Git, wut dat? [nmv-cs1-20-sept-17]

Definitions

(D)VCS

(Distributed) Version Control System

Goals

- Hold all the versions of all the files
- Hold all the dir tree
- Allow identification of dirs and files
- Provide utilities to manage it all
- DVCS, same but over distributed architecture

`diff` & `patch`

`diff` - compare files line by line

Important flags:

- `-w --ignore-white-space` ignore all w hite space.
- `-u -U NUM --unified[=NUM]` Output NUM (default 3) lines of unified context.

`patch` - apply a diff file to an original

```
SYNOPSIS
       patch [options] [originalfile [patchfile]]
       but usually just
       patch -p num <patchfile
```

Important flags

```
-pnum  or  --strip=num
Strip the smallest prefix containing num leading slashes from each file name
found in the  patch file. A  sequence of one or more adjacent slashes is
counted as a single slash. This controls how file names found in the patch
file are treated, in case you keep your files in a different directory than
the person who sent out the patch. For example, supposing the file name in the
patch file was

/u/howard/src/blurfl/blurfl.c

setting -p0 gives the entire file name unmodified, -p1 gives

u/howard/src/blurfl/blurfl.c

without the leading slash, -p4 gives

blurfl/blurfl.c

and not specifying -p at all just gives you blurfl.c.  Whatever you end up
with  is  looked  for
either in the current directory, or the directory specified by the -d option.

-r  --recursive
```

```
Recursively compare any subdirectories found```

#### Example
```

bash
$ diff toto.c toto-origin.c > correction.patch
$ bzip2 correction.patch
$ bzcat correction.patch.bz2 | patch -p 0 toto.c

Or recursively

$ cp -r Linux-2.6.17 linux-2.6.17-orig
$ cd linux-2.6.17
$ vim
$ cd ..
$ diff -r -u Linux-2.6.17-orig Liinux -2.6.17 > \
netw ork-driver-b44.patch
$ gzip -9 netw ork-driver-b44.patch
$ cd linux—2.6.17
$ zcat netw ork-driver-b44.patch.gz | patch -p 1

```
### History
History is a directed acyclic graph consisting of all possible versions one can compute from adjacent versions by applying

![Git-history](./images/git-history.png)

### Branch
A branch in Git is simply a lightweight movable pointer to one of the commits. The default branch name in Git is master. A

A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or

Branch of a version v[i] of a history is a sub-graph composed of the set of versions available from v[i] in the history gr

![Git-branch](./images/git-branch.png)
### Trunk
Trunk or main branch is the branch from the latest stable version.

![Git-trunk](./images/git-trunk.png)

### Sub-branch
Une sous-branche SB1 d'une branche B2 est le sous graphe composé de l'ensemble des versions d'une branche B1 n'appartenant

![Git-trunk](./images/git-sub_branch.png)

### Merge
On appelle merge toute version ayant un degre sortant strictement superieur a 1. Cette version correspond alors a la fusio

![Git-merge](./images/git-merge.png)

## Architecture interne git
Le nom du fichier est dans le repertoire

Un blob tree est un repertoire
Chaque blob dans git c'est 1 fichier

Les noms de fichier en ```projet.git``` viennent du flag ```--bare``` qui garde l'historique des fichiers, en gros que le

Git uses essentially 4 types of objects:
* Blob: Stores the files contents
* Tree: Contains the directories tree
* Commit: Contains versions of the repository
* Tag: Name for a version, either user-specified or `sha1` generated.

Blob, Tree and Commit are not mutable and shouldn't normally be deleted.
```

```
### Blobs
Blobs store file contents.
* They're identified with `sha1` of it's content
* Every new version of a file has a new blob
* Blob doesn't depend on name nor path
  * If a file is renamed, no new blob
  * If a file is moved, no new blob

![Git-blob](./images/git-blob.png)
### Trees
Stores the list of files in a dir:
* Tree is a set of pointers to blobs and other trees
* Tree associates filename (dirname) with every blob (tree) pointer
* A set of trees describes the state of a dir hierarchy at a given point

![Git-tree](./images/git-tree.png)
### Commits
To commit a file means to save it's version in a version manager.

#### CVS
![Git-cvs](./images/git-cvs.png)
CSV uses patch versionning, file by file
* [-] Hard to find a coherent state of the system
* [-] Accessing a version means reapplying all the patches

#### SVN
![Git-svn](./images/git-svn.png)
SVN uses global versionning on all the patches
* [+] Every version number represents a coherent version
* [-] Accessing a version requires re-applying all the patches

#### Git
![Git-git](./images/git-git.png)
Git uses global versionning for files
* [+] Every version number represents a coherent version
* [+] Direct acces to versions

Commit stores the state of a part of the repository at a given time. It contains:
* Pointer to Tree of which state we want to save
* Pointer to one or more other Commits to build history
* Name of the author, and of the commiter
* Description string

![Git-commit](./images/git-commit.png)

The difference between *commiter* and *author*:

As not everyone is allowed to commit, and common courtesy dictates to name the contributor, two fields are present. Commit

### Tags
Allows to identify objects by a name. Contains:
* pointer to _Blob_, _Tree_ or _Commit_
* A signature
### Overview
![Git-overview](./images/git-overview.png)


## Git Locally
So what is git? Git is simply a set of commands, a large one to it. In total over 145, one of which is the `git` command,
#### Regular repository
### ```git init```
Initializes an empty git repository.
```

