GIT « AVANCÉ »

DAVID PARSONS

20 JUIN 2017

1

THINGS TO KNOW

☺ Take home message ☺

- Commit often
  - Keep commits small and commit together only related changes
- Write clear and informative logs
  - A log should enable its reader to:
    1. Identify at a glance the rationale behind the commit
    2. Have detailed explanation if needed
  - Template for logs (from http://git-scm.com/book/ch5-2.html)

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely).

Configuring git

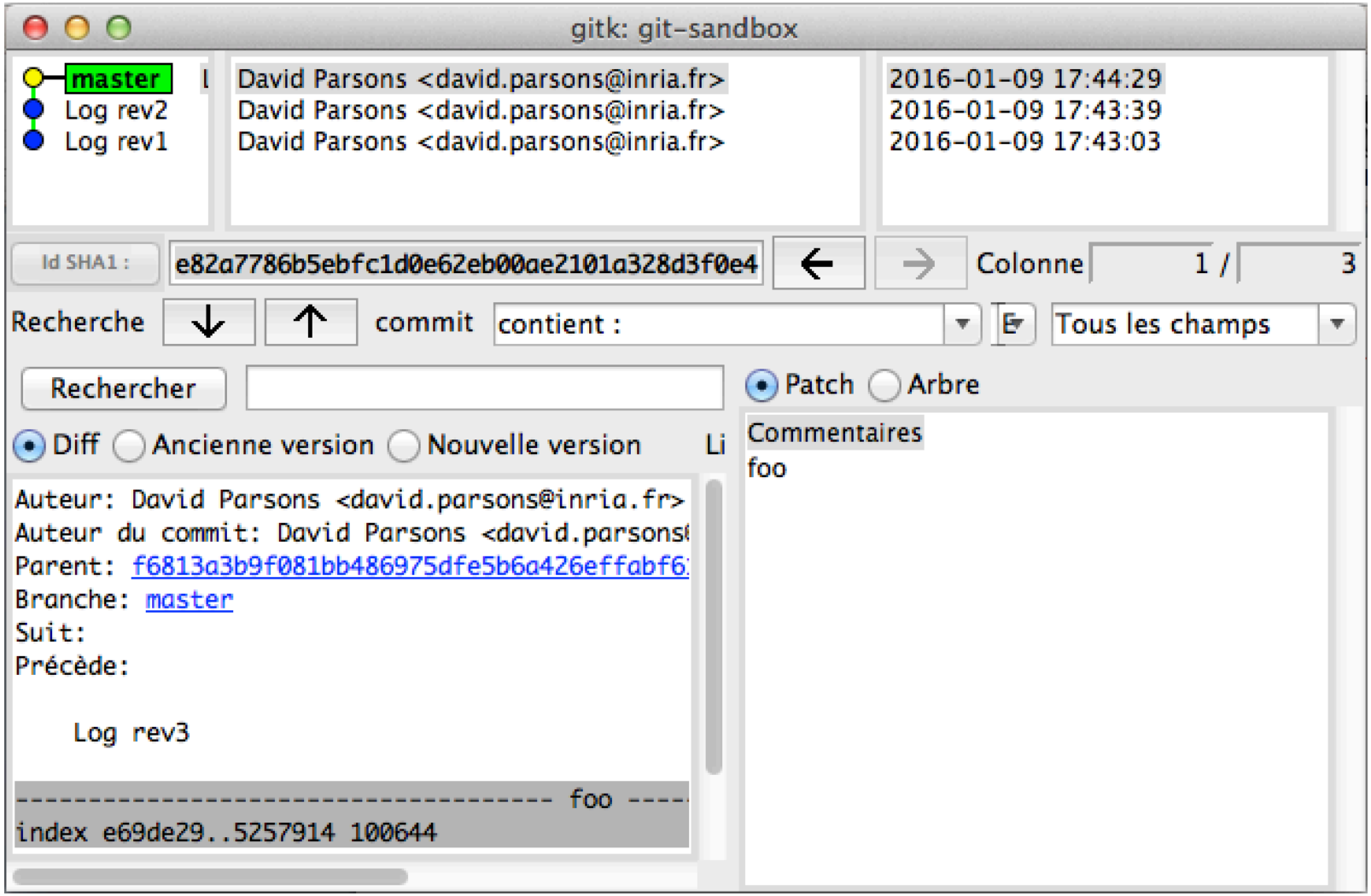
```
$ # Configure your name and e-mail address (almost mandatory)
$ git config --global user.name "David Parsons"
$ git config --global user.email david.parsons@inria.fr
$
$ # Configure the editor git will open when needed
$ git config --global core.editor nano
$
$ # Setup a few aliases, either simple shorthands...
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.st status
$
$ # ... or including options
$ git config --global alias.lg "log --pretty=format:\"%h - %an : %s\""
$
$ # You can even create new commands
$ git config --global alias.unstage "reset HEAD"
```

Tree-ish

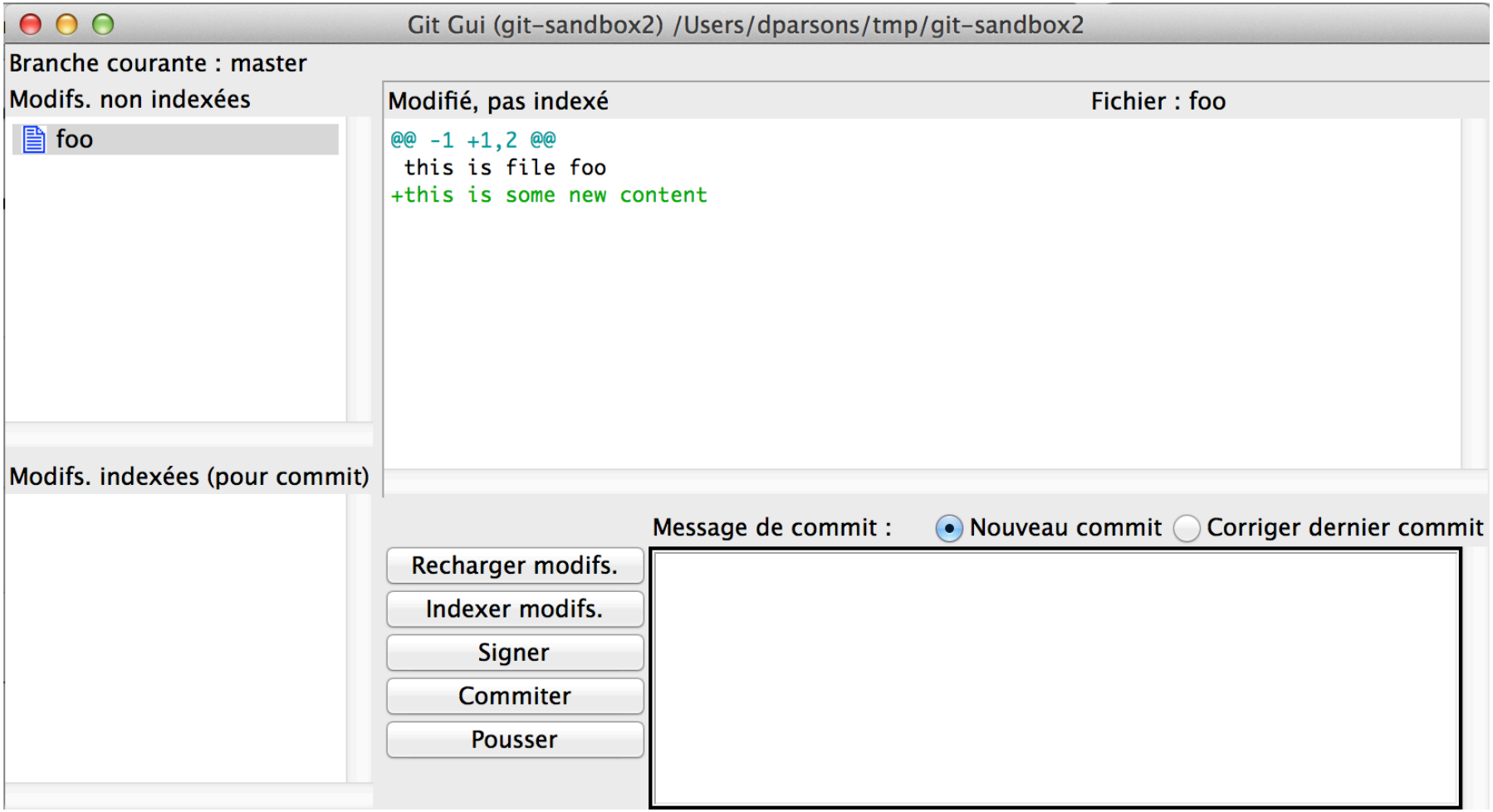
A commit can be identified in multiple ways, *e.g.* :

- Its SHA1 (possibly abbreviated)
- A reference (*e.g.* HEAD)
- An indirect reference : HEAD^, HEAD~, …
- A branch name
- …
- A tag

gitk


gitk: git-sandbox
master
Log rev2
Log rev1
David Parsons <david.parsons@inria.fr>
David Parsons <david.parsons@inria.fr>
David Parsons <david.parsons@inria.fr>
2016-01-09 17:44:29
2016-01-09 17:43:39
2016-01-09 17:43:03
Id SHA1 :
e82a7786b5ebfc1d0e62eb00ae2101a328d3f0e4
Colonne 1 / 3
Recherche
commit
contient :
Tous les champs
Rechercher
Patch
Arbre
Diff
Ancienne version
Nouvelle version
Commentaires
foo
Auteur: David Parsons <david.parsons@inria.fr>
Auteur du commit: David Parsons <david.parsons
Parent: f6813a3b9f081bb486975dfe5b6a426effabf6
Branche: master
Suit:
Précède:
Log rev3
foo
index e69de29..5257914 100644

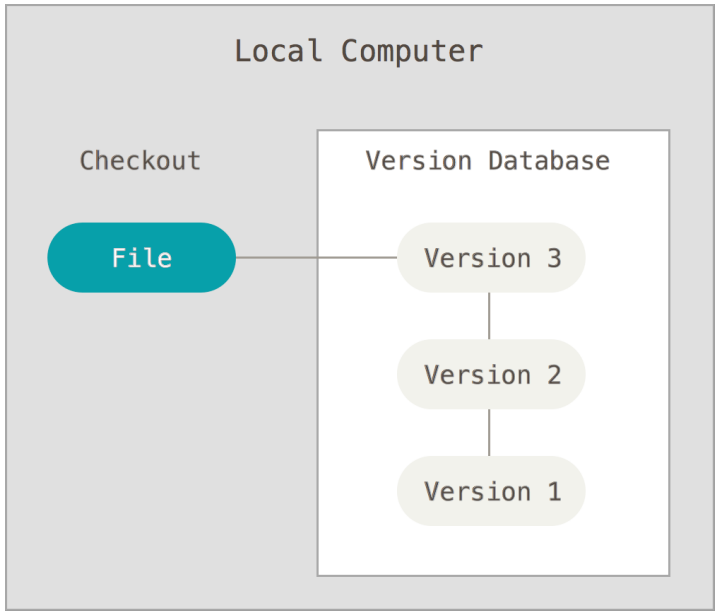
git-gui


Git Gui (git-sandbox2) /Users/dparsons/tmp/git-sandbox2
Branche courante : master
Modifs. non indexées
foo
Modifié, pas indexé
Fichier : foo
@@ -1 +1,2 @@
this is file foo
+this is some new content
Modifs. indexées (pour commit)
Message de commit :
Nouveau commit
Corriger dernier commit
Recharger modifs.
Indexer modifs.
Signer
Commiter
Pousser

2

FROM THE WORKING COPY TO THE REMOTE REPOSOTORY

Working copy + Local repo



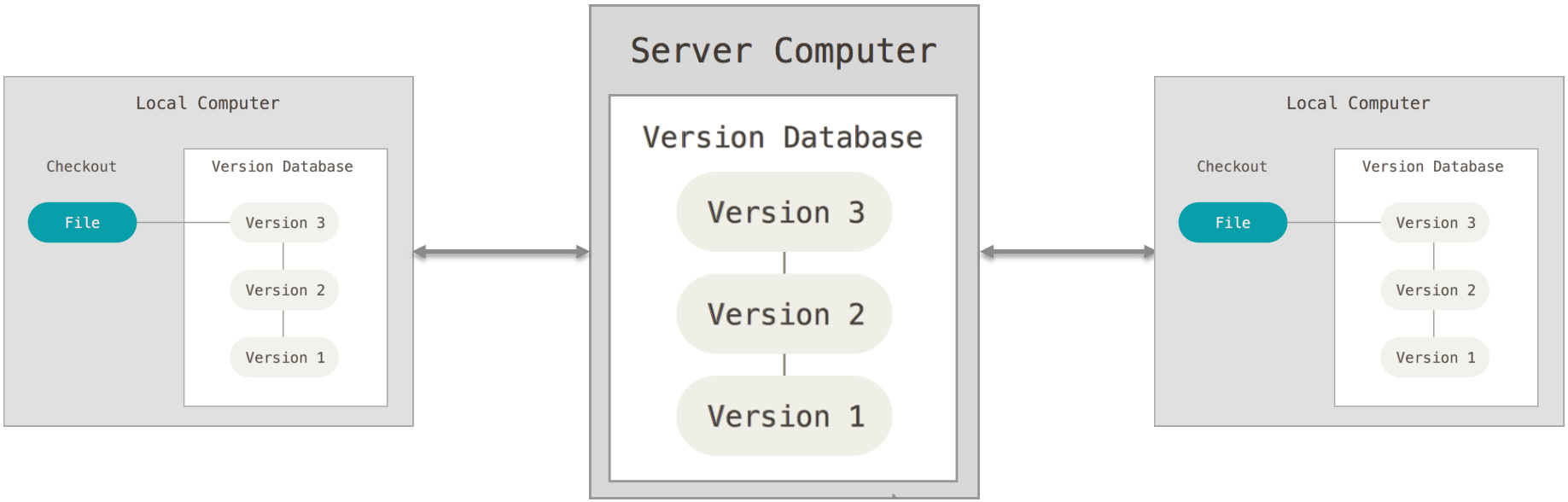
Each user has (on his own computer):

- His own local repository (stored in the `.git` repository)
- A working copy (a.k.a. checkout) of the tracked files

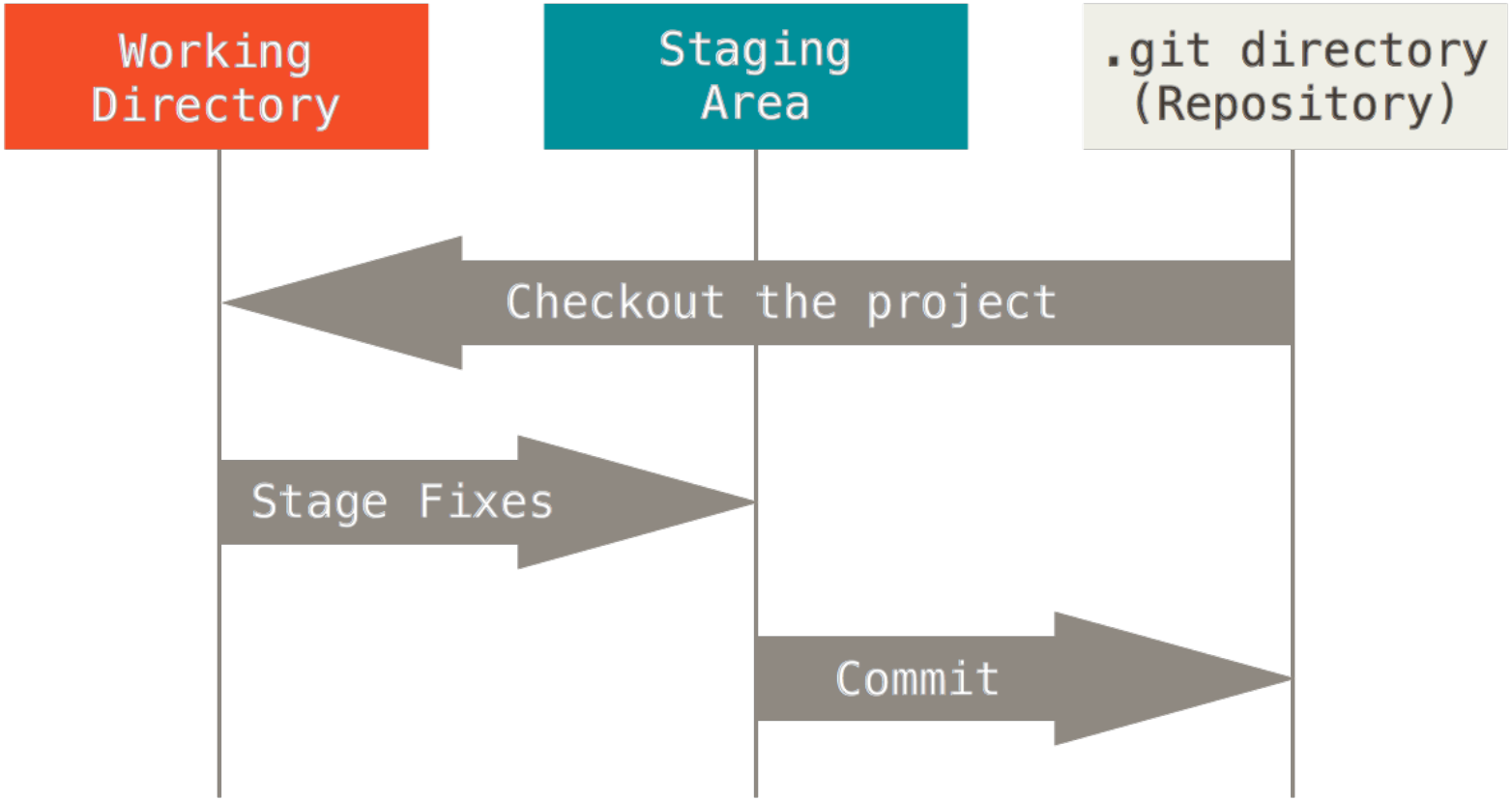
Remote repositories

Repositories can be synchronized with one another

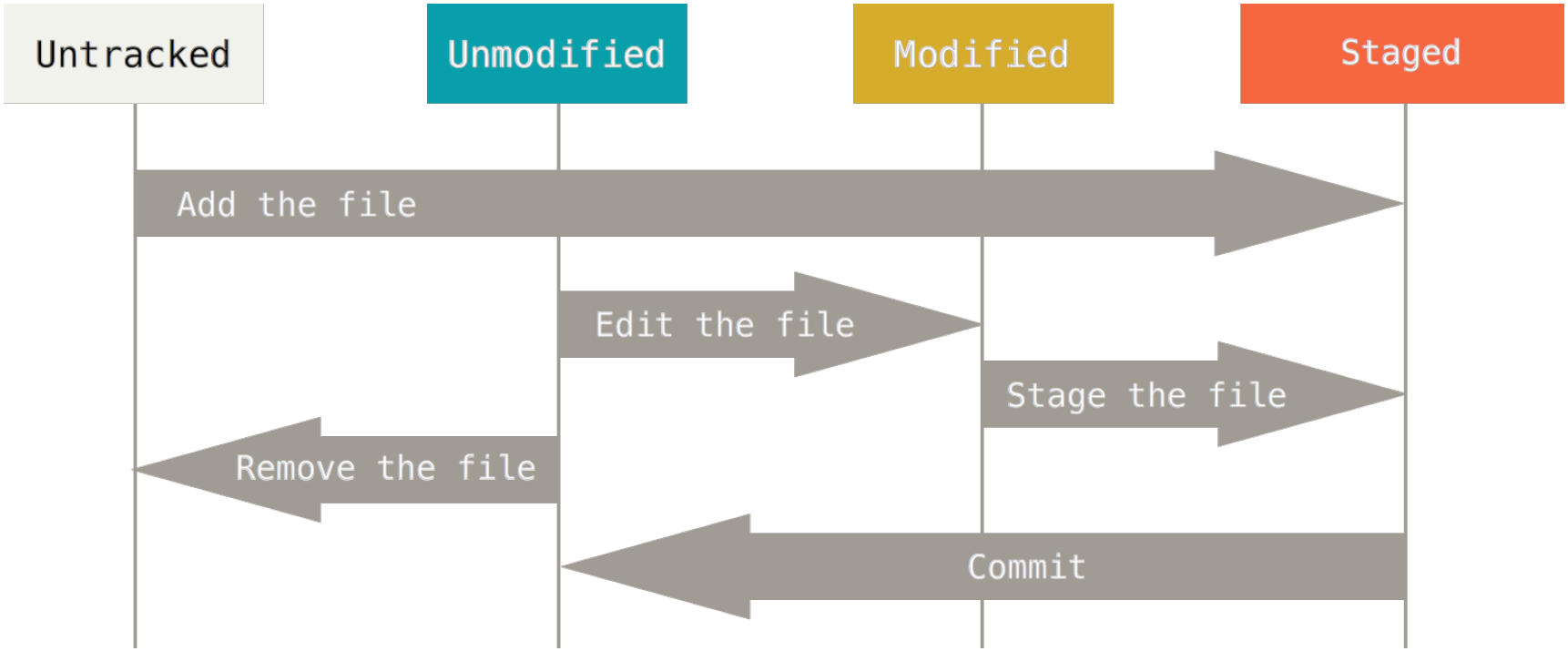
In the centralized workflow, each user synchronises his repository with a central repo (the remote) located on a server



The Staging Area (a.k.a. Index)



