git tutorial

Sébastien Brochet[1]

[1]Institut de Physique Nucléaire de Lyon – Université Claude Bernard Lyon 1

July 9, 2013

What's git?

- **DVCS**: distributed version control system
- Allow you to:
    - Keep track of changes in your code
    - **Document** those changes
    - **Share** your code / Collaborate with others
    - a lot more!

git vs CVS/SVN

- Right now, CMS uses CVS as VCS. Even if it (sort of) works, CVS has some major drawbacks:
  - **One central server** keeps all the data. If it crashes, everything is lost! (backup ...)
  - You absolutely need an Internet connection: no work on train or plane
  - Branching and tagging are inefficient.
  - It's **slow**!
- git fixes all these problems:
  - It's **distributed**: all the history is stored **locally** (ie, your computer).
  - No Internet connection required: since everything is local, you can commit whenever you want.
  - Branching / merging are cheap and very fast.
  - It's very (very) fast. A diff is instantaneous in git (takes sometines minutes on CVS ...)

git vs CVS/SVN

- If your history is stored locally, how do you share your code?

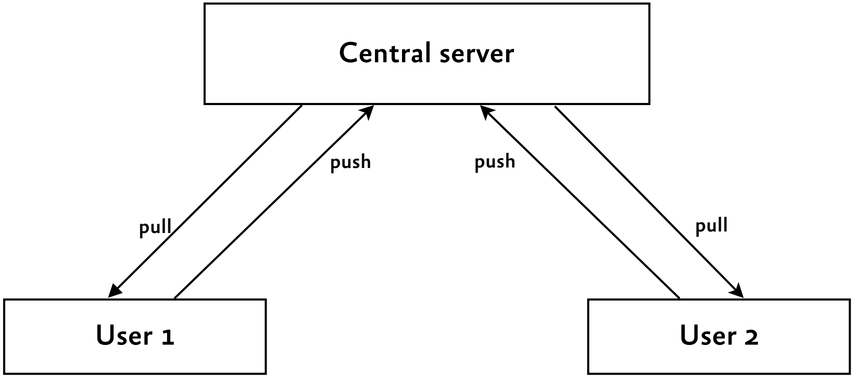
git vs CVS/SVN

- If your history is stored locally, how do you share your code?
- You can also have a central server on git:
  - Your history is still stored locally
  - You only need Internet access when you **pull** from or **push** to the central server!

git vs CVS/SVN

- If your history is stored locally, how do you share your code?
- You can also have a central server on git:
  - Your history is still stored locally
  - You only need Internet access when you **pull** from or **push** to the central server!
- Best of the two worlds:
  - You have everything you need to work (full history, commit, branching, merging, ...) locally
  - You can collaborate with others using the central server: when you want, you can sync your work with the central server (ie, retrieve other's work (**pull**) or send your work (**push**))

git server workflow



Key differences between CVS and git

- Same goal, different philosophy:
  - **Atomic operations**: operations either fail or succeed, no inconsistent state.
  - **Very important**: changesets (commits) refers to the **whole project**, and not to a single file like CVS. It is so **very** easy to revert a change (you don't have to track every single file you have changed like in CVS, just revert the commit).
  - Commit messages are mandatory: an empty message is not allowed. A convention exists between git users: commit messages are expected to start with a single line summarizing the change, then an empty line, followed by a more detailed description of the changes.
  - Each commit is identified by a commit hash, and not a revision number. The commit hash is a SHA-1 hash (like `bfc7747c4cf67a4aacc71d7a40337d2c3f73a886`) built using all files in the project.

History is everything

- **History is everything**: the most important thing for git is keeping track correctly of the history of your project.
- git will never allow an operation that results in a loss of history to the pushed centrally!
- Example:
    - You change a lot of files, introducing a bug. You commit, and push your changes centrally.
    - Someone notice there's a bug, and knows that your commit is responsible of it.
    - Instead of looking at each change separately, you prefer to revert (ie, remove) the commit.
    - An easy way for git is simply remove the commit from the history.
    - That's not what is done: git **never** edit history. Instead, a new commit is added at the top of the history, with the exact opposite of the commit to remove.