bash
$ mkdir proj
$ cd proj
$ git init
defaulting to local storage area

```
or in case of pre-existing repository:
```

bash
$ cd project
$ git init
$ git add .
$ git commit -alias
defaulting to local storage area

```
#### Server repository
Server version of the repository contains
* No files, only history
* No `proj/.git` directory, `proj.git/`
* No `remote` (origin)

**Example**
```

bash
$ git clone --bare
$ cat proj.git/config
[core]
...
bare = true
...

```
### Commit
![Git-commit-vs](./images/git-commit-vs.png)
#### Useful commands
* `add` This command updates the index using the current content found in the working tree, to prepare the content staged
* `commit` Stores the current contents of the index in a new commit along with a log message from the user describing the
* `reset HEAD` removes the reference of a file added with `add`
#### Examples
**Ex-1**
```

bash
$ echo "file1" > foo.txt
$ git add foo.txt
$ git commit -m "Commit description"

```
**Ex-2**
```

bash
$ mkdir project
$ cd project
$ git init
$ echo "toto" > foo.txt
$ git add foo.txt
$ git commit -m "Add foo.txt"
$ mkdir dir
$ echo "titi" > dit/bar.txt
$ git add dir/bar.txt
$ git commit -m "Add dir/bar.txt"
$ echo "tutu" > dir/bar.txt
```
```

$ git add dir/bar.txt
$ git commit -m "Modif dir/bar.txt"

```
![git-commands-vs-tree](./images/git-commands-vs-tree.png)
![git-commands-vs-tree-2](./images/git-commands-vs-tree-2.png)

### Branch
#### Basic commands
* `branch` list branches (active one flagged `*`)
* `branch <name>` create new branch
* `branch -m` rename branch
* `branch -d` delete branch
* `checkout` change (and/or create) active branch
* `show-branch` show branches and their commits
#### Example
```

bash
$ git branch

- master
  $ git branch myBranch
  $ git branch
  myBranch
  master
  $ git checkout myBranch
  $ git branch
- myBranch
  master

```
#### Commit's internal structure
```

bash
$ ls
foo.txt dir
$ git branch myBranch
$ git checkout myBranch
$ touch file1.txt
$ ls
dir file1.txt foo.txt
$ git add file1.txt
$ git commit -m "Add file1.txt"
$ git checkout master
$ ls
dir foo.txt
$ touch file2.txt
$ git add file2.txt
$ git commit -m "Add file2.txt"

```

```
Merge
`git-merge` - Join two or more development histories together

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the cur

Assume the following history exists and the current branch is "master":

             A---B---C topic
            /
```
```

```
       D---E---F---G master


Then "git merge topic" will replay the changes made on the topic branch since it diverged from master (i.e., E) until its
two parent commits and a log message from the user describing the changes.

            A---B---C topic
           /         \
       D---E---F---G---H master


The second syntax (<msg> HEAD <commit>...) is supported for historical reasons. Do not use it from the command line or in

The third syntax ("git merge --abort") can only be run after the merge has resulted in conflicts. git merge --abort will a
changes when the merge started (and especially if those changes were further modified after the merge was started), git me

Warning: Running git merge with non-trivial uncommitted changes is discouraged: while possible, it may leave you in a stat

### Rebase

If <branch> is specified, git rebase will perform an automatic git checkout <branch> before doing anything else. Otherwise

 If <upstream> is not specified, the upstream configured in branch.<name>.remote and branch.<name>.merge options will be u
 not have a configured upstream, the rebase will abort.

 All changes made by commits in the current branch but that are not in <upstream> are saved to a temporary area. This is t
 specified).

 The current branch is reset to <upstream>, or <newbase> if the --onto option was supplied. This has the exact same effect
 the reset.

 The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in o
 HEAD..<upstream> are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will b

 It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve a
 the merge failure with `git rebase --skip`. To check out the original <branch> and remove the `.git/rebase-apply` working

 Assume the following history exists and the current branch is "topic":

              A---B---C topic
             /
        D---E---F---G master


 From this point, the result of either of the following commands:
```