File Status Lifecycle



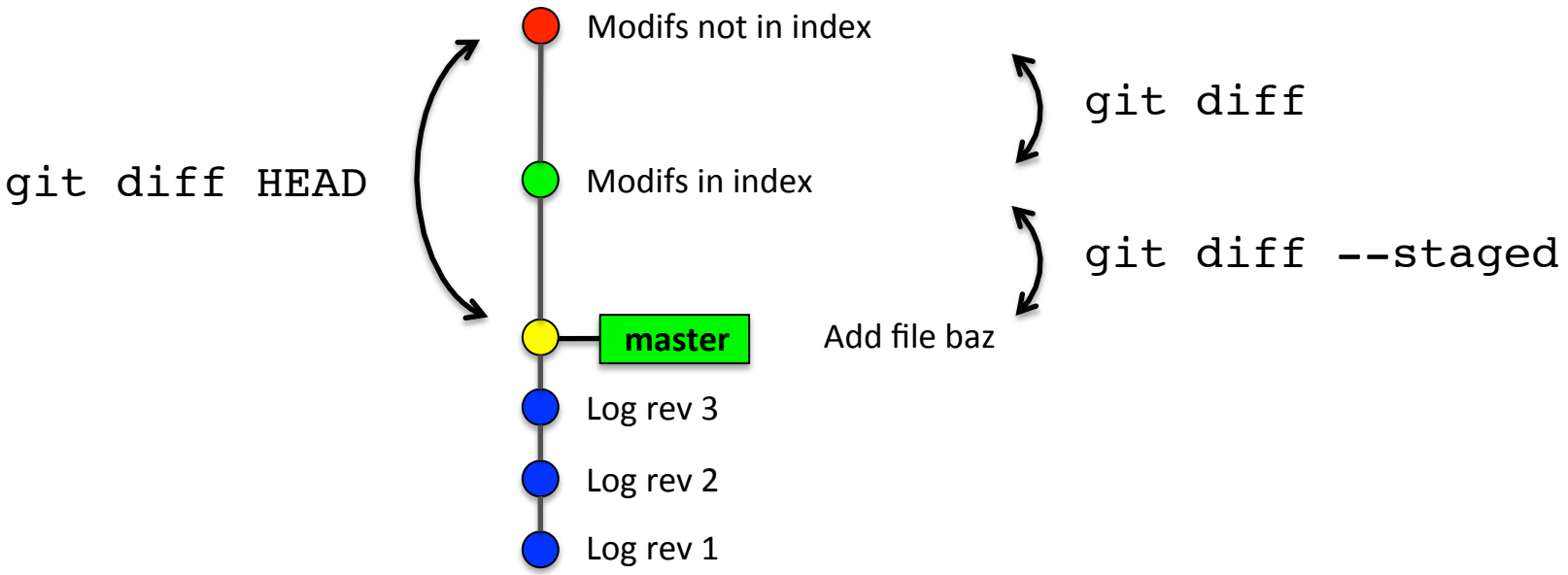
The corresponding commands…

… are quite straightforward :

- Add a file `git add`
- Edit a file
- Stage a file `git stage` (or `git add`)
- Remove a file `git rm`
- Commit a file `git commit`

NB : `git rm` actually deletes the file from your working copy. If you want to keep it as an untracked file, use the `--cached` option

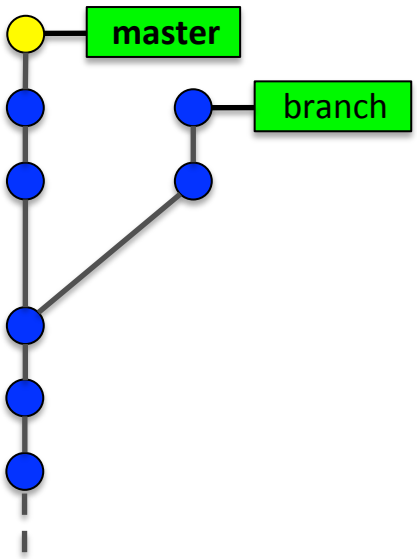
Illustrations : `diff`



3

WORKING WITH BRANCHES

What is a branch ?

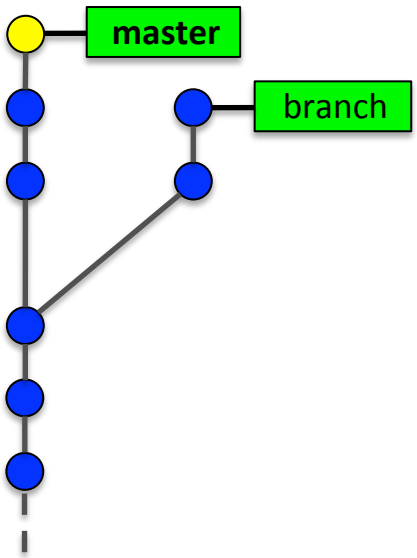


Each branch contains an alternative version of the same set of files

Git is meant for an extensive use of branches

« master » is the default name for the main branch, i.e. that which is checked out by default

Branch types



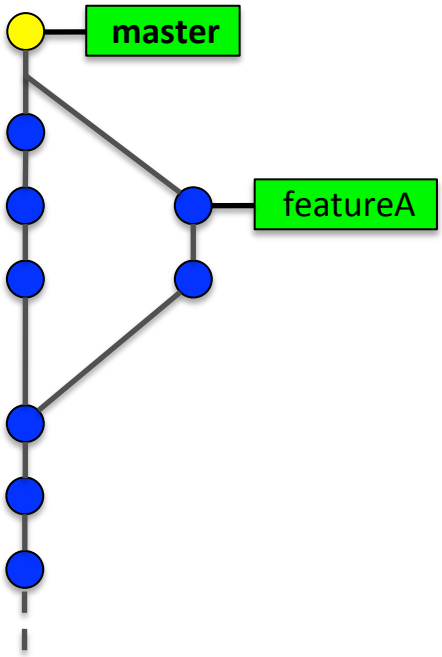
Git doesn't know about branch types, it is the usage you make of a branch that determines its "type"

A branch may remain local to a user's repo or be published (pushed) to the central repo

Common use cases for branches include:

- Feature
- Bugfix
- …

Merging branches



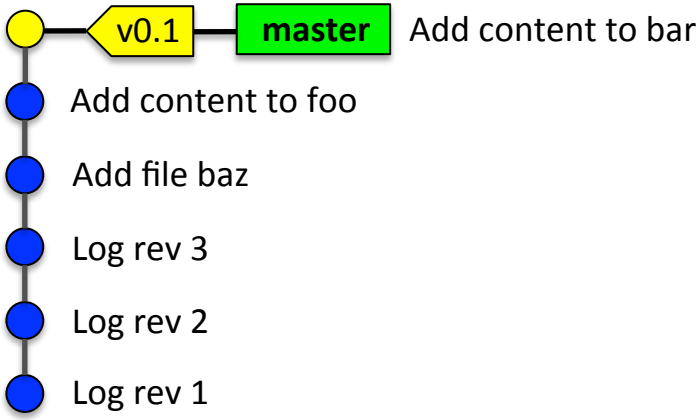
Here we have merged the branch featureA into the branch master. This means we have injected the constituent modification of featureA into master

This merge is embodied by a commit (unless it is a fast-forward) that has two parents and that include the changes made to both branches (here 5 sets of changes)

The branch featureA is untouched

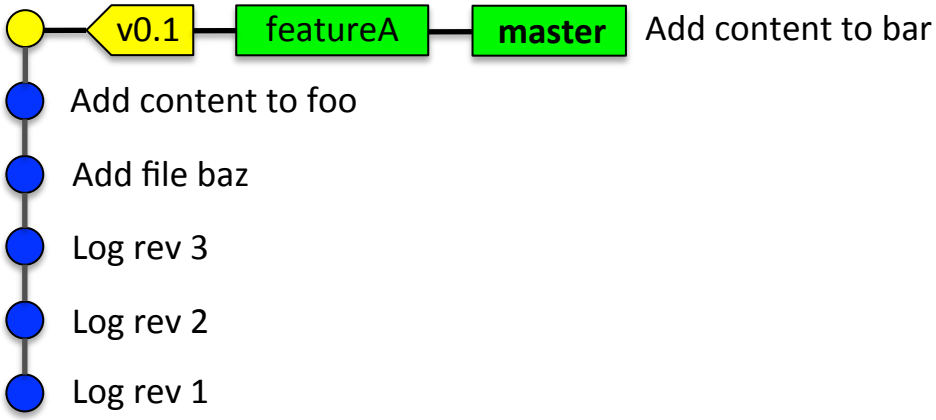
Branches

```
$ git branch
* master
$ git branch featureA
```



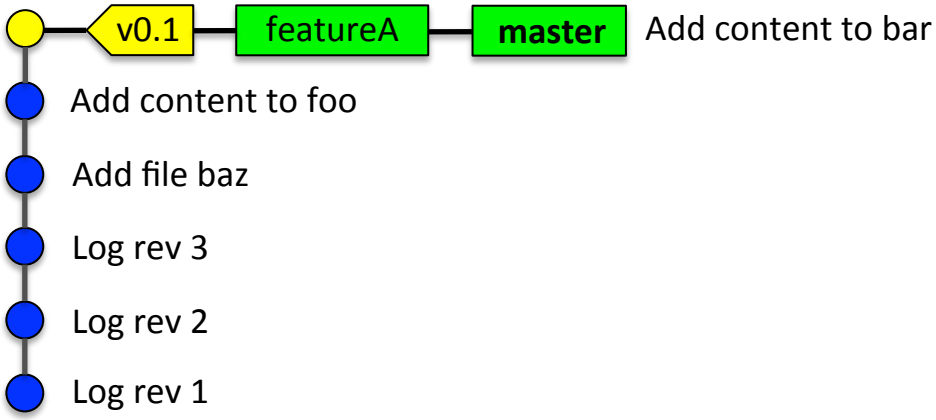
Branches

```
$ git branch
* master
$ git branch featureA
$
```



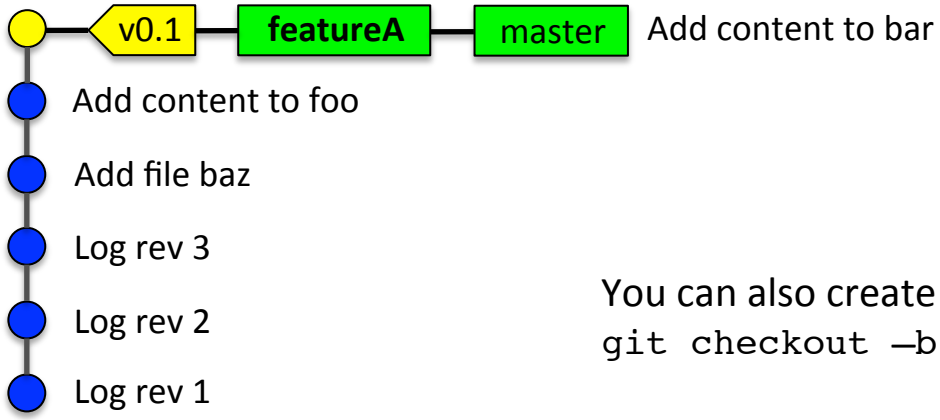
Branches

```
$ git branch
* master
$ git branch featureA
$ git branch
  featureA
* master
$
```



Branches

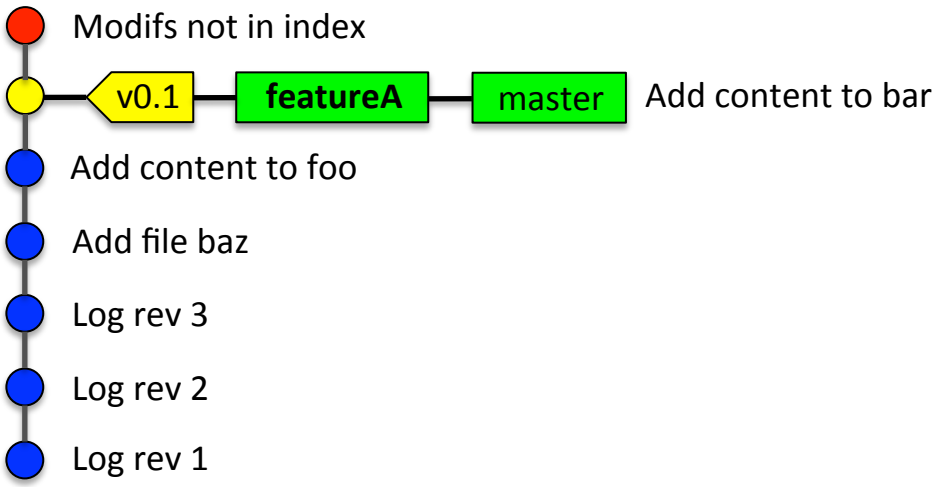
```
$ git branch
* master
$ git branch featureA
$ git branch
  featureA
* master
$ git checkout featureA
$ git branch
* featureA
  master
$
```



You can also create and checkout a new branch at once with
`git checkout –b newBranch`

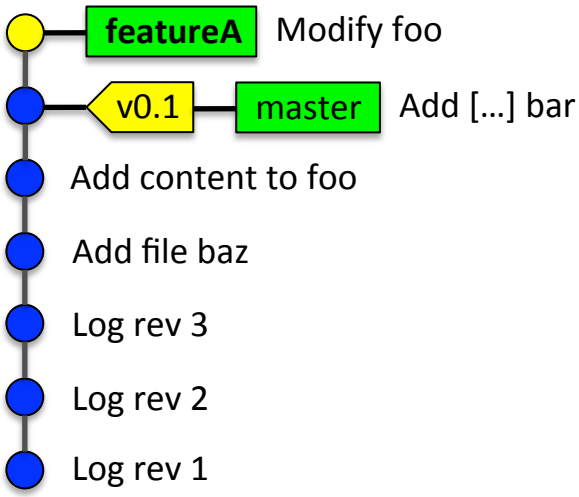
Branches

```
$ # Commit stuff in branch featureA
$ echo “Blabla” >> foo
$ git ci foo -m “Modify foo”
```



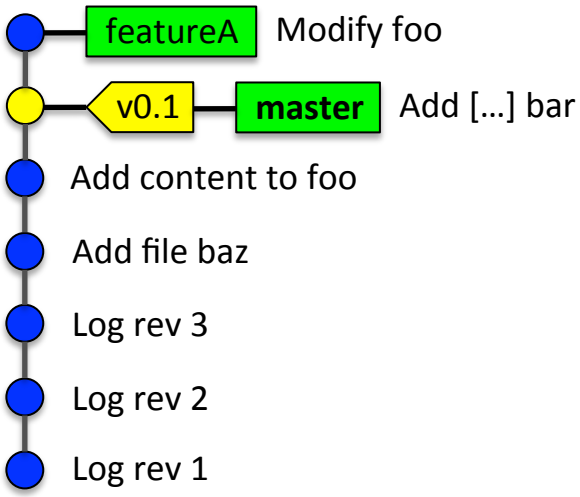
Branches

```
$ # Commit stuff in branch featureA
$ echo “Blabla” >> foo
$ git ci foo -m “Modify foo”
$
```



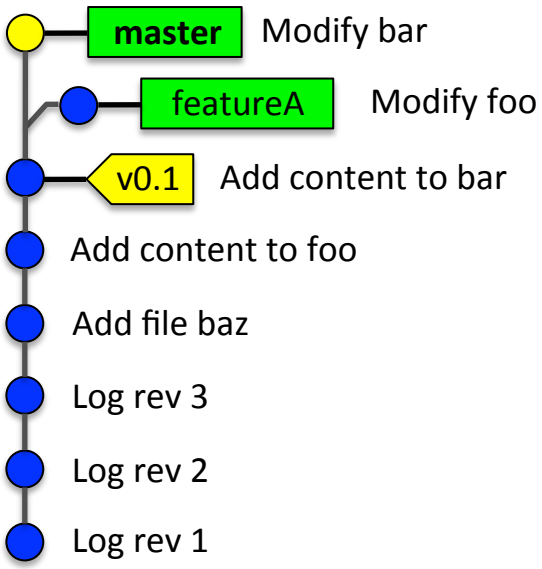
Branches

```
$ # Checkout branch master and commit stuff in it
$ git co master
$
```



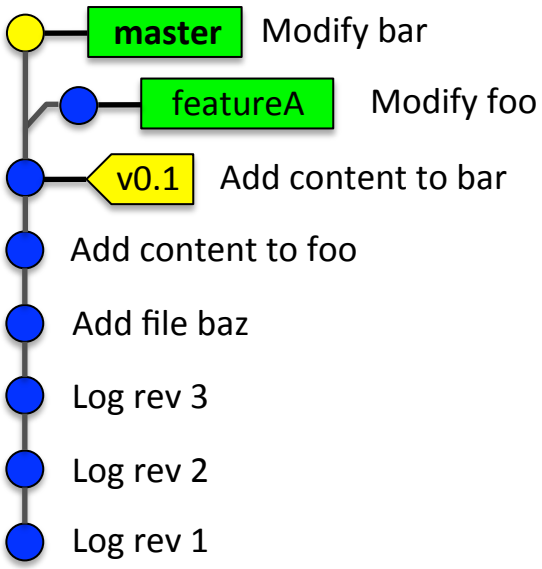
Branches

```
$ # Checkout branch master and commit stuff in it
$ git co master
$ echo “Blabla” >> bar
$ git ci foo -m “Modify bar”
$
```



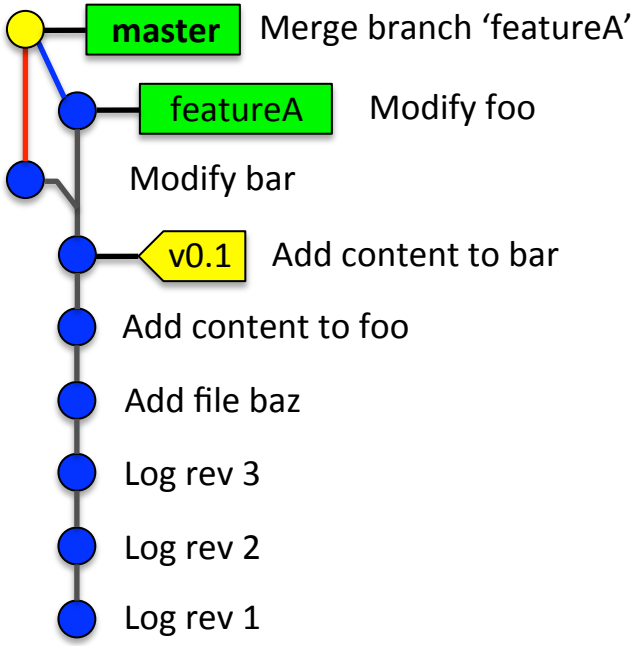
Merge without conflicts

```
$ git merge featureA
```



Merge without conflicts

```
$ git merge featureA
$
```



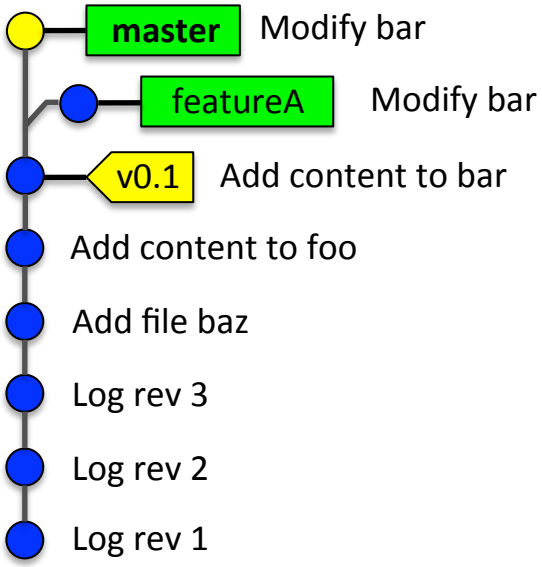
When there are no conflicts whatsoever, git merge automatically triggers the creation of the merge commit. It will open your core editor with a predefined log msg

4

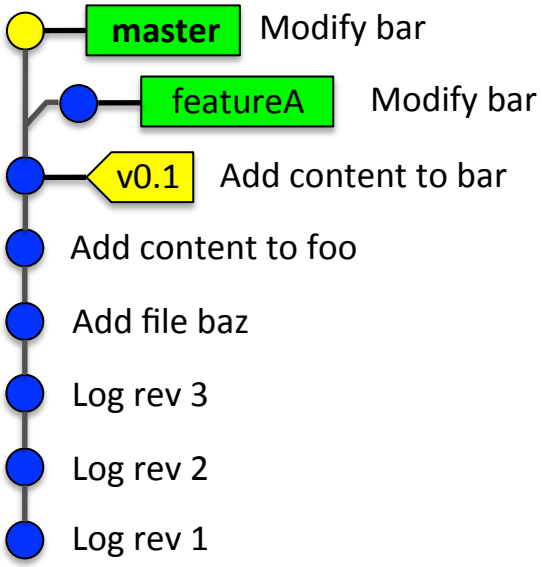
MANAGING CONFLICTS

Managing conflicts

```
$ git merge featureA
```

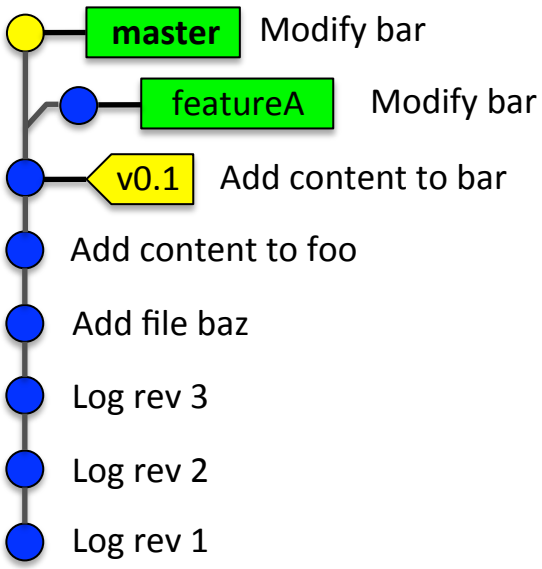


Managing conflicts




```
$ git merge featureA
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then
commit the result.
$
```

