GIT for Beginners

Anthony Baire

Université de Rennes 1

November 14, 2013

Objectives

- Understand the basics about version control systems
- Getting started with GIT
  - working with a local repository
  - synchronising with a remote repository
  - setting up a server

Summary

1. About Version Control Tools
2. Overview of GIT
3. Working locally
4. Branching & merging
5. Interacting with a remote repository
6. Administrating a server
7. Extras

Part 1.
About Version Control Tools

- Definition
- Use cases
- Base concepts
- History

What is a version control system ?

From: `http://en.wikipedia.org/wiki/Revision_control`

Revision control [...] *is the management of changes to documents, computer programs, large web sites, and other collections of information.*

Changes are usually identified by a number or letter code, termed the "revision number" [...]. *For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on.*

Each revision is associated with a timestamp and the person making the change.

Revisions can be compared, restored, and with some types of files, merged.

Use case 1: keeping an history

The life of your software/article is recorded from the beginning

- at any moment you can revert to a previous revision [1]
- the history is browseable, you can inspect any revision [2]
  - when was it done ?
  - who wrote it ?
  - what was change ?
  - why ?
  - in which context ?
- all the deleted content remains accessible in the history

[1] let's say your not happy with your latest changes

[2] this is useful for understanding and fixing bugs

Use case 2: working with others

VC tools help you to:

- share a collection of files with your team
- merge changes done by other users
- ensure that nothing is accidentally overwritten
- ~~know who you must blame when something is broken~~

Use case 3: branching

You may have multiple variants of the same software, materialised as **branches**, for example:

- a main branch
- a maintainance branch *(to provide bugfixes in older releases)*
- a development branch *(to make disruptive changes)*
- a release branch *(to freeze code before a new release)*

VC tools will help you to:

- handle multiple branches concurrently
- merge changes from a branch into another one

Use case 4: working with external contributors

VC tools help working with third-party contributors:

- it gives them visibility of what is happening in the project
- it helps them to submit changes (patches) and
  it helps you to integrate these patches
- forking the development of a software and merging it back into mainline[3]

[3] decentralised tools only

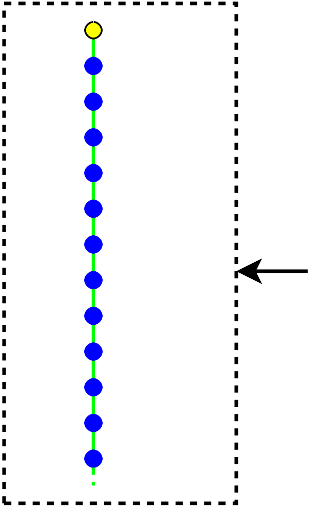
Use case 5: scaling

Some metrics[4] about the Linux kernel (developed with GIT):

- about 10000 changesets in each new version (every 2 or 3 months)
- 1000+ unique contributors

[4] source: the Linux Foundation

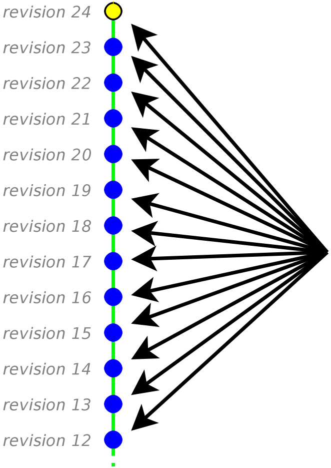
Some illustrations



The Repository

it contains the full history of your project (all revisions from the beginning)

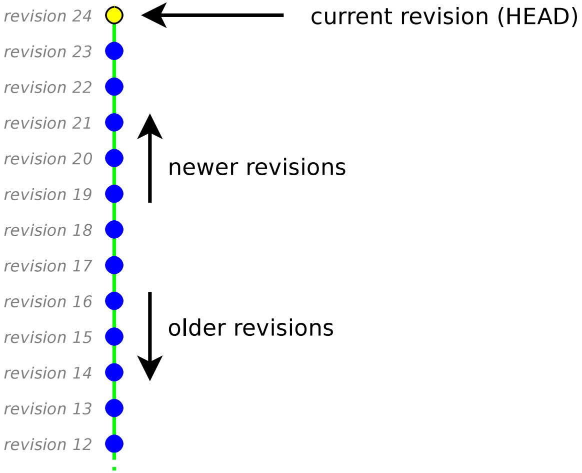
Some illustrations



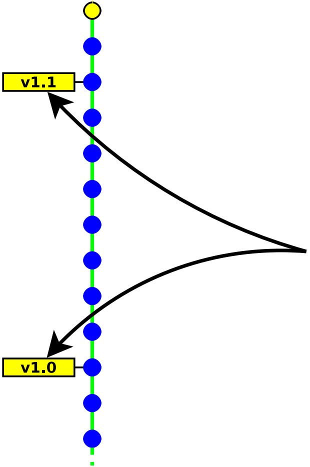
Revisions

each revision:
- introduces changes from the previous revision
- has an identified author
- contains a textual message describing the changes

Some illustrations



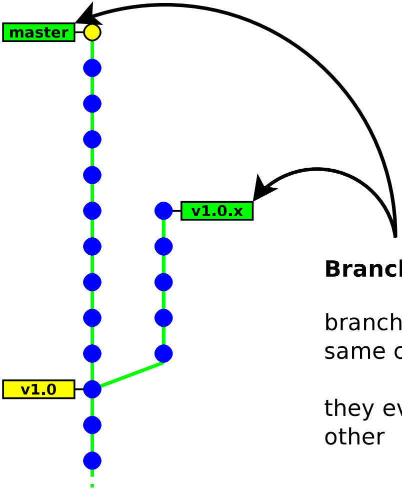
Some illustrations



Tags

a tag identifies a particular revision
(typically each release of the software)

Some illustrations

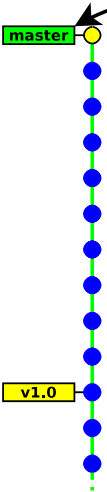


Branches

branches are different variants of the same collection of files

they evolve independently of each other

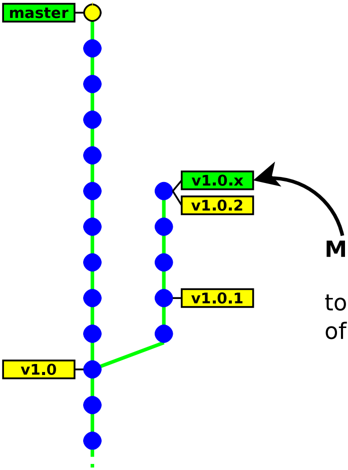
Some illustrations



Main branch

where the everyday development happens

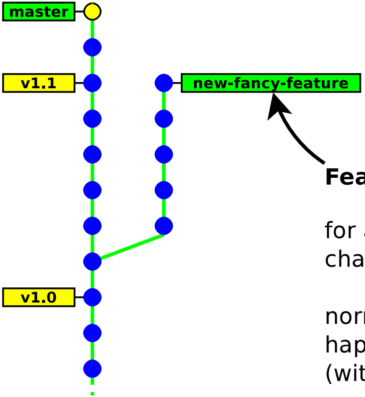
Some illustrations



Maintainance branch

to issue bug fixes for older releases of the software

Some illustrations

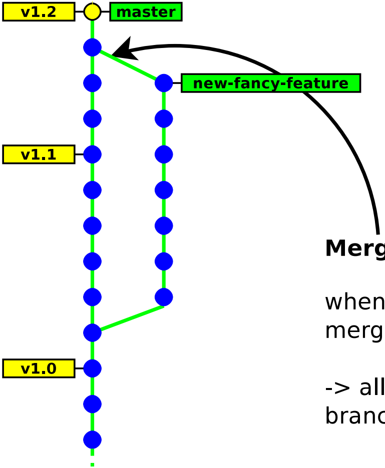


Feature branch

for a new feature requiring intrusive changes in the code

normal development continues to happen in the master branch (without disturbance)

Some illustrations

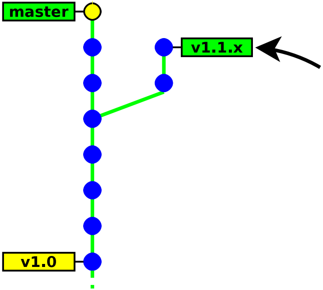


Merging

when the new feature is ready, it can merged back into the master branch

-> all changes done in the feature branch are imported

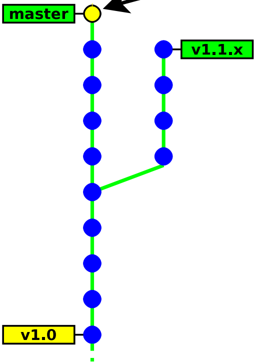
Some illustrations



Release branch

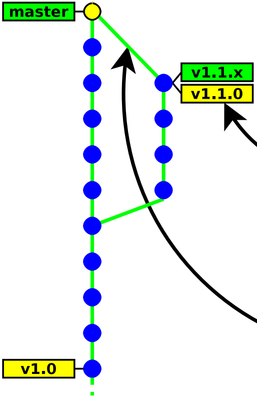
to prepare the next release
- the code is frozen
- only bug fixes are accepted

Some illustrations



meanwhile developments continue
in the master branch

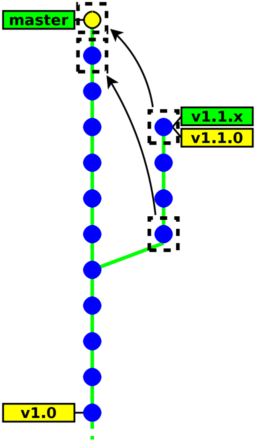
Some illustrations



New release

when the code is ready, the new version is released
- the release branch becomes a maintainance branch
- bug fixes can be merged back into the main branch

Some illustrations



Cherry picking

it may not be desirable to merge all the commits into the other branch (e.g. a bug may need a different fix)

-> it is possible to apply each commit individually

Taxinomy

Architecture:

- **centralised** → everyone works on the same unique repository
- **decentralised** → everyone works on his own repository

Concurrency model:

- **lock before edit** (mutual exclusion)
- **merge after edit** (may have conflicts)

History layout:

- **tree** (merges are not recorded)
- **direct acyclic graph**

Atomicity scope: **file** vs **whole tree**

GIT

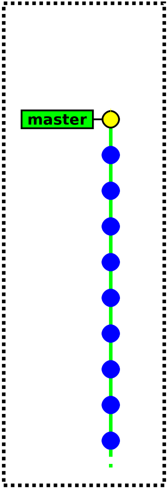
Other technical aspects

Space efficiency: storing the whole history of a project requires storage space *(storing every revision of every file)*
→ most VC tools use delta compression to optimise the space *(except Git which uses object packing instead)*

Access method: A repository is identified with a URL. VC tools offer multiple ways of interacting with remote repositories.