bash
git rebase master
git rebase master topic

```
 would be:

                     A'--B'--C' topic
                    /
        D---E---F---G master


 NOTE: The latter form is just a short-hand of git checkout topic followed by git rebase master. When rebase exits topic w

 If the upstream branch already contains a change you have made (e.g., because you mailed a patch which was applied upstre

              A---B---C topic
             /
        D---E---A'---F master


 will result in:

                       B'---C' topic
```

```
                     /
        D---E---A'---F master
```

Here is how you would transplant a topic branch based on one branch to another, to pretend that you forked the topic bran

First let’s assume your topic is based on branch next. For example, a feature developed in topic depends on some function

```
    o---o---o---o---o  master
         \
          o---o---o---o---o  next
                           \
                            o---o---o  topic
```

We want to make topic forked from branch master; for example, because the functionality on which topic depends was merged

```
     o---o---o---o---o  master
         |            \
         |             o'--o'--o'  topic
          \
           o---o---o---o---o  next
```

We can get this using the following command:

`git rebase --onto master next topic`

Another example of `--onto` option is to rebase part of a branch. If we have the following situation:

```
                             H---I---J topicB
                            /
                   E---F---G  topicA
                  /
     A---B---C---D  master
```

then the command

`git rebase --onto master topicA topicB`

would result in:

```
                  H'--I'--J'  topicB
                 /
                | E---F---G  topicA
                |/
     A---B---C---D  master
```

This is useful when topicB does not depend on topicA.

A range of commits could also be removed with rebase. If we have the following situation:

```
    E---F---G---H---I---J  topicA
```

then the command

`git rebase --onto topicA~5 topicA~3 topicA`

would result in the removal of commits F and G:

```
    E---H'---I'---J'  topicA
```

This is useful if F and G were flawed in some way, or should not be part of topicA. Note that the argument to --onto and t

In case of conflict, git rebase will stop at the first problematic commit and leave conflict markers in the tree. You can
edit, you need to tell Git that the conflict has been resolved, typically this would be done with

`git add <filename>`

After resolving the conflict manually and updating the index with the desired resolution, you can continue the rebasing pr

```
`git rebase --continue`

Alternatively, you can undo the git rebase with

`git rebase --abort`

### Rebase vs Merge
```

bash
$ git checkout maBranche
$ git merge master

```
![git-rebase-v-merge-1](./images/git-rebase-v-merge-1.png)
```

bash
$ git checkout myBranche
$ git rebase master

```
![git-rebase-v-merge-2](./images/git-rebase-v-merge-2.png)
```

bash
$ git checkout branche2
$ git rebase -onto master branche1 branche2

```



### Mistakes
**If the mistake is already pushed, the only acceptable model is _revert_. Other solutions may induce incoherence.**
#### Amend
Modify the last commit
```

bash
$ ls
foo.txt dir
$ touch bar.txt
$ git commit -m "Ajou d'un fichier."
$ git add bar.txt
$ git commit --amend -m "Ajout d'un fichier"