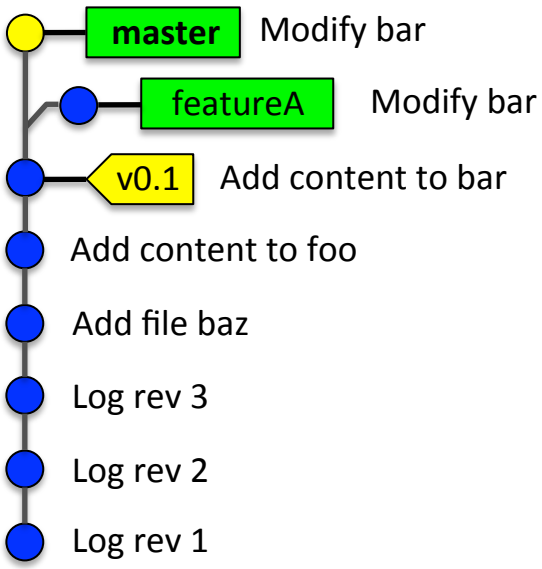
Managing conflicts



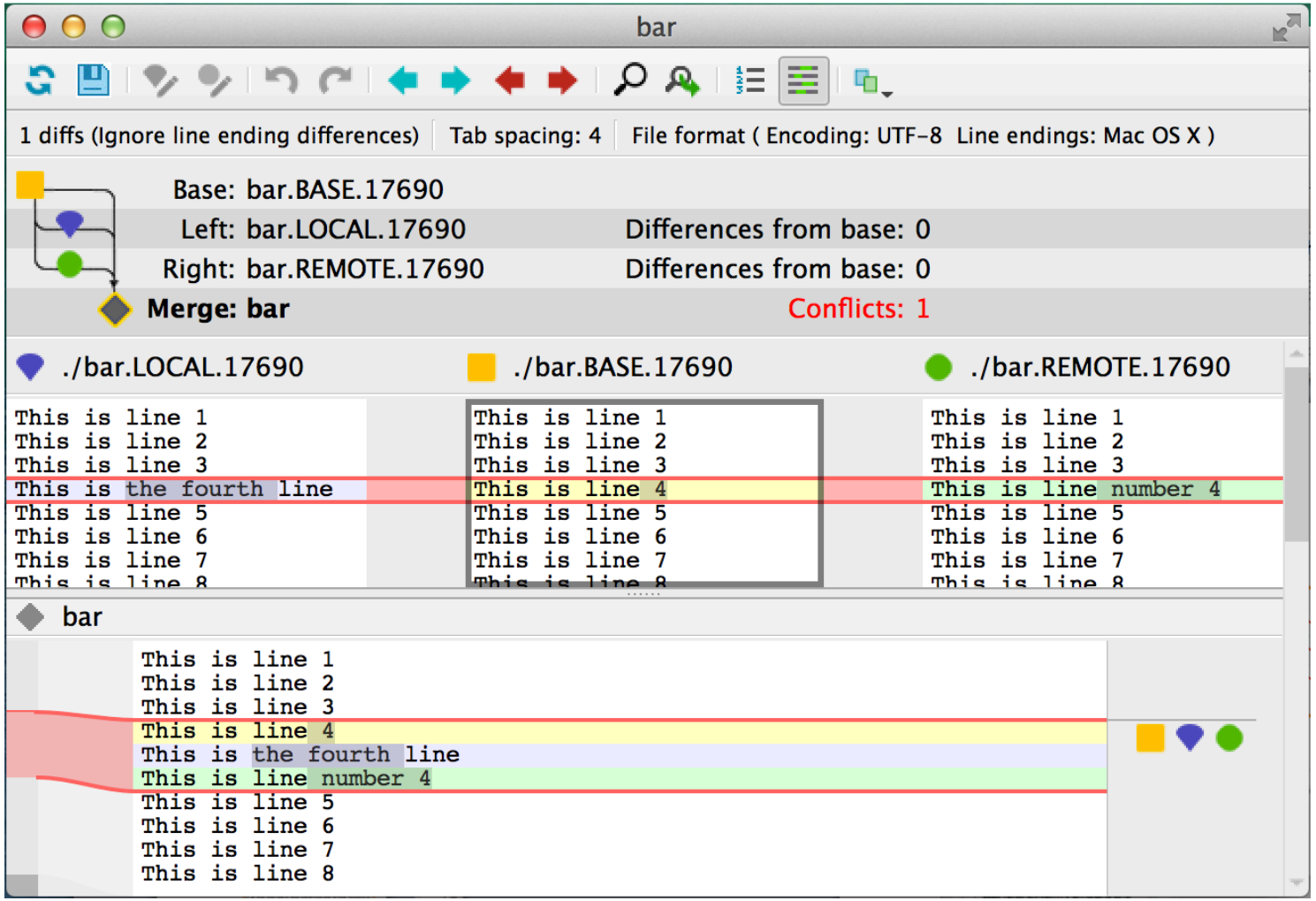

```
$ git merge featureA
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then
commit the result.
$
$ cat bar
This is line 1
This is line 2
This is line 3
<<<<<<< HEAD
This is the fourth line
=======
This is line number 4
>>>>>>> featureA
This is line 5
This is line 6
This is line 7
This is line 8
$
```

Managing conflicts

```
$ # You can edit the conflicting files directly
and then stage and commit them, or you can use
a merge tool
$
$ git mergetool
```

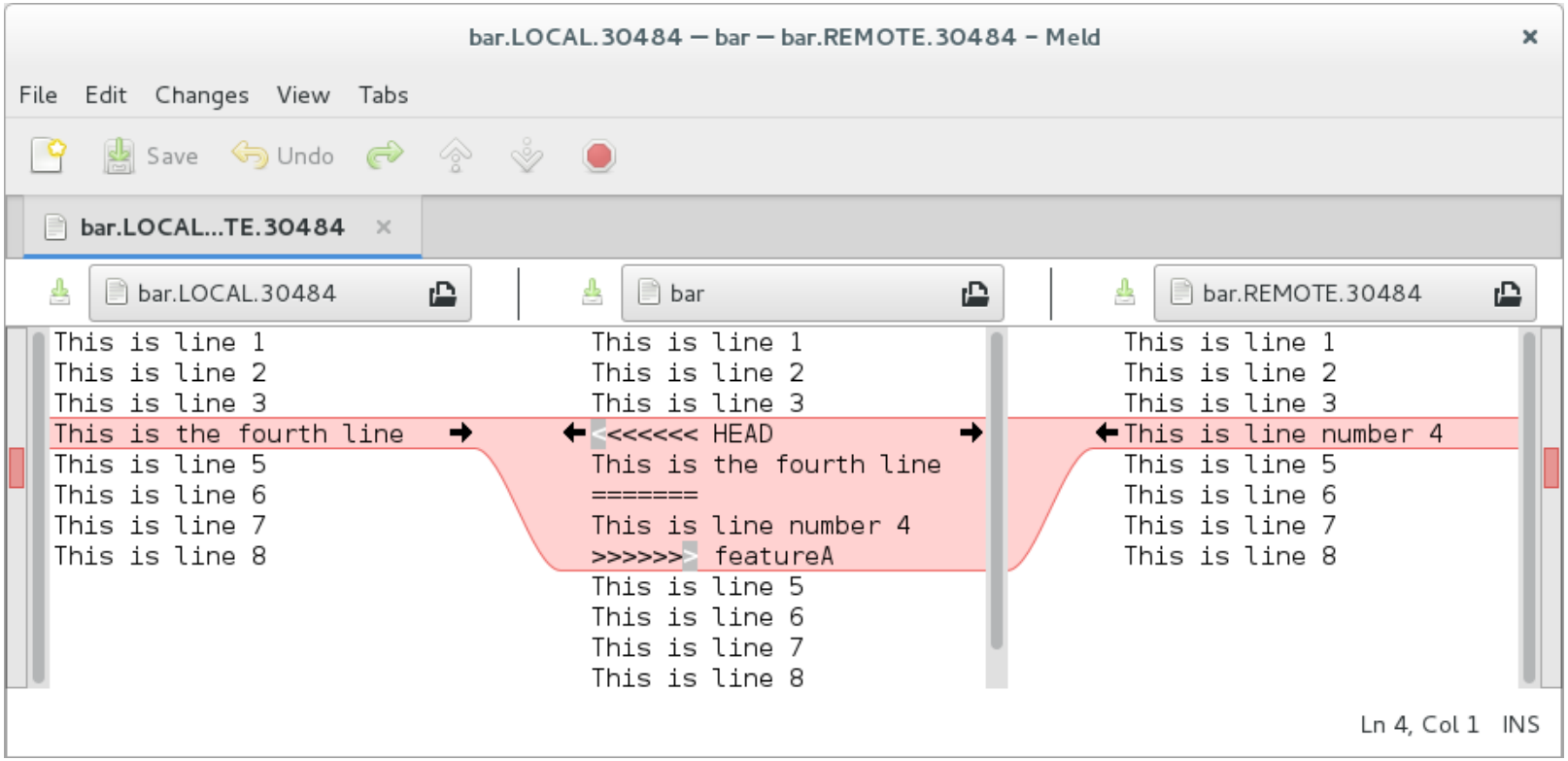


mergetool


bar
1 diffs (Ignore line ending differences)
Tab spacing: 4
File format (Encoding: UTF-8 Line endings: Mac OS X)
Base: bar.BASE.17690
Left: bar.LOCAL.17690
Differences from base: 0
Right: bar.REMOTE.17690
Differences from base: 0
Merge: bar
Conflicts: 1
./bar.LOCAL.17690
./bar.BASE.17690
./bar.REMOTE.17690
This is line 1
This is line 2
This is line 3
This is the fourth line
This is line 4
This is line number 4
This is line 5
This is line 6
This is line 7
This is line 8
bar

Configuring a mergetool

```
$ git config --global merge.tool meld
$ git config --global mergetool.meld.cmd 'meld $LOCAL $MERGED $REMOTE'
$ git config --global mergetool.meld.trustExitCode false
$
```



5
OTHER THINGS TO KNOW

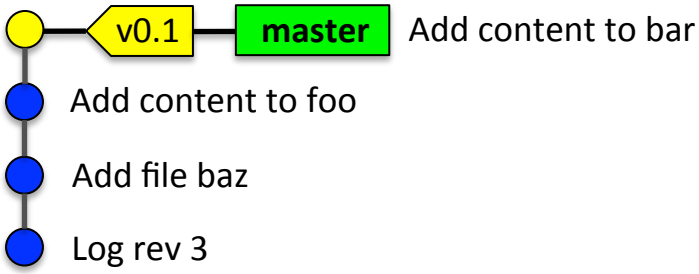
Tags

A tag is typically used to mark a release point.

There are 2 types of tags:

- Lightweight (just a pointer to a specific commit)
- Annotated (full object, contains additional info, can be signed)

```
$ # Create an annotated tag named "v0.1"
$ git tag -a v0.1 -m “version 0.1”

$ # List tags
$ git tag
v0.1
$
```



Tags

Tags have to be pushed and fetched manually:

```
git push origin <tagname>
```
or
```
git push origin --tags
git fetch origin --tags
```

Detached head ?

You are in a detached head state when you are not “on” any branch

Most of the time, this happens when you checkout anything that is not a branch (e.g. a tag). You can also use `git checkout –detach`

When in detached head, you can commit as you like ; but be wary of the garbage collector, it could very well erase your work if you don’t pay attention !

