Conventions for working on Quagga

March 9, 2017

This is a living document describing the processes and guidelines for working on Quagga. You *must* read Section “REQUIRED READING”, before contributing to Quagga. Suggestions for updates, via the quagga-dev list, are welcome.

Contents

OBJECTIVES

The objectives of the Quagga project are to develop and implement high quality routing protocols and related software, maximising:

- Free software:
  - Quagga is and will remain a copyleft, free software project
  - Some non-core parts may be available under compatible, permissive licenses to facilitate code sharing, where contributors agree.
  - The test and integration orchestration infrastructure shall be free software, developed similarly to the rest of Quagga. Proprietary conformance suites may be among the test tools orchestrated.
- Openness and transparency
  - The business of the project shall be conducted on its public email lists, to the greatest extent possible.
  - The design of the software will be governed by open discussion on the public email lists.
  - Participants shall endeavour to be transparent about their interests in the project, and any associations likely to be relevant.
- Ethical behaviour:
  - The licenses of all copyright holders will be respected, and the project will err in their favour where there is reasonable doubt or legal advice to that effect.
  - Participants will behave with respect for others, and in a manner that builds and maintains the trust needed for productive collaboration.

See also the Section on CODE OF CONDUCT.

Governance

Quagga is a Sociocracy, as it has been since its earliest days.

Quagga was forked from GNU Zebra by Paul Jakma, who holds the domain name. Governance was soon devolved to a collective group, the maintainers, consisting of those who regularly contributed and reviewed code. The details can easily be changed.

You are free to use reason to *persuade* others to adopt some alternative. If, after that, you truly can not abide by what is mutually agreeable, you are asked to do the honourable thing: take your copy of the code, make your apologies, and be on your way with good grace.

Those who repeatedly violate the Code of Conduct will be asked to leave.

Holding of project assets

One or more mature, independent trustees, with technical and free software experience, will be appointed as the executor(s) for key assets of the project to ensure continuity, such as the domain name.

Should a corporate vehicle ever be created to hold such assets it **must**:

- Publish up to date accounts on a regular business.

- Generally operate openly and transparently.
- Have control distributed, with a significant degree of control held independent of any contributors with business interests in the software.
- Carry out no other business itself that may be seen to conflict or compete with the business of others in the community.
- Have all officers disclose all interests that could be seen to have a bearing on the project, as far as is reasonable.

It not clear at this time that the overheads and potential liabilities of such a vehicle would be appropriate for this project. These principles should though still be applied, where possible, to any non-corporate body formed around the project.

CODE OF CONDUCT

Participants will treat each other with respect and integrity. Participants will build and treasure the trust that is required for parties to successfully collaborate together on free software. Particularly when those parties may have competing interests. The following principles and guidelines should be followed to foster that trust:

- Participants should be open about their goals, and their interests.
  - Business associations with other participants should be disclosed, so far as is reasonable and where applicable. E.g., if there is voting on matters, or in endorsements or objections to contributions.
  - Other associations and interests that may be relevant should be disclosed, to the degree necessary to avoid any perception by others of conflicts of interests or of deception.
  - Be open about your goals, so as to maximise the common understanding and minimise any misunderstandings and disputes.
- Design should be done in the open
  - Do your design on list, ahead of significant implementation. Avoid "Surprise!" development where possible.
  - Where significant implementation work must be done behind closed doors, accept that you may be asked to rework it, potentially from scratch once you take it public.
  - Get "buy in" from others ahead of time, to avoid disappointment.
- Interaction
  - Feedback on design should be constructive, thoughtful and understanding.
  - Avoid personalising matters. Discuss the idea, the code, the abstract subject and avoid unnecessary personal pronouns.
  - Avoid language that paints either party into a corner. Leave some room for doubt. Ask questions, rather than make assertions, where possible.
- Disputes should be resolved through calm, analytic discussion
  - Separate out as much of the matter under dispute into principles that can be agreed on, and into the objective domain (by measurement or logic).
  - Seek ways to resolve any remaining subjective differences by alternate paths that can accommodate both sides, e.g., through abstraction or modularisation.
  - Aim for Win-Win.
- Respect others
  - Avoid passive-aggressive behaviours. E.g., tit-for-tat non-constructive behaviour. Be explicit.
  - It is acceptable for management to allocate resources on development according to their

need. It is not acceptable to try use external, management intervention to over-turn positions held by participants.