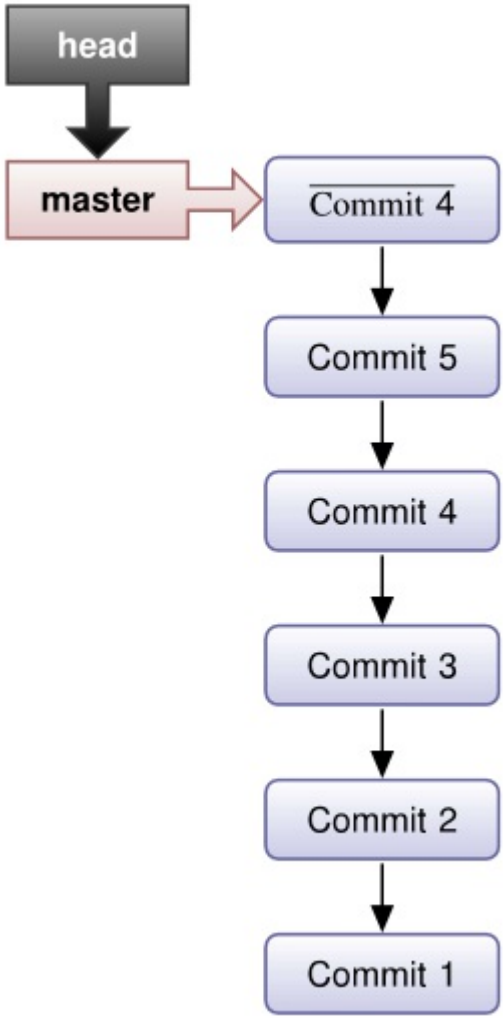
```



#### Revert
To cancel a commit with a different commit
```

bash
$ git branch master
$ cat fichierl.txt
Premiere version de Fl
$ cat fichier2.txt
Premiere version de F2
```
```

$ echo "Deuxieme version de F1" > fichier1.txt
$ git add fichierl.txt
$ git commit -m "Add fichierl.txt"
$ echo "Deuxieme version de F2" > fichier2.txt
$ git add fichier2.txt
$ git commit -m "Add fichier2.txt"
$ git revert HEAD^
$ cat fichierl . txt
Premiere version de Fl
$ cat fichier2. txt
Deuxieme version de F2
```




## Reset

Restore ancient commit

Commit related files are not really deleated w ith reset. They're flagged and only deleted w ith `git gc` , if they're old enough.

`--hard`

- restores the commit reference (active branch)
- restores index
- restores data

**regular (no flags)**
```

- restores the commit reference (active branch)
- restores the index

`---soft`

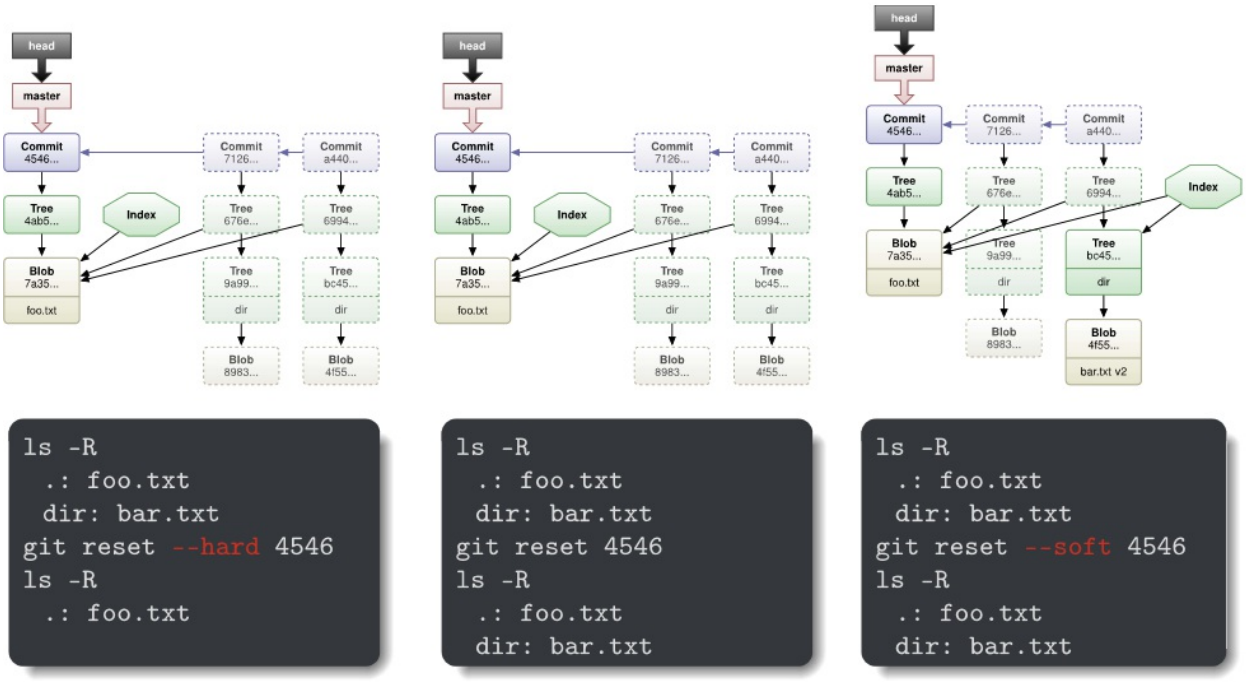
- restores the commmit reference (active branch)