But git is kind, it tells you that and what to do …

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name
```

Stash

Literally “garder sous la coude”

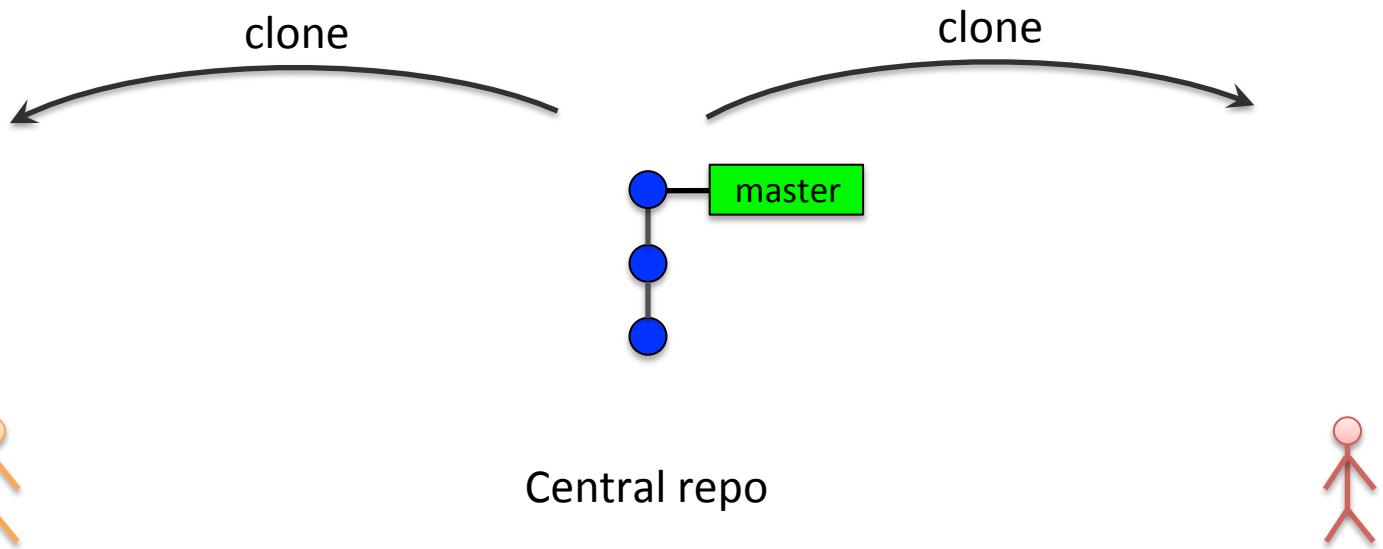
```
# You’re working on something and you need to switch to something else
$ git co something_else
error: Your local changes to the following files would be overwritten
by checkout
$ git stash [save]
Saved working directory and index state WIP on master: […]
$ git stash list
stash@{0}: WIP on master: […]
$ git co something_else # Now we can (WD is clean) !
# Do things on something_else
$ git co master # Back to master
$ git stash pop # Back to initial state
```

6
INTERACTING WITH REMOTES

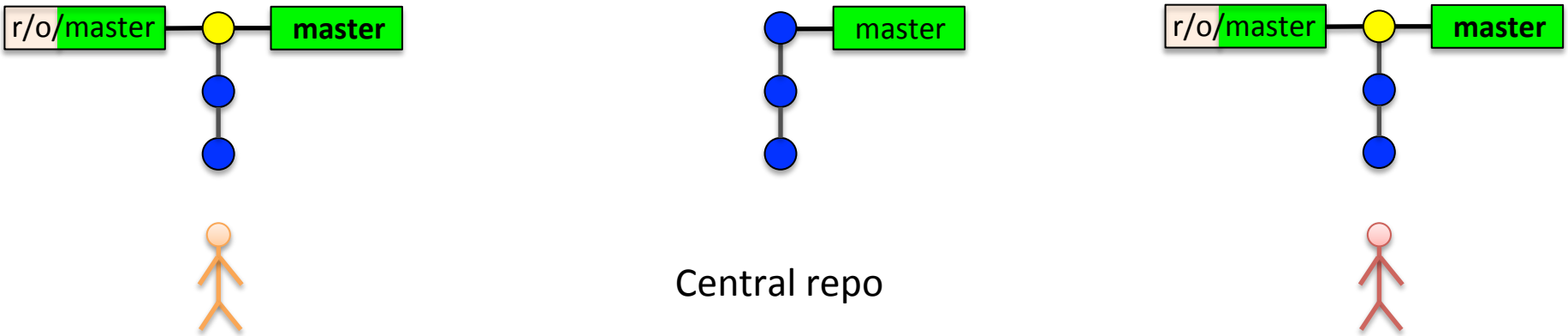
Working in parallel

Scenario

Working in parallel

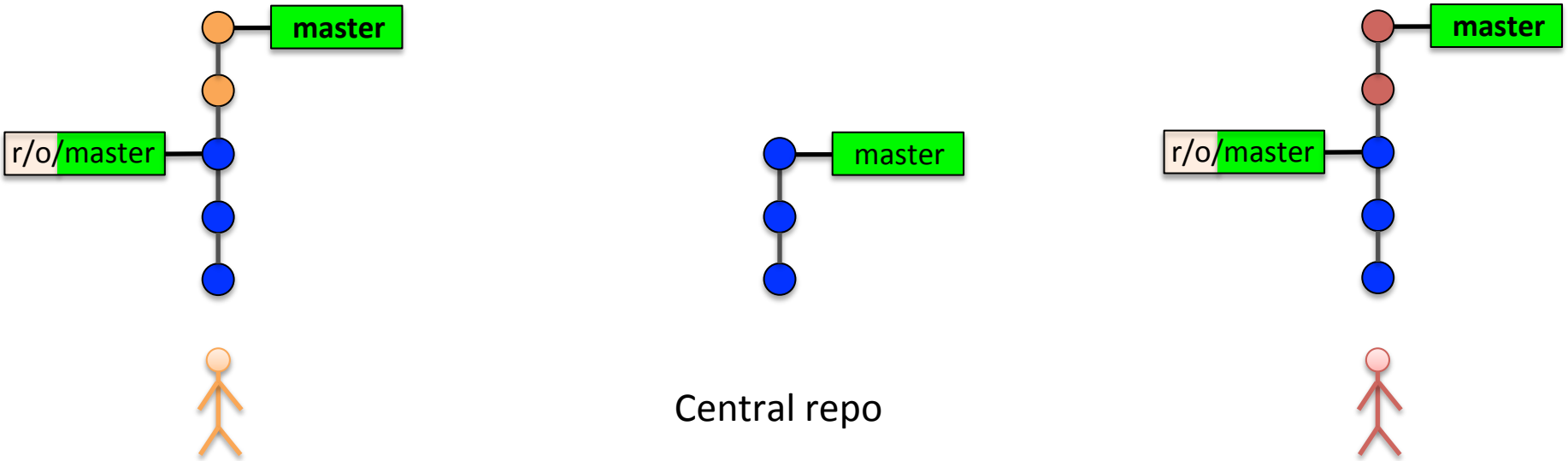


Working in parallel



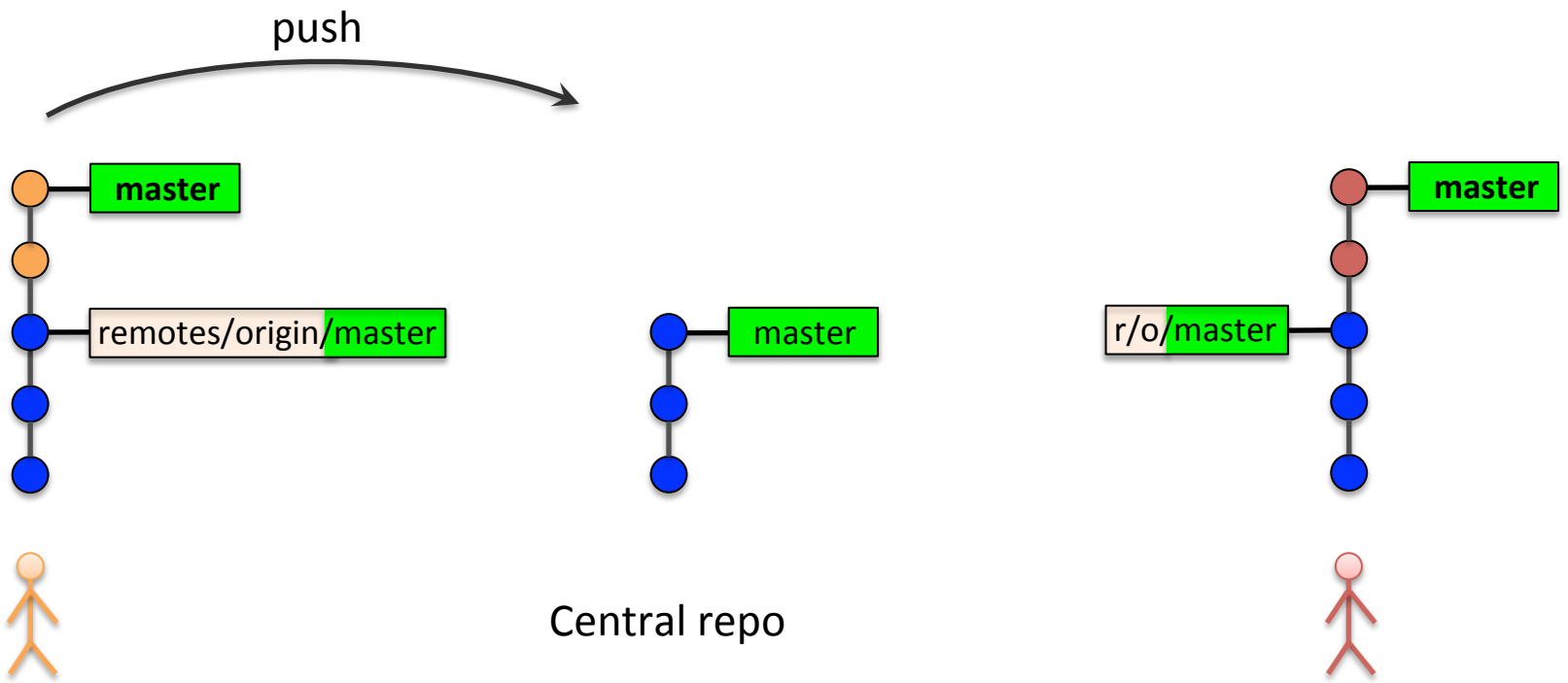
Central repo

Working in parallel

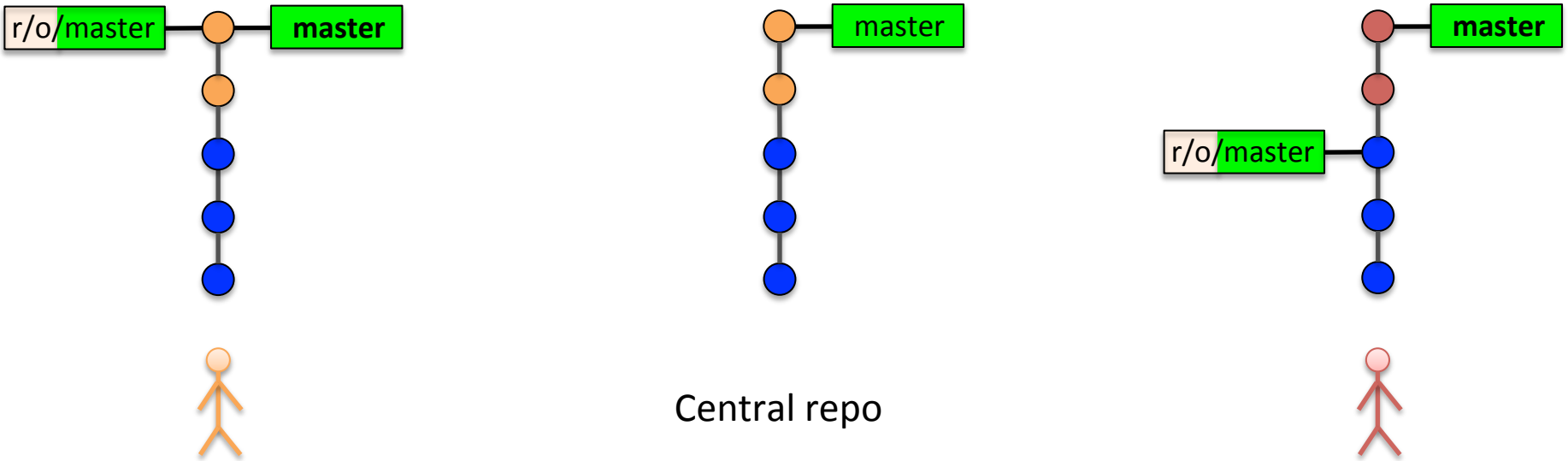


Central repo

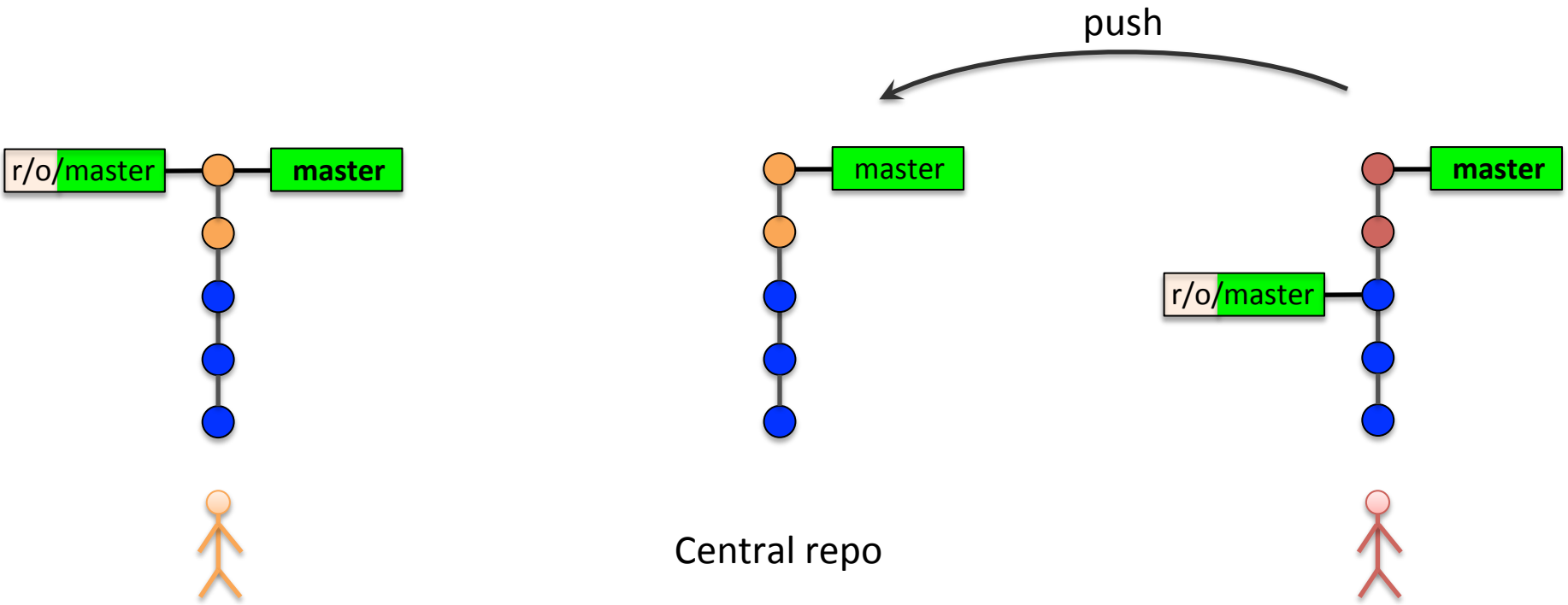
Working in parallel



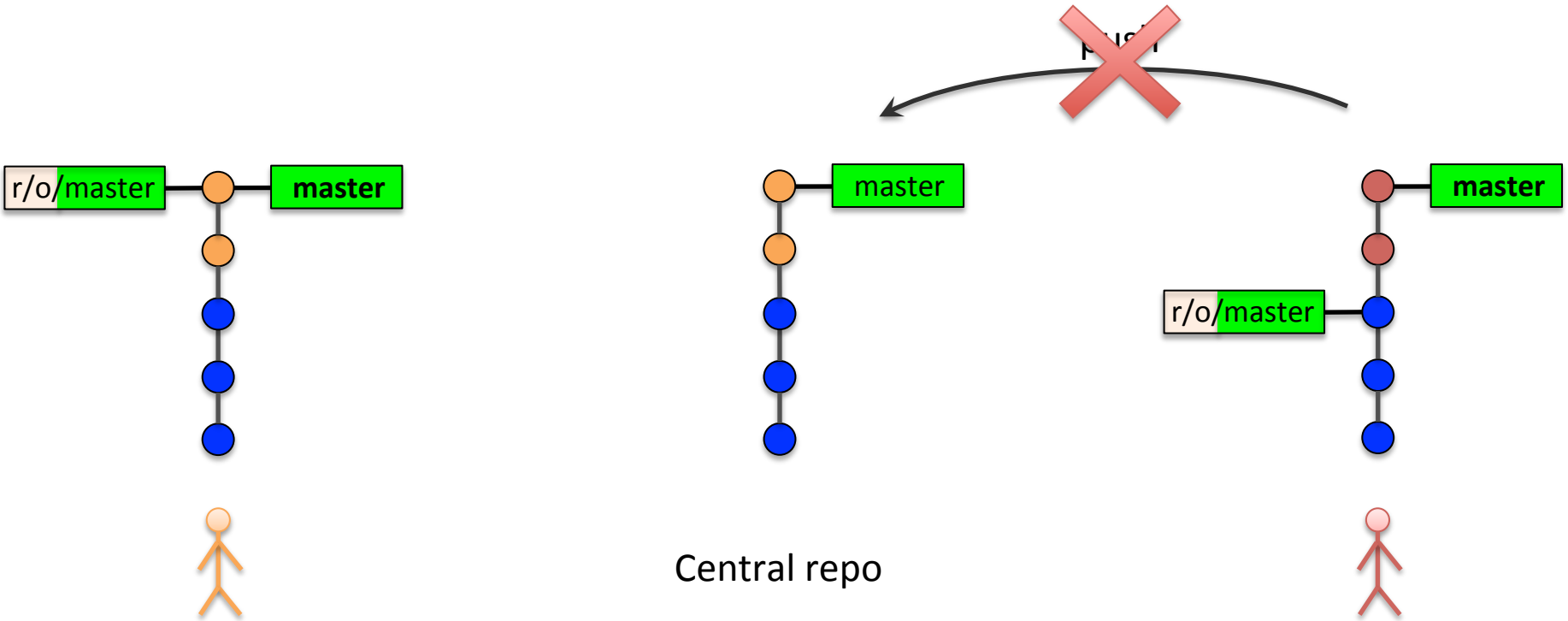
Working in parallel



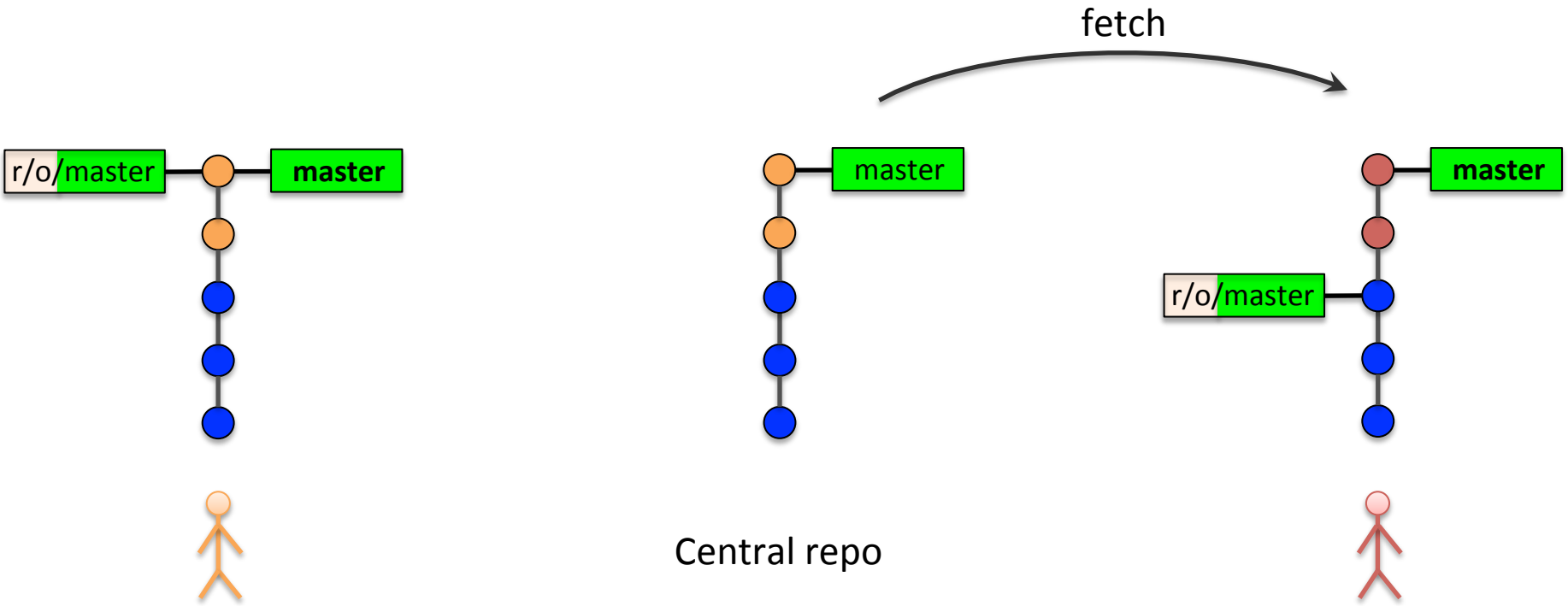
Working in parallel



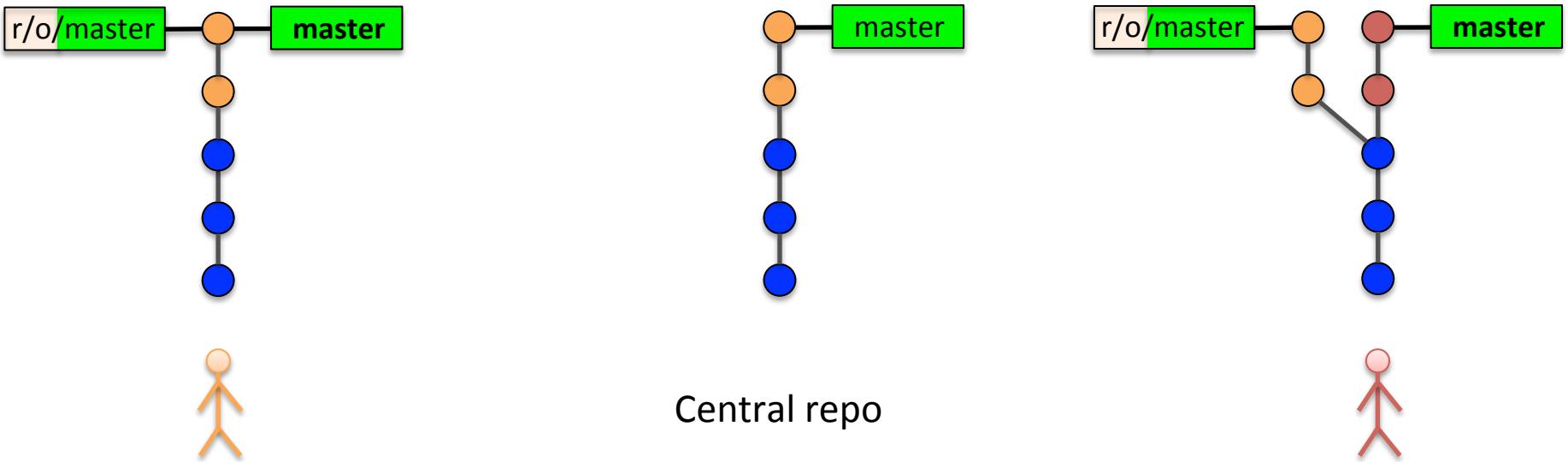
Working in parallel



Working in parallel

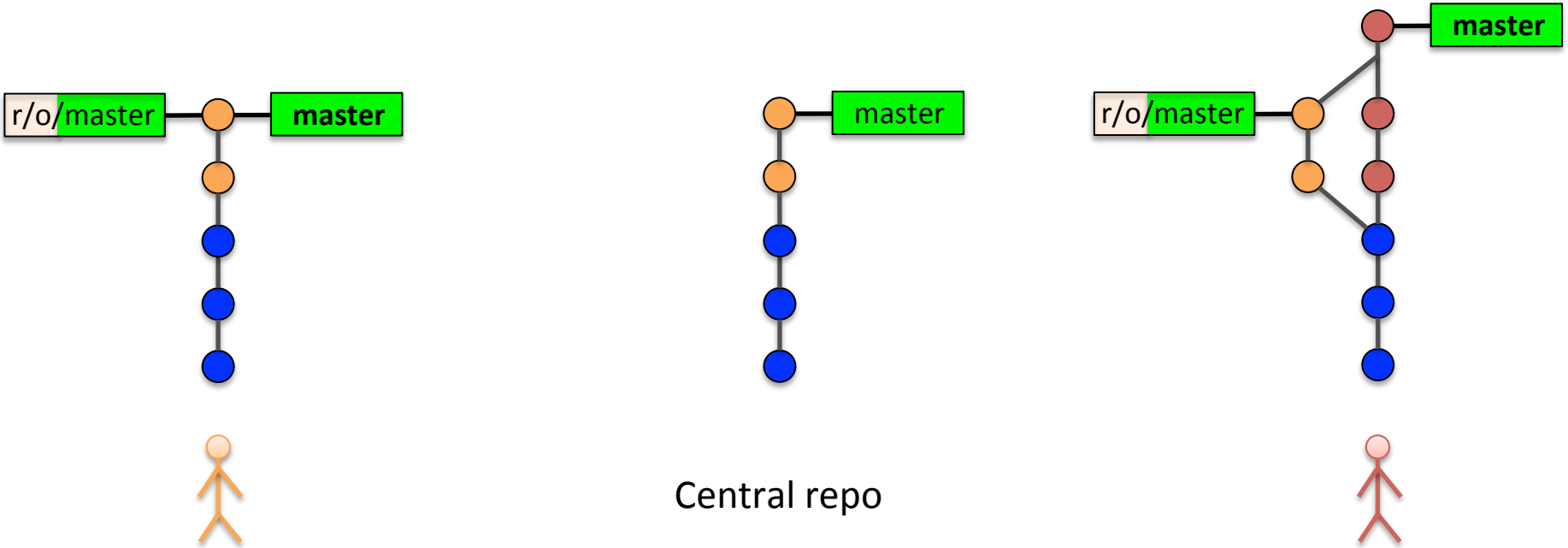


Working in parallel



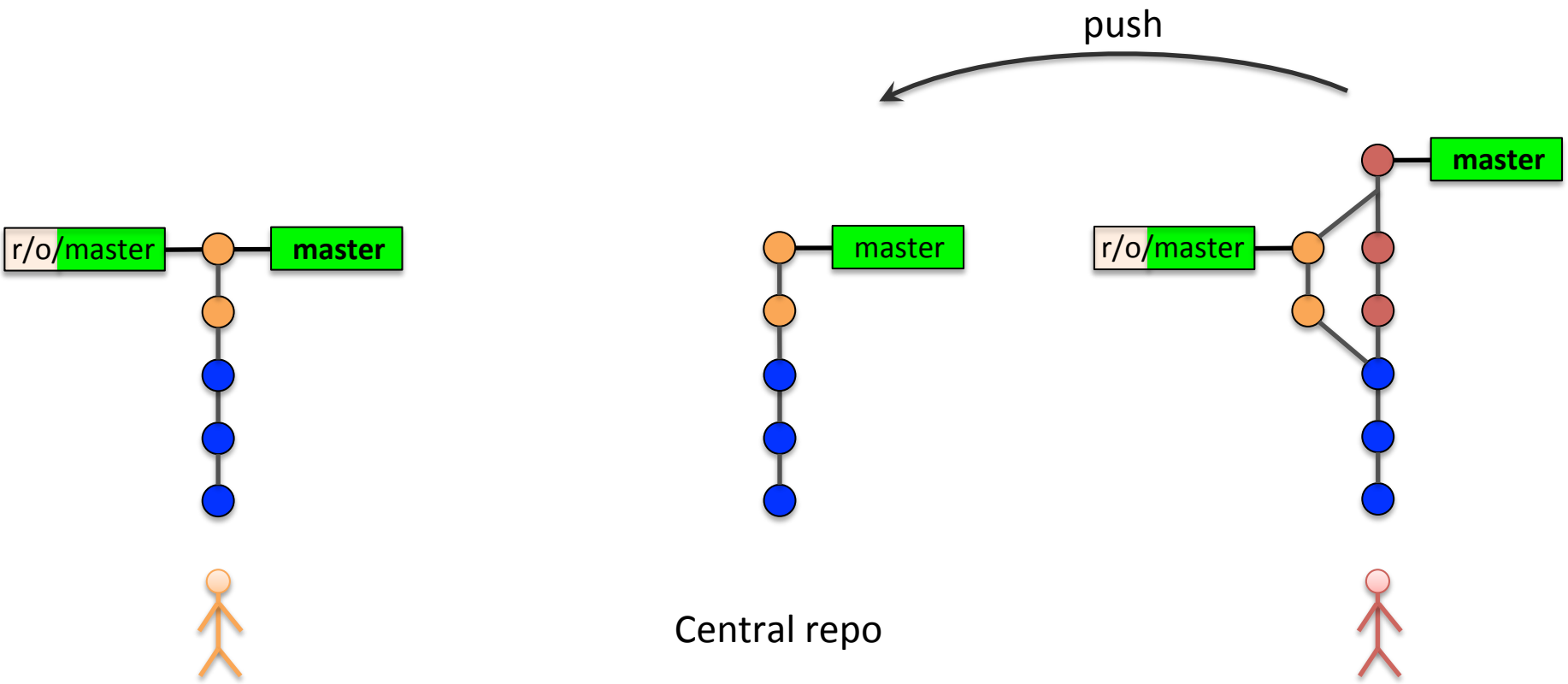
Central repo

Working in parallel

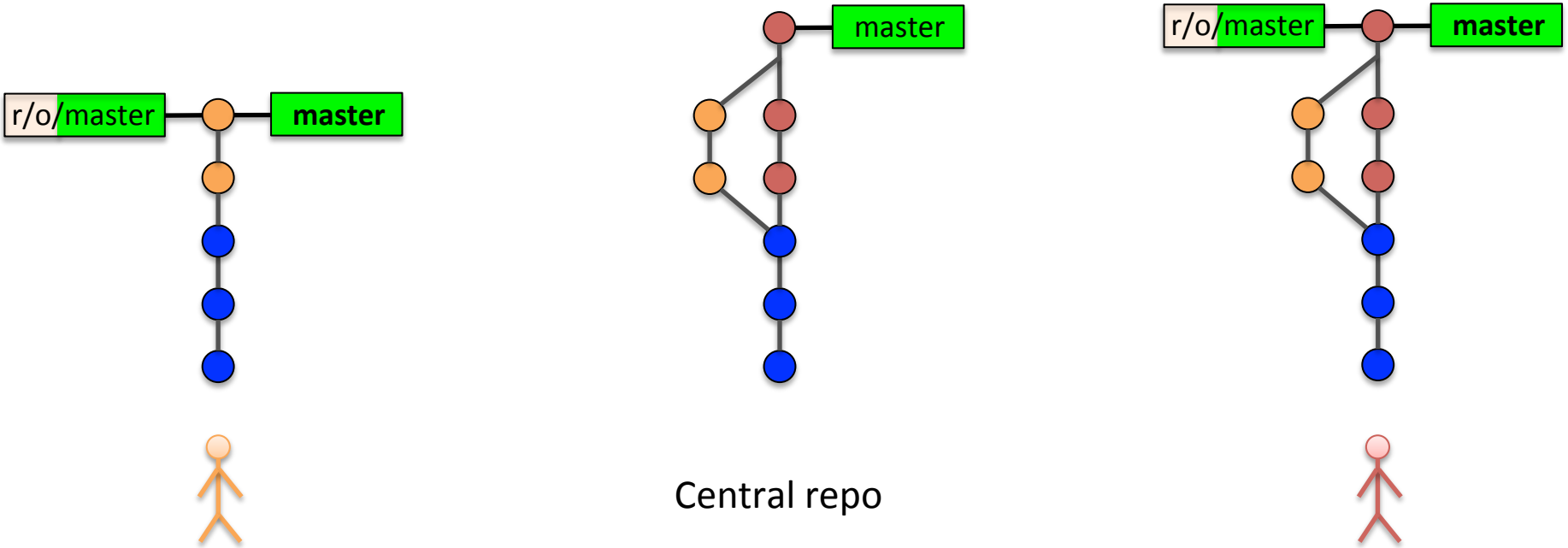


Central repo

Working in parallel

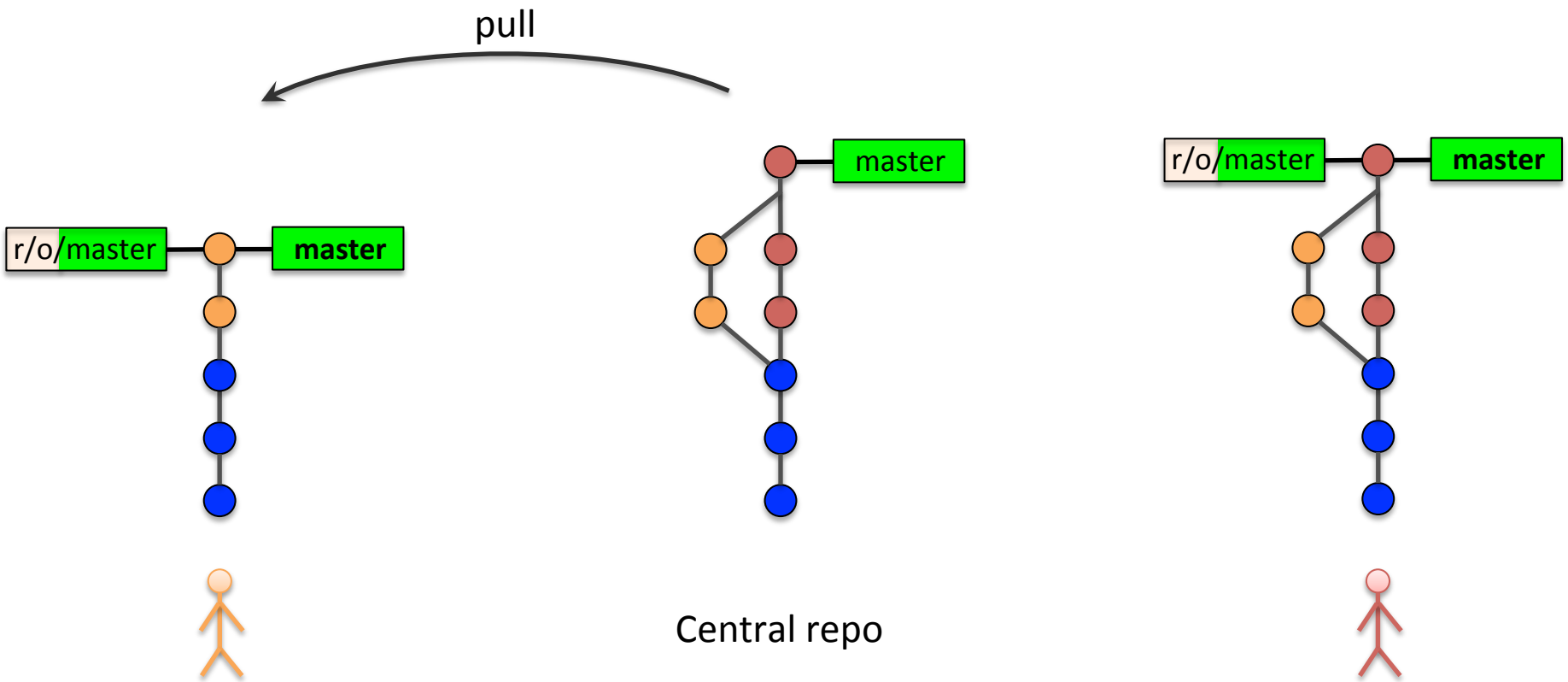


Working in parallel

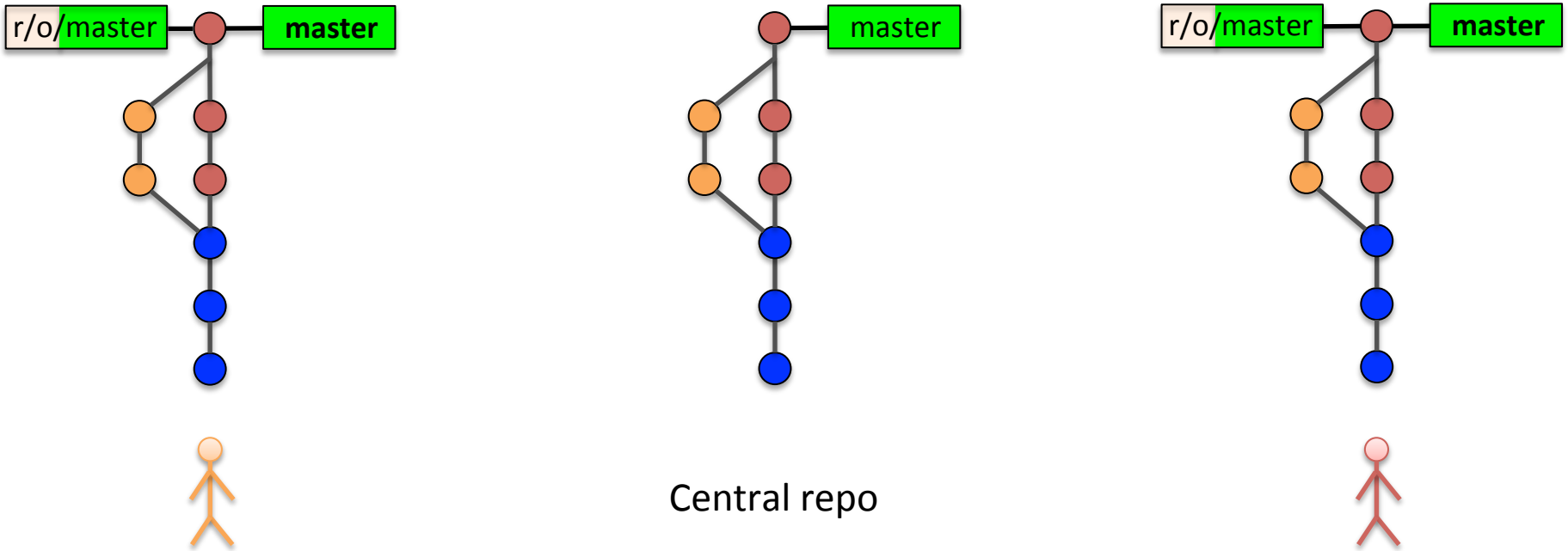


Central repo

Working in parallel



Working in parallel

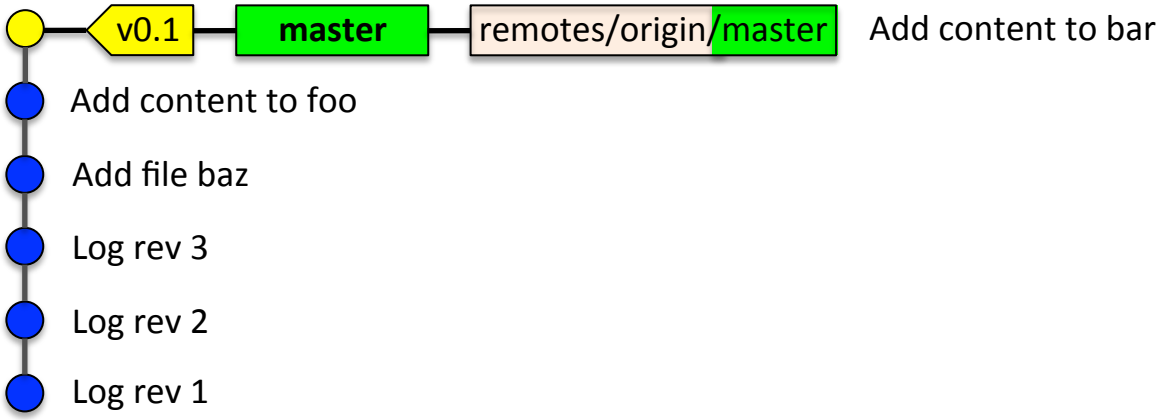


Central repo

Details of the corresponding Commands

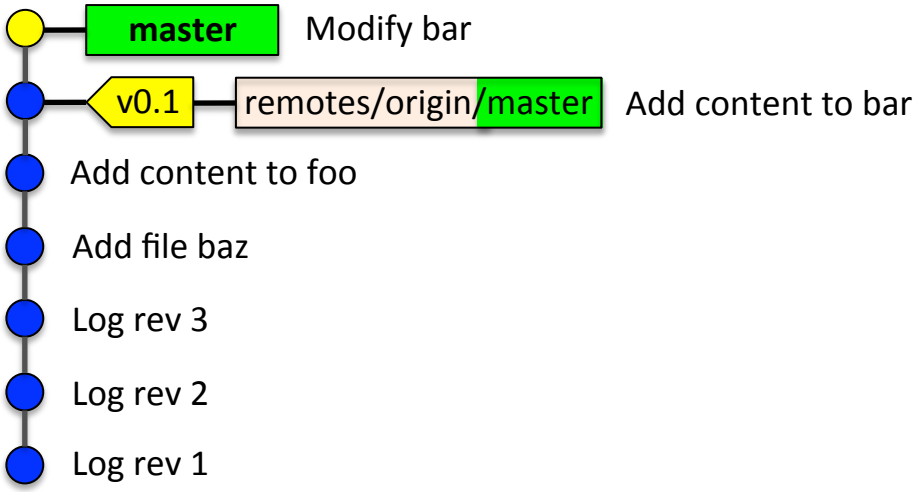
Remote Branches