- dedicated protocol (*svn:// git://*)
- direct access to a local repository (*file://path* or just *path*)
- direct access over SSH (*ssh:// git+ssh:// svn+ssh://*)
- over http (*http:// https://*)

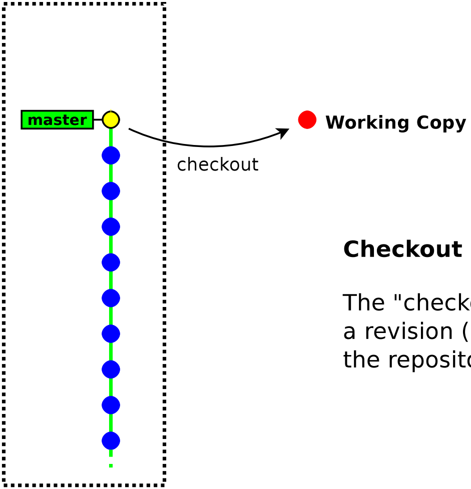
Creating new revisions



A repository is an opaque entity,
it cannot be edited directly

We will first need to extract
a local copy of the files

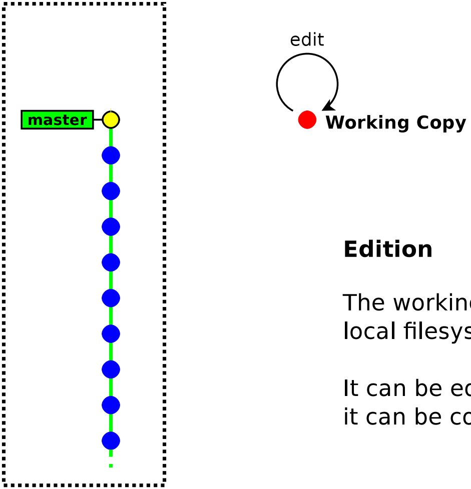
Creating new revisions



Checkout

The "checkout" command extracts a revision (usually the latest) from the repository.

Creating new revisions

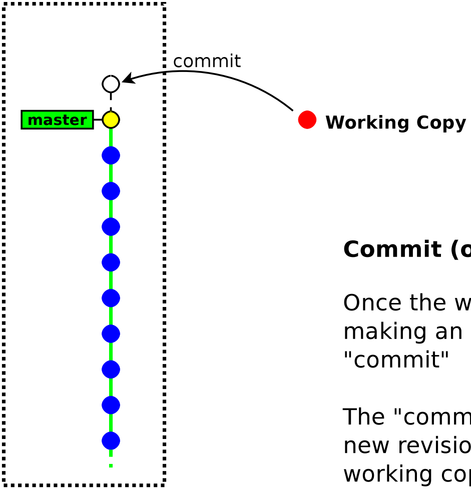


Edition

The working copy is hosted in the local filesystem

It can be edited with any editor, it can be compiled, ...

Creating new revisions

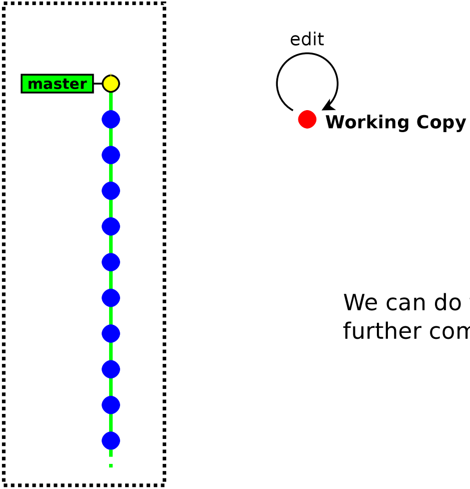


Commit (or Checkin)

Once the working copy is ready for making an new revision, we do a "commit"

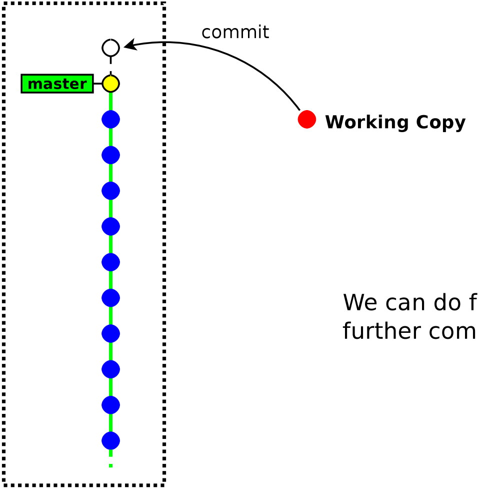
The "commit" command creates a new revision from the current working copy

Creating new revisions



We can do further editions and further commits...

Creating new revisions



We can do further editions and further commits...

What shall be stored into the repository ?

You should store all files that are not generated by a tool:

- source files (`.c .cpp .java .y .l .tex` ...)
- build scripts / project files (`Makefile configure.in Makefile.am CMakefile.txt wscript .sln`)
- documentation files (`.txt README ...`)
- resource files (images, audio, . . .)

You should not store generated files
(*or you will experience many unneccessary conflicts*)

- `.o .a .so .dll .class .jar .exe .dvi .ps .pdf`
- source files / build scripts when generated by a tool (like autoconf, cmake, lex, yacc)

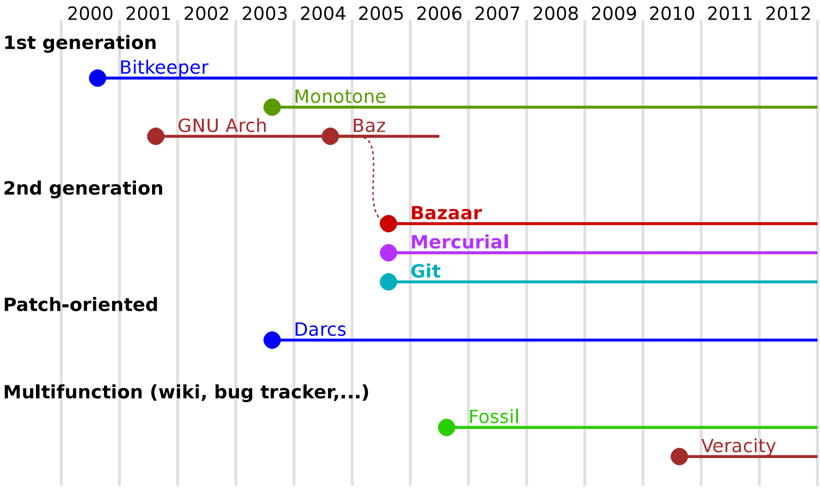
Guidelines for committing

- commit often
- commit independent changes in separate revisions
- in commit messages, describe the rationale behind of your changes (*it is often more important than the change itself*)

History (Centralised Tools)

- 1^{st} generation *(single-file, local-only, lock-before-edit)*
  - 1972: **SCCS**
  - 1982: **RCS**
  - 1985: PVCS
- 2^{nd} generation *(multiple-files, client-server, merge-before-commit)*
  - 1986: **CVS**
  - 1992: Rational ClearCase
  - 1994: Visual SourceSafe
- 3^{rd} generation *(+ repository-level atomicity)*
  - 1995: Perforce
  - 2000: **Subversion**
  - + many others

History (Decentralised tools)



Part 2.
Overview of GIT

- History
- Git's design & features
- User interfaces

History

- before 2005: Linux sources were managed with Bitkeeper (proprietary DVCS tool)
- April 2005: revocation of the free-use licence (because of some reverse engineering)
- No other tools were enough mature to meet Linux's dev constraints (distributed workflow, integrity, performance).
  ⇒ Linus Torvald started developing Git
- June 2005: first Linux release managed with Git
- December 2005: Git 1.0 released

Git Design objectives

- distributed workflow (decentralised)
- easy merging (`merge` deemed more frequent than `commit`)
- integrity (protection against accidental/malicious corruptions)
- speed & scalability
- ~~ease of use~~

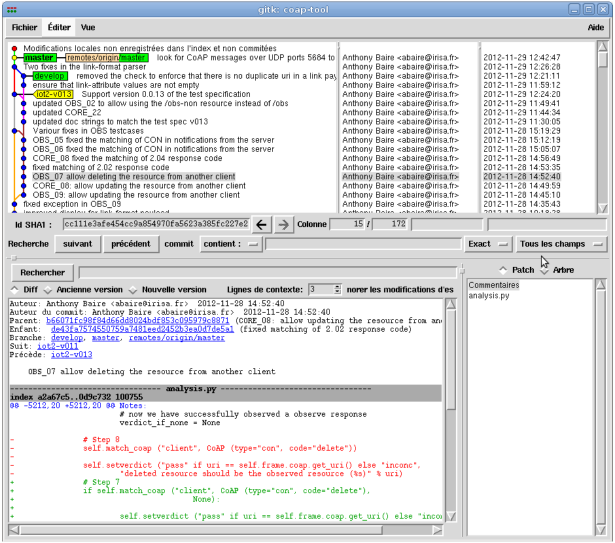
Git Design choices

- Easily hackable
  - simple data structures (blobs, trees, commits, tags)
  - no formal branch history
    (a branch is just a pointer to the last commit)
  - low-level commands exposed to the user
- Integrity
  - cryptographic tracking of history (SHA-1 hashes)
  - tag signatures (GPG)
- Merging
  - pluggable merge strategies
  - staging area (index)
- Performance
  - no delta encoding

