OPeNDAP Software Release Process

James Gallagher* Dan Holloway† Ethan Davis‡

2004/01/09, Revision: 1.18

Contents

1 Introduction

Deciding when to release software is hard. The choices depend on the long-term goals of the group doing the work and which features they think should be bundled together. However, this paper is not about choosing which features go in which release. This paper describes a process that starts once the features have been determined and are implemented in the software. Once we decide to release, we will follow the process described here.

*jgallagher@opendap.org
†dholloway@opendap.org
‡edavis@ucar.edu

2 Release Checklists

This section contains some checklists that we have developed over time to help make the process of releasing software more predictable. Collecting these lists in one place should help coördinate our group and also make it easier to fix broken parts of, or add missing parts to, the process.

All the OPeNDAP software is held in one CVS module. Even so, each discrete piece of software (a server, the library of common code, *et c.* is in its own directory. Each source directory has its own version number. With the exception of the server dispatch software (which is in `DODS/etc`) all of the software is held in directories under `DODS/src` and each of those directories contains a separate deliverable.

In practice, we release the OPeNDAP software mostly *en masse*. However, if a fix for a particular component really needs to be released quickly, we can increment just that component's version number and release it alone. That is, suppose the current release is version 2.1 and we add an important fix to `asciival` which is at version 2.1.6. We can bump the version number of `asciival` to 2.1.7 and put that in the 2.1 release. We don't need to release all the other components.

When we decide to there are enough new features and/or fixes for a new release, the version number of the release as a whole is increased. Appendix A has information on version numbering.

2.1 Making a new release

Making a new software release can be roughly divided into two parts. In the first part we start making Release Candidates (essentially beta releases). As problems are found in these, we fix those problems and produce a successive candidate. This iterative process continues until the current Release Candidate has no major problems and can be made a formal release.

Here are the steps used to iterate over making release candidates toward making a formal release:

1. Create a release branch in CVS
2. Start documentation efforts:
   (a) Gather information needed for documentation (web pages, README/INSTALL docs, and formal manuals):
      - the new features and major bug fixes (this information should be part of the ChangeLog files).
      - the system requirements.
      - compatibility with previous versions.
      - the list of the systems on which the code has been built (include compiler version information).

      This list will be the rough draft of the release notes.
   (b) Create web pages for this release.
   (c) Create release notes. (Take the list made above, and check it into the documentation DODS CVS tree at DODS-doc/notes.) The release notes will eventually consist of a list of the important

changes, followed by a list of explanations for each change. Here's an example from release 3.4:

```
Code changes
1.  Using libcurl instead of libwww.
2.  URL dereferencing done with curl instead of Perl.
...

CODE CHANGES
1. The DODS core now links to libcurl instead of libwww for WWW
functions.  The libcurl library is smaller than libwww, is being
actively developed, and is MT-safe.

2. The server dispatch scripts no longer use the Perl HTTP, MIME and
LWP classes.  Instead, URL dereferencing is done with curl, a
freestanding binary that is part of the libcurl build.
...
```

(d) Start the process of adding that information to the Guides.

It's important to get started on these tasks early because they are complicated and, unlike a build or a test suite, there's no automated way to determine when they are 'ready.'

3. Iterate through the Release Candidates until no major bugs remain. See Section 2.1.1 for the steps involved in making individual release candidates.
4. Evaluate the software (See Section 2.1.2) to determine if there are issues with the source not yet addressed by the process of release candidate preparation.
5. Mark the branch as a formal release (e.g., `cvs tag release-3-4-3FCS`). NB: I prefer to do this *after* making the source files. If you use this ordering of the steps and find a problem, make sure to re-tag the branch (see `cvs tag -h` for information about moving (or forcing) a tag.
6. Make source and binary distribution files. See Section 2.1.3
7. Place source and binary on `ftp` site
8. Update web pages for formal release

2.1.1 Making release candidates

1. Prepare the code to be a Release Candidate (see Section 3.1).
2. Run nightly builds until the tests are clean.
3. Update all `version.h` files.
4. Tag CVS branch as a release candidate (e.g., `cvs tag release-3-4RC1`).
5. Make source tar files (see Section 2.1.3)
6. Put source distribution files on the `ftp` site
7. Check that web site is up-to-date (i.e., so that people can get the release candidate).
8. Announce release candidate source code.
9. Tag and merge the branch (see Section 3.2).

10.

Optional

Make binary builds; release.

11. If there are major bugs, fix them and redo this list.
12. If there are no major bugs, we are done making release candidates

Note that the Release Candidate process concerns the source code and binary builds only. Once we've decided to make a formal release, There are other things that must be done *before* it is ready to become a formal release. See Section 2.1.2 for details on the steps required before a formal release can be made.

2.1.2 Before making a formal release

Think about each of the following questions before moving on from release candidates to a formal release:

- Do we have binaries for all the platforms we support?
- Double check information needed for documentation:
  - new features and major bug fixes.
  - system requirements.
  - compatibility with previous versions.
  - the list of the systems on which the code has been built.
- Is the documentation sufficient? Check README and INSTALL files in the `src`, `etc` and `docs` directories and the NEWS file in the top-level directory. The text documentation files in `docs` are used when making binary distributions, the ones in the `src` directories are included with the source distributions and the ones in `etc` are included in both.
- Do the formal manuals need to be updated? If so, this process should be started (however, in general the release should not wait on the manuals).
- Is the web-site current?
- Can we all answer questions about the installation and use of the software? Do we all understand the software's limitations?