```
$ # Commit stuff
$ echo "This is the content of bar" > bar
$ git ci bar -m "Modify bar”
```



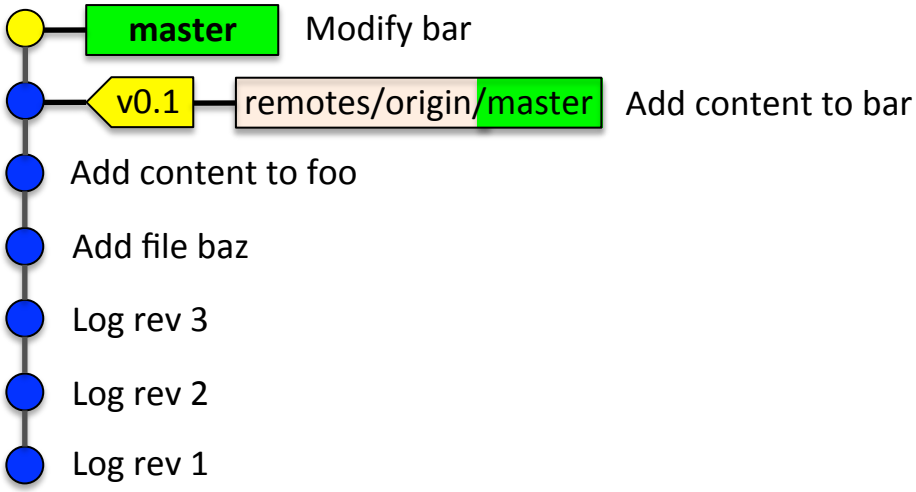
Remote Branches

```
$ # Commit stuff
$ echo "This is the content of bar" > bar
$ git ci bar -m "Modify bar"
$
```



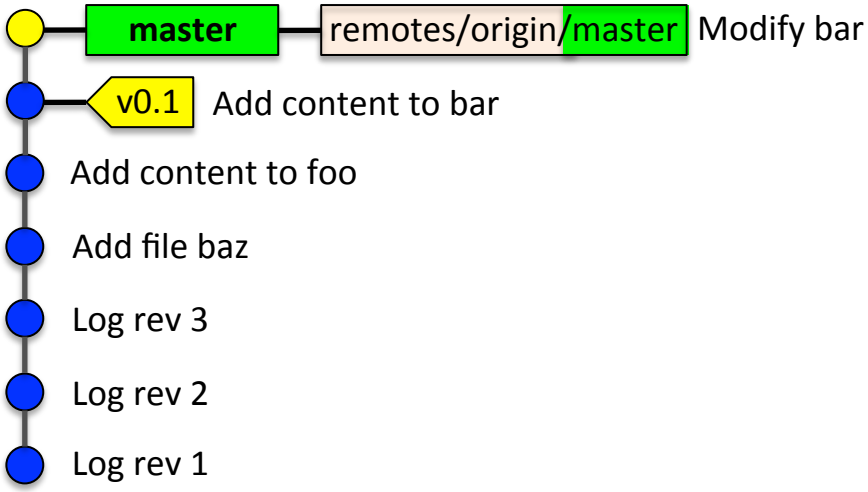
Push

```
$ # Push your commits onto the remote repository
$ git push
```



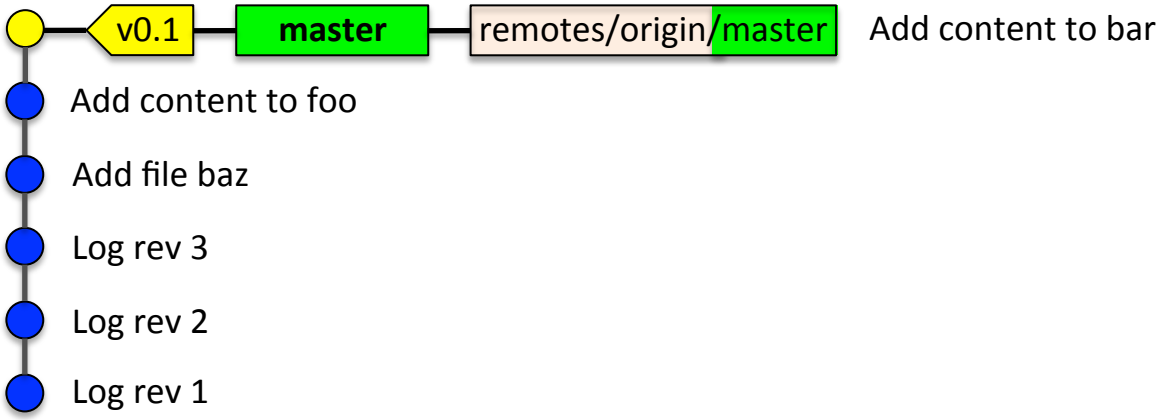
Push

```
$ # Push your commits onto the remote repository
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /Users/dparsons/tmp/git-sandbox2/../git-sandbox
   119dfba..a8f9783  master -> master
$
```



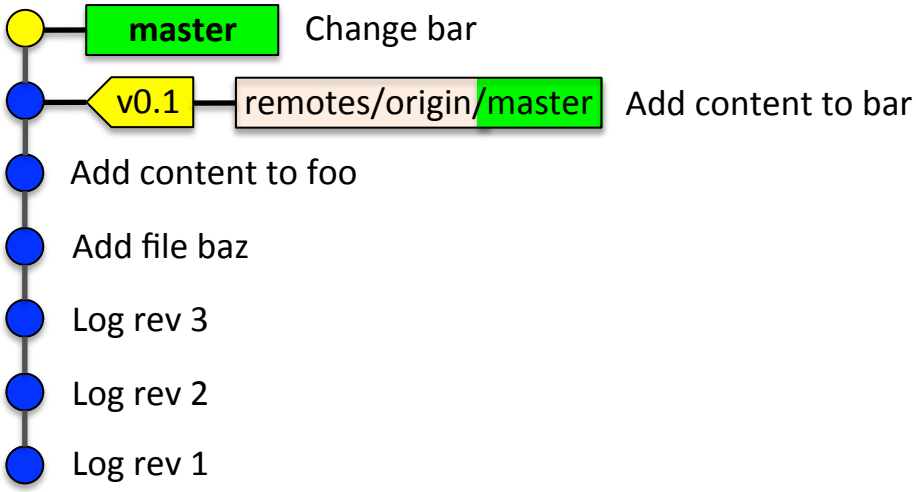
Other user's viewpoint

```
$ # Commit stuff
$ echo "This is bar's content" > bar
$ git ci bar -m "Change bar"
```



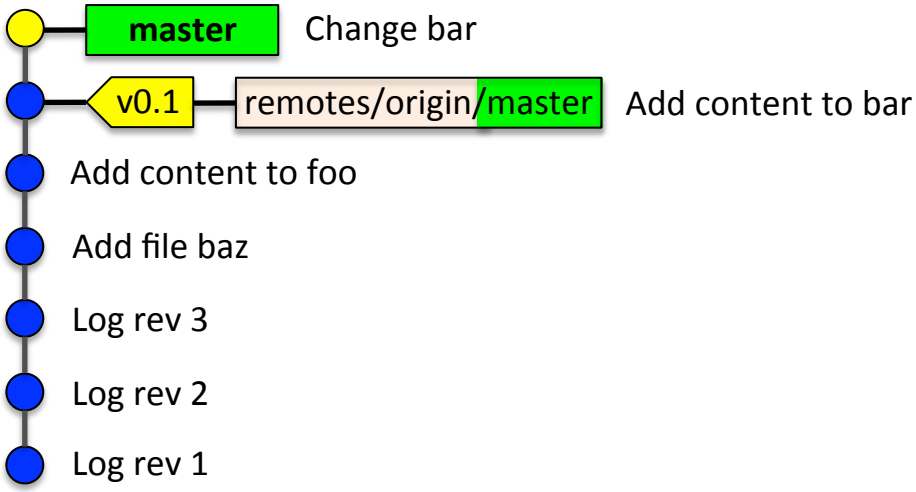
Other user's viewpoint

```
$ # Commit stuff
$ echo "This is bar's content" > bar
$ git ci bar -m "Change bar"
$
```



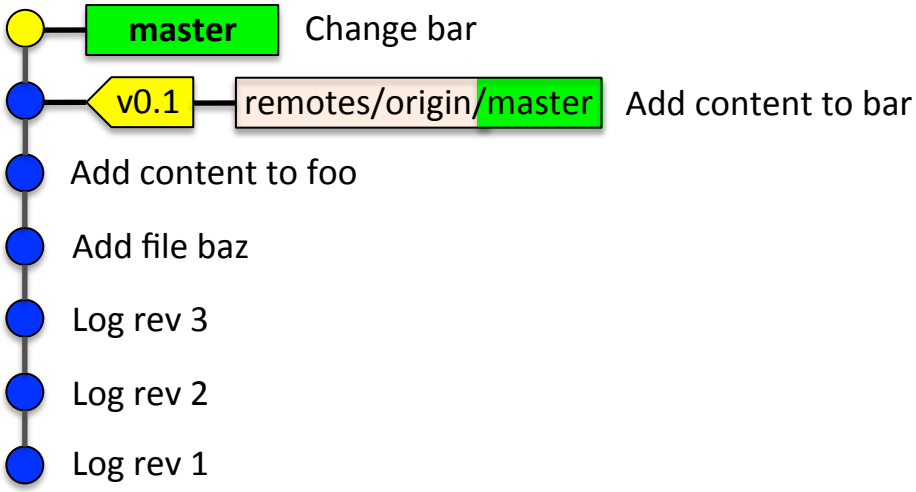
Other user's viewpoint

