Pacemaker Development

Working with the Pacemaker Code Base

, Written by the Pacemaker project contributors

Pacemaker Development: Working with the Pacemaker Code Base

by

Abstract

This document has guidelines and tips for developers interested in editing Pacemaker source code and submitting changes for inclusion in the project.

Table of Contents

List of Examples

Chapter 1. Frequently Asked Questions

1.1. Who is this document intended for?

Anyone who wishes to read and/or edit the Pacemaker source code. Casual contributors should feel free to read just this FAQ, and consult other sections as needed.

1.2. Where is the source code for Pacemaker?

The source code for Pacemaker [https://github.com/ClusterLabs/pacemaker] is kept on GitHub [https://github.com/], as are all software projects under the ClusterLabs [https://github.com/ClusterLabs] umbrella. Pacemaker uses Git [https://git-scm.com/] for source code management. If you are a Git newbie, the gittutorial(7) man page [http://schacon.github.io/git/gittutorial.html] is an excellent starting point. If you're familiar with using Git from the command line, you can create a local copy of the Pacemaker source code with: `git clone https://github.com/ClusterLabs/pacemaker.git`

1.3. What are the different Git branches and repositories used for?

- The master branch [https://github.com/ClusterLabs/pacemaker/tree/master] is the primary branch used for development.
- The 2.0 branch [https://github.com/ClusterLabs/pacemaker/tree/2.0] contains the latest official release, and normally does not receive any changes. During the release cycle, it will contain release candidates for the next official release, and will receive only bug fixes.
- The 1.1 branch [https://github.com/ClusterLabs/pacemaker/tree/1.1] is similar to both the master and 2.0 branches, but for the 1.1 release series. The 1.1 branch receives only backports of certain bug fixes and backward-compatible features from the master branch. During the release cycle, it will contain release candidates for the next official 1.1 release.
- The 1.0 repository [https://github.com/ClusterLabs/pacemaker-1.0] is a frozen snapshot of the 1.0 release series, and is no longer developed.
- Messages will be posted to the developers@clusterlabs.org [http://clusterlabs.org/mailman/listinfo/developers] mailing list during the release cycle, with instructions about which branches to use when submitting requests.

1.4. How do I build from the source code?

See INSTALL.md [https://github.com/ClusterLabs/pacemaker/blob/master/INSTALL.md] in the main checkout directory.

1.5. What coding style should I follow?

You'll be mostly fine if you simply follow the example of existing code. When unsure, see the relevant section of this document for language-specific recommendations. Pacemaker has grown and evolved organically over many years, so you will see much code that doesn't conform to the current guidelines. We discourage making changes solely to bring code into conformance, as any change requires developer time for review and opens the possibility of adding bugs. However, new code should follow the guidelines, and it is fine to bring lines of older code into conformance when modifying that code for other reasons.

1.6. How should I format my Git commit messages?

See existing examples in the git log. The first line should look like `change-type: affected-code: explanation` where `change-type` should be `Fix` or `Bug` for most bug fixes, `Feature` for new features, `Log` for changes to log messages or handling, `Doc` for changes to documentation or comments, or `Test` for changes in CTS and regression tests. You will sometimes see `Low`, `Med` (or `Mid`) and `High` used instead for bug fixes, to indicate the severity. The important thing is that only commits with `Feature`, `Fix`, `Bug`, or `High` will automatically be included in the change log for the next release. The `affected-code` is the name of the component(s) being changed, for example, `controller` or `libcrmcommon` (it's more free-form, so don't sweat getting it exact). The `explanation` briefly describes the change. The git project recommends the entire summary line stay under 50 characters, but more is fine if needed for clarity. Except for the most simple and obvious of changes, the summary should be followed by a blank line and then a longer explanation of *why* the change was made.

1.7. How can I test my changes?

Most importantly, Pacemaker has regression tests for most major components; these will automatically be run for any pull requests submitted through GitHub. Additionally, Pacemaker's Cluster Test Suite (CTS) can be used to set up a test cluster and run a wide variety of complex tests. This document will have more detail on testing in the future.

