Keeping tagged versions of code and parameters by GIT

Håkan T. Johansson
f96hajo@chalmers.se

Ronja Thies
ronja.thies@chalmers.se

5th March 2014

Contents

1 Introduction

GIT[1] is a *distributed* version control system (VCS). Distributed means that there is *no* (central) repository, i.e. none which by design is more important, general or valid than any other.

[1] http://git-scm.com/

1.1 Meritocracy — building trust

In a distributed model, a preferred repository, i.e. one which is used as upstream by users, can not be enforced by technical means. Repositories become preferred sources by "natural selection", in this case by building trust.

Simple example: A repository $\mathbb{A}$ maintained by user $\mathcal{A}$ only become used by others if they feel that the code or parameters in that repository is good and useful to them. They then base their work on $\mathbb{A}$. This does not mean that $\mathcal{A}$ must produce all (or even any) code or parameters, just that $\mathcal{A}$ is showing good judgement in which commits $\mathcal{A}$ pull from other users and place in $\mathbb{A}$. It may happen that eventually $\mathcal{A}$ spends less effort on maintaining the tree $\mathbb{A}$. Then e.g. the tree $\mathbb{D}$ (which inherits from $\mathbb{A}$) of user $\mathcal{D}$ could become the de-facto standard synchronisation source.

1.2 Plain vanilla

We believe that a plain GIT workflow is advantageous in the setting of Nuclear Physics experiment computing. We also think that the GIT tools (including other third-party helper utilities) are best used as they are, and not wrapped in any custom fashion by the collaboration[2][3].

1.3 GIT in three paragraphs

Each version of a file in a git repository is stored as a "blob", which is the content and an unique identifer of that content (an `SHA1` hash). A directory of files is a list of names and file contents, the latter described by their unique identifiers. An entire directory tree is stored in the same fashion, recursively, i.e. with each subdirectory as an entry in the parent directory. A commit is a particular version of a directory tree, by reference. Each commit also has references to the predecessor commit(s)[4] and a commit message. Many commits together make the development graph. Each of these items above is a GIT object, identified by its `SHA1` hash, see Figure 1.

GIT is a collection of tools to manipulate such repositories. A command has the general form `git command args...`. In addition to the examples in this document, there are many resources on the web, and likely manual pages in your UNIX computer: `man git`.

Each GIT working directory has the full commit history in a subdirectory `.git`. With the GIT tools, one commit changes from the working tree to the repository. And move the working directory "around" in the project

[2]Simplifying software installation as GIT+utilities come with any sane OS distribution.
[3]Not precluding contributions to the GIT or utility projects!
[4]Plural for merges.