Git Commands

Version Control Layer	Local commands	**add** annotate apply archive bisect blame **branch** check-attr **checkout** cherry-pick **clean** **commit** **diff** filter-branch grep **help** **init** **log** **merge** **mv** notes rebase rerere **reset** revert **rm** shortlog show-branch **stash** **status** submodule **tag** whatchanged
	Sync with other repositories	**am** bundle **clone** daemon fast-export fast-import **fetch** **format-patch** http-backend http-fetch http-push imap-send mailsplit **pull** **push** quiltimport **remote** request-pull send-email shell update-server-info
	Sync with other VCS	archimport cvsexportcommit cvsimport cvsserver **svn**
	GUI	citool **difftool** **gitk** **gui** instaweb **mergetool**
VC Low-Level Layer		checkout-index check-ref-format cherry commit-tree **describe** diff-files diff-index diff-tree fetch-pack fmt-merge-msg for-each-ref fsck **gc** get-tar-commit-id ls-files **ls-remote** ls-tree mailinfo merge-base merge-file merge-index merge-one-file mergetool--lib merge-tree mktag mktree **name-rev** pack-refs parse-remotes patch-id prune read-tree receive-pack reflog replace rev-list rev-parse send-pack **show** **show-ref** sh-setup stripspace symbolic-ref update-index update-ref upload-archive **verify-tag** write-tree
Utilities		**config** var web--browse
Database Layer		cat-file count-objects hash-object index-pack pack-objects pack-redundant prune-packed relink repack show-index unpack-file unpack-objectsupload-pack verify-pack
Database (blobs, trees, commits, tags)		

Git GUIs: gitk → browsing the history



```
Modifications locales non enregistrées dans l'index et non commitées
master remotes/origin/master look for CoAP messages over UDP ports 5684 to   Anthony Baire <abaire@irisa.fr>  2012-11-29 12:42:47
Two fixes in the link-format parser                                        Anthony Baire <abaire@irisa.fr>  2012-11-29 12:26:28
  develop removed the check to enforce that there is no duplicate uri in a link pay  Anthony Baire <abaire@irisa.fr>  2012-11-29 12:21:11
  ensure that link-attribute values are not empty                          Anthony Baire <abaire@irisa.fr>  2012-11-29 11:59:12
iot2-v013 Support version 0.0.13 of the test specification                 Anthony Baire <abaire@irisa.fr>  2012-11-29 12:24:20
  updated OBS_02 to allow using the /obs-non resource instead of /obs      Anthony Baire <abaire@irisa.fr>  2012-11-29 11:49:41
  updated CORE_22                                                          Anthony Baire <abaire@irisa.fr>  2012-11-29 11:44:34
  updated doc strings to match the test spec v013                          Anthony Baire <abaire@irisa.fr>  2012-11-29 11:30:05
  Variour fixes in OBS testcases                                           Anthony Baire <abaire@irisa.fr>  2012-11-28 15:19:29
  OBS_05 fixed the matching of CON in notifications from the server        Anthony Baire <abaire@irisa.fr>  2012-11-28 15:12:19
  OBS_06 fixed the matching of CON in notifications from the server        Anthony Baire <abaire@irisa.fr>  2012-11-28 15:05:07
  CORE_08 fixed the matching of 2.04 response code                         Anthony Baire <abaire@irisa.fr>  2012-11-28 14:56:49
  fixed matching of 2.02 response code                                     Anthony Baire <abaire@irisa.fr>  2012-11-28 14:53:35
  OBS_07 allow deleting the resource from another client                   Anthony Baire <abaire@irisa.fr>  2012-11-28 14:52:40
  CORE_08: allow updating the resource from another client                 Anthony Baire <abaire@irisa.fr>  2012-11-28 14:49:59
  OBS_09: allow updating the resource from another client                  Anthony Baire <abaire@irisa.fr>  2012-11-28 14:45:10
fixed exception in OBS_09                                                  Anthony Baire <abaire@irisa.fr>  2012-11-28 14:35:43
```

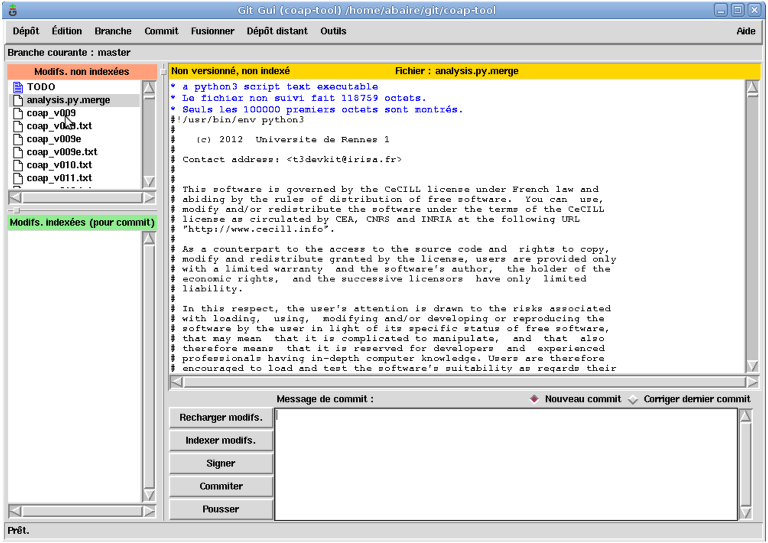
Id SHA1 : cc111e3afe454cc9a854970fa5623a385fc227e2 Colonne 15 / 172

Lignes de contexte: 3

```
Auteur: Anthony Baire <abaire@irisa.fr>  2012-11-28 14:52:40
Auteur du commit: Anthony Baire <abaire@irisa.fr>  2012-11-28 14:52:40
Parent: b66071fc98f84d66dd8024bdf853c095979c8871 (CORE_08: allow updating the resource from an
Enfant:  de43fa7574550759a7481eed2452b3ea0d7de5a1 (fixed matching of 2.02 response code)
Branche: develop, master, remotes/origin/master
Suit: iot2-v011
Précède: iot2-v013

    OBS_07 allow deleting the resource from another client

-------------------------------- analysis.py --------------------------------
index a2a67c5..0d9c732 100755
@@ -5212,20 +5212,20 @@ Notes:
                        # now we have successfully observed a observe response
                        verdict_if_none = None

-               # Step 8
-               self.match_coap ("client", CoAP (type="con", code="delete"))
-
-               self.setverdict ("pass" if uri == self.frame.coap.get_uri() else "inconc",
-                       "deleted resource should be the observed resource (%s)" % uri)
+               # Step 7
+               if self.match_coap ("client", CoAP (type="con", code="delete"),
+                                       None):
+
+                       self.setverdict ("pass" if uri == self.frame.coap.get_uri() else "inco
```

Commentaires
analysis.py

Git GUIs: `git gui` → preparing commits



```
* a python3 script text executable
* Le fichier non suivi fait 118759 octets.
* Seuls les 100000 premiers octets sont montrés.
#!/usr/bin/env python3
#
#   (c) 2012  Universite de Rennes 1
#
# Contact address: <t3devkit@irisa.fr>
#
#
# This software is governed by the CeCILL license under French law and
# abiding by the rules of distribution of free software.  You can  use,
# modify and/or redistribute the software under the terms of the CeCILL
# license as circulated by CEA, CNRS and INRIA at the following URL
# "http://www.cecill.info".
#
# As a counterpart to the access to the source code and  rights to copy,
# modify and redistribute granted by the license, users are provided only
# with a limited warranty  and the software's author,  the holder of the
# economic rights,  and the successive licensors  have only  limited
# liability.
#
# In this respect, the user's attention is drawn to the risks associated
# with loading,  using,  modifying and/or developing or reproducing the
# software by the user in light of its specific status of free software,
# that may mean  that it is complicated to manipulate,  and  that  also
# therefore means  that it is reserved for developers  and  experienced
# professionals having in-depth computer knowledge. Users are therefore
# encouraged to load and test the software's suitability as regards their
```

3rd party GUIs

- Turtoise git (Windows)
- Gitx (MacOS-X)
- Smartgit (java, multiplatform)
- Eclipse git plugin

Part 3. Working locally

- creating a repository
- adding & committing files
- the staging area (or index)

Create a new repository

`git init` *myrepository*

This command creates the directory *myrepository*.