```
$ git push
```



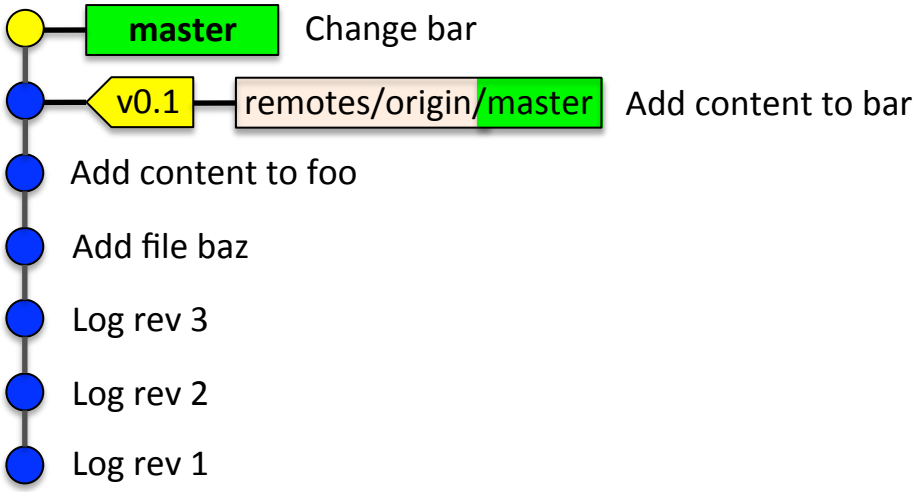
Other user's viewpoint

```
$ git push
To /Users/dparsons/tmp/git-sandbox/.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '[…]'
hint: Updates were rejected because the tip of your current branch is
hint: behind its remote counterpart. Integrate the remote changes
hint: (e.g. 'git pull ...') before pushing again. See the 'Note about
hint: fast-forwards' in 'git push --help' for details.
$
```



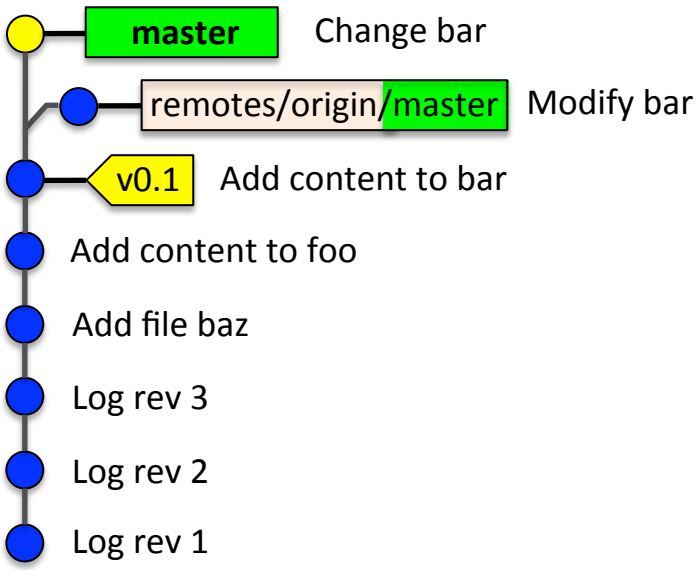
Other user's viewpoint

```
$ git fetch
```



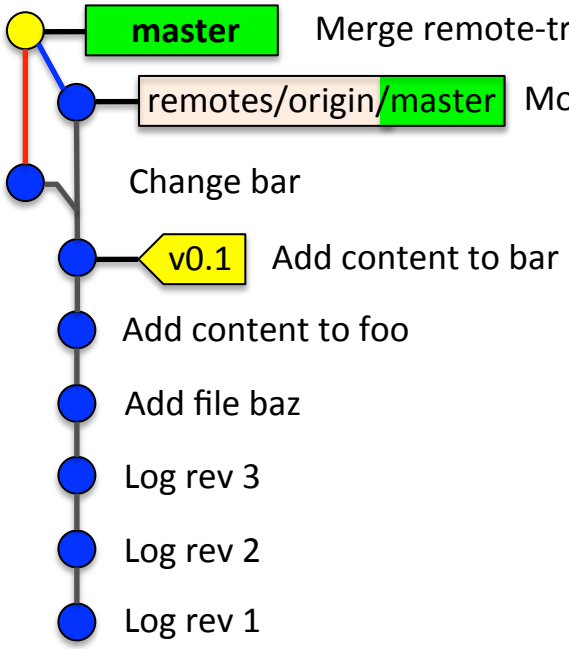
Other user's viewpoint

```
$ git fetch
From /Users/dparsons/tmp/git-sandbox/
   119dfba..a8f9783  master     -> origin/master
$
```



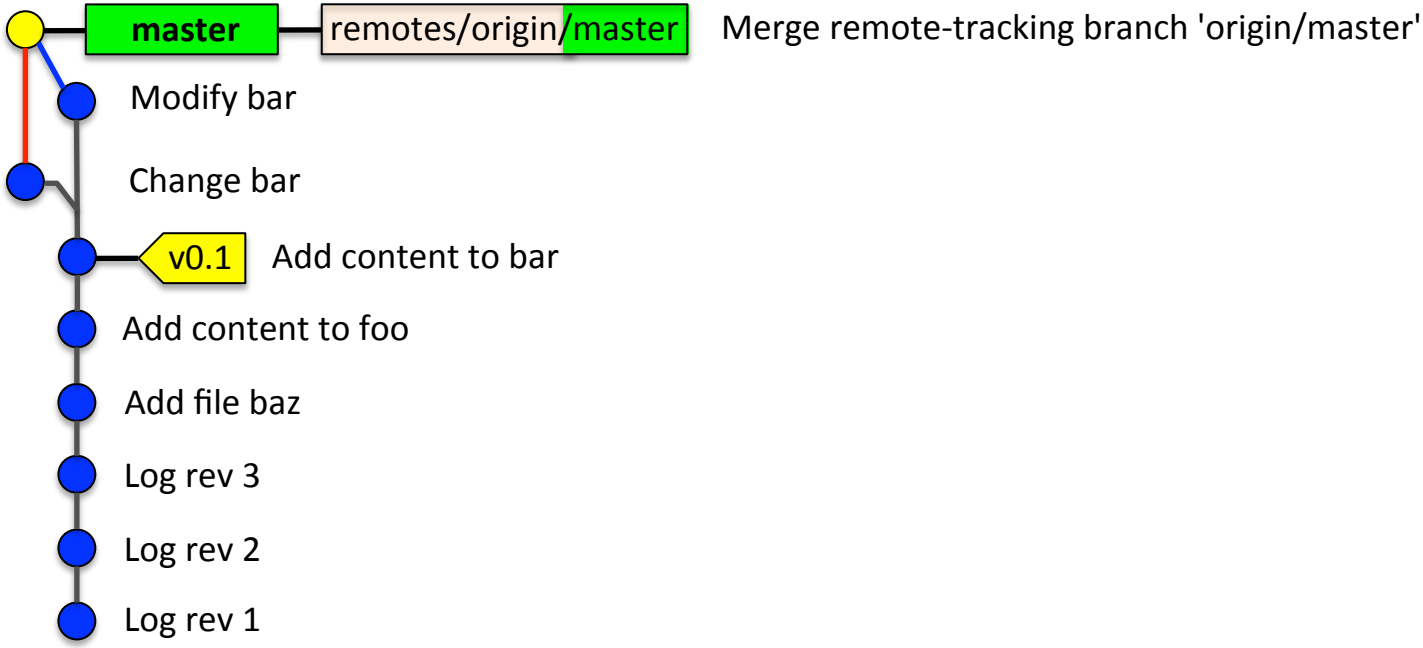
Other user's viewpoint

```
$ git merge origin/master
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then commit the result.
$ git mergetool
$ git ci
$
```



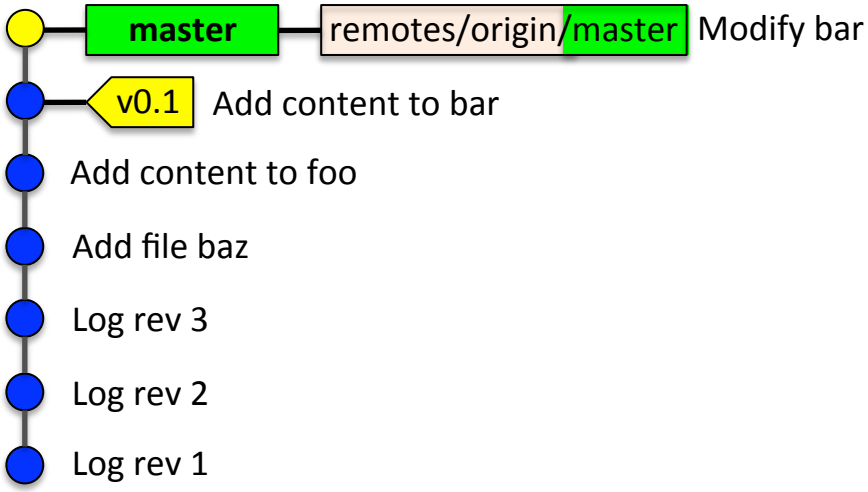
Other user's viewpoint

```
$ git push
$
```



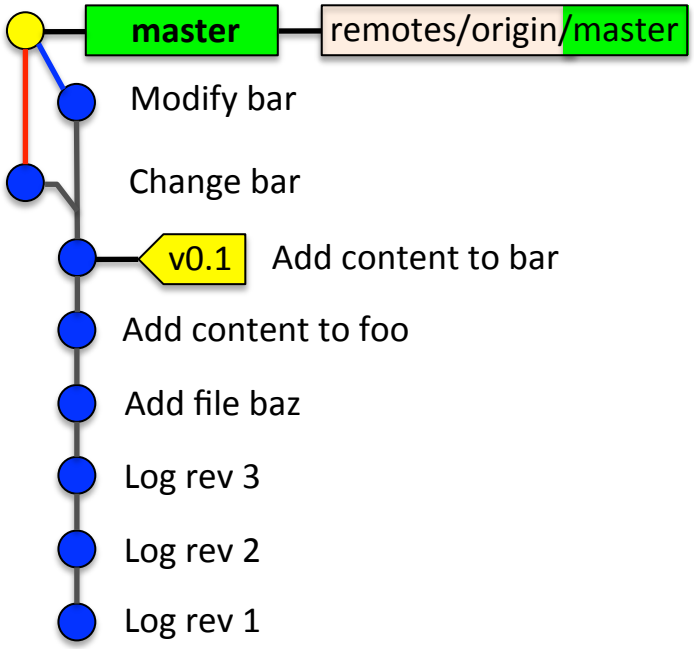
Pull

```
$ git pull
```



Pull

```
$ git pull
[…]
$
```



More about remotes and remote branches

About remotes

- Remote branches are local references that you cannot move (moved automatically on communication with the remote)
- `origin` is the default name of the default remote
- Add a remote:

```
git remote other add git://…
```

- Fetch remote branches:

```
git fetch other
```

- Push master to remotes/other/master

```
git push other master
```

Remote tracking branches

- Remote tracking branches are local branches that have a direct relationship to a remote branch (useful to simplify operations such as push, pull, merge, rebase, …)
- Make current branch track remotes/other/master:

```
git branch –u other/master
```

- List local branches and their remote counterpart:

```
git branch -vv
```

Annoying things

- Delete a remote branch:

```
git push origin --delete featA
```

- Push a tag:

```
git push origin mytag or
git push --tags
```

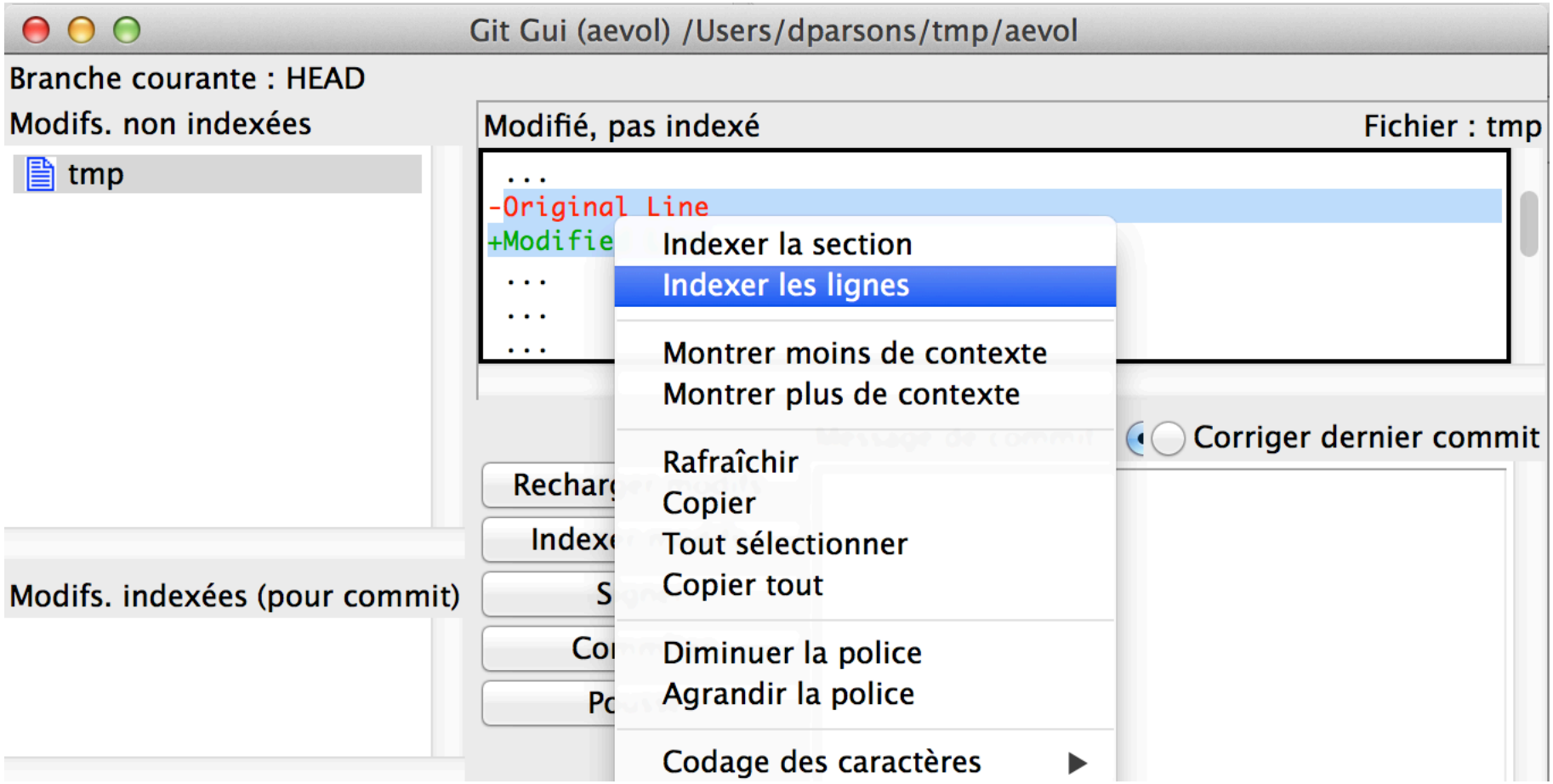
7
Partial commits

Add -p

Works but… a little tedious !

```
$ git add -p
diff --git a/file b/file
index e72f11f..75ad869 100644
--- a/file
+++ b/file
@@ -1,7 +1,7 @@
 ...
 ...
 ...
-  Original line
+  Modified line
 ...
 ...
 ...
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]?
```

git gui


Git Gui (aevol) /Users/dparsons/tmp/aevol
Branche courante : HEAD
Modifs. non indexées
tmp
Modifié, pas indexé
Fichier : tmp
...
-Original Line
+Modifie
...
...
...
Indexer la section
Indexer les lignes
Montrer moins de contexte
Montrer plus de contexte
Rafraîchir
Copier
Tout sélectionner
Copier tout
Diminuer la police
Agrandir la police
Codage des caractères
Corriger dernier commit
Modifs. indexées (pour commit)

8

Rewriting History (gentle intro)

Rewriting History ?

Remember you have a local repository ?

Everything you haven't published (i.e. pushed) yet is strictly local to your repo. It is known by you and no one else. Since then, what prevents you from modifying it ?

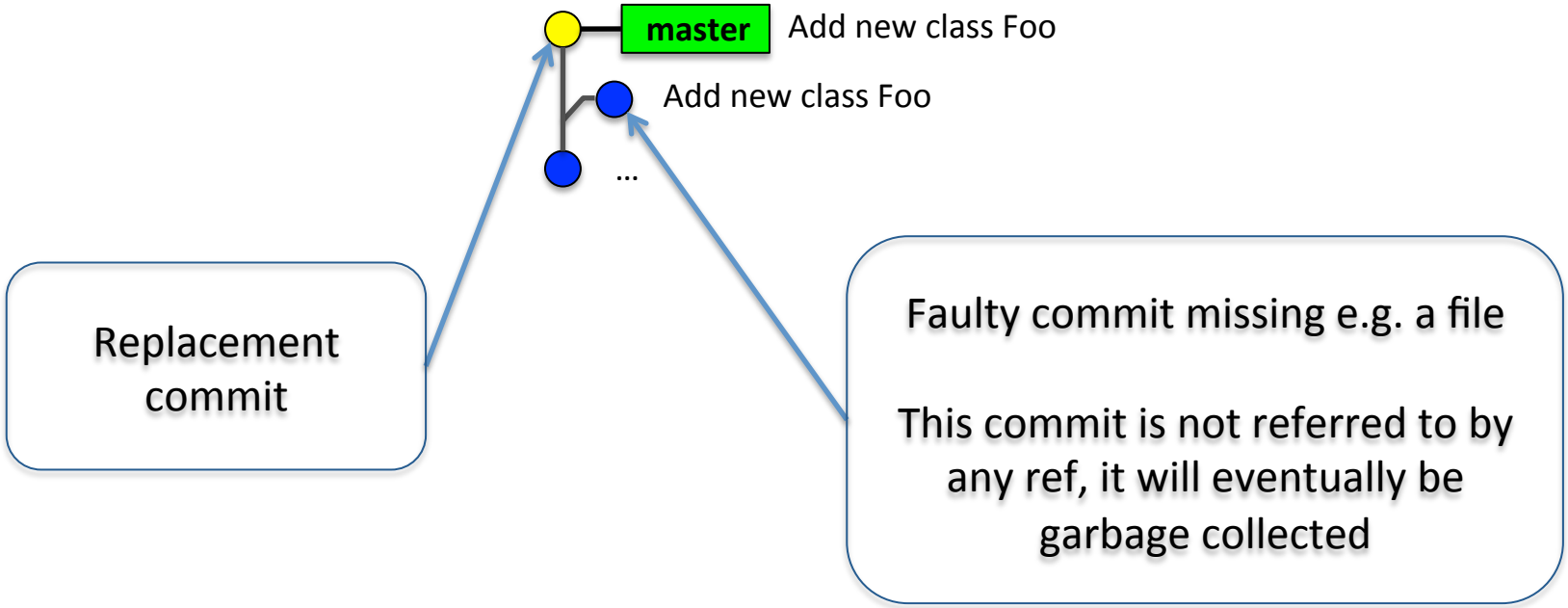
This is one of my favourite things about git, I can be stupid and appear not to be !

```
# You've just commited something and realize you forgot to add a file
git add the_forsaken_file
git commit --amend
# No one saw you ;)
```

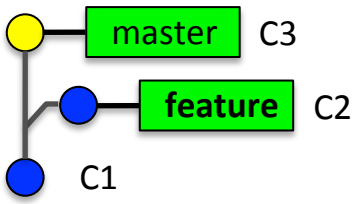
WARNING: Do not do that if you've pushed the faulty commit !!!!!

Commit --amend

Let's look at what our commit graph looks like

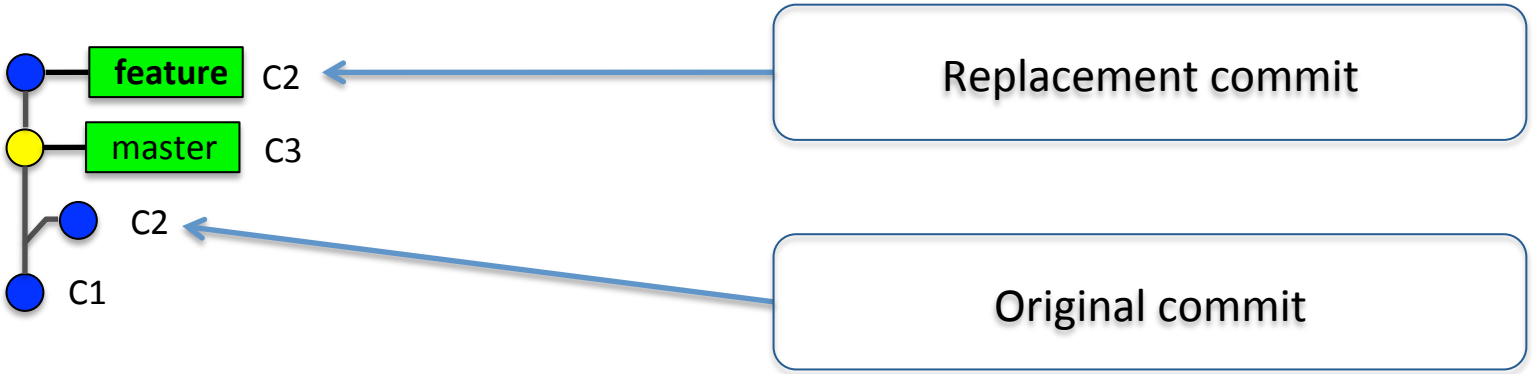


Basic `rebase`



Branch master has been updated since you started to write your feature. You would like to benefit from these updates (or perhaps have an up-to-date feature to propose as a pull-request ?)

