DIRAC Release procedure using Branches and Tags

Author: R. Graciani

Version: 3.0, 2011-03-04

Based on: http://nvie.com/posts/a-successful-git-branching-model/

Precedents

DIRAC project is using SVN as code repository. We need to produce releases for an increasing number of user communities. Among other things, this means that several "active" releases have to be maintained (bug fixes applied) to accommodate the different life cycles of the releases in the different communities.

Releases are named with the following convention DIRAC_vXrY. X is incremented when there are changes on the compiled code, mayor changes on the interfaces or important changes in the functionality. Y is incremented when new features are added or there are minor changes in the interfaces (i.e., they are backwards compatible). An extra pZ is added to the release name when bug fixes (patches) are introduced. In almost all cases bug fixes are relevant for all "active" releases.

The current release procedure occurs in several steps. First, each system creates their own "tags". Then, the release manager collects the relevant tag from each system and creates a global DIRAC tag (DIRAC_vXpY-pre1). From this tag a release candidate is produced, deployed, tested. Updates and patches are committed to the trunk and new system "tags" are created as necessary. The procedure is iterated increasing the cardinal of the "pre" suffix until a convergence is found. At this point, the last list of tags is collected and the final release DIRAC_vXrY is created.

After this point, only bug fixes are allowed. The procedure to do it is highly **non-standard**. Fixes are committed to the HEAD. For the affected system, a new system "tag" is created copying the "tag" used in the last release for that version. The affected file is updated (removed and copied from the HEAD, edited on the tag,...). And a DIRAC_vXrYpZ release is defined from the new set of systems "tags".

This bug fix procedure has several problems:

- It requires changing tagged files.
- Applying the same fix to other active releases is not foreseen.
- When the trunk has evolved from the release being patched, fixes may be difficult to apply.

There are other alternatives, like checking out the affected tag, making the fix and committing to a new tag, even more non-standard.

To overcome these problems, we propose to actively introduce the use of Branches in the procedure as well as a more formal bug found and bug fix report procedure. The use of branches implies more discipline on the developers but the aim is to simplify the work to maintained "patched" "active" releases. The proposal is described in the following.

Different Branches and their usage

HEAD or "trunk" branch: it is the master branch of the code, has unlimited life. We should put all the effort to keep the trunk in a "working" state. Making use of Development branches when there are changes that imply a testing phase. This is an important change with respect to the current mode of work where development was being done directly on the trunk.

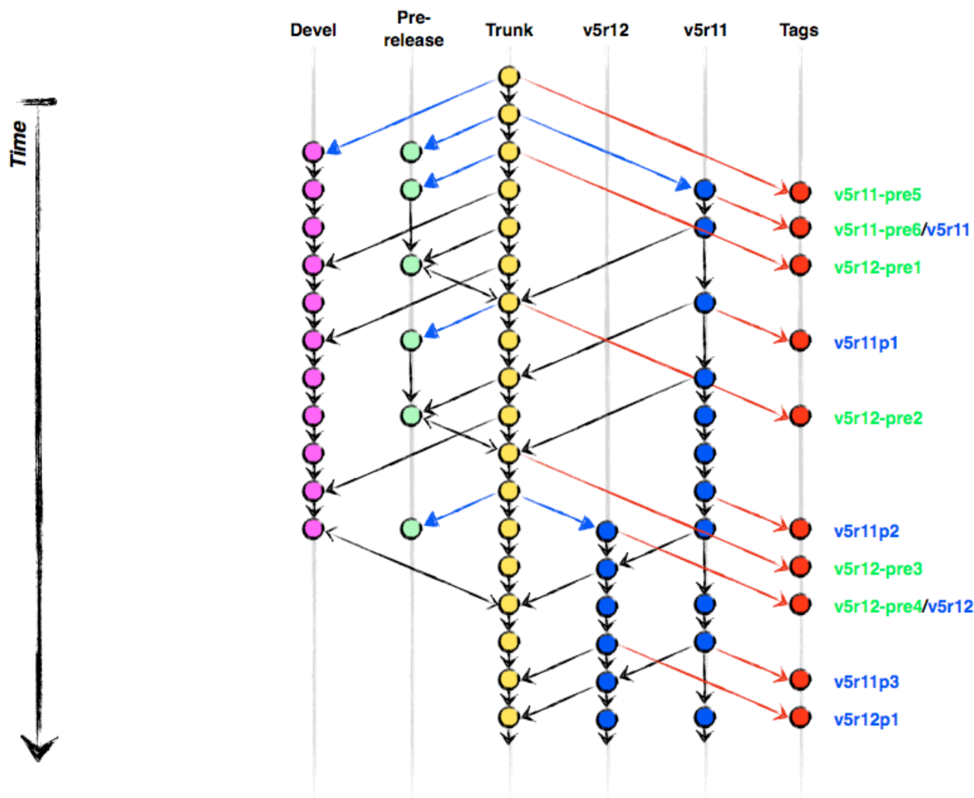
Development or Feature branches: they are created when the new code is likely to require a