REQUIRED READING

Note well: By proposing a change to Quagga, by whatever means, you are implicitly agreeing:

- To licence your contribution according to the licence of any files in Quagga being modified, *and* according to the COPYING file in the top-level directory of Quagga, other than where the contribution explicitly and clearly indicates otherwise.
- That it is your responsibility to ensure you hold whatever rights are required to be able to contribute your changes under the licenses of the files in Quagga being modified, and the top-level COPYING file.
- That it is your responsibility to give with the contribution a full account of all interests held and claims in the contribution; such as through copyright, trademark and patent laws or otherwise; that are known to you or your associates (e.g. your employer).

Before contributing to Quagga, you *must* also read Section COMMIT MESSAGES.

You *should* ideally read the entire document, as it contains useful information on the community norms and how to implement them.

Please note that authorship and any relevant other rights information should be *explicitly* stated with the contribution. A “Signed-off-by” line is *not* sufficient. The “Signed-off-by” line is not used by the Quagga project.

You may document applicable copyright claims to files being modified or added by your contribution. For new files, the standard way is to add a string in the following format near the beginning of the file:

```
Copyright (C) <Year> <name of person/entity>[, optional contact details]
```

When adding copyright claims for modifications to an existing file, please preface the claim with “Portions:” on a line before it and indent the “Copyright . . . ” string. If such a case already exists, add your indented claim immediately after. E.g.:

```
Portions:
  Copyright (C) <Year> <Entity A> ....
  Copyright (C) <Year> <Your details> [optional brief change description]
```

GUIDELINES FOR HACKING ON QUAGGA

GNU coding standards apply. Indentation follows the result of invoking GNU indent (as of 2.2.8a) with the —nut argument.

Originally, tabs were used instead of spaces, with tabs are every 8 columns. However, tab’s interoperability issues mean space characters are now preferred for new changes. We generally only clean up whitespace when code is unmaintainable due to whitespace issues, to minimise merging conflicts.

Be particularly careful not to break platforms/protocols that you cannot test.

Parsers or packet-writers of data from untrusted parties, particularly remote ones, *MUST* use the lib/stream bounded-buffer abstraction, and use its checked getters and putters. Twiddling of pointers based on contents of untrusted data is *strongly* discouraged - any such code is not acceptable, unless there are very good reasons (e.g. compatibility with external or old code that is not easily rewritten).

New code should have good comments, which explain why the code is correct. Changes to existing code should in many cases upgrade the comments when necessary for a reviewer to conclude that the change has no unintended consequences.

Each file in the Git repository should have a git format-placeholder (like an RCS Id keyword), somewhere very near the top, commented out appropriately for the file type. The placeholder used for Quagga (replacing <dollar> with $) is:

```
$QuaggaId: <dollar>Format:%an, %ai, %h<dollar> $
```

See line 2 of HACKING.tex, the source for this document, for an example.

This placeholder string will be expanded out by the 'git archive' commands, which is used to generate the tar archives for snapshots and releases.

Please document fully the proper use of a new function in the header file in which it is declared. And please consult existing headers for documentation on how to use existing functions. In particular, please consult these header files:

lib/log.h logging levels and usage guidance

[more to be added]

If changing an exported interface, please try to deprecate the interface in an orderly manner. If at all possible, try to retain the old deprecated interface as is, or functionally equivalent. Make a note of when the interface was deprecated and guard the deprecated interface definitions in the header file, i.e.:

```
/* Deprecated: 20050406 */
#if !defined(QUAGGA_NO_DEPRECATED_INTERFACES)
#warning "Using deprecated <libname> (interface(s)|function(s))"
...
#endif /* QUAGGA_NO_DEPRECATED_INTERFACES */
```

This is to ensure that the core Quagga sources do not use the deprecated interfaces (you should update Quagga sources to use new interfaces, if applicable), while allowing external sources to continue to build. Deprecated interfaces should be excised in the next unstable cycle.

Note: If you wish, you can test for GCC and use a function marked with the 'deprecated' attribute. However, you must provide the warning for other compilers.

If changing or removing a command definition, *ensure* that you properly deprecate it - use the _DEPRECATED form of the appropriate DEFUN macro. This is *critical.* Even if the command can no longer function, you *MUST* still implement it as a do-nothing stub.