1.8. What is Pacemaker's license?

Except where noted otherwise in the file itself, the source code for all Pacemaker programs is licensed under version 2 or later of the GNU General Public License (GPLv2+ [https://www.gnu.org/licenses/gpl-2.0.html]), its headers and libraries under version 2.1 or later of the less restrictive GNU Lesser General Public License (LGPLv2.1+ [https://www.gnu.org/licenses/lgpl-2.1.html]), its documentation under version 4.0 or later of the Creative Commons Attribution-ShareAlike International Public License (CC-BY-SA-4.0 [https://creativecommons.org/licenses/by-sa/4.0/legalcode]), and its init scripts under the Revised BSD [https://opensource.org/licenses/BSD-3-Clause] license. If you find any deviations from this policy, or wish to inquire about alternate licensing arrangements, please e-mail the developers@ClusterLabs.org [mailto:developers@clusterlabs.org] mailing list. Licensing issues are also discussed on the ClusterLabs wiki [https://wiki.clusterlabs.org/wiki/License].

1.9. How can I contribute my changes to the project?

Contributions of bug fixes or new features are very much appreciated! Patches can be submitted as pull requests [https://help.github.com/articles/using-pull-requests/] via GitHub (the preferred method, due to its excellent features [https://github.com/features/]), or e-mailed to the developers@clusterlabs.org [http://clusterlabs.org/mailman/listinfo/developers] mailing list as an attachment in a format Git can import. Authors may only submit changes that they have the right to submit under the open source license indicated in the affected files.

1.10. What if I still have questions?

Ask on the developers@clusterlabs.org [http://clusterlabs.org/mailman/listinfo/developers] mailing list for development-related questions, or on the users@clusterlabs.org [http://clusterlabs.org/mailman/listinfo/users] mailing list for general questions about using Pacemaker. Developers often also hang out on freenode's [http://freenode.net/] #clusterlabs IRC channel.

Chapter 2. General Guidelines for All Languages

Copyright

When copyright notices are added to a file, they should look like this:

Example 2.1. Copyright Notice Format

Copyright YYYY[-YYYY] the Pacemaker project contributors

The version control history for this file may have further details.

The first YYYY is the year the file was originally published. The original date is important for two reasons: when two entities claim copyright ownership of the same work, the earlier claim generally prevails; and copyright expiration is generally calculated from the original publication date. [1]

If the file is modified in later years, add -YYYY with the most recent year of modification. Even though Pacemaker is an ongoing project, copyright notices are about the years of *publication* of specific content.

Copyright notices are intended to indicate, but do not affect, copyright *ownership*, which is determined by applicable laws and regulations. Authors may put more specific copyright notices in their commit messages if desired.

[1] See the U.S. Copyright Office's "Compendium of U.S. Copyright Office Practices" [https://www.copyright.gov/comp3/], particularly "Chapter 2200: Notice of Copyright", sections 2205.1(A) and 2205.1(F), or "Updating Copyright Notices" [https://techwhirl.com/updating-copyright-notices/] for a more readable summary.

Chapter 3. C Coding Guidelines

Style Guidelines

Pacemaker is a large, distributed project accepting contributions from developers with a wide range of skill levels and organizational affiliations, and maintained by multiple people over long periods of time. The guidelines in this section are not technically better than alternative approaches, but make project management easier.

Many of these simply ensure stylistic consistency, which makes reading, writing, and reviewing code easier.

C Boilerplate

Every C file should start with a short copyright notice:

```
/*
 * Copyright <YYYY[-YYYY]> the Pacemaker project contributors
 *
 * The version control history for this file may have further details.
 *
 * This source code is licensed under <LICENSE> WITHOUT ANY WARRANTY.
 */
```

`<LICENSE>` should follow the policy set forth in the `COPYING` [https://github.com/ClusterLabs/pacemaker/blob/master/COPYING] file, generally one of "GNU General Public License version 2 or later (GPLv2+)" or "GNU Lesser General Public License version 2.1 or later (LGPLv2.1+)".

Header files should additionally protect against multiple inclusion by defining a unique symbol.

```
#ifndef MY_HEADER_NAME__H
#  define MY_HEADER_NAME__H

// header code here

#endif // MY_HEADER_NAME__H
```

Public API header files should additionally declare "C" compatibility for inclusion by C++, and give a Doxygen file description. For example:

```
#ifdef __cplusplus
extern "C" {
#endif

/*!
 * \file
 * \brief My brief description here
 * \ingroup core
```

```
 */

// header code here

#ifdef __cplusplus
}
#endif
```

Line Formatting

- Indentation must be 4 spaces, no tabs.
- Do not leave trailing whitespace.
- Lines should be no longer than 80 characters unless limiting line length significantly impacts readability.