- the repository is located in *myrepository/*`.git`
- the (initially empty) working copy is located in *myrepository/*

```
$ pwd
/tmp
$ git init helloworld
Initialized empty Git repository in /tmp/helloworld/.git/
$ ls -a helloworld/
.  ..  .git
$ ls helloworld/.git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Note: The `/.git/` directory contains your whole history,
⚠ **do not delete it**[5]

[5] unless your history is merged into another repository

Commit your first files

```
git add file

git commit [ -m message ]
```

```
$ cd helloworld
$ echo 'Hello World!' > hello
$ git add hello
$ git commit −m ”added file 'hello'”
[master (root−commit) e75df61] added file 'hello'
 1 files changed, 1 insertions(+), 0 deletions(−)
 create mode 100644 hello
```

Note: “master” is the name of the default branch created by `git init`

The staging area (aka the “index”)

Usual version control systems provide two spaces:

- the **repository**
  (the whole history of your project)

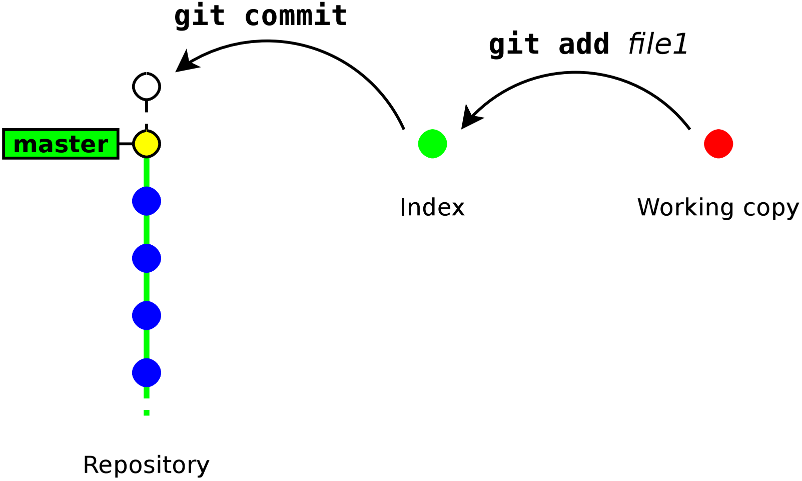
- the **working tree** (or **local copy**)
  (the files you are editing and that will be in the next commit)

Git introduces an intermediate space : the **staging area** (also called **index**)

The index stores the files scheduled for the next commit:

- `git add` *files* → copy files into the index
- `git commit` → commits the content of the index

The staging area (aka the “index”)



Update a file

```
$ echo 'blah blah blah' >> hello
$ git commit
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   hello
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git complains because the index is unchanged (nothing to commit)

→ We need to run `git add` to copy the file into the index

```
$ git add hello
$ git commit -m "some changes"
[master f37f2cf] some changes
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Bypassing the index[6]

Running `git add` & `git commit` for every iteration is tedious.

GIT provides a way to bypass the index.

```
git commit file1 [ file2 ...]
```

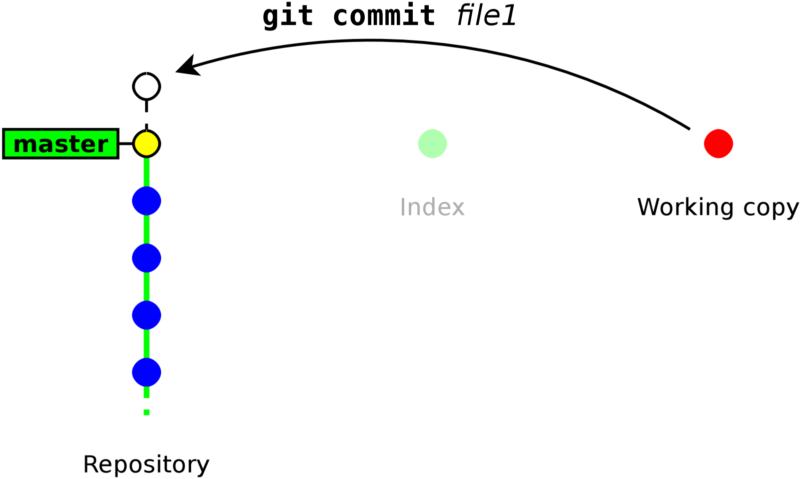
This command commits files (or dirs) directly from the working tree

Note: when bypassing the index, GIT ignores new files:

- "`git commit .`" commits only files that were present in the last commit (updated files)
- "`git add . && git commit`" commits everything in the working tree (including new files)

[6] also named "partial commit"

Bypassing the index



Deleting files

`git rm` *file*

→ remove the file from the index and from the working copy

`git commit`

→ commit the index

```
$ git rm hello
rm 'hello'
$ git commit -m "removed hello"
[master 848d8be] removed hello
 1 files changed, 0 insertions(+), 3 deletions(-)
 delete mode 100644 hello
```

Showing differences

`git diff` [*rev_a* [*rev_b*]] [-- *path* . . .]

→ shows the differences between two revisions *rev_a* and *rev_b* (*in a format suitable for the* `patch` *utility*)

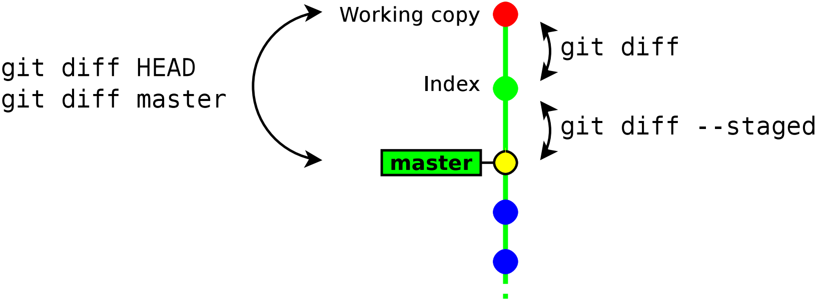
- by default *rev_a* is the **index**
- by default *rev_b* is the **working_copy**

`git diff --staged` [*rev_a*] [-- *path* . . .]

→ shows the differences between *rev_a* and the index

- by default *rev_a* is `HEAD` *(a symbolic references pointing to the last commit)*

About git diff and the index



Diff example

```
$ echo foo >> hello
$ git add hello
$ echo bar >> hello

$ git diff
--- a/hello
+++ b/hello
@@ -1,2 +1,3 @@
 Hello World!
 foo
+bar

$ git diff --staged
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World!
+foo

$ git diff HEAD
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 Hello World!
+foo
+bar
```

Resetting changes

```
git reset [ --hard ] [ -- path ...]
```

`git reset` cancels the changes in the index (and possibly in the working copy)

- `git reset` drops the changes staged into the index[7], but the working copy is left intact
- `git reset --hard` drops all the changes in the index **and** in the working copy

[7] it restores the files as they were in the last commit

Resetting changes in the working copy

`git checkout -- `*path*

This command restores a file (or directory) as it appears in the index (thus it drops all unstaged changes)

```
$ git diff HEAD
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 Hello World!
+foo
+bar
$ git checkout -- .
$ git diff HEAD
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World!
+foo
```

Other local commands

- `git status` → show the status of the index and working copy
- `git show` → show the details of a commit (metadata + diff)
- `git log` → show the history
- `git mv` → move/rename a file[8]
- `git tag` → creating/deleting tags (to identify a particular revision)

[8] note that `git mv` is strictly equivalent to: "`cp` *src dst* `&& git rm` *src* `&& git add` *dst*" (file renaming is not handled formally, but heuristically)

Exercises

1. create a new repository
2. create a new file, add it to the index and commit it
3. launch `gitk` to display it. Keep the window open and hit F5 after each command (to visualise the results of your commands)
4. modify the file and make a new commit
5. rename the file (try both ways, and do `git status` before committing)
   - using `git mv`
   - renaming the file manually, then using `git rm` and `git add`
6. delete the file and commit it
7. create two new files and commit them. Then modify their content in the working copy and display the changes with `git diff`
8. add one file into the index but keep the other one. Display the changes between:
   - the index and the working copy
   - the last commit and the index
   - the last commit and the working copy
9. run `git reset` to reset the index
10. run `git reset --hard` to reset the index and the working copy

Part 4.
Branching & merging

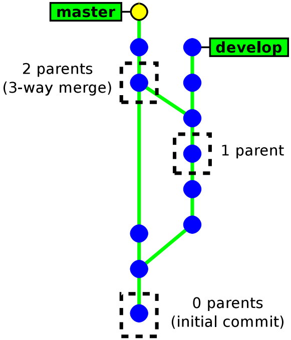
- How GIT handles its history
- Creating new branches
- Merging & resolving conflicts

How GIT handles its history

Each **commit** object has a list of **parent commits**:

- 0 parents → initial commit
- 1 parent → ordinary commit
- 2+ parents → result of a `merge`

→ This is a Direct Acyclic Graph



How GIT handles its history

- There is no formal “branch history”
  → a **branch** is just a pointer on the latest commit.
  (*git handles branches and tags in the same way internally*)

- Commits are identified with **SHA-1 hash** (160 bits) computed from:
  - the commited files
  - the meta data (commit message, author name, . . .)
  - the hashes of the parent commits

  → A commit id (hash) identifies **securely** and **reliably** its content and all the previous revisions.

Creating a new branch

```
git checkout -b new_branch [ starting_point ]
```

- *new_branch* is the name of the new branch
- *starting_point* is the starting location of the branch (possibly a commit id, a tag, a branch, . . .). If not present, git will use the current location.

```
$ git status
# On branch master
nothing to commit (working directory clean)
$ git checkout -b develop
Switched to a new branch 'develop'
$ git status
# On branch develop
nothing to commit (working directory clean)
```

Switching between branches

`git checkout [-m]` *branch_name*

```
$ git status
# On branch develop
nothing to commit (working directory clean)
$ git checkout master
Switched to branch 'master'
```

Note: it may fail when the working copy is not clean. Add `-m` to request merging your local changes into the destination branch.

```
$ git checkout master
error: Your local changes to the following files would be overwritten by
checkout: hello
Please, commit your changes or stash them before you can switch branches.
Aborting
$ git checkout -m master
M hello
Switched to branch 'master'
```

Merging a branch

`git merge` *origin_branch*

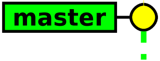
This will merge the changes in *origin_branch* into the current branch.