Failure to follow this causes grief for systems administrators, as an upgrade may cause daemons to fail to start because of unrecognised commands. Deprecated commands should be excised in the next unstable cycle. A list of deprecated commands should be collated for each release.

See also Section SHARED LIBRARY VERSIONING below.

YOUR FIRST CONTRIBUTIONS

Routing protocols can be very complex sometimes. Then, working with an Opensource community can be complex too, but usually friendly with anyone who is ready to be willing to do it properly.

- First, start doing simple tasks. Quagga's patchwork is a good place to start with. Pickup some patches, apply them on your git trie, review them and send your ack't or review comments. Then, a maintainer will apply the patch if ack't or the author will have to provide a new update. It help a lot to drain the patchwork queues. See http://patchwork.quagga.net/project/quagga/list/
- The more you'll review patches from patchwork, the more the Quagga's maintainers will be willing to consider some patches you will be sending.
- start using git clone, pwclient http://patchwork.quagga.net/help/pwclient/

```
$ pwclient list -s new
ID    State        Name
--    -----        ----
179   New          [quagga-dev,6648] Re: quagga on FreeBSD 4.11 (gcc-2.95)
181   New          [quagga-dev,6660] proxy-arp patch
[...]

$ pwclient git-am 1046
```

HANDY GUIDELINES FOR MAINTAINERS

Get your cloned trie:

```
git clone vjardin@git.sv.gnu.org:/srv/git/quagga.git
```

Apply some ack't patches:

```
pwclient git-am 1046
  Applying patch #1046 using 'git am'
  Description: [quagga-dev,11595] zebra: route_unlock_node is missing in "show ip[v6] r
  Applying: zebra: route_unlock_node is missing in "show ip[v6] route <prefix>" comman
```

Run a quick review. If the ack't was not done properly, you know who you have to blame.

Push the patches:

```
git push
```

Set the patch to accepted on patchwork

```
pwclient update -s Accepted 1046
```

COMPILE-TIME CONDITIONAL CODE

Please think very carefully before making code conditional at compile time, as it increases maintenance burdens and user confusion. In particular, please avoid gratuitous —enable-…

switches to the configure script - typically code should be good enough to be in Quagga, or it shouldn't be there 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. I.e. this:

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

rather than:

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

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

COMMIT MESSAGES

The commit message requirements are:

- The message *MUST* provide a suitable one-line summary followed by a blank line as the very first line of the message, in the form:

  ```
  topic: high-level, one line summary
  ```

  Where topic would tend to be name of a subdirectory, and/or daemon, unless there's a more suitable topic (e.g. 'build'). This topic is used to organise change summaries in release announcements.

- It should have a suitable "body", which tries to address the following areas, so as to help reviewers and future browsers of the code-base understand why the change is correct (note also the code comment requirements):
  - The motivation for the change (does it fix a bug, if so which? add a feature?)
  - The general approach taken, and trade-offs versus any other approaches.
  - Any testing undertaken or other information affecting the confidence that can be had in the change.
  - Information to allow reviewers to be able to tell which specific changes to the code are intended (and hence be able to spot any accidental unintended changes).

- The commit message *must* give details of all the authors of the change, beyond the person listed in the Author field. Any and all affiliations which may have a bearing on copyright in any way should be clearly stated, unless those affiliations are already obvious from other details, e.g. from the email address. This would cover employment and contracting obligations (give details).

  Note: Do not rely on "Signed-off-by" for this, be explicit.

- If the change introduces a new dependency on any code or other copyrighted material, please explicitly note this. Give details of what that external material is, the copyright licence the material may be used under, and the nature of the dependency.

The one-line summary must be limited to 54 characters, and all other lines to 72 characters.

Commit message bodies in the Quagga project have typically taken the following form:

- An optional introduction, describing the change generally.
- A short description of each specific change made, preferably:
  - file by file
    - function by function (use of "ditto", or globs is allowed)

Contributors are strongly encouraged to follow this form.

This itemised commit messages allows reviewers to have confidence that the author has self-reviewed every line of the patch, as well as providing reviewers a clear index of which changes are intended, and descriptions for them (C-to-english descriptions are not desirable - some discretion is useful). For short patches, a per-function/file break-down may be redundant. For longer patches, such a break-down may be essential. A contrived example (where the general discussion is obviously somewhat redundant, given the one-line summary):

```
zebra: Enhance frob FSM to detect loss of frob

Add a new DOWN state to the frob state machine to allow the barinator to
detect loss of frob.

* frob.h: (struct frob) Add DOWN state flag.
* frob.c: (frob_change) set/clear DOWN appropriately on state change.
* bar.c: (barinate) Check frob for DOWN state.
```

Please have a look at the git commit logs to get a feel for what the norms are.