Distributed git

DVCS : Distributed version control system

Remote push

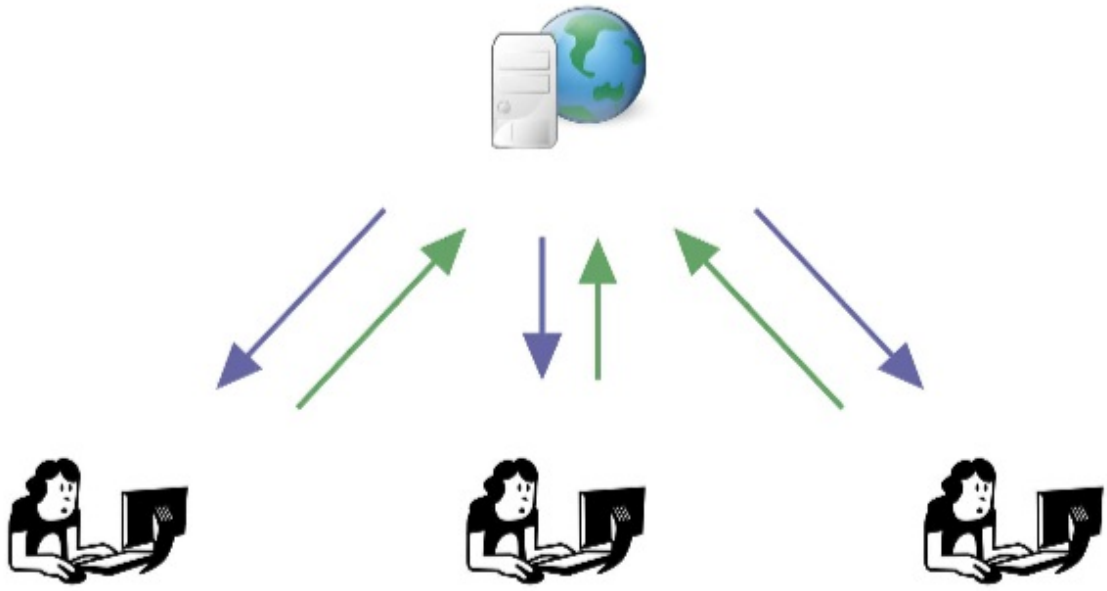
Origin c'est un alias de la machine / facon de contacter la machine sur laquelle se trouve le clone, ca se change



