command            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

8.2.2 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	44
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_L2VNI_ADD	63
ZEBRA_L2VNI_DEL	64
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

CHAPTER 9

VTYSH

See also:

Command Line Interface

9.1 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*.

9.1.1 Command Extraction

When VTYSH is a built, a Perl script named `extract.pl` searches the FRR codebase looking for `DEFUN`'s. It extracts these `DEFUN`'s, transforms them into `DEFSH`'s and appends them to `vtysh_cmd.c`. Each `DEFSH` contains the name of the command plus `_vtysh`, as well as a flag that indicates which daemons the command was found in. When the command is executed in VTYSH, this flag is inspected to determine which daemons to send the command

to. This way, commands are only sent to the daemons that know about them, avoiding spurious errors from daemons that don't have the command defined.

The extraction script contains lots of hardcoded knowledge about what sources to look at and what flags to use for certain commands.

9.1.2 Special DEFUNs

In addition to the vanilla `DEFUN` macro for defining CLI commands, there are several VTYSH-specific `DEFUN` variants that each serve different purposes.

`DEFSH` Used almost exclusively by generated VTYSH code. This macro defines a `cmd_element` with no handler function; the command, when executed, is simply forwarded to the daemons indicated in the daemon flag.

`DEFUN_NOSH` Used by daemons. Has the same expansion as a `DEFUN`, but `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.

9.1.3 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`.

9.2 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 NULL-terminated strings, whose content are CLI commands. Here is a typical message from VTYSH to a daemon:

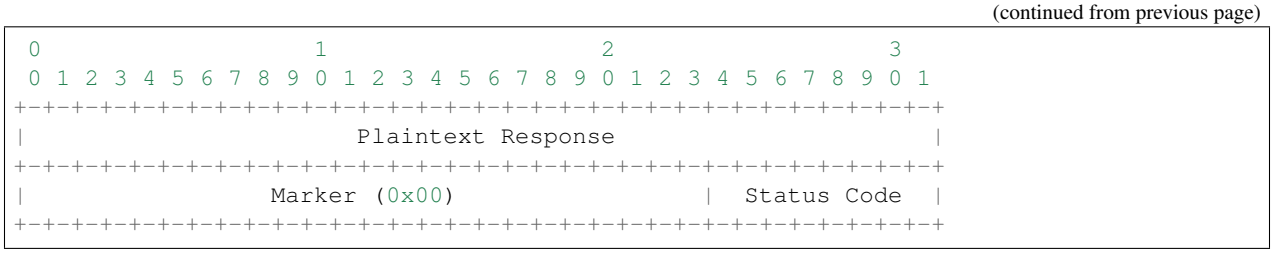
```
Request

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 NULL-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.

Index

Symbols

C

D

H

X