Note that the commit message format follows git norms, so that "git log –oneline" will have useful output.

HACKING THE BUILD SYSTEM

If you change or add to the build system (configure.ac, any Makefile.am, etc.), please heck that the following things still work:

- make dist
- resulting dist tarball builds
- out-of-tree builds

This can be achieved by running 'make distcheck'

The quagga.net site relies on make dist to work to generate snapshots. It must work. Common problems are to forget to have some additional file included in the dist, or to have a make rule refer to a source file without using the srcdir variable.

RELEASE PROCEDURE

To make a release:

- Edit configure.ac, bump the version and commit the change with a "release: <version>" subject.

The 'release.sh' script should then be used. It should be run with 2 arguments, the release tag for the release to be carried, and the tag of the previous release, e.g.:

```
release.sh quagga-1.1.1 quagga-1.1.0
```

The 'release.sh' will carry out these steps for you:

- Tag the appropriate commit with a release tag (follow existing conventions), with:

  git tag -u

- Create a fresh tar archive of the quagga.net repository, and do a test build. Use git archive to ensure it consists of files in the repository, and to carry out the keyword expansions. Do NOT do this in a subdirectory of the Quagga sources, autoconf will think it's a sub-package and fail to include neccessary files.

  ```
      git archive ... <quagga-release-tag> | tar xC ..

      autoreconf -i && ./configure && make && make dist-gzip
  ```

- Similarly test the dist tarball produced. This is the tarball to be released. This is important.
- Sign the dist tarball to be released

  ```
  gpg -u 54CD2E60 -a --detach-sign quagga-0.99.99.99.tar
  ```

The 'release.sh' script, if finishes successfully, will print out instructions on the files it has created and the details on remaining steps to be carried out to complete the release. Which roughly are:

- Upload the release tarball, its PGP signature, and the full changelog to the public release area on Savannah
- Add the version number on https://bugzilla.quagga.net/, under Administration, Products, "Quagga", Edit versions, Add a version.
- Post a news entry on Savannah
- Send a mail to quagga-dev and quagga-users

If any errors occur, move tags as needed and start over again with the release.sh script. Do not try to append stuff to tarballs, as this has produced non-standards-conforming tarballs in the past.

[TODO: collation of a list of deprecated commands. Possibly can be scripted to extract from vtysh/vtysh_cmd.c]

TOOL VERSIONS

Require versions of support tools are listed in INSTALL.quagga.txt. Required versions should only be done with due deliberation, as it can cause environments to no longer be able to compile quagga.

SHARED LIBRARY VERSIONING

[this section is at the moment just gdt's opinion]

Quagga builds several shared libaries (lib/libzebra, ospfd/libospf, ospfclient/libsopfapiclient). These may be used by external programs, e.g. a new routing protocol that works with the zebra daemon, or ospfapi clients. The libtool info pages (node Versioning) explain when major and minor version numbers should be changed. These values are set in Makefile.am near the definition of the library. If you make a change that requires changing the shared library version, please update Makefile.am.

libospf exports far more than it should, and is needed by ospfapi clients. Only bump libospf for changes to functions for which it is reasonable for a user of ospfapi to call, and please err on the side of not bumping.

There is no support intended for installing part of zebra. The core library libzebra and the included daemons should always be built and installed together.

GIT COMMIT SUBMISSION

The preferred method for submitting changes is to provide git commits via a publicly-accessible git repository, which the maintainers can easily pull.

The commits should be in a branch based off the Quagga.net master - a "feature branch". Ideally there should be no commits to this branch other than those in master, and those intended to be submitted. However, merge commits to this branch from the Quagga master are permitted, though strongly discouraged - use another (potentially local and throw-away) branch to test merge with the latest Quagga master.

Recommended practice is to keep different logical sets of changes on separate branches - "topic" or "feature" branches. This allows you to still merge them together to one branch (potentially local and/or "throw-away") for testing or use, while retaining smaller, independent branches that are easier to merge.