```
$ git status
# On branch master
nothing to commit (working directory clean)
$ git merge develop
 Merge made by recursive.
 dev    |     1 +
 hello  |     4 +++-
 2 files changed, 4 insertions(+), 1 deletions(-)
 create mode 100644 dev
```

Notes about merging

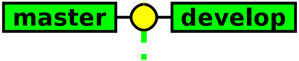
- The result of `git merge` is immediately committed (unless there is a conflict)
- The new commit object has **two parents**.
  → the merge history is recorded
- `git merge` applies only the changes since the last common ancestor in the origin branch.
  → if the branch was already merged previously, then only the changes since the last `merge` will be merged.

Branching example



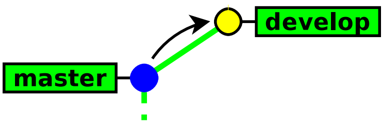
Branching example

```
git checkout -b develop
```



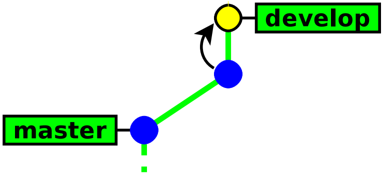
Branching example

```
git commit
```



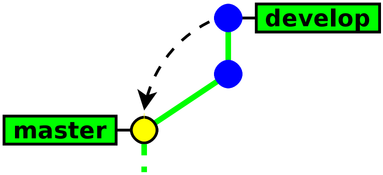
Branching example

```
git commit
```



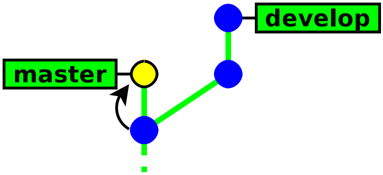
Branching example

```
git checkout master
```



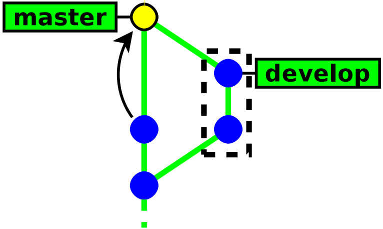
Branching example

```
git commit
```



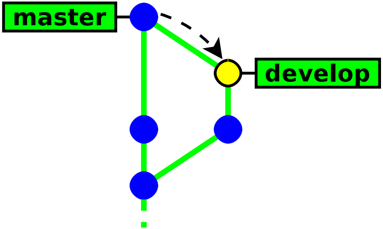
Branching example

```
git merge develop
```



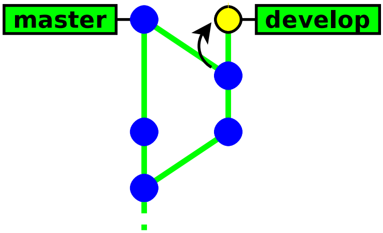
Branching example

```
git checkout develop
```



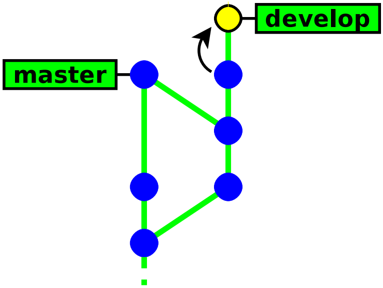
Branching example

```
git commit
```



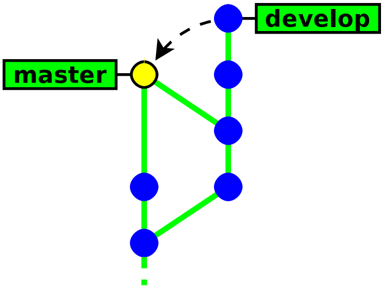
Branching example

```
git commit
```

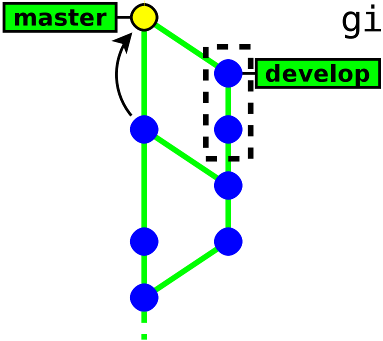


Branching example

```
git checkout master
```

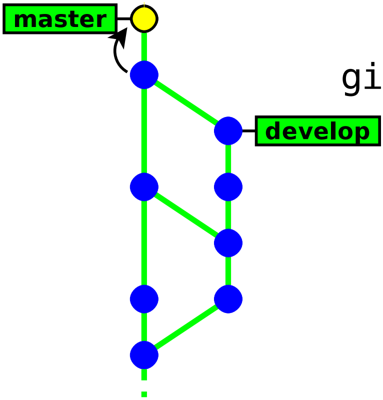


Branching example



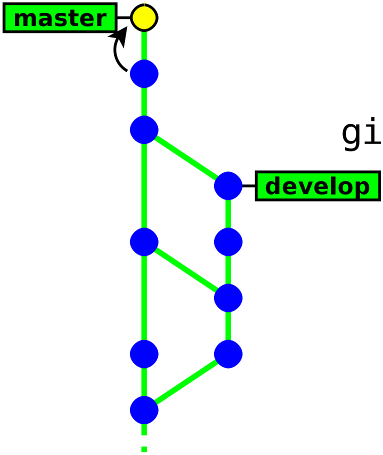
```
git merge develop
```

Branching example



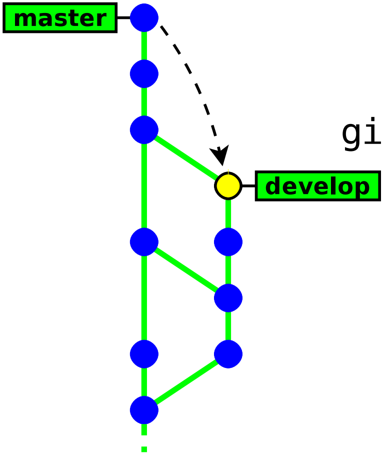
```
git commit
```

Branching example



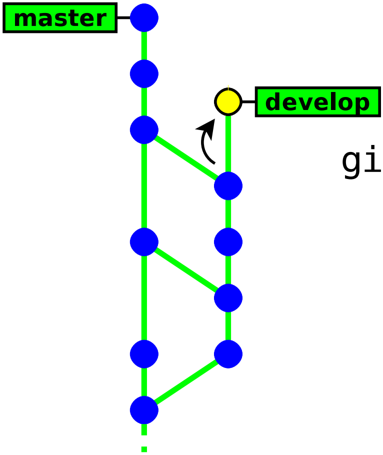
```
git commit
```

Branching example



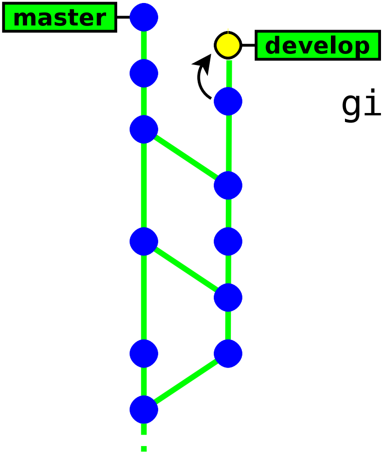
```
git checkout develop
```

Branching example



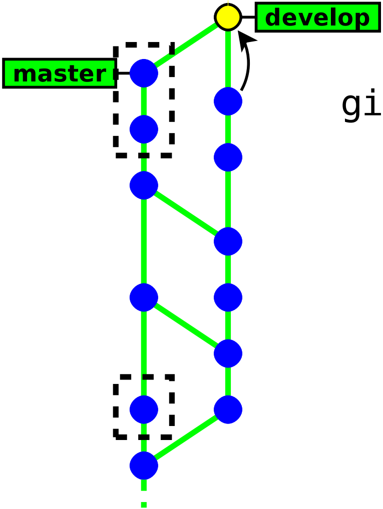
git commit

Branching example



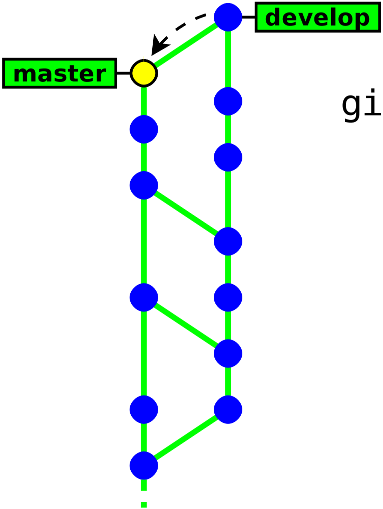
```
git commit
```

Branching example



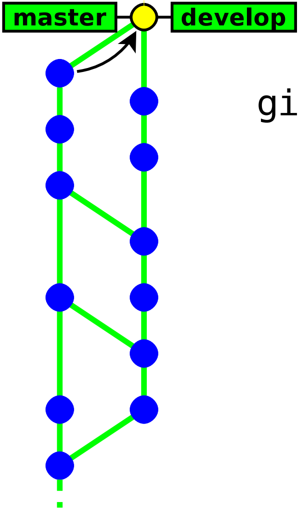
```
git merge master
```

Branching example



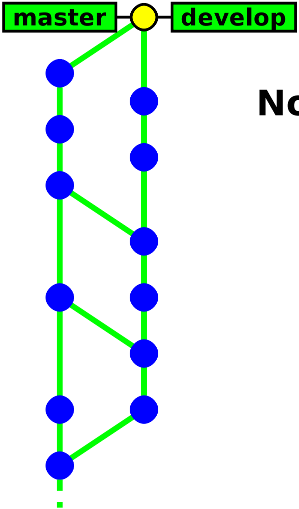
```
git checkout master
```

Branching example