2.1.3 Making source and binary distribution files for a formal release

1. Check out a clean copy of the release branch using CVS's `export` command (e.g., `cvs export -r release-3-4RC3 -d DODS-3.4 DODS`).
2. Remove any source/directories that we don't intend to include in this release. Since the trunk often contains more software than we are actually releasing, this is an important step.
3. Verify that this software builds: `./configure`, `make World`, `make tar`
   NB: It's often a good idea to see the build as it happens and to get a log. To do that, use `tee` as in: `make World 2>&1 | tee World.log`. If you're using `csh` change `2>&1 |` to `|&`.

- Build binaries and run tests.
- Build binary distribution files (`make binary-tar`)
- Remove the binaries: `make distclean`
- Add the version numbers to the directory names: `make update-version`
- Make the source code tar files: `make source-tar` (at the top-level) or `make tar` in a `DODS/src/`*component* directory.
- Test both the source and binary distributions!
  (a) Expand all the binary distribution files to make sure they hold software that works.
  (b) Expand the source distribution files and verify they build.

3 CVS Checklists

Using CVS can be complicated. This section contains some short checklists for common CVS operations.

3.1 Prepare the code to be a Release Candidate

NOTE: For the DAP software, some of these steps must be done in the `DODS_ROOT` and `DODS_ROOT/etc` directories in addition to `DODS_ROOT/src/dap`. For the other software, follow these in just the `DODS/src/`*component* directory.

1. Check that the `configure` script is up-to-date WRT `etc/aclocal.m4`. Use `make configure`.
2. Rebuild the dependencies using `make depend`.
3. Examine the directory for files that should have been removed or are leftover from development/testing and remove those by hand. In some cases files will need to be removed from CVS also (`cvs rm -f` *files*).
4. Run the tests. Do this starting from a clean set of sources (`make clean; make check`).
5. Update release number in `version.h`. Use the three part version numbers. Check this file in and add today's date as the log entry. This makes figuring out when a particular version was made simpler (although there are other ways). See Appendix A for information about version numbers.
6. Update the `ChangeLog`. Scan changed log files using `cvsdate ">`*date of last version*"."[1]
7. Make sure that all files in the directory are checked into CVS. Use `cvs ci`. Use the `-l` flag with `cvs ci` in `DODS_ROOT` to check in only that directory and avoid recurring through all the software.
8. Examine the INSTALL and README files. Are they correct? Anything new missing?
9. Look at the README/INSTALL files in DODS/etc *and* DODS/docs. Are they correct? The first set of README/INSTALL files are sent with both source and binaries, the second set are bundled into the binary distributions only.

[1] `cvsdate` is short command that runs `cvs log` and parses the output. A copy should be in `DODS/etc`.

3.2 Tag and merge the branch

1. Go to the CVS directory that holds the release code and tag it with the same release number as is in the version.h file.
2. Merge back into the trunk. To merge the changes made in the branch software back to the trunk, do the following:
   (a) Go to the CVS directory that holds the trunk software.
   (b) Use `cvs update -j` *`release-branch-tag`* for the first merge
   (c) Use `cvs update -j` *`previous-merge-tag`* `-j` *`release-branch-tag`* for subsequent merges[2]

   See Appendix A for an explanation of the relation between version numbers and CVS tags.
3. In the `ChangeLog` file in the trunk software, record the cvs command used to do the merge and the arguments to `-j` that should be used for the next merge.
4. To test for conflicts use:

   ```
   alias conflict='grep -l "<<<<<<" `find ! -type d ! -name *.cft`'
   ```

   and run the results through the `resolve_conflicts.pl` script. Some files get munged by this so examine the result.[3]
5. After merging each directory, add a note in ChangeLog and check in the code using `Merged with release-<rev>` as the log entry.
6. Save and check in the changes that result from the merge.
7. Try to compile the newly merged code. Resolve problems introduced by the merge.

A Version Numbers

Version/revision numbers in DODS are formed from two or three numbers separated by a dot. Here's how to interpret the version numbers:

x.y.z

where

- x: This is the major version number. Two different major versions are almost certain to be incompatible. This is used to indicate a big change in the way the software works.

[2] There's an argument you can pass to `cvs update` that will suppress expansion of `$Id$` and `$Log$` symbols. Using this will cut down on spurious conflicts (but you don't get those symbols expanded). You can resolve the conflicts by hand or use the `resolve_conflicts.pl` script.

[3] `resolve_conflicts.pl` keeps a copy of the original file in *`filename`*`.cft`.

- y: This is the minor version number. This indicates an important change in the software. Two different minor versions maybe incompatible in some ways but mostly work together.

  Each minor version has a corresponding tag in CVS *that is a branch*. That is, you can checkout/update using `-r release-x-y` and get the latest code for that `major.minor` release number. Fixes for the release should get checked in on the appropriate branch.