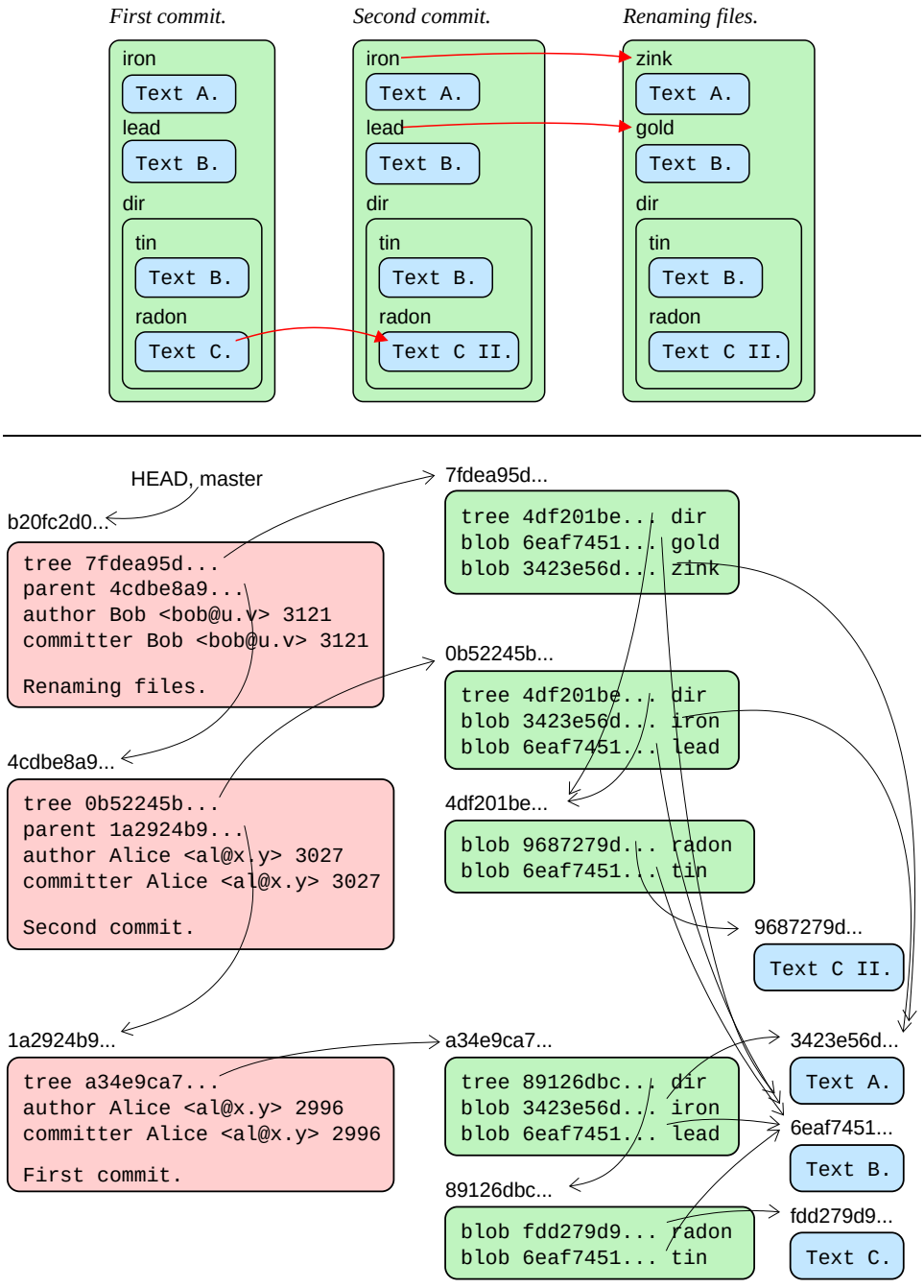

Figure 1: The upper part shows three versions of a small project with four files, two in a subdirectory. The second commit changes the contents of a file, and the third renames two files. The lower part shows the objects of a GIT repository representing the above revision history, evolving from the bottom up in the picture. Everything is an object: content + SHA1 hash. (The hashes have been abbreviated for clarity.)

history by checking different commits out. Commit chains or entire repositories can be fetched or cloned from other GIT directories (repositories). A GIT repository can also be set up as "bare", in which case it has no working set of files. They are suitable as points for code distribution.

2 Workflow

2.1 Tutorial by example

The workflow is described as working examples below. (The `test-proj` sandbox does exist — please try by executing the commands!)

Some preliminaries:

0. Getting the name right in commits is nice (change to yours):

   ```
   git config --global user.name "Alice"
   ```

   ```
   git config --global user.email "alice@xyz.org"
   ```

   Colourised output!

   ```
   git config --global color.ui true
   ```

   But do not mark whitespace at end of lines like severe errors:

   ```
   git config --global color.diff.whitespace "blue ul"
   ```

2.1.1 Hacking away

1. Check out a project directory from any (normal) upstream repository:

   ```
   git clone land@lx-pool.gsi.de:/u/landcvs/test-proj.git
   ```

   This creates a working directory `test-proj`, with the current `master` branch of the source repository. The entire history of the project is placed in the subdirectory `test-proj/.git`.

   ```
   cd test-proj
   ```

2. View the history of the project[5]:

   ```
   git log --all --oneline --decorate --color
   ```

3. Then, hack it a bit (e.g. modify a comment):

   ```
   emacs hello.sh
   ```

4. One should commit early and often. But first (compile and) test the new version:

   ```
   ./hello.sh
   ```

[5] It is suggested to run this command in a separate terminal window, to easily see the output (not having to remember it) when doing more complex GIT manipulations.

5. One should commit early and often. But before etching the changes in history, inspect them[6]!

   `git diff`

   Depending on the kind of changes, the option `--word-diff` may be a useful companion as it reduces the change-marks from lines to the changed words. [7]

6. Commits are created from the staging area (also known as the "index"). Add a modified file to the index:

   `git add hello.sh`

   After being staged, the modifications are no longer seen by `git diff`. The staged differences can be viewed by `git diff --cached`.

7. Perform the commit!

   `git commit`

   An editor[8] will open for entering a commit message. Shortly describe what has been changed.

2.1.2 Bare repository

Bare repositories (without a working set of files) are useful both as distribution points and for synchronisation when working at different systems. They by convention have `.git` as extension on the directory name.