Setup your identify and your text editor

- git needs to know who you are!
- You need to configure your identify only once, on each computer you'll use (this configuration is stored either at repository level or in your home)

```
$ git config --global user.name "Sébastien Brochet"
$ git config --global user.email "s.brochet@ipnl.in2p3.fr"
```

- git uses UTF-8! Say yes to accentuated letters, Cyrillic or Chinese characters, ...!
- You'll also need a text editor for your commit message. By default, git use Vi, but you can change that. For example, to use emacs, simply run

```
$ git config --global core.editor emacs
```

Create a new repository

- Create a new repository

```
$ mkdir project
$ cd project/
$ git init
```

- Clone an existing project

```
$ git clone "https://github.com/cms-sw/cmssw.git"
$ cd cmssw
```

File states

- *Untracked*: this file is unknown to git and not managed

File states

- *Untracked*: this file is unknown to git and not managed
- *Unmodified* / *Committed*: no modification to this file

File states

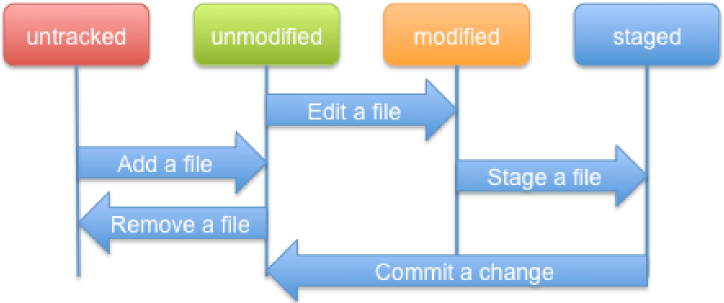
- *Untracked*: this file is unknown to git and not managed
- *Unmodified / Committed*: no modification to this file
- *Modified*: file modified, but not taken into account on the next commit

File states

- *Untracked*: this file is unknown to git and not managed
- *Unmodified* / *Committed*: no modification to this file
- *Modified*: file modified, but not taken into account on the next commit
- *Staged*: file added, deleted, moved or modified ; will be taken into account on next commit.

File states

- *Untracked*: this file is unknown to git and not managed
- *Unmodified* / *Committed*: no modification to this file
- *Modified*: file modified, but not taken into account on the next commit
- *Staged*: file added, deleted, moved or modified ; will be taken into account on next commit.



Create a file

```
$ touch README # README status: untracked
```

- Use `git status` to have an overview of the states of all files in the project:

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# README
```

Add a file

```
$ git add README # README status: untracked  →  staged

$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   README
#
```

- `README` is ready to be committed

Your first commit

```
$ git commit # The editor pops up (usually vim) ; enter the commit
    message, save and quit (:wq for vim), or
$ git commit -m "My first commit"
# [master (root-commit) bad13af] My first commit
#  1 file changed, 0 insertions(+), 0 deletions(-)
#  create mode 100644 README
```

- You can see the hash of your commit (`bad13af`), and what's changed (+1 file, no insertion nor deletions)
- To see the history, use `git log`

```
$ git log
# commit bad13af1848f716e434f7c9ad8bfd452826f0cd4
# Author: Sébastien Brochet <s.brochet@ipnl.in2p3.fr>
# Date:   Mon Jul 8 10:07:26 2013 +0200
#
#     My first commit
```

Edit file

```
$ echo "Hello world" > README # README is now modified
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
    directory)
#
# modified:   README
#
# no changes added to commit (use "git add" and/or "git commit -a")
```

- To see the differences before committing, use `git diff`

```
$ git diff
diff --git a/README b/README
index e69de29..802992c 100644
--- a/README
+++ b/README
@@ -0,0 +1 @@
+Hello world
```

Edit file

```
$ git add README # Stage file for commit
$ git commit -m "Blabla into README" # Commit staged files
# [master 669d569] Blabla into README
#  1 file changed, 1 insertion(+)

$ git log
# commit 669d5697f9e540653d7cd25eab4f4e411c7ace1b
# Author: Sébastien Brochet <s.brochet@ipnl.in2p3.fr>
# Date:   Mon Jul 8 10:22:18 2013 +0200
#
#     Blabla into README
#
# commit bad13af1848f716e434f7c9ad8bfd452826f0cd4
# Author: Sébastien Brochet <s.brochet@ipnl.in2p3.fr>
# Date:   Mon Jul 8 10:07:26 2013 +0200
#
#     My first commit
```

Show commit content

- To show the content of a commit, use `git show`