- z: This is the sub-minor version number. These are used to indicate that problems with a particular version of the software have been found and fixed. Different sub-minor versions should work together except that a problem fixed might cause slightly different behavior.

B Version Numbers and CVS Tags

Each sub-minor version number corresponds to a real release of software. Each has a tag that looks like `release-x-y-z`. However, this tag is *not* a branch. This might seem confusing at first, but it makes working with CVS simpler. Here's an example. Lets say we have made a release of version 3.1.0 and some problems have been found. We can commit changes to that release until we are happy with the fixes. At that time we then tag *that point* on the 3-1 branch as 3-1-1 and make the source code tar files for version 3.1.1. See Figure 1.

Figure 1: CVS branching

Why make the non-branch tags? Because what we really need are place holders for merging the changes back onto the trunk. See the Using CVS paper for a discussion of CVS merging strategy. While we work on the 3.1 code, fixing bugs and tweaking features, other completely new features are added to the trunk code.

C Targets in the Makefiles

Listed here are targets in the Makefiles that are used to help make releases.

`Makefile` If the `Makefile.in` has changed, rebuild the `Makefile` either by running `config.status` or `configure`. In source code that has just been exported from CVS, you will need to run `configure` to build the `Makefile`.

`depend` (Re)build the dependencies. This target rebuilds the dependencies in the `Makefile.in`. You should not have to use this target with code that's been exported from CVS; if you do, then it is a sign that something is wrong with the `Makefile.in` that is in CVS. You'll need to go back to code that is in CVS, run make depend and then check in the resulting `Makefile.in`.

`check-version` Check that the version number in the `Makefile` matches the version number in the version.h file. If this returns an error, you need to go back to code in CVS, re-run `configure` and then export the code.

`update-version` Read the current version from the `version.h` file and append it to the CWD's name (i.e., `jg-dods` becomes `jg-dods-3.1.2`). Don't run this in the directories under CVS control (those that have the 'CVS' subdirs in them). Only run this in an exported directory.

`all` Build all the main programs in the directory.

`check` Run the tests for this this software. In some cases, you have to install the software (using the install target) before this does anything except return an obscure error. Bummer. This should have been run by the nightly build, so running it when making the source or binary tars is not completely necessary...

`install` Copy the binaries from this directory into the `bin`, `lib`, `etc` and `include` directories.

`binary-tar` Build a tarball that contains the binaries built by 'all' and installed by 'install'. The binaries are tarred in the directories where they are installed, not where they are built.

`clean` Remove most generated files (i.e., executables, object files, core files, emacs backup files, ...).

`distclean` Remove all generated files except grammar files, the `Makefile` and `configure`.

`tar` Make a source code tarball. Run distclean before using this target to make the source tar balls. Note that in the top-level `Makefile` this is called `source-tar`.

If all goes well, the targets should be run in the following order after running `configure`: `check-version` (should return OK), `update-version` (should set the directory name so that the source tar balls have the correct names), `all` (builds the code), `install` (installs it), `binary-tar` (creates the binary distributions for this platform), `distclean` (gets rid of the same before making the source tar balls), `tar` (makes the source code tar balls).

D ChangeLog

```
$Log: release-process.tex,v $
Revision 1.18  2004/01/09 20:32:51  jimg
Fixed a latex-typo.

Revision 1.17  2004/01/09 20:16:55  jimg
Editing...

Revision 1.16  2004/01/08 18:34:00  jimg
Added info about NEWS files.

Revision 1.15  2004/01/08 00:37:07  jimg
Minor formatting changes.

Revision 1.14  2004/01/06 23:01:47  jimg
Added an item about testing the distribution files.

Revision 1.13  2004/01/06 22:03:49  jimg
I added some additional information to the check lists and tried to tie them
together a bit more.

Revision 1.12  2003/12/08 16:41:02  tom
added information about release notes, fixed some typos.

Revision 1.11  2003/12/04 22:42:08  jimg
Fixed an error in the merge recipe!

Revision 1.10  2003/09/24 22:28:18  edavis
Added some details to the "Making release candidates" and the "Making
source tar files" sections.

Revision 1.9  2003/06/30 23:07:19  edavis
Added examples of commands for several steps. Added some detail
to the documentation requirements.

Revision 1.8  2003/06/26 15:26:41  tom
fixed \figureplace usage (added \figpath)

Revision 1.7  2003/06/24 23:25:57  jimg
Added figure; changed to \figureplace. Latex works but the figure is big,
hyperlatex almost works. The web pages have a snazzy new look about them...

Revision 1.6  2003/06/24 22:39:09  tom
put it into dods-paper class. other little adjustments

Revision 1.5  2003/06/24 21:56:05  jimg
I added some hypelatex stuff, but I'm really pretty clueless about it all.

Revision 1.4  2003/06/20 18:02:35  jimg
Changes. I'm done with this for a while even though it could use more work...

Revision 1.3  2003/06/19 01:47:46  jimg
Added hyperlatex (but not tested) and a short abstract.

Revision 1.2  2003/06/19 01:40:43  jimg
First pass at formatting.
```