- Create a bare repository.

  ```
  cd ..
  mkdir test-proj.git
  cd test-proj.git
  git init --bare
  ```

  It can be populated either by pulling from somewhere, or by pushing. By pushing branch `master` from repository `test-proj` (`bare` is the local name (handle) for the bare repository):

  ```
  cd ../test-proj
  git remote add bare ../test-proj.git
  git push bare master
  ```

[6] Also this benefits from running in a separate terminal, while performing the next step.

[7] If you see entire files mysteriously changed, did something change the formatting? Newline changes on Windows?

[8] In case disaster strikes and an unusable editor (like VI) appears, prepend the command by e.g. `GIT_EDITOR=nano`. Emergency instructions: type `:q` to evacuate VI.

- The circle is now complete by you or someone else cloning from your new (public?) repository:

  `git clone` *`url-path-to-.../test-proj.git`*

- One can also fetch a branch into an existing repository.

  `git remote add friend1` *`url-path-to-.../test-proj.git`*

  `git fetch friend1 master:test1`

  `git checkout test1`

  `friend1` is the local handle for the remote repository, `master` the name of the branch to be fetched and `test1` the name the branch will have locally. The `checkout` changes the working directory to be at branch `test1` for testing.

- Once satisfied that `test1` is a good development, it can be merged with your active branch, e.g. `master`:

  `git checkout master`

  `git merge test1`

It is suggested to keep bare repositories such that only one user can write to each. That way it is clear who has introduced (pulled) the contained commits.

3 Archive

There is one caveat: archiving (of abandoned branches).

3.1 Central memory

For a collaboration, it is necessary to be able to reproduce results and plots long after production. This includes intermediate results, that may have come from code or parameter sets that were subsequently found bad and have been abandoned. This can potentially become quite a head-ache when a distributed VCS is used, as dead-end trees are abandonded even by their own creators. Some sort of central memory is needed.

"You Will be Assimilated. Resistance is Futile."
The Borg, Star Trek.

This document describes a way to keep an *archive* of such (possibly) historical code and parameter sets. Handled, of course, in a GIT repository — see Figure 2. Note that such an archive repository does not imply endorsement of any contained code or parameters. It exists just as an convenience to keep the sources and parameters used to produce plots and results that have been shown more or less officially.

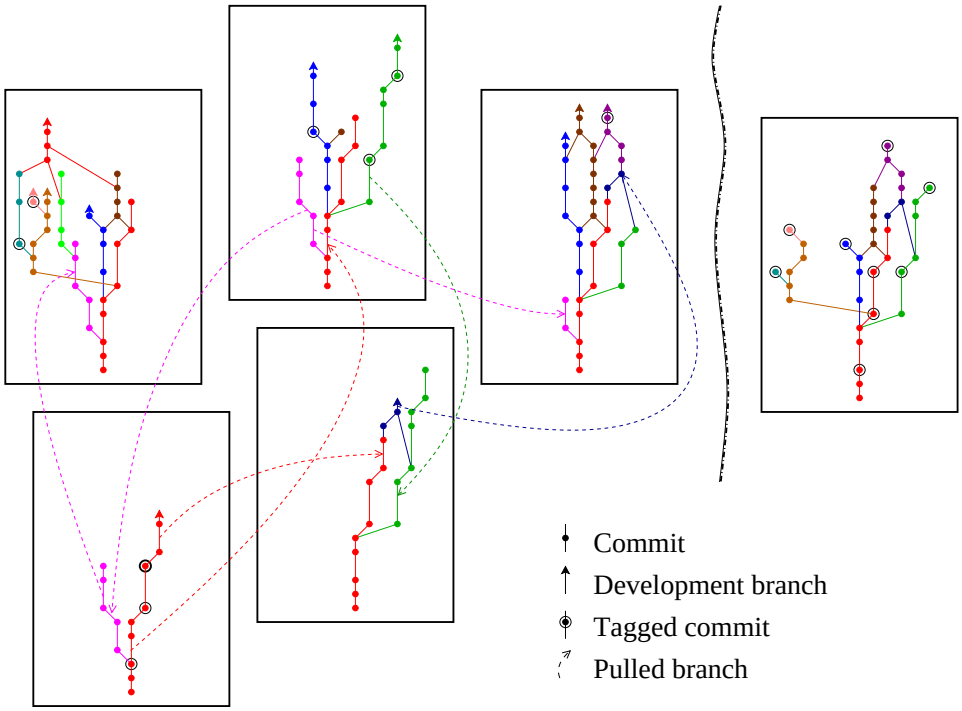

Figure 2: To the left: five repositories with common ancestry, with development influences from each other. To the right: an "archive" repository with tags from the left repositories. Only commits leading up to the tagged versions become part of the archive.