```
$ git rebase master
First, rewinding head to replay your work on top of it…
Applying: …
```



Less basic `rebase`

Rebase literally means “set a new base for a branch”

However, if it’s trivial to tell where a branch ends, it’s not that simple for where it starts...

And what should be the new base for our branch is a similar matter…

So, given your commit tree, rebasing a branch is telling git to

1. “**Pick**” a branch
2. “**Cut**” it (but where ?)
3. “**Graft**” it (onto another part of the tree)

Less basic `rebase`

If you specify nothing, you **pick** the current branch, **cut** it at its LCA (last common ancestor) with its remote tracking branch and **graft** it at this exact same location (which means you've achieved to do nothing in a complicated way)

The most common case is to give a single arg, i.e. `git rebase master`. In that case, you **pick** the current branch, **cut** it at its LCA with master and **graft** it at the tip of master

You can also specify where to **cut** with the option `--root`, where to **graft** with the option `--onto` and where to **pick** (last argument)

```
$ git rebase --root <commit> --onto <new-base> <branch-to-rebase>
$ git rebase <upstream> --onto <new-base> <branch-to-rebase>
```

9
Merge or rebase ?

Prefer `fetch` over `pull`

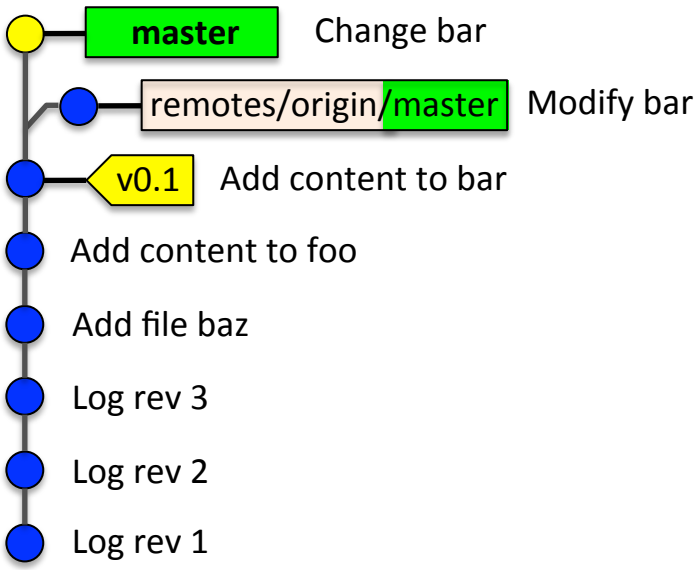
The `pull` command is most of the time equivalent to a `fetch` followed by a `merge`.

So what you are really asking git to do when you `pull` is to merge your work with something you know nothing about (!)

To come around this problem, start by fetching what's new from the remote and have a look at it. If what you really want is a merge, you can do it. But this time, you do it knowingly

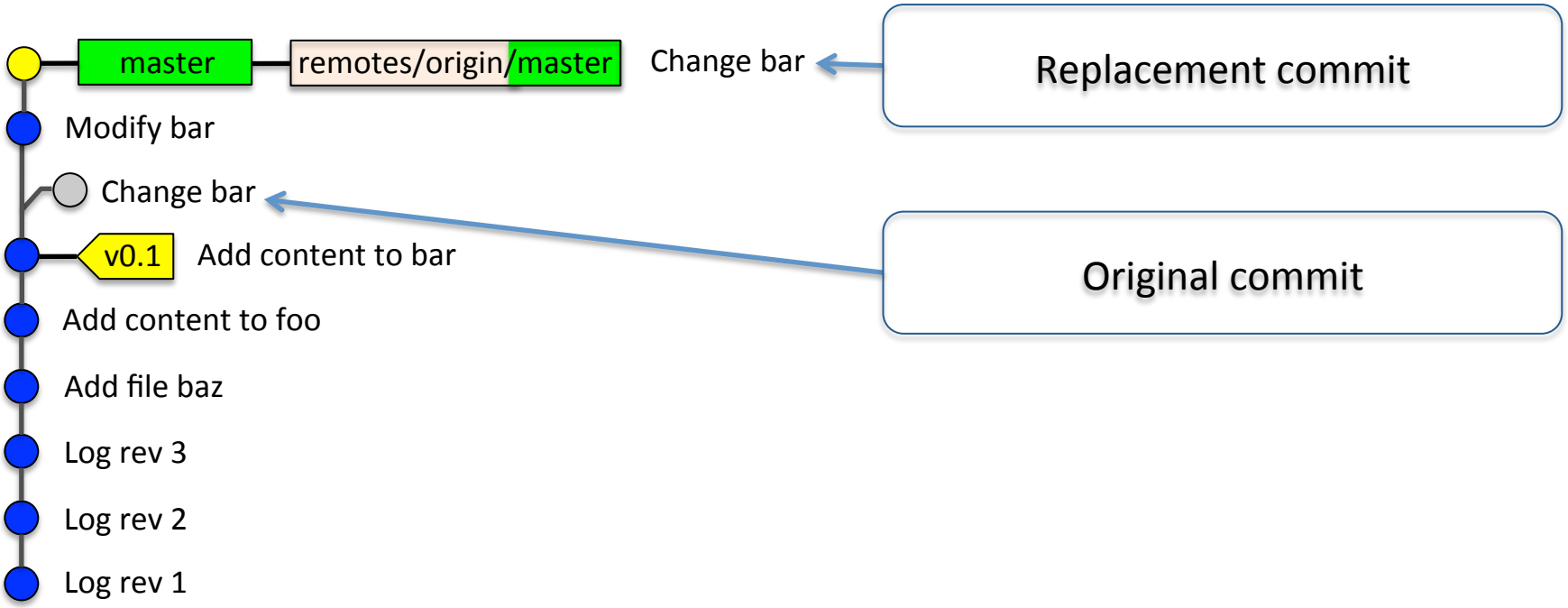
Rebase

```
$ git fetch
$ git rebase
```



Rebase

```
$ git fetch
$ git rebase
…
$
```



Rebase only local commits

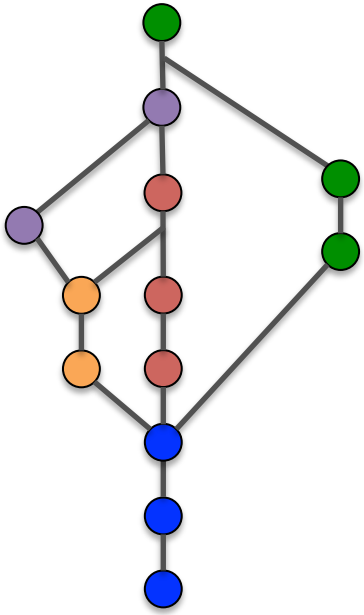
- Rebasing basically means re-writing the history of what happened. This is a very powerful feature.

  But as always, great power comes with great responsibility:

DO NOT REBASE COMMITS THAT EXIST OUTSIDE YOUR REPOSITORY

Merge or rebase ?

Which one do you like most ?

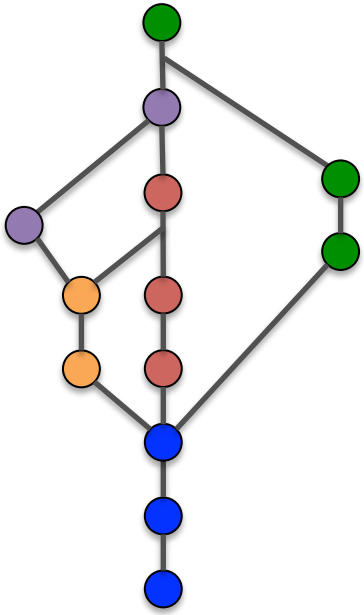


Merge

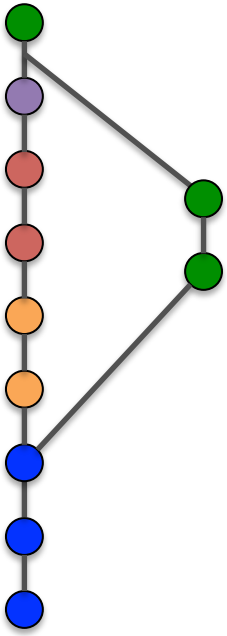
Rebase

Merge or rebase ?

Or maybe this mixed one ?



Merge



Mixed

Rebase

10
Undoing things

Unstage

Unstaging an entire file is very easy, git tells you how to do it:

```
$ git status
  (use "git reset HEAD <file>..." to unstage)
$ git reset HEAD <file>
```

Don’t want to have to remember this command ? Create an alias:

```
$ git config --global alias.unstage "reset HEAD"
$ git unstage <file>
```

If you don’t want to unstage the entire file but only some parts of it, the easiest solution is probably git gui (but you could also use `git reset -p`)

Unmodify a file

Again, git is kind enough to prompt you for actions you might want to do:

```
$ git status
  (use "git checkout -- <file>..." to discard changes …)
$ git co -- <file>
```

And again you can create an alias:

```
$ git config --global alias.unmod "checkout --"
$ git unmod <file>
```

If you don't want to unmodify a whole file but only some parts of it, the easiest solution is probably a difftool (but you could also use `git co -p`)

Undo a commit

- If the commit has not been published yet

```
$ git rebase –i
...
```

This way, you can thoroughly remove the faulty commit from the history

- If it has (been published)

```
$ git revert <commit-to-undo>
```

This will create a new commit whose diff is the inverse of that of the commit to undo

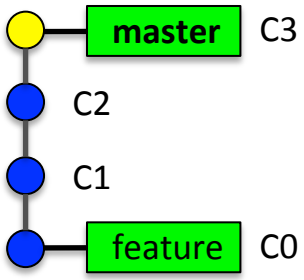
git reset

Git reset can become very handy when things are beginning to get awry

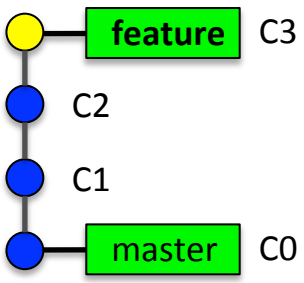
It allows you to make a branch point anywhere you want.

Let's say you have committed stuff in master when what you really wanted was to commit them in feature:

What you have

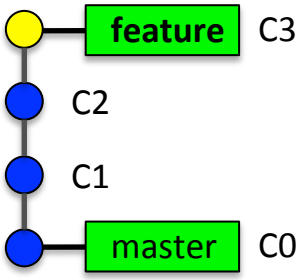


What you want

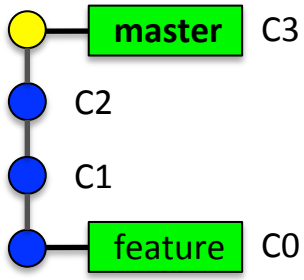


git reset

What you want



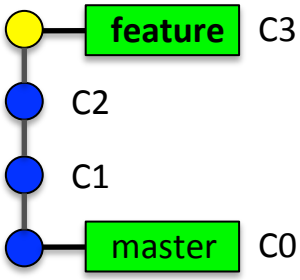
What you have



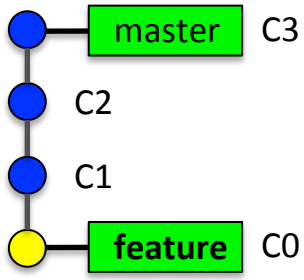

```
$ git co feature
```

git reset

What you want



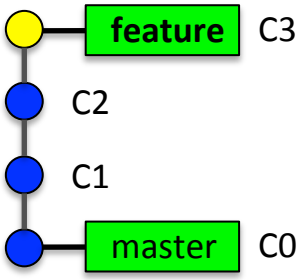
What you have



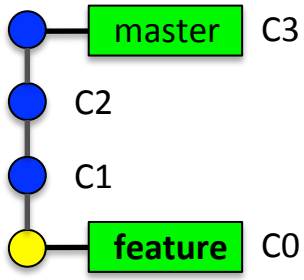

```
$ git co feature
Switched to branch 'feature'
$ # ?
```

git reset

What you want



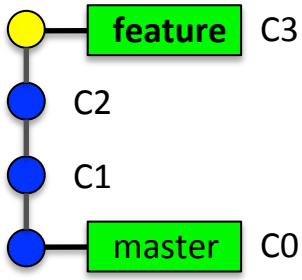
What you have



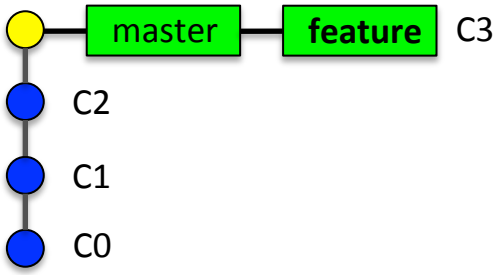

```
$ git co feature
Switched to branch 'feature'
$ git (merge | rebase | reset --hard) master
```

git reset

What you want



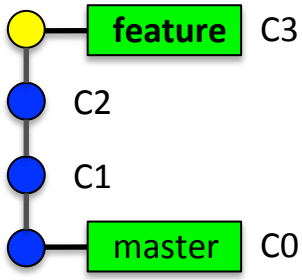
What you have



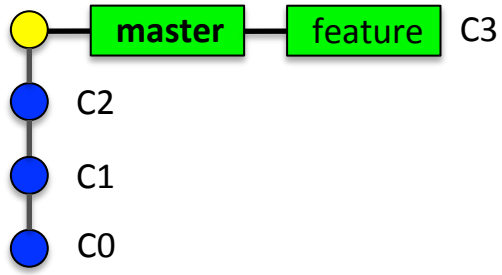

```
$ git co feature
Switched to branch 'feature'
$ git (merge | rebase | reset --hard) master
...
$
```

git reset

What you want



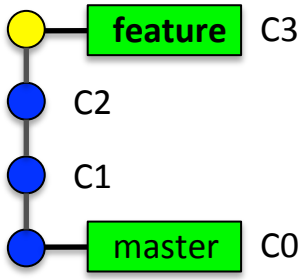
What you have



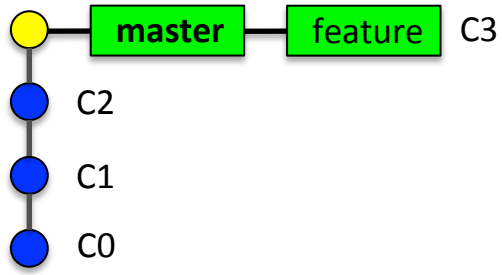

```
$ git co feature
Switched to branch 'feature'
$ git (merge | rebase | reset --hard) master
...
$ git co master
Switched to branch 'master'
$ # ?
```

git reset

What you want



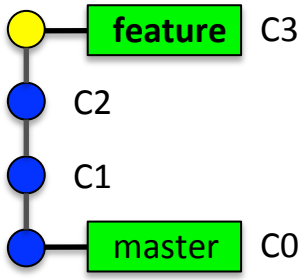
What you have



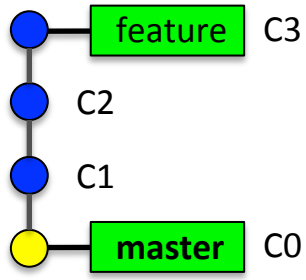

```
$ git co feature
Switched to branch 'feature'
$ git (merge | rebase | reset --hard) master
...
$ git co master
Switched to branch 'master'
$ git reset --hard bdfa5a # ref-to-C0
```

git reset

What you want



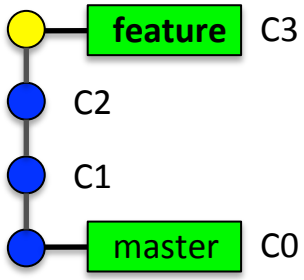
What you have




```
$ git co feature
Switched to branch 'feature'
$ git (merge | rebase | reset --hard) master
...
$ git co master
Switched to branch 'master'
$ git reset --hard bdfa5a # ref-to-C0
HEAD is now at bdfa5ab C0
$
```

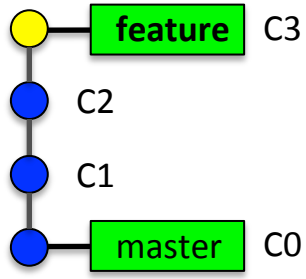
git reset

What you want



BINGO !

What you have




```
$ git co feature
Switched to branch 'feature'
$ git (merge | rebase | reset --hard) master
...
$ git co master
Switched to branch 'master'
$ git reset --hard bdfa5a # ref-to-C0
HEAD is now at bdfa5ab C0
$ git co feature
$
```

soft, mixed or hard ?

There are 3 main options to `git reset` : soft, mixed and hard.

There are also 3 steps that can be done:

1. Move the current branch
2. Update the staging area
3. Update the working directory

- --soft does only 1.
- --mixed does 1. and 2.
- --hard goes all the way to 3.

Cheat-sheet

	HEAD	Index	Workdir	WD Safe?
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	**NO**
`checkout [commit]`	HEAD	YES	YES	YES
File Level				
`reset (commit) [file]`	NO	YES	NO	YES
`checkout (commit) [file]`	NO	YES	YES	**NO**

More undoing…

`git-filter-branch` is a very powerful tool, it allows you to apply "filters" on each revision of a branch.

This can be useful e.g. when you've committed a file that should never have been added (binary file, confidential information, …)

```
$ git filter-branch –tree-filter 'rm -f <file>' HEAD
```

An alternative to `git-filter-branch` for "cleansing bad data out of a git repo" is worth mentioning : BFG Repo-Cleaner (https://rtyley.github.io/bfg-repo-cleaner/)

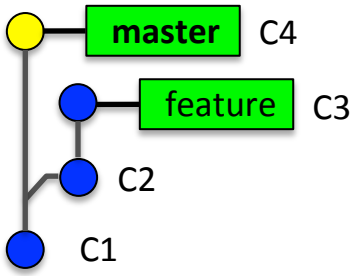
11
Cherry picking

Cherry picking

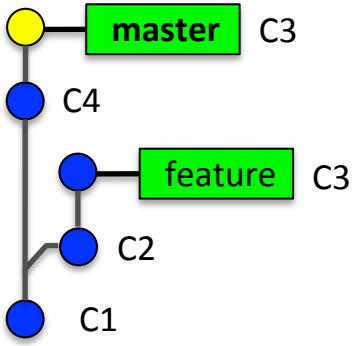
Cherry picking is similar to rebasing in the same way that checking out is similar to resetting. You can do the same kind of things except that you don't reset anything

Cherry picking is really easy and very handy.

It allows you to apply the changes introduced by one or more commits on top of HEAD.




```
$ git cherry-pick <C3>
```



12
Interactive rebase

Interactive rebase

Remember what we did with `git commit --amend`? Now imagine you did the exact same thing but have already added a few commits on top of the faulty one. You can't use `commit --amend` because it only allows to modify the very latest commit.

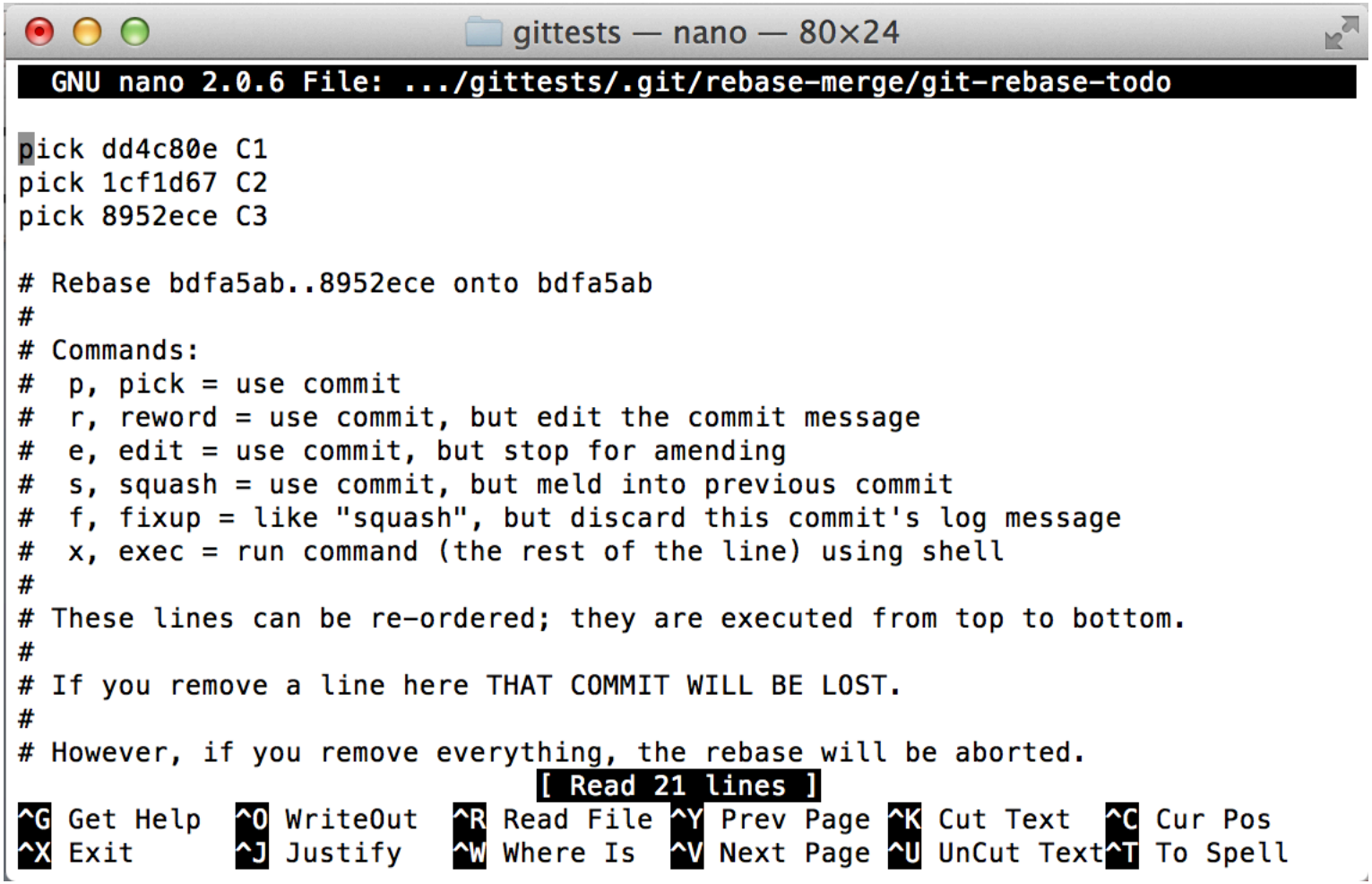
This is when the interactive rebase becomes handy (even though this particular rewrite could be done with non-interactive rebase)

Let's see what the command looks like