testing phase, specially when changing functionality or interfaces for existing modules. **They must be created in a "devel" subdirectory** and named after the developer working on it or the task is is trying to achieve; i.e. branches/devel/ricardo or branches/devel/newstager. **When the development becomes stable it must be "reintegrated" into the trunk**. During the development process the developers working on the branch are responsible from merging updates from the trunk. This should minimize the risk of having a non working trunk. If the development gets to a dead end, it has to be deleted. In either case it has a limited life cycle.

Pre-Release branches: **they are created from a well define set of tags defined from the trunk**. And **they are placed in a "pre" directory and are named vXrY-preZ**. In all other aspects, they are identical to Development branches. Once the testing of the pre-release is done and all fixes are committed **it must be "reintegrated" to the trunk before preparing the next pre-release or release candidate**. All tags defining the branch must have been created from the trunk.

Release branches: they are created when the last release candidate of a series is successfully tested and requires no additional changes. **The same tags are used to create a new branch name vXrY, located under a "rel" directory**. Bug fixes are possible in these branches until they are declared obsolete.

The life cycle of the different branches are depicted in the following figure.



Some important considerations about SVN branches and merges

From the point of view of SVN all directories (trunk, tags, branches, or any other) are identical. The repository only has the concept of directories (or files) that are copied from others and thus they can be mutually merged provided they do not have conflicting changes. SVN also knows how to reintegrate a copied directory into its source. One needs to merge all updates from the source, SVN calculates the differences and applies them to the source and, once the modified source version is committed, SVN forgets the common history.

This has some important implications. In order to merge directories (or files), one of them must be a copy of the other. Branches, to be reintegrated to the trunk or from where bug fixes are merged back to the trunk, can not be created (copied from) from tags, rather they must be created by copying from the trunk, using the same "revision" of the trunk from which the tag was produced.

Branches can thus be used in two different ways:

- **Development mode**: created from the trunk, developed, eventually merging fixes from the trunk, and finally being reintegrated to the trunk or abandoned. This is the distributed equivalent to doing a cvs checkout, working on your local copy, updating from time to time from the repository and finally committing all your local changes to the repository. In the above, **this mode is used for development and release candidate branches**.
- **Release mode**: created from the trunk, patch fixed in place or from an older branch, and the fixes merged back to the trunk or a newer branch. Merges can only go in this direction, from an older branch into a newer branch, to avoid incorporating new functionality to older releases. **This is the mode for release branches**.

From a practical point it view it means that:

- Branches are created from a copy of a given revision of the trunk.
- Fixes are propagated from older release branches to newer ones until the trunk is reached, from there each development branch is free to take them on board at any time.
- New features are committed to the trunk either directly (small changes or fixes) or by reintegrating a development branch (bigger changes).
- During the preparation of a new release commits to the trunk, either directly or from a development branch, should be limited to avoid interferences.
- Between releases new development can be committed to the trunk once they have been properly tested on his own branch to avoid breaking the functionality.

For convenience, branches are grouped in 3 directories "devel", "pre" and "rel" in the SVN repository.

The release Procedure

Following the same structure of the branches we propose to have 4 types of releases for which distribution tars are to be created:

- Development releases: they come directly from the head of a given "devel" branch and are used to test them beyond local tests (installation on servers, pilots, clients,..). They can be recreated as often as necessary. This includes releases created from the head of the trunk.
- Release candidates: they come from the tag associated to a given "pre" branch. They are use to test the candidates and eventually certify final release candidates
- Production releases: they come from the tag associated to the last release candidate of the series.
- Patch releases: in this case the tag comes from the corresponding "rel" branch.