3.2 Tag archive

The tag archive is realised as a bare repository stored as

`/u/landcvs/gittags/`*`projectname`*`.git`

The reason for keeping it under the `landcvs` user is to limit the number of users that can write directly to it. In fact, no user should ever write to it as it is updated by a script[9].

The reason the archive is updated by a script is that it can be quite a challenge to get specific commits into a repository, without making mistakes. Commits to be archived are therefore first pushed to a bare staging archive, on the `land` user account:

`/u/land/gitstage/`*`projectname`*`.git`

[9]To avoid mistakes also by those having access to the `landcvs` account, it is **forbidden** to clone or fetch a tag archive using `landcvs` as username in the URL. Note that this is not enforced by technical means — users with access need to exercise their skills. Besides, one would rarely clone from this repository.

If this repository is somehow destroyed or scrambled, it can easily be recreated without any ill effects on the archive repository.

Since the archive is not a development tree, it contains no branches. It is very likely to become a rather "bushy" tree (graph) of commit heritage lines as its sole purpose is to store snapshots of code and parameters that were used at various stages, and possibly abandoned later.

To easily move the commits around, tags are used as handles, but are themselves not important. To put a commit into the archive, first tag it and push the tag to the staging archive. Then tell the fetcher script (which runs as `landcvs` in a CRON job), to fetch that tag. This is done by (as `land`) calling:

```
landaddgittag projectname name-date Name
```

where *`projectname`* is the project (e.g. `land02` or `s393`), *`name-date`* a tag as one word and *`Name`* your name (for the benefit of the script logfile).

4 Tagged results

As a GIT commit represents a snapshot of the tracked directory tree, one `SHA1` hash reference to the current commit is enough to precisely describe the state of code or parameters. The `SHA1` can be retrieved at any time by:

```
git describe --dirty --always --tags --long
```

and may look like "`htj-20131031-0-g42178d7`".[10] This is on the form `[TAG-N-g]SHA1[-dirty]`[11], where `TAG` is the most recent tag, `N` steps away from the current commit `SHA1`. The mark `-dirty` appears if the working directory has uncommitted changes[12].

The idea is now very simple: by attaching the `SHA1` of both the used analysis code and parameters[13] to results presented, they can be traced back and reproduced.

4.1 Tagged ROOT-files

To allow tracability, the intermediate analysis results must be marked. Since our analysis process has a clear information flow from sources to destination of the form:

$$\text{raw data} + \text{analysis code} + \text{calib.parameters} \rightarrow \text{intermediate}, \qquad (1)$$

[10] The GIT installtion at GSI is ancient and does not support the `--dirty` option. LAND02 knows how to work around this.

[11] Note the `-g`, which is not part of the `SHA1`.

[12] In which case the entire identifier is of limited use.

[13] In the future also auxillary scripts used to generate plots etc. should be tagged. But let us do one step at a time.

where intermediate is a ROOT-file, and

$$\text{intermediate} \rightarrow \text{results} = \text{figures} \ \& \ \text{values}, \tag{2}$$

one important ingredient is to mark our ROOT-files.

LAND02-generated output depends on:

- The source code.
- The calibration parameters.
- The command line.

The two first can be uniquely identified by embedding the `git-describe` identifiers of the source and parameter directories, at compile and run-time, respectively[14], into the ROOT-file. The command line can be embedded as is[15] (this is not implemented yet).

ROOT-files produced by Ralf's TRACKER should pass these identifiers along, and in addition obtain similar identifiers describing the source and parameters used for particle tracking. (Not implemented yet.)

4.2 Tagged figures

Histograms produced from trees in marked ROOT-files can be annotated by using a small script: `tagplot.C`. This places a small identifying text in the lower left corner of the current figure, see example in Figure 3. (Default here, optional there...) It can also generate a pure text output, suitable for mail or other media (e.g. marking a full presentation instead of every picture). Usage when standing in a LAND02 experiment directory (`land02/Sxxx`):

```
.L ../scripts/tagplot.C++
h509->Draw("Inz:Inaoverz")
tagplot()
```

Finally, even publications might be marked at the end, just like the arcane funding agency contract numbers: "These results were obtained using analysis software with version-tag c0ffee." Which then references a script or other description, that in turn has the code and parameter references included.

[14] Due to some parameters actually being compiled into LAND02 instead of being parsed at run-time, they need to be identified also at compile time, but this is a minor historical technical issue (handled by the tools).