Pointers

- The * goes by the variable name, not the type:

```
char *foo;
```

- Use a space before the * and after the closing parenthesis in a cast:

```
char *foo = (char *) bar;
```

Function Definitions

- In the function definition, put the return type on its own line, and place the opening brace by itself on a line.
- For functions with enough arguments that they must break to the next line, align arguments with the first argument.
- When a function argument is a function itself, use the pointer form.

```
static int
function_name(int bar, const char *a, const char *b,
              const char *c, void (*d)())
{
```

- If a function name gets really long, start the arguments on their own line with 8 spaces of indentation:

```
static int
really_really_long_function_name_this_is_getting_silly_now(
        int bar, const char *a, const char *b,
        const char *c, const char *d)
```

```
{
```

Control Statements (if, else, while, for, switch)

- The keyword is followed by one space, then left parenthesis without space, condition, right parenthesis, space, opening bracket on the same line. `else` and `else if` are on the same line with the ending brace and opening brace, separated by a space.
- Always use braces around control statement blocks, even if they only contain one line. This makes code review diffs smaller if a line gets added in the future, and avoids any chance of bad indenting making a line incorrectly appear to be part of the block.
- Do not put assignments in `if` or `while` conditionals. This ensures that the developer's intent is always clear, making code reviews easier and reducing the chance of using assignment where comparison is intended.

```
a = f();
if (a < 0) {
    statement1;
} else if (some_other_condition) {
    statement2;
} else {
    statement3;
}
```

- In a `switch` statement, `case` is indented one level, and the body of each `case` is indented by another level. The opening brace is on the same line as `switch`.

```
switch (expression) {
    case 0:
        command1;
        break;
    case 1:
        command2;
        break;
    default:
        command3;
}
```

Operators

- Operators have spaces from both sides.
- Do not rely on operator precedence; use parentheses when mixing operators with different priority.
- No space is used after opening parenthesis and before closing parenthesis.

```
x = a + b - (c * d);
```

Best Practices

The guidelines in this section offer technical advantages.

New Struct and Enum Members

In the public APIs, always add new struct members to the end of the struct. This allows us to maintain backward API/ABI compatibility (as long as the application being linked allocates structs via API functions).

This generally applies to enum values as well, as the compiler will define enum values to 0, 1, etc., in the order given, so inserting a value in the middle will change the numerical values of all later values, making them backward-incompatible. However, if enum numerical values are explicitly specified rather than left to the compiler, new values can be added anywhere.

Documentation

All public API header files, functions, structs, enums, etc., should be documented with Doxygen comment blocks, as Pacemaker's online API documentation [http://clusterlabs.org/pacemaker/doxygen/] is automatically generated via Doxygen. It is helpful to document private symbols in the same way, with an `\internal` tag in the Doxygen comment.

Symbol Naming

- All file and function names should be unique across the entire project, to allow for individual tracing via `PCMK_trace_files` and `PCMK_trace_functions`, as well as making detail logs easier to follow.
- Any exposed symbols in libraries (non-`static` function names, type names, etc.) must begin with a prefix appropriate to the library, for example, `crm_`, `pe_`, `st_`, `lrm_`. This reduces the chance of naming collisions with software linked against the library.
- Time intervals are sometimes represented in Pacemaker code as user-defined text specifications (e.g. "10s"), other times as an integer number of seconds or milliseconds, and still other times as a string representation of an integer number. Variables for these should be named with an indication of which is being used (e.g. `interval_spec`, `interval_ms`, or `interval_ms_s` instead of `interval`).

Memory Allocation

Always use calloc() rather than malloc(). It has no additional cost on modern operating systems, and reduces the severity of uninitialized memory usage bugs.

Logging

- When format strings are used for derived data types whose implementation may vary across platforms (`pid_t`, `time_t`, etc.), the safest approach is to use `%lld` in the format string, and cast the value to `(long long)`.
- Be sure *not* to pass `NULL` as an argument to satisfy `%s` format specifier in logging and more generally `printf` style functions. When mere `<null>` string is a sufficient output representation in such case, there is `crm_str` convenient macro. Ternary operator is an apparent choice otherwise.

Regular Expressions

- Use `REG_NOSUB` with `regcomp()` whenever possible, for efficiency.