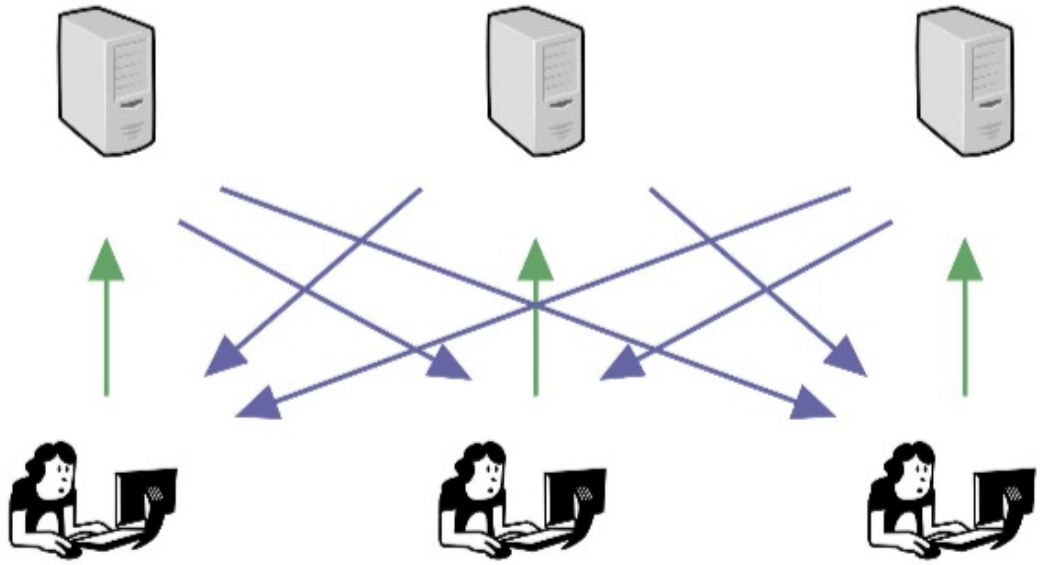
Workflows

4 w orkflow s possible

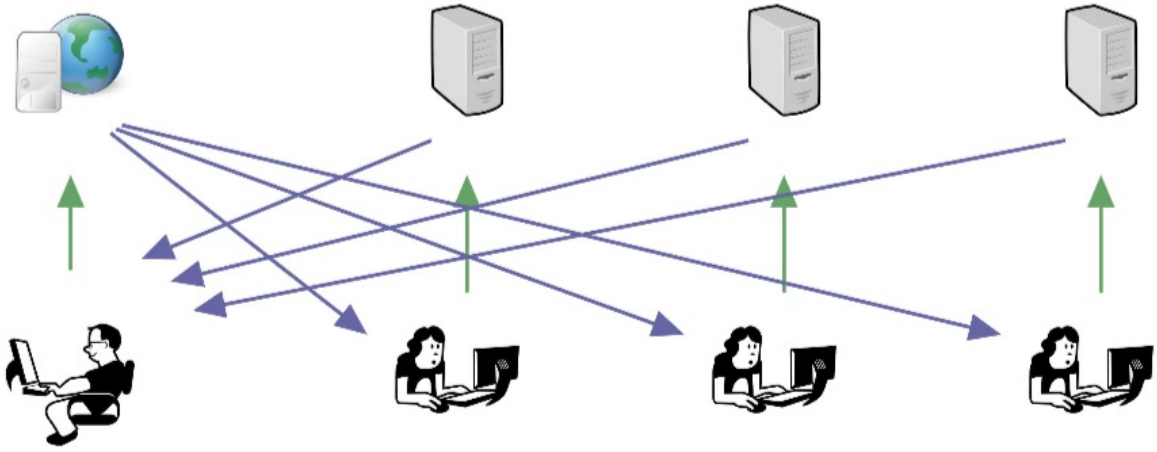
- ***Centralized Workflow*** Centralized repo as in CVS in SVN.