```
$ git rebase --interactive
```

What you're rebasing and onto what, follows the same rules as non-interactive rebase. The difference is that you will get to decide what to do with each commit to be replayed

Interactive rebase



13
Other interesting features

git bisect

You've just found a major bug and need to know when it was introduced.

Don't panic, git bisect is here to help you, it using dichotomy to locate the first "bad" commit quickly

```
$ git bisect start
$ git bisect bad # Current version is bad
$ git bisect good <a-good-old-version>
Bisecting: 675 revisions left to test after this (roughly 10 steps)
$ git bisect good|bad
…
```

git subrepo

Alternatives: submodules, subtrees

Git subrepo allows you to work with embedded git repositories

```
# Clone an existing repo as a subrepo (in a subdirectory)
$ git subrepo clone <url> [<subdir>]

# Create an embedded git repo
$ git subrepo init <subdir>

# Pull from upstream
$ git subrepo pull <subdir>

# Push to upstream
$ git subrepo push <subdir>
```

git annex

Alternatives: lfs

Git annex provides an interesting solution to version large files, it basically handles symlinks and provides tools to manage the actual files behind the links

```
# Prepare an existing git repo for git annex
$ git annex init

$ git annex add <file>
$ git ci
# => commits a symlink and stores the file in .git/annex
```

Git annex can be used with a wide variety of types of remote storage spaces (special remotes)

Hooks

Hooks are custom scripts that can be automatically triggered when certain important actions occur

Client-side hooks:

pre-commit, prepare-commit-msg, commit-msg, post-commit

pre-rebase, post-rewrite, post-checkout, post-merge, pre-push, pre-auto-gc

Server-side hooks:

pre-receive, update, post-receive

On platforms such as forges or gitlabs/github, some (many ?) predefined hooks can be set up in a matter of minutes

Hooks can be used e.g. to send notifications, run tests prior to commits, enforce any kind of policies, …

Multiple remotes, push requests

Many prjects on github/lab allow third-party contributions by a mechanism called pull-request. This mechanism can also be used internally for code-review

Multiple remotes can be very usefull in the context of workflows including pull-requests but also in other cases (git annex, migrations, …)

Rerere

One of the most annoying things that a git-user has to do is to resolve conflicts.

Even more annoying would be to have to resolve the same conflict several times. Sadly, this happens. Mostly when relying heavily on rebase (?)

Rerere helps you avoiding this situation by **re**using **re**corded **re**solutions

Thank you

Antenne INRIA Lyon la Doua

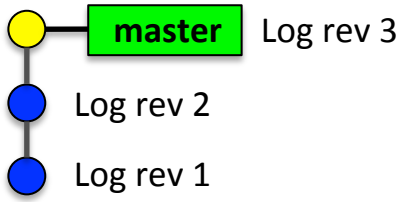
www.inria.fr

A1

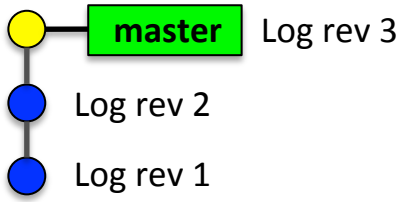
Illustration of the Main Commands

Illustrations : `status`

```
$ ls
foo      bar
$ git status
On branch master

nothing to commit, working directory clean
$
```



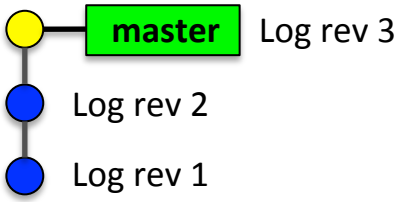
Illustrations : untracked file




```
$ ls
foo      bar
$ git status
On branch master

nothing to commit, working directory clean
$
$ # Create a new file baz
$ echo "A new file…" > baz
$
$ git status
On branch master
Untracked files:
  (use "git add <file>…" to include in what
will be committed

        baz

nothing added to commit but untracked files
present (use "git add" to track)
$
```

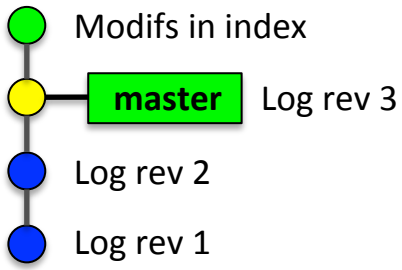
Illustrations : add

```
$ git add baz
```



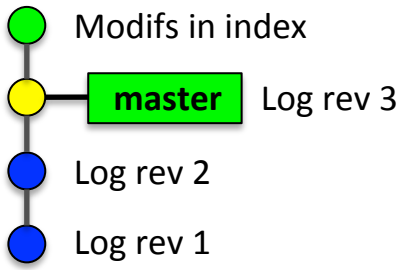
Illustrations : add

```
$ git add baz
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   baz

$
```

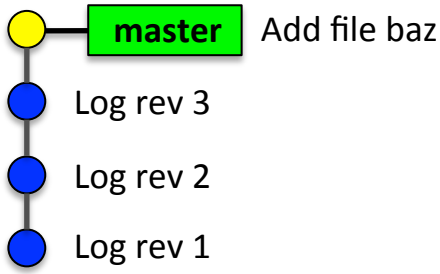


Illustrations : commit

```
$ git add baz
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   baz

$ git commit
```

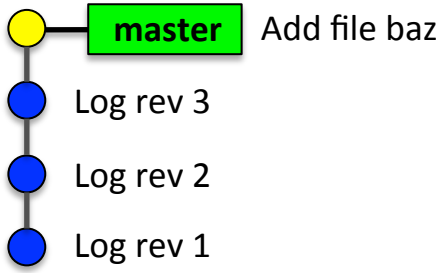


Illustrations : commit




```
$ git add baz
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   baz

$ git commit
[master 98aa7ff] Add file baz
 1 file changed, 1 insertion(+)
 create mode 100644 baz
$ git status
On branch master
nothing to commit, working directory clean
$
```

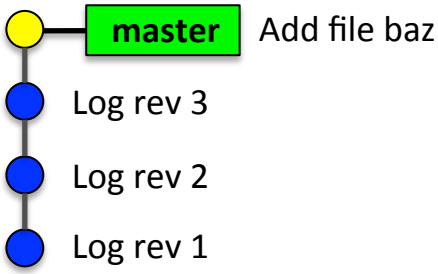
Illustrations : log




```
$ git log
commit 579001a8a6f83cb6547440ba319f7210916ece6b
Author: David Parsons <david.parsons@inria.fr>
Date:   Sat Jan 9 17:53:11 2016 +0100

    Add file baz

commit e82a7786b5ebfc1d0e62eb00ae2101a328d3f0e4
Author: David Parsons <david.parsons@inria.fr>
Date:   Sat Jan 9 17:44:29 2016 +0100

    Log rev3

commit f6813a3b9f081bb486975dfe5b6a426effabf61c
Author: David Parsons <david.parsons@inria.fr>
Date:   Sat Jan 9 17:43:39 2016 +0100

    Log rev2

commit f667b6e8847d20c67968c2b2d8a85fac7076251f
:
```

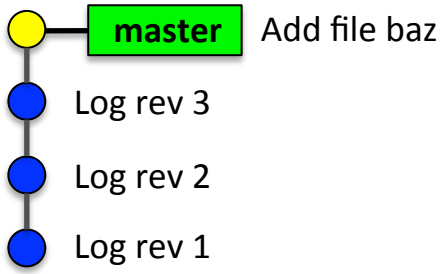
Illustrations : log

```
$ git log --pretty=oneline
579001a8a6f83cb6547440ba319f7[…] Add file baz
e82a7786b5ebfc1d0e62eb00ae210[…] Log rev3
f6813a3b9f081bb486975dfe5b6a4[…] Log rev2
f667b6e8847d20c67968c2b2d8a85[…] Log rev1
$
$ git log --pretty=format:"%h - %an : %s"
579001a - David Parsons : Add file baz
e82a778 - David Parsons : Log rev3
f6813a3 - David Parsons : Log rev2
f667b6e - David Parsons : Log rev1
$
```



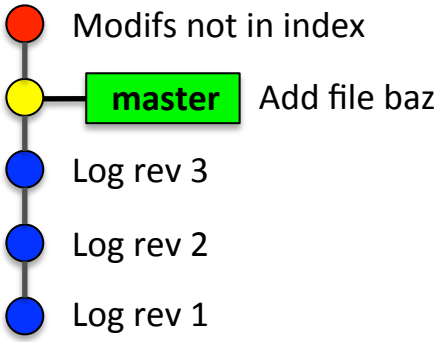
Illustrations : edit tracked file

```
$ # Add content to foo
$ echo "Added content" >> foo
```

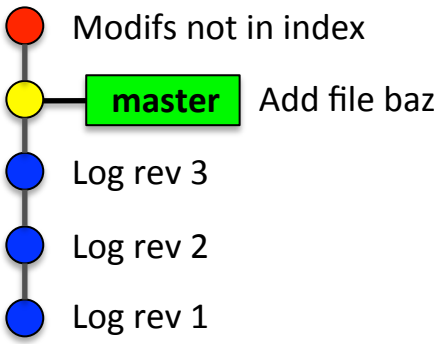


Illustrations : edit tracked file

```
$ # Add content to foo
$ echo "Added content" >> foo
$
```



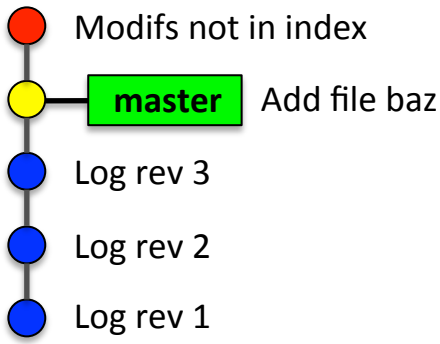
Illustrations : edit tracked file




```
$ # Add content to foo
$ echo "Added content" >> foo
$
$ 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:   foo

no changes added to commit (use "git add" and/
or "git commit -a")
$
```

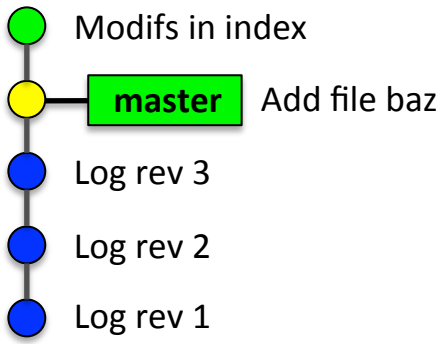
Illustrations : `stage`

```
$ # Mark changes in foo as ‘to be committed’
$ git stage foo
```



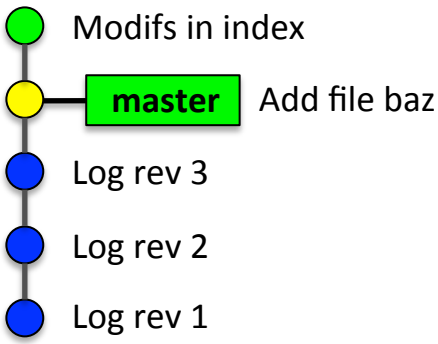
Illustrations : stage

```
$ # Mark changes in foo as ‘to be committed’
$ git stage foo
$
$ # The exact same thing could have been done
with git add foo
$
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   foo

$
```

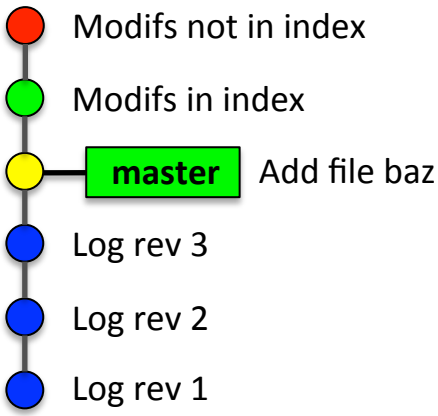


Index and Working Copy

```
$ # Add content to bar
$ echo "Content added to bar" >> bar
```

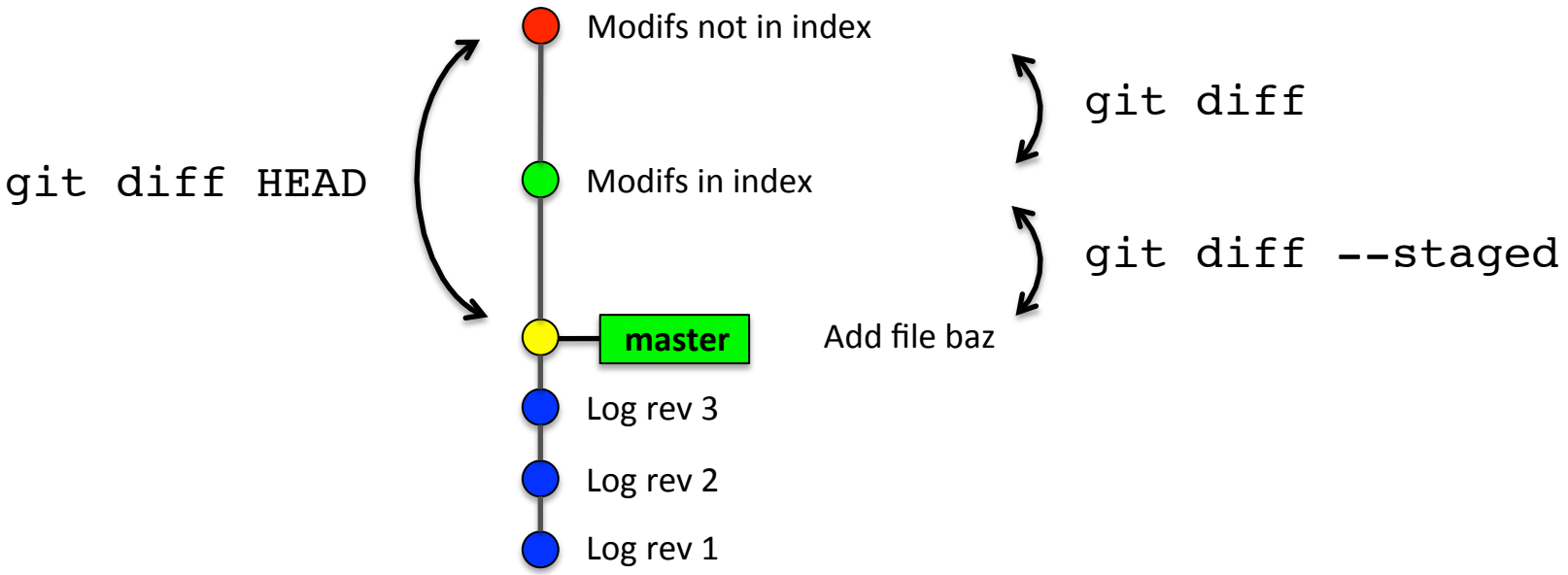


Index and Working Copy

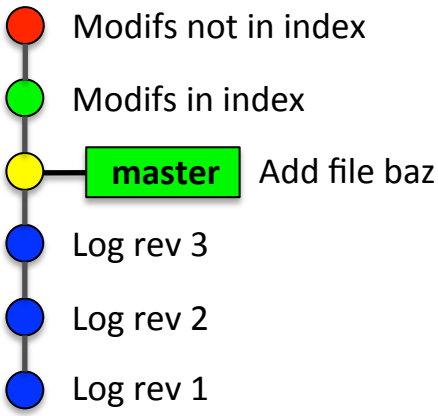



```
$ # Add content to bar
$ echo "Content added to bar" >> bar
$
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   foo

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:   bar

$
```

Illustrations : `diff`



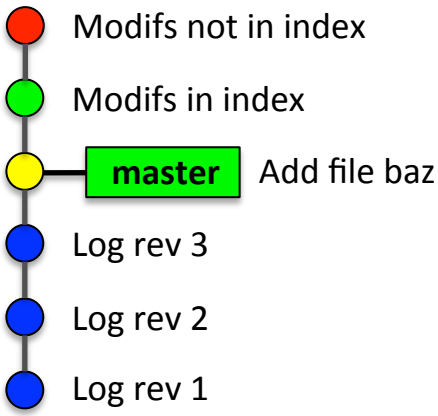
Index and Working Copy



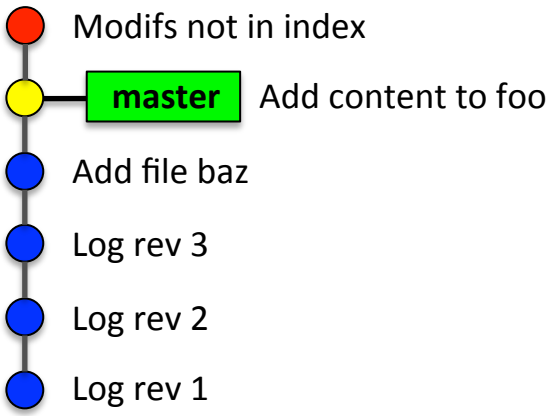

```
$ # Add more content to foo
$ echo "more content" >> foo
$
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   foo

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:   bar
    modified:   foo

$
```

Bypassing the index

```
$ # Commit working copy state of foo
$ git commit foo
```



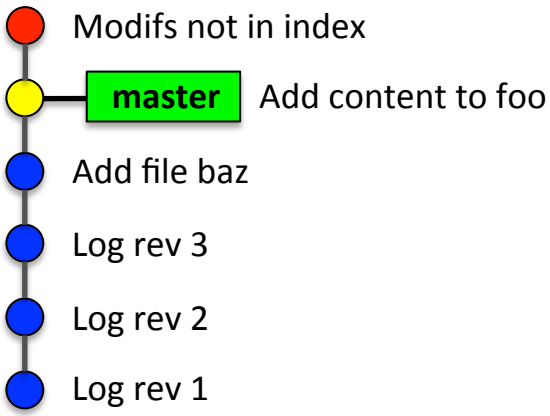
Bypassing the index



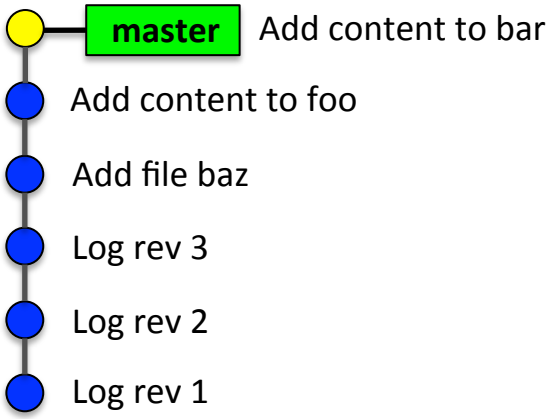

```
$ # Commit working copy state of foo
$ git commit foo
[master 5e60acc] Add content to foo
 1 file changed, 2 insertions(+)
$
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will
be committed)
  (use "git checkout -- <file>..." to discard
changes in working directory)

    modified:   bar

$
```

Bypassing the index

```
$ # Commit working copy state of all tracked
files
$
$ git commit -a -m "Add content to bar”
```



Bypassing the index




```
$ # Commit working copy state of all tracked
files
$
$ git commit -a -m "Add content to bar"
[master 119dfba] Add content to bar
 1 file changed, 1 insertion(+)
$ git status
On branch master
nothing to commit, working directory clean
$
```

Configuring git

```
$ # Configure your name and e-mail address (almost mandatory)
$ git config --global user.name "David Parsons"
$ git config --global user.email david.parsons@inria.fr
$
$ # Configure the editor git will open when needed
$ git config --global core.editor nano
$
$ # Setup a few aliases, either simple shorthands...
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.st status
$
$ # ... or including options
$ git config --global alias.lg "log --pretty=format:\"%h - %an : %s\""
$
$ # You can even create new commands
$ git config --global alias.unstage "reset HEAD"
```

Tree-ish

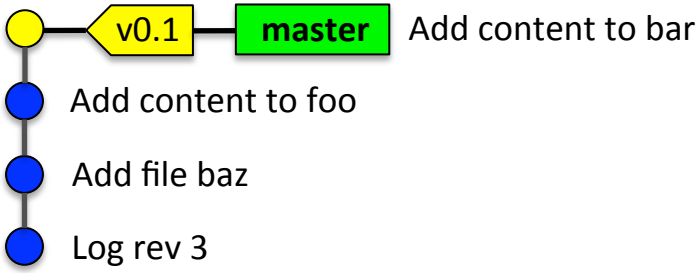
A commit can be identified in multiple ways, *e.g.* :

- Its SHA1 (possibly abbreviated)
- A reference (*e.g.* HEAD)
- An indirect reference : HEAD^, HEAD~, …
- A branch name
- …
- A tag

Tags

Tagging a commit simply means giving it a name. This commit can then be referenced by this tag

```
$ # Create a tag named "v0.1"
$ git tag v0.1

$ # List tags
$ git tag
v0.1
$
```



Tags are not pushed by default, use

`git push origin <tagname>` or `git push --tags`