```
$ git show HEAD
# commit 669d5697f9e540653d7cd25eab4f4e411c7ace1b
# Author: Sébastien Brochet <s.brochet@ipnl.in2p3.fr>
# Date:   Mon Jul 8 10:22:18 2013 +0200

#     Blabla into README

diff --git a/README b/README
index e69de29..802992c 100644
--- a/README
+++ b/README
@@ -0,0 +1 @@
+Hello world
```

- Note: **HEAD** is a special alias for the current changeset. In our case, HEAD is `669d5697f9e540653d7cd25eab4f4e411c7ace1b`

Summary of commands

- `git help <cmd>`: print help about <cmd>
- `git init`
- `git clone`
- `git status`
- `git add <file>`
- `git rm/mv <file> [<to>]`
- `git commit`
- `git log`
- `git diff`

Interaction with the central server

- Now that you have some commits, you want to share them with your collaborators.
- All you need to do is to **push** your changes to the remote server.
- First, let check the remote server configured for your project

```
$ git remote
```

- If nothing is returned, you have no remote server configured. Let's add one (more details on how to create remote repository on github later)

```
$ git remote add origin https://github.com/blinkseb/ipnl-tuto.git #
    origin is the name of the remote server (you can have as many
    remote as you want!)
$ git remote
origin
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/blinkseb/ipnl-tuto.git
  Push  URL: https://github.com/blinkseb/ipnl-tuto.git
  HEAD branch: (unknown)
```

Interaction with the central server

- You can now push your changes to the remote `origin`

```
$ git push -u origin master
# Counting objects: 6, done.
# Delta compression using up to 4 threads.
# Compressing objects: 100% (2/2), done.
# Writing objects: 100% (6/6), 466 bytes | 0 bytes/s, done.
# Total 6 (delta 0), reused 0 (delta 0)
# To https://github.com/blinkseb/ipnl-tuto.git
#  * [new branch]      master - > master
# Branch master set up to track remote branch master from origin.
```

- The `-u` flags is needed only for the **first push**. It sets the remote `origin` as the default one for the next push and pull
- `origin` is the name of the remote
- `master` is the name of the branch to push. The default branch is always named **master**.

Interaction with the central server

- To retrieve the changes from remote, use `git pull`

```
$ git pull origin master
# Already up-to-date.
```

- **Very important**: when pulling changes from remote, all the files changed remotely **must be in a unmodified state locally**, otherwise the pull will fail!.
    - Either **commit** the modified files, **stash** them, or revert them to their initial state.
- You may have merge conflicts if the same part of the file has been modified locally and remotely. It's off-topic, see here for more details.

Interaction with the central server

- The **push** will fail if there were push done by someone else between your last pull and your push. This will lead to the following scenario:

```
$ git push origin master
# To https://github.com/blinkseb/ipnl-tuto.git
#  ! [rejected]        master  →  master (fetch first)
# error: failed to push some refs to
    'https://github.com/blinkseb/ipnl-tuto.git'
# hint: Updates were rejected because the remote contains work that you do
# hint: not have locally. This is usually caused by another repository pushing
# hint: to the same ref. You may want to first merge the remote changes (e.g.,
# hint: 'git pull') before pushing again.
# hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- git tells you it refuses to push because "the remote contains work that you do not have locally". All you need to do is to pull in order to incorporate remote changes with yours, and then push

```
$ git pull --rebase origin master
$ git push origin master
```

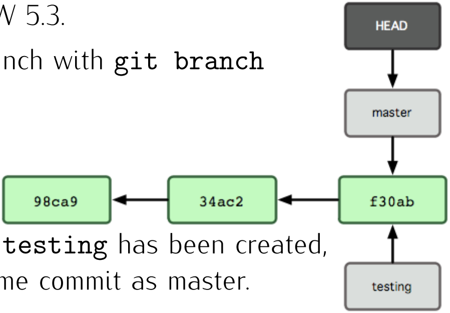
A note about rebase

- On the previous slide, I used `git pull --rebase origin master`. Notice the `--rebase` flag.
- If you have done some commits after your last pull, **always use the `--rebase` flag for `git pull`**!
- **rebase** rebuilds your local history, putting all your local commits done after your last pull **on top of the history**, avoiding the need of a merge commit.
- VERY IMPORTANT: NEVER EVER **rebase commits that you have pushed to a remote repository!** Rebase only **local** commits!
- More details about rebase here