- Be sure to use `regfree()` appropriately.

vim Settings

Developers who use `vim` to edit source code can add the following settings to their `~/.vimrc` file to follow Pacemaker C coding guidelines:

```
" follow Pacemaker coding guidelines when editing C source code files
filetype plugin indent on
au FileType c   setlocal expandtab tabstop=4 softtabstop=4 shiftwidth=4 textwidth=
autocmd BufNewFile,BufRead *.h set filetype=c
let c_space_errors = 1
```

Chapter 4. Python Coding Guidelines

Python Boilerplate

If a Python file is meant to be executed (as opposed to imported), it should have a `.in` extension, and its first line should be:

```
#!@PYTHON@
```

which will be replaced with the appropriate python executable when Pacemaker is built. To make that happen, add an AC_CONFIG_FILES() line to configure.ac, and add the file name without .in to .gitignore (see existing examples).

After the above line if any, every Python file should start like this:

```
""" <BRIEF-DESCRIPTION>
"""

# Pacemaker targets compatibility with Python 2.7 and 3.2+
from __future__ import print_function, unicode_literals, absolute_import, division

__copyright__ = "Copyright <YYYY[-YYYY]> the Pacemaker project contributors"
__license__ = "<LICENSE> WITHOUT ANY WARRANTY"
```

`<BRIEF-DESCRIPTION>` is obviously a brief description of the file's purpose. The string may contain any other information typically used in a Python file docstring [https://www.python.org/dev/peps/pep-0257/].

The `import` statement is discussed further in the section called "Python Future Imports".

`<LICENSE>` should follow the policy set forth in the `COPYING` [https://github.com/ClusterLabs/pacemaker/blob/master/COPYING] file, generally one of "GNU General Public License version 2 or later (GPLv2+)" or "GNU Lesser General Public License version 2.1 or later (LGPLv2.1+)".

Python Compatibility

Pacemaker targets compatibility with Python 2.7, and Python 3.2 and later. These versions have added features to be more compatible with each other, allowing us to support both the 2 and 3 series with the same code. It is a good idea to test any changes with both Python 2 and 3.

Python Future Imports

The future imports used in the section called "Python Boilerplate" mean:

- All print statements must use parentheses, and printing without a newline is accomplished with the `end=' '` parameter rather than a trailing comma.

- All string literals will be treated as Unicode (the `u` prefix is unnecessary, and must not be used, because it is not available in Python 3.2).
- Local modules must be imported using `from . import` (rather than just `import`). To import one item from a local module, use `from .modulename import` (rather than `from modulename import`).
- Division using `/` will always return a floating-point result (use `//` if you want the integer floor instead).

Other Python Compatibility Requirements

- When specifying an exception variable, always use `as` instead of a comma (e.g. `except Exception as e` or `except (TypeError, IOError) as e`). Use `e.args` to access the error arguments (instead of iterating over or subscripting `e`).
- Use `in` (not `has_key()`) to determine if a dictionary has a particular key.
- Always use the I/O functions from the `io` module rather than the native I/O functions (e.g. `io.open()` rather than `open()`).
- When opening a file, always use the `t` (text) or `b` (binary) mode flag.
- When creating classes, always specify a parent class to ensure that it is a "new-style" class (e.g. `class Foo(object):` rather than `class Foo:`)
- Be aware of the bytes type added in Python 3. Many places where strings are used in Python 2 use bytes or bytearrays in Python 3 (for example, the pipes used with `subprocess.Popen()`). Code should handle both possibilities.
- Be aware that the `items()`, `keys()`, and `values()` methods of dictionaries return lists in Python 2 and views in Python 3. In many case, no special handling is required, but if the code needs to use list methods on the result, cast the result to list first.
- Do not raise or catch strings as exceptions (e.g. `raise "Bad thing"`).
- Do not use the `cmp` parameter of sorting functions (use `key` instead, if needed) or the `__cmp__()` method of classes (implement rich comparison methods such as `__lt__()` instead, if needed).
- Do not use the `buffer` type.
- Do not use features not available in all targeted Python versions. Common examples include:
  - The `html`, `ipaddress`, and `UserDict` modules
  - The `subprocess.run()` function
  - The `subprocess.DEVNULL` constant
  - `subprocess` module-specific exceptions

Python Usages to Avoid

Avoid the following if possible, otherwise research the compatibility issues involved (hacky workarounds are often available):

- long integers

- octal integer literals
- mixed binary and string data in one data file or variable
- metaclasses
- `locale.strcoll` and `locale.strxfrm`
- the `configparser` and `ConfigParser` modules
- importing compatibility modules such as `six` (so we don't have to add them to Pacemaker's dependencies)

Formatting Python Code

- Indentation must be 4 spaces, no tabs.
- Do not leave trailing whitespace.
- Lines should be no longer than 80 characters unless limiting line length significantly impacts readability. For Python, this limitation is flexible since breaking a line often impacts readability, but definitely keep it under 120 characters.
- Where not conflicting with this style guide, it is recommended (but not required) to follow PEP 8 [https://www.python.org/dev/peps/pep-0008/].
- It is recommended (but not required) to format Python code such that `pylint --disable=line-too-long,too-many-lines,too-many-instance-attributes,too-many-arguments,too-many-statements` produces minimal complaints (even better if you don't need to disable all those checks).