So, basically there are 2 use cases. Creating a distribution from the head of a directory (devel bruch or trunk) or from a previously defined tag (also a directory in the repository). The dirac tools will be updated to work in this way.

The procedure to prepare a new release it then as follows:

- **Development is frozen**: the release manager announces the start of the procedure and ask

developers to reintegrate branches with developments to be included in the new release.

- **Preparation of release candidates**: the trunk is tagged, the tag is used to create a "pre" branch, distribution tars are created. The branch is used to commit fixes or do some development, if necessary, before it is reintegrated to the trunk to prepare the next release candidate.
- **Creation of the release**: when the last release candidate certification procedure is passed with satisfaction, the same tag is used to create the "rel" branch. Distribution tars are created.
- **Patch releases**: the release manager decides when accumulated bug fixes on the "rel" branch deserve a new distribution, checks that all bug fixes committed to older versions are properly merged. A new tag is created from the branch and distribution tars are created.
- **End of support**: at some point the branch is frozen an no new patch releases will be created. All fixes applied to the branch must be merged to the next "active" release if not already done. At this point the branch is removed or moved into an "old" directory.

The bug fix Procedure

This is one of the most important change in the procedure, that implies extra discipline from the developers to assure fixes are properly propagated to all "active" releases that is the main purpose of this proposal.

The bug fix procedure should be as follows:

- The problem is detected. It must be reported for future reference.
- A developer finds the reason and looks for solution. The bug report should be updated as necessary.
- The fix must be applied on the oldest rel branch for which is relevant (not necessarily corresponding to the same release where the problem was detected).
- From this branch the fix is merged into newer rel branches and the trunk.

This assures that at any point in time we can create new patch releases from any "active" version that includes fixes for all the "known" bugs.

There are 2 key points in this procedure:

- **Proper report of bugs and their fix**. It is the responsibility of the developer working on the issue to make sure a proper bug report is created and filled.
- **Correct application to all affected branches**. Depending on the experience of the developer, he will commit the fix to the proper branch and propagate the bug fix, or simply inform the release manager, by properly updating the bug report.

For development branches, it is the responsibility of the developer of the branch to incorporate bug fixes from the trunk if appropriated or else they will be taken once the branch is merged back to the trunk. Even if the release manager has the duty to check that fixes in older branches are merged into newer branches at the time of preparing a new release candidate or patched release, this can only be done if bug reports are properly fixed and all bug fix commits are commented as such.

Given the number of current developers and the available manpower we must keep a manageable number of "active" versions; i.e., a small number of release branches should be kept. The minimum is one production release, a second one while the deployment of the new version occurs by the different communities, and the trunk.

SVN branching on eclipse

This is simply an interface to “svn copy”. From a certain point of the working directory: Team → Branch. Browse to find the destination in the branches tree. The copy is done using the local revision of eclipse project, but not the local changes. In general, one should update from the repository and commit changes before creating new branches.

This procedure can be used to create development branches from the trunk.

SVN merge on eclipse

This is an interface to “svn merge”. From the place where we want to merge into (at the same level as the branch was created): Team → Merge. In the “URL” tab, browse the branches tree to locate the appropriated source. The “Preview” button allows to see what are the differences. The changes are made in the local working copy, and a synchronization perspective is opened. If any conflicts arise, they must be fixed and marked as solved. In order to be able to commit the merged version of the files the project must be previously updated from the repository (as for any other commit).

This procedure is used to propagate bug fixes from “rel” branches in to the trunk and from the trunk into “devel” branches.

SVN reintegrate on eclipse

This is an interface to “svn merge –reintegrate”. From the place we want to reintegrate to (generally the trunk in the procedure describe in this document). Previous to the integrate is is convenient to merge into the branch all fixes committed to the trunk (see previous section). Then one should move to the trunk and update from repository. In Team → Merge, go to the “Reintegrate” tab and browse the branches tree to locate the appropriated source. The “Preview” button allows to see what are the differences. The changes are made in your local working copy, and a synchronization perspective is opened. If any conflicts arise, they must be fixed and marked as solved. In order to be able to commit the merged version of the files the project must be previously updated from the repository (as for any other commit).

This procedure is used to close a “devel” or a “pre” branch incorporating the changes to the trunk.