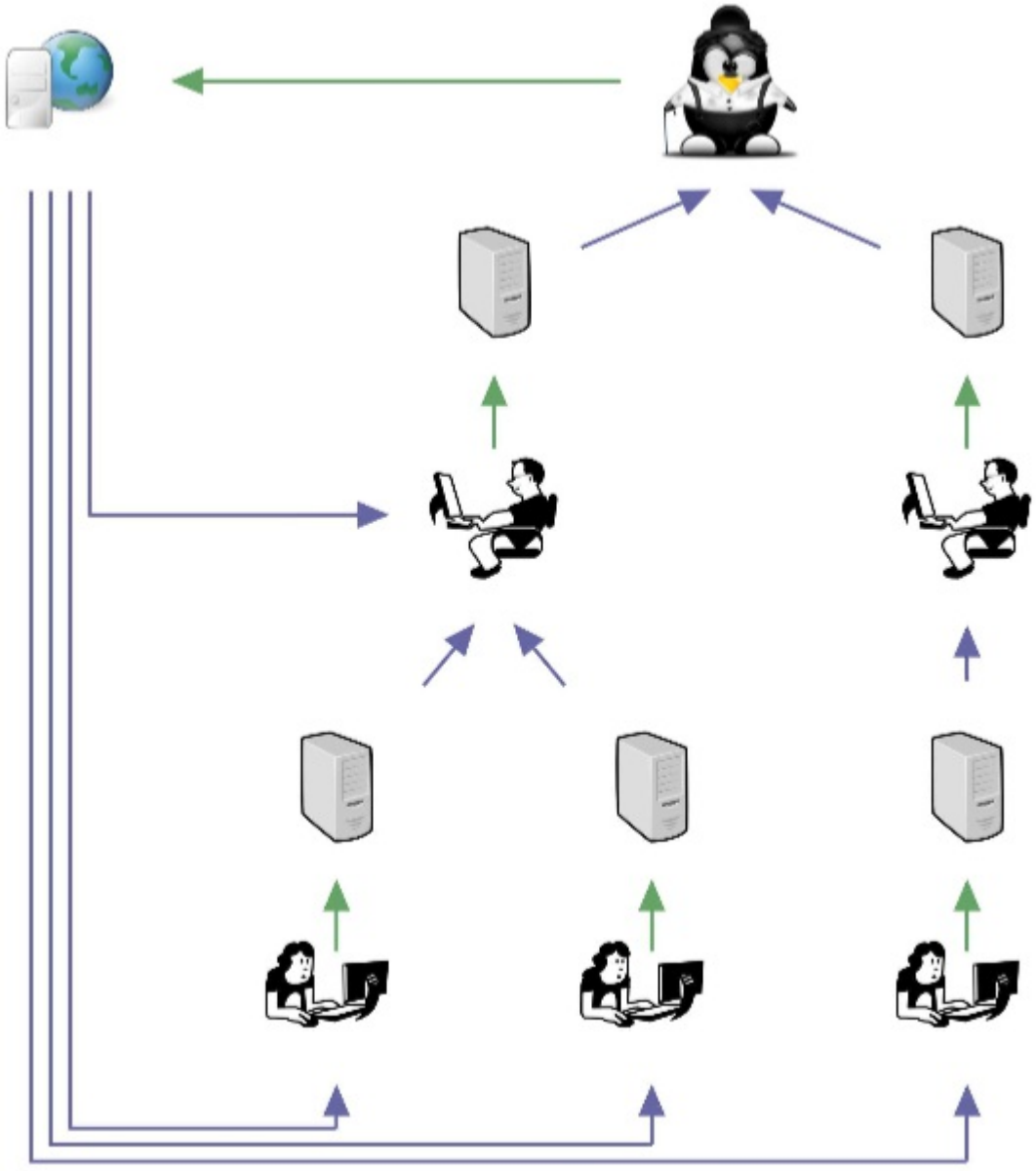
- ***Cooperative and Decentralized Workflow*** One repo per developer, synchros all around, traditional Arch model