Chapter 5. Evolution of the project

Foreword

This section is currently not meant as a definite summary of how Pacemaker got into where it stands now, but rather to provide few valuable pointers an entusiasts (presumably software archeologist type of person) may find useful. Moreover, well-intentioned contributors to Pacemaker project may want to review them occasionally since, as the famous quote has it, "those who do not learn history are doomed to repeat it".

For anything more talkative with less emphasis on actual code, other places will serve better for the time being (and if not, should not be too hard to volunteer extensions to those writeups):

- main entry at ClusterLabs community wiki [https://wiki.clusterlabs.org/wiki/Pacemaker]
- entry at wikipedia.org [https://en.wikipedia.org/wiki/Pacemaker_(software)]
- brief section dedicated to Pacemaker in Digimer's tutorial regarding setting up the cluster with the old Red Hat Cluster Suite like stack [https://www.alteeve.com/w/AN!Cluster_Tutorial_2#What_about_Pacemaker.3F]

Common ancestor: *heartbeat* project

Pacemaker can be considered as a spin-off from 'heartbeat', original comprehensive HA suite started by Alan Robertson, and some portions of code are shared, at least on the conceptual level if not verbatim, till today, even if the effective percentage continually declines. Note that till Pacemaker 2.0, it also used to stand for one (and initially the only) of supported messaging back-ends (removal of this support made for one such notable drop of reused code), see also pre-2.0 commit 55ab749bf [https://github.com/ClusterLabs/pacemaker/commit/55ab749bf0f0143bd1cd050c1bbe302aecb3898e].

The codebase for heartbeat used to be hosted at http://hg.linux-ha.org, but since that does not appear reliably available recently, an archive checkout from 2016 is shared at as a dedicated read-only repository [https://gitlab.com/poki/archived-heartbeat], and anchored there, the most notable commits are:

- initial check-in of what turned up to be the basis for Pacemaker later on [https://gitlab.com/poki/archived-heartbeat/commit/bb48551be418291c46980511aa31c7c2df3a85e4]
- drop of now-detached Pacemaker code [https://gitlab.com/poki/archived-heartbeat/commit/74573ac6182785820d765ec76c5d70086381931a]

Regarding the Pacemaker's split from heartbeat, it evolved stepwise (as opposed to one-off cut), and the last step of full dependency is depicted in The Corosync Cluster Engine [https://www.kernel.org/doc/ols/2008/ols2008v1-pages-85-100.pdf#page=14] paper, fig. 10. This article also provides a good reference regarding wider historical context of the tangentially (and deeper in some cases) meeting components around that time.

Influence of *heartbeat* on Pacemaker

On a closer look, we can identify these things in common:

- extensive use of data types and functions of GLib [https://wiki.gnome.org/Projects/GLib]

- Cluster Testing System (CTS) is inherited from initial implementation by Alan Robertson
- …

Notable Restructuring Steps in the Codebase

File renames may not appear as notable … unless one runs into complicated `git blame` and `git log` scenarios, so some more massive ones may be stated as well.

- watchdog/*sbd* functionality spin-off:
  - start separating, eb7cce2a1 [https://github.com/ClusterLabs/pacemaker/commit/eb7cce2a172a026336f4ba6c441dedce42f41092]
  - finish separating, 5884db780 [https://github.com/ClusterLabs/pacemaker/commit/5884db78080941cdc4e77499bc76677676729484]
- daemons' rename for 2.0 (in chronological order)
  - start of moving daemon sources from their top-level directories under new `/daemons` hierarchy, 318a2e003 [https://github.com/ClusterLabs/pacemaker/commit/318a2e003d2369caf10a450fe7a7616eb7ffb264]
  - `attrd` → `pacemaker-attrd`, 01563cf26 [https://github.com/ClusterLabs/pacemaker/commit/01563cf2637040e9d725b777f0c42efa8ab075c7]
  - `lrmd` → `pacemaker-execd`, 36a00e237 [https://github.com/ClusterLabs/pacemaker/commit/36a00e2376fd50d52c2ccc49483e235a974b161c]
  - `pacemaker_remoted` → `pacemaker-remoted`, e4f4a0d64 [https://github.com/ClusterLabs/pacemaker/commit/e4f4a0d64c8b6bbc4961810f2a41383f52eaa116]
  - `crmd` → `pacemaker-controld`, db5536e40 [https://github.com/ClusterLabs/pacemaker/commit/db5536e40c77cdfdf1011b837f18e4ad9df45442]
  - `pengine` → `pacemaker-schedulerd`, e2fdc2bac [https://github.com/ClusterLabs/pacemaker/commit/e2fdc2baccc3ae07652aac622a83f317597608cd]
  - `stonithd` → `pacemaker-fenced`, 038c465e2 [https://github.com/ClusterLabs/pacemaker/commit/038c465e2380c5349fb30ea96c8a7eb6184452e0]
  - `cib daemon` → `pacemaker-based`, 50584c234 [https://github.com/ClusterLabs/pacemaker/commit/50584c234e48cd8b99d355ca9349b0dfb9503987]