```
git merge develop
```

Branching example



Note:

now the two branches share **exactly** the same history

How Git merges files ?

If the same file was independently modified in the two branches, then Git needs to merge these two variants

- **textual files** are merged on a per-line basis:
  - lines changed in only one branch are automatically merged
  - if a line was modified in the two branches, then Git reports a conflict. Conflict zones are enclosed within <<<<<<< >>>>>>>

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

- **binary files** always raise a conflict and require manual merging

Merge conflicts

In case of a conflict:

- **unmerged files** (those having conflicts) are left **in the working tree** and marked as "unmerged"[9]
- **the other files** (free of conflicts) and the metadata (commit message, parents commits, ...) are automatically added **into the index** (the staging area)

[9] Git will refuse to commit the new revision until all the conflicts are explicitely resolved by the user

Resolving conflicts

There are two ways to resolve conflicts:

- either edit the files manually, then run

  `git add` *file* → to check the file into the index

  or

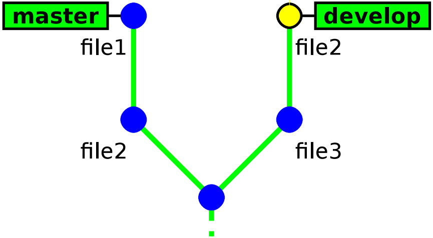
  `git rm` *file* → to delete the file

- or with a conflict resolution tool(xxdiff, kdiff3, emerge, ...)

  `git mergetool [` *file* `]`

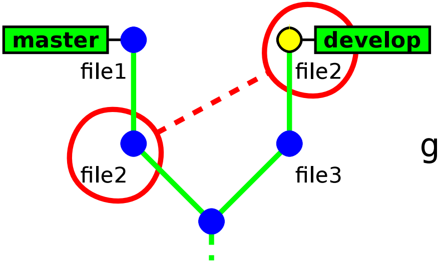
Then, once all conflicting files are checked in the index, you just need to run `git commit` to commit the merge.

Conflict example



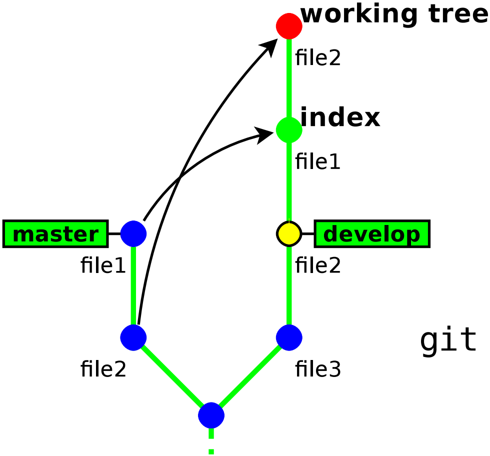
Conflict example

!! conflict !!



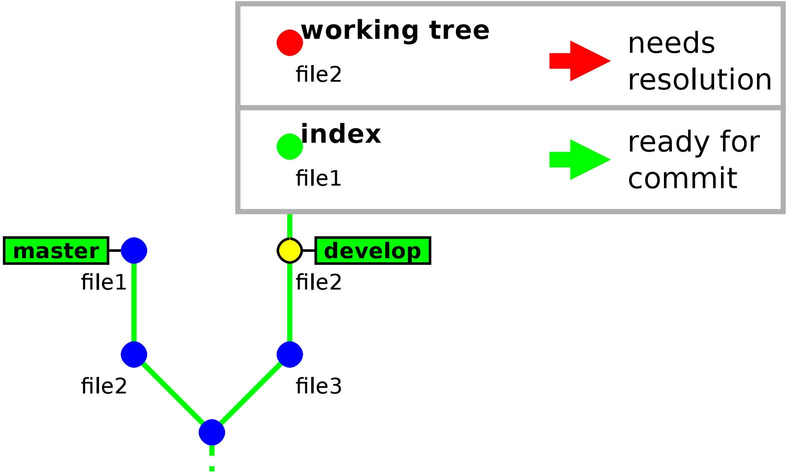
```
git merge master
```

Conflict example

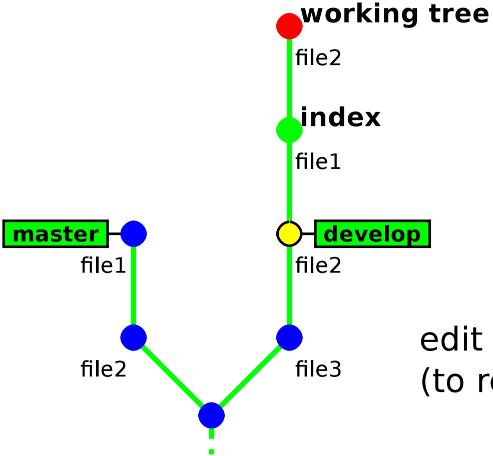


```
git merge master
```

Conflict example

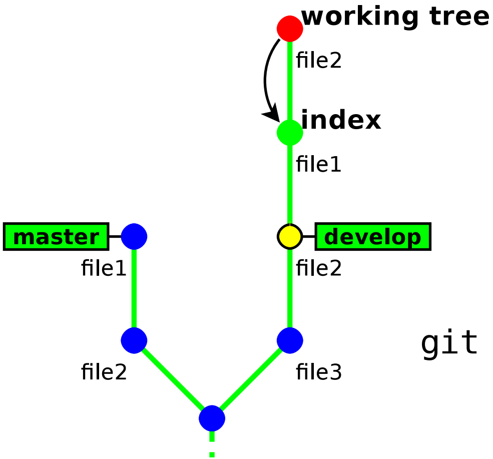


Conflict example



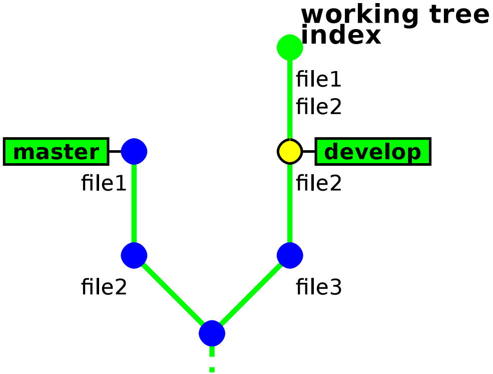
edit file2
(to resolve conflicts)

Conflict example

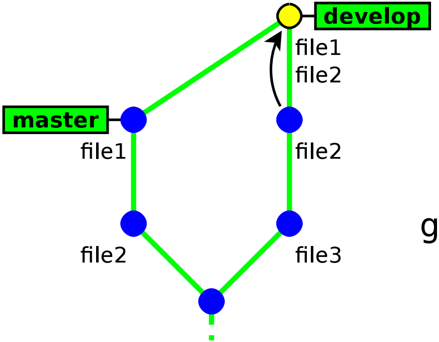


```
git add file2
```

Conflict example



Conflict example



```
git commit
```

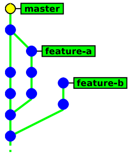
Deleting branches

`git branch -d` *branch_name*

This command has some restrictions, it cannot delete:

- the current branch (`HEAD`)
- a branch that has not yet been merged into the current branch