All content guidelines in Section PATCH SUBMISSION apply.

PATCH SUBMISSION

- For complex changes, contributors are strongly encouraged to first start a design discussion on the quagga-dev list *before* starting any coding.
- Send a clean diff against the 'master' branch of the quagga.git repository, in unified diff format, preferably with the '-p' argument to show C function affected by any chunk, and with the -w and -b arguments to minimise changes. E.g:

  git diff -up mybranch..remotes/quagga.net/master

It is preferable to use git format-patch, and even more preferred to publish a git repository (see Section GIT COMMIT SUBMISSION).

If not using git format-patch, Include the commit message in the email.

- After a commit, code should have comments explaining to the reviewer why it is correct, without reference to history. The commit message should explain why the change is correct.
- Include NEWS entries as appropriate.
- Include only one semantic change or group of changes per patch.
- Do not make gratuitous changes to whitespace. See the w and b arguments to diff.
- Changes should be arranged so that the least controversial and most trivial are first, and the most complex or more controversial are last. This will maximise how many the Quagga maintainers can merge, even if some other commits need further work.
- Providing a unit-test is strongly encouraged. Doing so will make it much easier for maintainers to have confidence that they will be able to support your change.
- New code should be arranged so that it easy to verify and test. E.g. stateful logic should be separated out from functional logic as much as possible: wherever possible, move complex logic out to smaller helper functions which access no state other than their arguments.
- State on which platforms and with what daemons the patch has been tested. Understand that if the set of testing locations is small, and the patch might have unforeseen or hard to fix consequences that there may be a call for testers on quagga-dev, and that the patch may be blocked until test results appear.

  If there are no users for a platform on quagga-dev who are able and willing to verify -current occasionally, that platform may be dropped from the “should be checked” list.

PATCH APPLICATION

- Only apply patches that meet the submission guidelines.
- If the patch might break something, issue a call for testing on the mailing-list.
- Give an appropriate commit message (see above), and use the –author argument to git-commit, if required, to ensure proper attribution (you should still be listed as committer)
- Immediately after commiting, double-check (with git-log and/or gitk). If there’s a small mistake you can easily fix it with ‘git commit –amend ..’
- When merging a branch, always use an explicit merge commit. Giving –no-ff ensures a merge commit is created which documents “this human decided to merge this branch at this time”.

STABLE PLATFORMS AND DAEMONS

The list of platforms that should be tested follow. This is a list derived from what quagga is thought to run on and for which maintainers can test or there are people on quagga-dev who are able and willing to verify that -current does or does not work correctly.

- BSD (Free, Net or Open, any platform)
- GNU/Linux (any distribution, i386)
- Solaris (strict alignment, any platform)
- future: NetBSD/sparc64

The list of daemons that are thought to be stable and that should be tested are:

- zebra
- bgpd
- ripd
- ospfd
- ripngd

Daemons which are in a testing phase are

- ospf6d
- isisd
- watchquagga

USEFUL URLs

- The project homepage is at:

  https://www.quagga.net
- Bugs can be reported via Bugzilla at:

  https://bugzilla.quagga.net
- Buildbot runs CI tests, and is at:

  https://buildbot.quagga.net

  It tests commits and jobs submitted on local changes via 'buildbot try ... ' for developers.
- Patchwork tracks any patches emailed to the quagga-dev list, and is at:

  https://patchwork.quagga.net/project/quagga/list/

BUILDBOT

The buildbot client can be used to test changes before committing, with "buildbot try".

- Ask for a buildbot account
- Install the buildbot client
- Configure it, e.g.:

```
$ cat ~/.buildbot/options
try_master = 'radia.quagga.net:8031'
try_username = 'paul'
try_password = 'password123'
try_vc = 'git'
try_branch = 'master'
try_wait = True
$ buildbot try -c pb --get-builder-names
using 'pb' connect method
The following builders are available for the try scheduler:
build-fedora-24
...
```

- You can then submit your local changes to try build:

  ```
  $ buildbot try -c pb
  ```

  or use the -b argument to limit to a specific builder (recommended).

  ```
  $ buildbot try -c pb -b build-distcheck
  ```

- To test a series of locally committed change use git diff:

  ```
  git diff <base rev>.. | buildbot try -c pb --vc git \
   -b build-centos-7  --branch=volatile/next --diff=- -p 1
  ```