- ***Integration-Manager Workflow*** One repo per developer and a manager w ho merges cleans and yells



- ***Dictator and Lieutenants Workflow*** Mail-based, linux kernel method

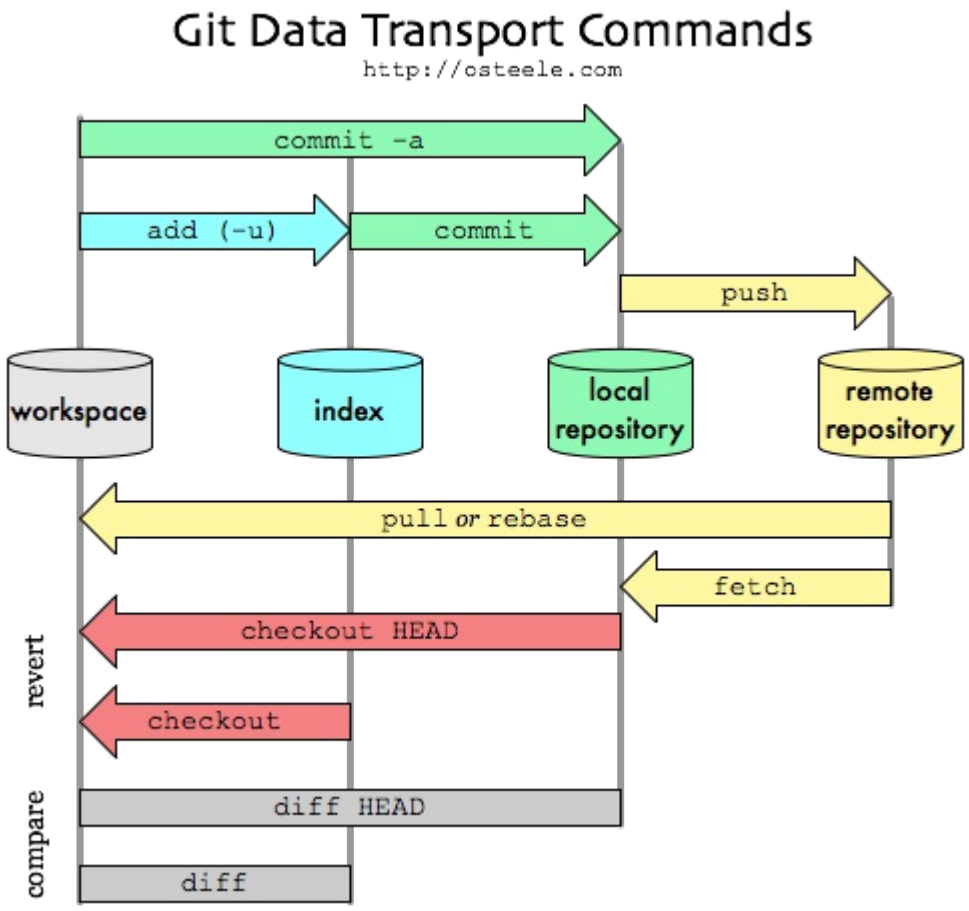


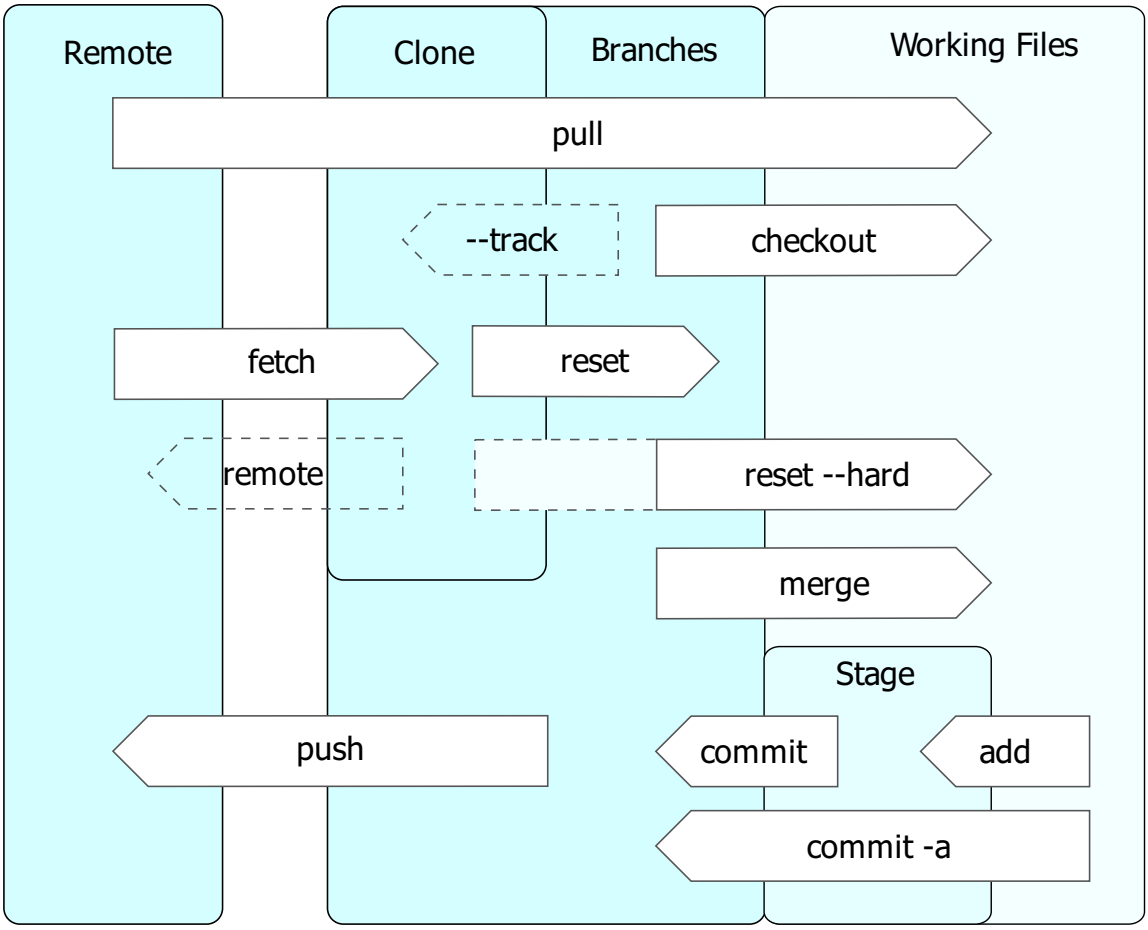
Conclusions

- [+] The fastest one to apply patches
- [+] Easy to setup in server mode
- [+] Small pow erful tools (Unix spirit)
- [+] Active development
- [-] Learning curve

Extra ressources

- Interactive Git cheat sheet

- 
Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
checkout
revert
diff HEAD
diff
compare

- 
Remote
Clone
Branches
Working Files
pull
--track
checkout
fetch
reset
remote
reset --hard
merge
Stage
push
commit
add
commit -a