```
$ git branch -d feature-a
Deleted branch feature-a (was 45149ea).
$ git branch -d feature-b
error: The branch 'feature-b' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-b
'.
$ git branch -d master
error: Cannot delete the branch 'master' which you are currently on.
```



→ `git branch -d` is safe[10]

[10] unlike `git branch -D` which **deletes unconditionally** (⚠) the branch

Exercises

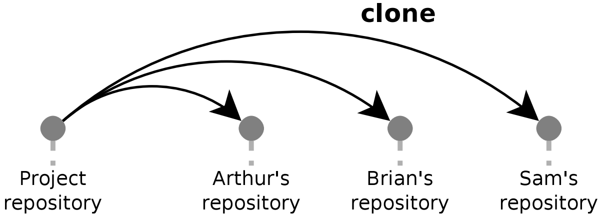
0. use “`gitk --all`” to display all branches
   (and remember to hit F5 after each command to visualise the changes)
1. create a new branch named “develop”
2. make some commits in this branch
3. go back to branch “master” and make some commits
4. merge branch “develop” into “master”
5. make a new commit in each branch so as to generate a conflict (edit the same part of a file)
6. merge branch “develop” into “master”, and fix the conflict
7. merge “master” into “develop”

Part 5.

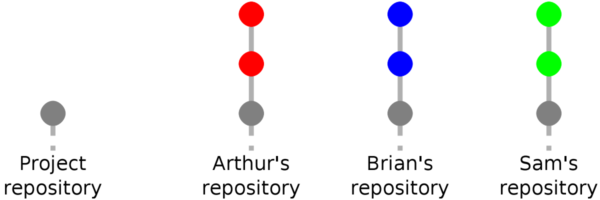
Interacting with a remote repository

- Overview
- Creating a shared repository
- Configuring a remote repository
- Sending changes (push)
- Receiving changes (pull)

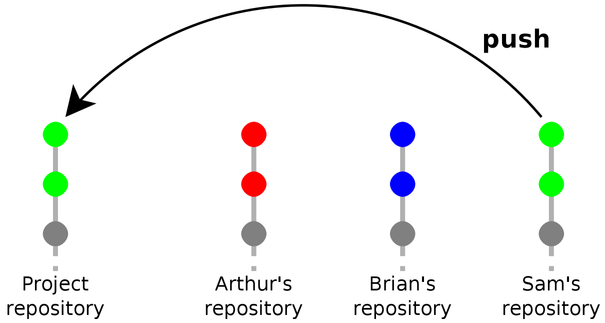
Team Workflow



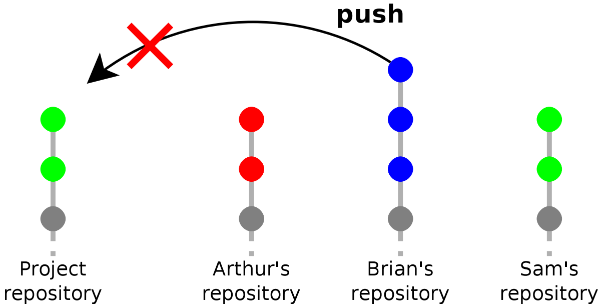
Team Workflow



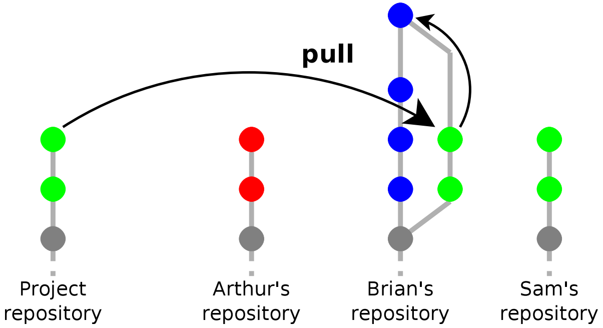
Team Workflow



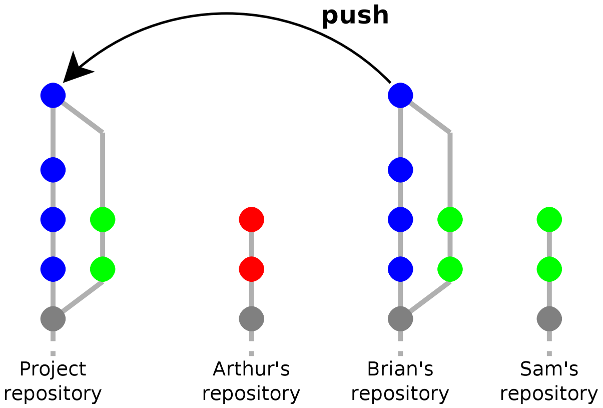
Team Workflow



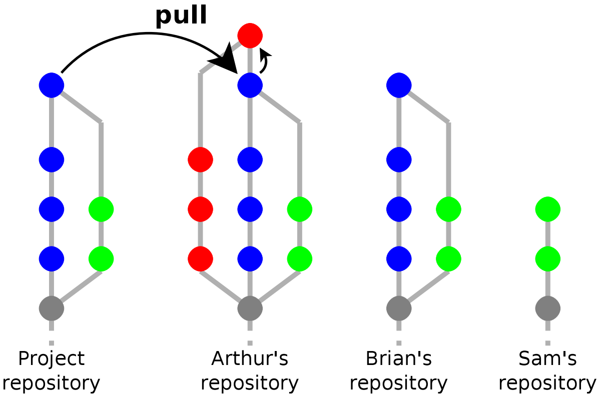
Team Workflow



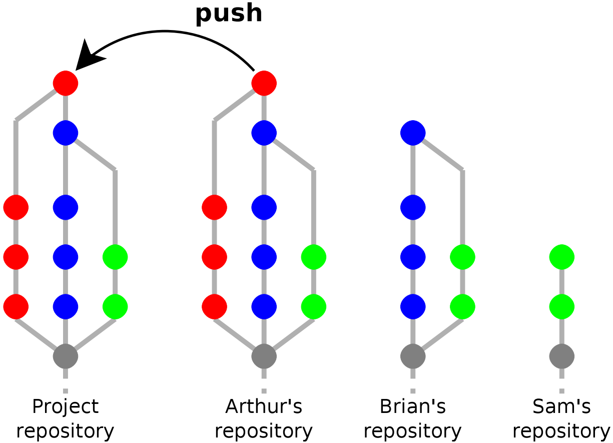
Team Workflow



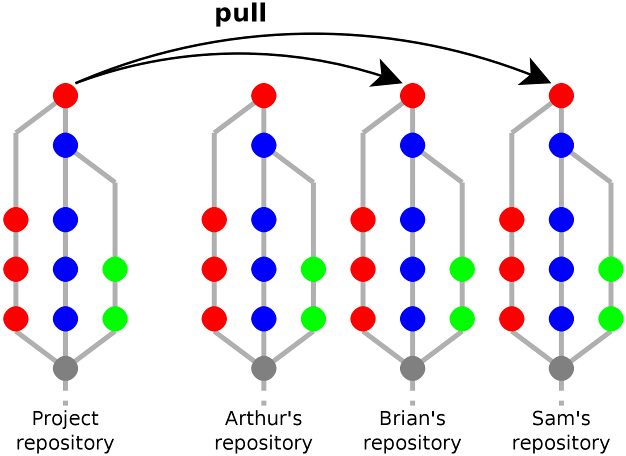
Team Workflow



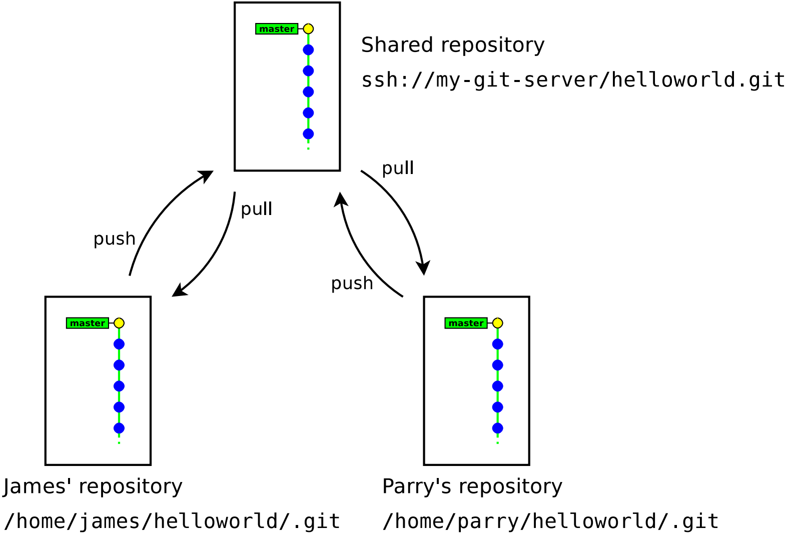
Team Workflow



Team Workflow



Simple workflow (Centralised)



Shared repository

`ssh://my-git-server/helloworld.git`

push

pull

pull

push

James' repository

`/home/james/helloworld/.git`

Parry's repository

`/home/parry/helloworld/.git`

How git handles remote repositories

- Remote repositories are mirrored within the local repository
- It is possible to work with multiple remote repositories
- Each remote repository is identified with a local alias. When working with a unique remote repository, it is usually named **origin**[11]
- Remote branches are mapped in a separate namespace: `remote/`*name*/*branch*.
  Examples:
  - `master` refers to the local `master` branch
  - `remote/origin/master` refers to the `master` branch of the remote repository named `origin`

[11] default name used by `git clone`

Adding a remote repository

```
git remote add name url
```

- *name* is a local alias identifying the remote repository
- *url* is the location of the remote repository

Examples:

```
$ git remote add origin /tmp/helloworld.git

$ git remote add origin ssh://username@scm.gforge.inria.fr/gitroot/helloworld/
helloworld.git
```

Pushing (uploading) local changes to the remote repository

```
git push [ --tags ]
```

- `git push` examines each branch, then:
    - if the branch exists both locally **and** remotely, then the local changes (commits) are propagated to the remote branch
    - if not, then nothing is uploaded
      (thus new local branches are considered private)

- In case of conflict `git push` will fail and require to run `git pull` first

- Tags are not uploaded by defaut. Runninng `git push --tags` will upload **all** local tags to the remote repository

Pushing a new branch to the remote repository

`git push -u` *destination_repository refspec*

- explicit variant of `git push`: the local reference *refspec* (a branch or a tag) is pushed to the remote *destination_repository*
- `-u` will set up an **upstream relationship** between the local and the remote branch so that remote changes are merged when running `git pull` (this is usually what you want)

```
$ git push
No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
error: failed to push some refs to '/tmp/helloworld.git/'
$ git push -u origin master
To /tmp/helloworld.git/
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Fetching (downloading) changes from the remote repository

```
git fetch
```

`git fetch` updates the local mirror of the remote repository:

- it downloads the new commits from the remote repository
- it updates the references `remote/`*remote_name/** to match their counterpart in the remote repository.

  Example: the branch `remote/origin/master` in the local repository is updated to match the new position of the branch `master` in the remote reposity

Merging remote changes into the current local branch

Changes in the remote repository can be merged explicitely into the local branch by running `git merge`

```
$ git status
# On branch master
$ git fetch
...
$ git merge origin/master
```

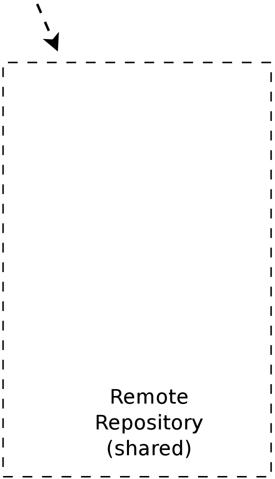
In practice, it is more convenient to use `git pull`, which is an alias to `git fetch` + `git merge`

`git pull`

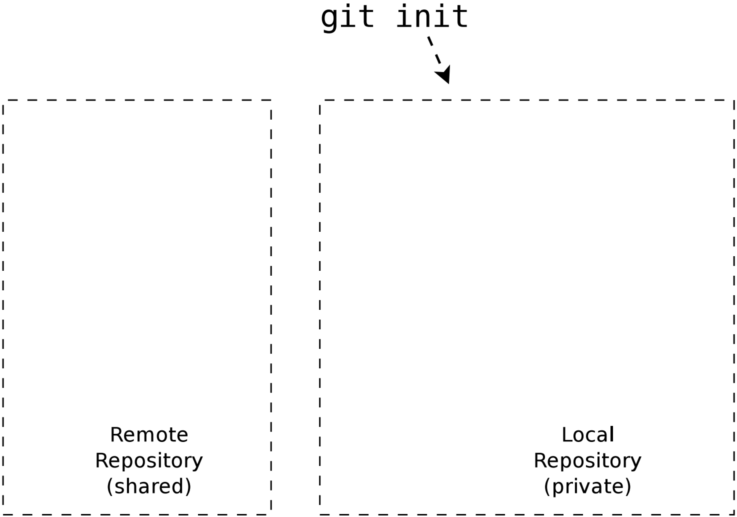
```
$ git pull
```

Remote example

```
git init --bare --shared
```



Remote example



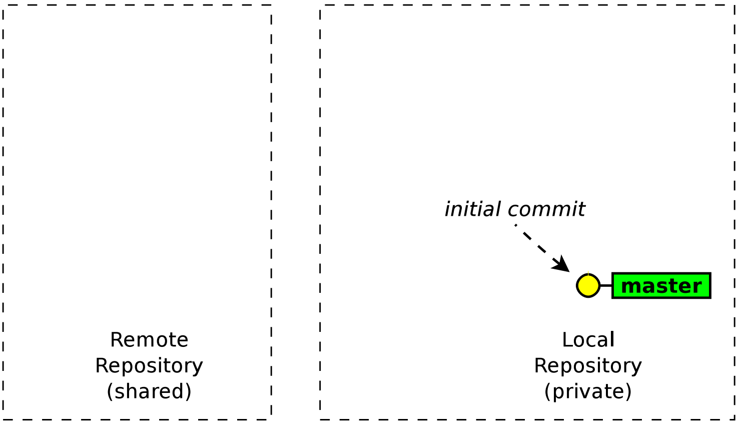
git init

Remote Repository (shared)