Branching and tagging

- A very powerful feature of git is cheap branching. A branch can be seen as a branch in the history tree
- Let's see a concrete example. Suppose you work on an analysis which need to be compatible for CMSSW 5.3 and for CMSSW 6.2. You'll need two branches, `master` for development on CMSSW 6.2, and `testing` for CMSSW 5.3.
- You can create a branch with `git branch`

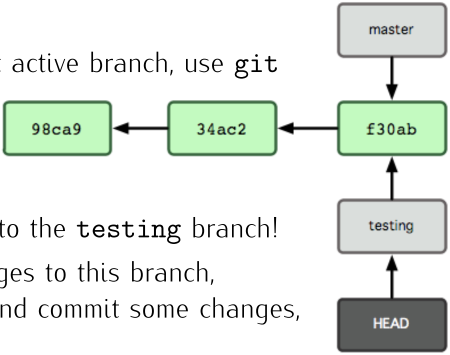
```
$ git branch testing
```



- See that the branch `testing` has been created, and points to the same commit as master.
- You are still on the `master` branch! (see where HEAD points to)

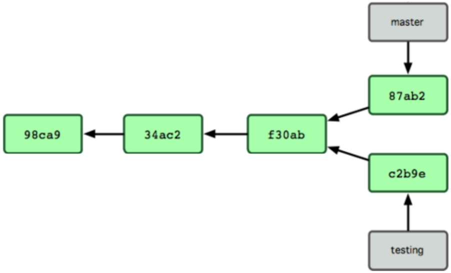
Changing branch

- In order to change the current active branch, use `git checkout`



```
$ git checkout testing
```

- Note that now, HEAD points to the `testing` branch!
- If now you commit some changes to this branch, checkout the master branch, and commit some changes, the history diverges.



Merge branch

- Suppose you created a branch **`new_feature`** to develop a new feature of your analysis. You broke the code, but it was on your own branch, so everyone else was able to use the analysis using the **`master`** branch.
- Your work is done: everything works at it should. You want now to **merge** the branch **`new_feature`** into **`master`** so everybody can enjoy your new feature.
- It's done very easily using **`git merge`**

```
$ git checkout master # Be sure to be on the master branch
$ git merge new_feature # Merge the new_feature branch inside the
    active branch (master)
```

- Every commit on **`new_feature`** is now part of **`master`**. You can safely remove your develop branch

```
$ git branch -d new_feature # git won't delete the branch if it has
    not been merge previously!
```

Some words about tagging

- Tagging is another great and cheap feature of git. You can assign an alias to a commit hash. It's very useful to identify a precise moment in the lifetime of your analysis for example (ie, version of code used to launch a PAT production, or for example, v1 of software released to public, etc.)
- To tag, simply use `git tag`

```
$ git tag -a my_tag_name # Create a annotated tag. You'll need to
    enter a tag message
$ git push --tags # Push the tags to the default remote
```

Github

- Github is a social collaboration website, where users can create an unlimited number of **public** git repositories, for free.
- CMSSW code have been migrated from CERN's CVS to github: https://github.com/cms-sw/cmssw
- In order to keep contributing, you'll need a github account. Please register!
- Don't forget to create your public / private keys for your github account. More details here
- I've created a github organization for the IPNL CMS group: https://github.com/organizations/IPNL-CMS. The plan is to have a centralized place where all the tools used by the group can be stored.
- Send me an email with your github account so I can add you as a member!

Various links

- git website: http://git-scm.com/
- Pro Git, a free book on git, available in French or English: http://git-scm.com/book
- TryGit, an interactive tutorial to learn git: http://try.github.io/
- Github: https://github.com/
- A real world example of git power: the linux kernel! https://github.com/mirrors/linux