[15] Which just shifts the interpretation work to the next level, but at least maintains traceability.

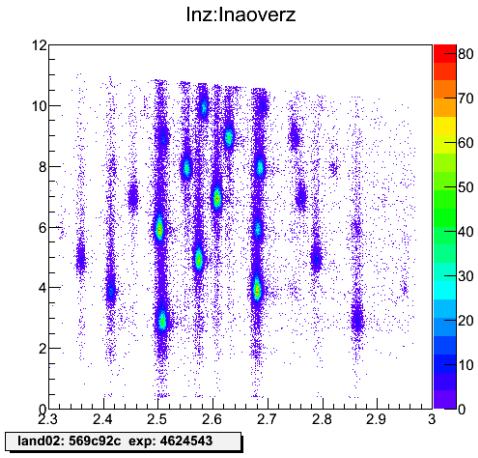

Figure 3: Figure with `SHA1` hash tags in the lower left corner.

5 Guidance to the tag archive, by example

The workflow including interaction with the archive repository is described as a working example below. (The `test-proj` sandbox does exist — please try by executing the commands! For production use, `test-proj` would be e.g. `land02`, `s393` or some other collaboration GIT project name[16].)

1. Check out a project directory from any (normal) upstream repository:

   ```
   git clone \
     land@lx-pool.gsi.de:/u/landcvs/test-proj.git
   cd test-proj
   git log --all --oneline --decorate --color
   ```

2. Then, hack it a bit (e.g. modify a comment):

   ```
   emacs hello.sh
   ```

3. Check, stage and commit the changes in your repository:

   ```
   git diff
   git add hello.sh
   git commit
   ```
   (Enter a correlated message.)

4. This commit is an important mile-stone in development — tag it! (E.g. on the form `name-date[-feature]`.)

   ```
   git tag smith-19830123
   ```

[16] Contact `f96hajo@chalmers.se` if your favourite project does not have an archive yet.

5. The tagged commit has been used to produce some results or plots shown somewhat "officially". Collaborative behaviour requires:

   Push it to the staging area for the archive (on the `land` account):

   ```
   git remote add stagetags \
     land@lx-pool.gsi.de:/u/land/gitstage/test-proj.git
   ```

   ```
   git push stagetags smith-19830123
   ```

6. Tell the automatic fetch-script to do its work (giving the tag, and your name for the script log files)!

   ```
   ssh land@lx-pool.gsi.de
   ```

   ```
   landaddgittag test-proj smith-19830123 Smith
   ```

7. Check that the tag was accepted into the archive:

   ```
   tail -f /u/landcvs/gittags/addtagslog.txt
   ```

Some other useful operations:

- The following will create a complete copy of one project repository in the central tag-keeping archive. (Effectively a backup, if one so wants.) (It will complain a bit, since there is no `HEAD` set (which is how the archive is — it is not a development tree):

  ```
  git clone \
    land@lx-pool.gsi.de:/u/landcvs/gittags/test-proj.git
  ```

  ```
  cd test-proj
  ```

  ```
  git log --all --oneline --decorate --color
  ```

- Fetching single tags is also possible (create empty repository, add remote, fetch):

  ```
  mkdir test-proj
  ```

  ```
  cd test-proj
  ```

  ```
  git init
  ```

  ```
  git remote add archive \
    land@lx-pool.gsi.de:/u/landcvs/gittags/test-proj.git
  ```

  ```
  git fetch --no-tags archive \
    refs/tags/smith-19830123:refs/tags/smith-19830123
  ```

6 GIT cheat-sheet

A list of useful GIT commands.

- to list all branches (active is marked with *):
  `git branch`
- to change the working tree to a certain branch:
  `git checkout <branch>`
- to add a remote repository to fetch from or push to:
  `git remote add <remotename> <url>`
- to push a certain branch (to a remote repository):
  `git push <remotename> <branchtopush>:<branchnameonremote>`
- to push a certain commit (to a remote repository):
  `git push <remotename> <commithash>`
- to tag a certain commit:
  `git tag <tagname> <commithash>`
- to push a certain tag (to a remote repository):
  `git push <remotename> <tagname>`
- to initialize a GIT repository:
  `git init`
- to initialize a bare (i.e. no working files) GIT repository:
  `git init --bare`
- to pull a certain tag:
  `git pull <remotename> <tagname>`
- to see the commits in a nice and colored way:
  `git log --all --decorate --graph --oneline --color`
- to add a file to the index (staging area for next commit):
  `git add <filename>`
- to make a commit:
  `git commit`