Chapter 6. Advanced Hacking on the Project

Foreword

This section aims to be a gentle introduction (or perhaps, rather a summarization of advanced techniques we developed for backreferences) to how deal with the Pacemaker internals effectively. for instance, how to:

- debug with an ease
- verify various interesting interaction-based properties

or simply put, all that is in the interest of the core contributors on the project to know, master, and (preferably) also evolve — way beyond what is in the presumed repertoire of a generic contributor role, which is detailed in other sections of this guide.

Therefore, if you think you will not benefit from any such details in the scope of this section, feel free to skip it.

Debugging

In the GNU userland tradition, preferred way of debugging is based on *gdb* (directly or via specific frontends atop) that is widely available on platforms (semi)supported with Pacemaker itself.

To make some advanced debugging easier, we maintain a script defining some useful helpers in `extra/gdb/gdbhelpers` file, which you can make available in the debugging session easily when invoking it as `gdb -x <path-to-gdbhelpers> ...`.

From within the debugger, you can then invoke the new `pcmk` command that will guide you regarding other helper functions available, so we won't replicate that here.

Working with mocked daemons

Since the Pacemaker run-time consists of multiple co-operating daemons as detailed elsewhere, tracking down the interaction details amongst them can be rather cumbersome. Since rebuilding existing daemons in a more modular way as opposed to clusters of mutually dependent functions, we elected to grow separate bare-bones counterparts built evolutionary as skeletons just to get the basic (long-term stabilized) communication with typical daemon clients going, and to add new modules in their outer circles (plus minimalistic hook support at those cores) on a demand-driven basis.

The code for these is located at `maint/mocked`; for instance, `based-notifyfenced.c` module of `based.c` skeleton mocking `pacemaker-based` daemon was exactly to fulfill investigation helper role (the case at hand was also an impulse to kick off this very sort of maintenance support material, to begin with).

Non-trivial knowledge of Pacemaker internals and other skills are needed to use such devised helpers, but given the other way around, some sorts of investigation may be even heftier, it may be the least effort choice. And when that's the case, advanced contributors are expected to contribute their own extensions

they used to validate the reproducibility/actual correctness of the fix along the actual code modifications. This way, the rest of the development teams is not required to deal with elaborate preconditions, be at guess, or even forced to use a blind faith regarding the causes, consequences and validity regarding the raised issues/fixes, for the greater benefit of all.

Appendix A. Revision History

Revision History		
Revision 1-0	Tue Jul 26 2016	AndrewBeekhof<andrew@beekhof.net>, KenGaillot<kgaillot@redhat.com>
Convert coding guidelines and developer FAQ to Publican document		
Revision 1-1	Mon Aug 29 2016	KenGaillot<kgaillot@redhat.com>
Add Python coding guidelines, and more about licensing		
Revision 2-0	Fri Jan 12 2018	KenGaillot<kgaillot@redhat.com>
Drop support for Python 2.6		
Revision 2-1	Tue Sep 18 2018	JanPokorný<poki@redhat.com>
Start documenting notable evolutionary points		
Revision 2-2	Fri Dec 7 2018	KenGaillot<kgaillot@redhat.com>
Update FAQ and C guidelines		
Revision 2-3	Mon May 13 2019	KenGaillot<kgaillot@redhat.com>, JanPokorný<poki@redhat.com>
Update copyright notice policy, and some external references		
Revision 2-4	Fri 17 May 2019	JanPokorný<poki@redhat.com>
Start capturing hacking howto for advanced contributors		

Index