Local Repository (private)

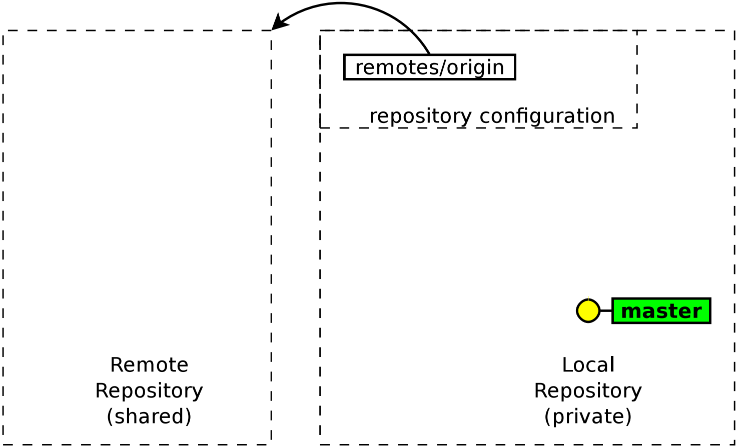
Remote example

```
git commit
```



Remote example

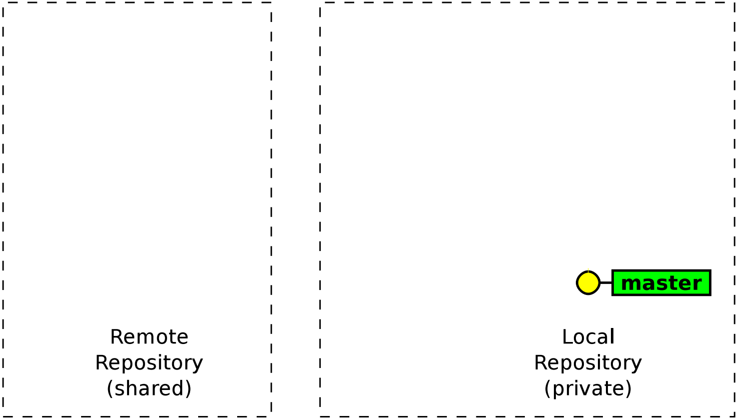
```
git remote add origin shared_url
```



Remote example

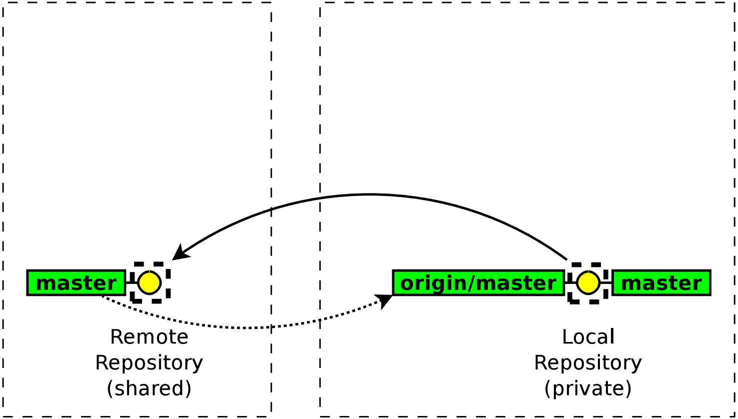
```
git push
```

-> nothing to be pushed !!



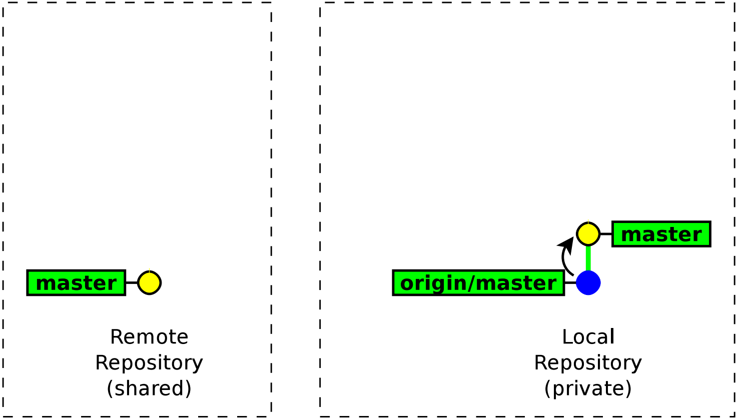
Remote example

```
git push -u origin master
```



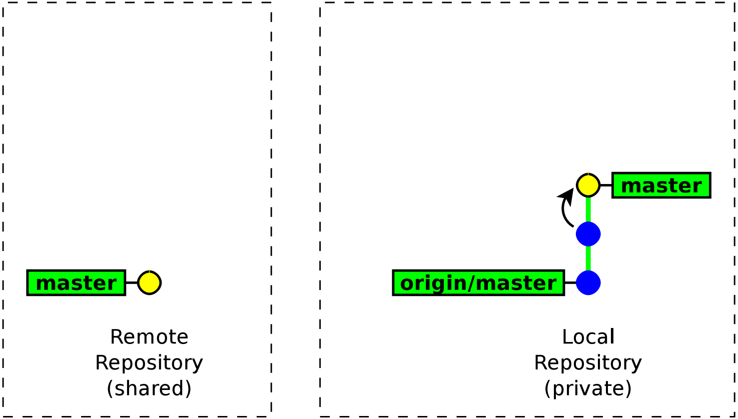
Remote example

```
git commit
```



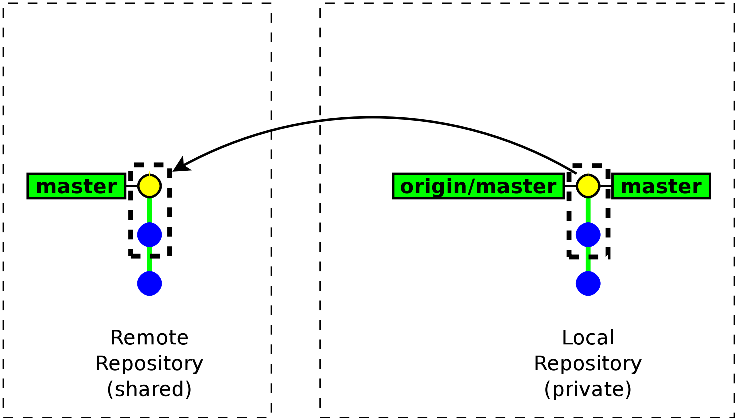
Remote example

```
git commit
```



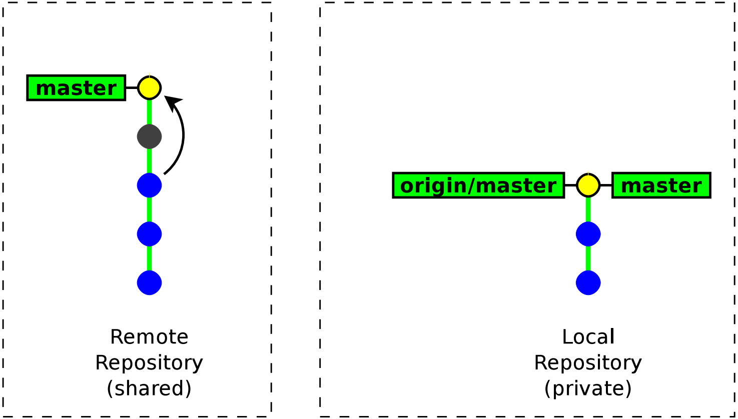
Remote example

```
git push
```



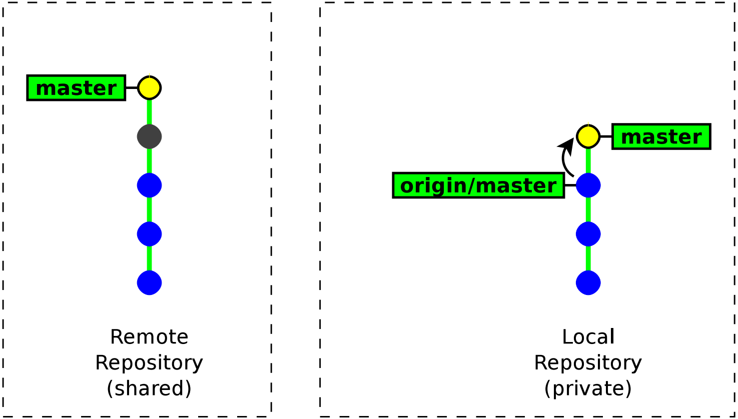
Remote example

another developer pushes his two commits



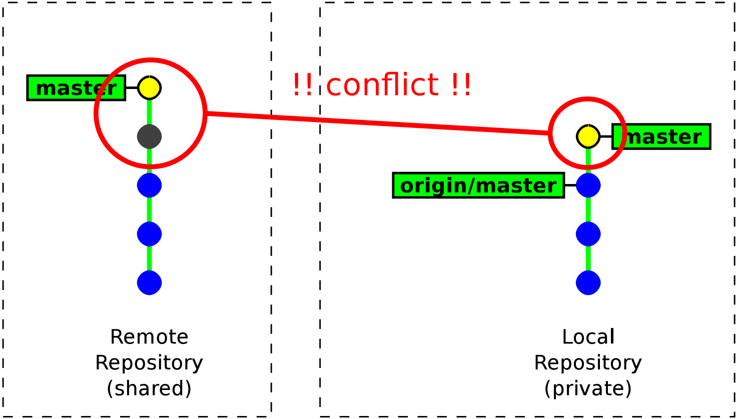
Remote example

```
git commit
```



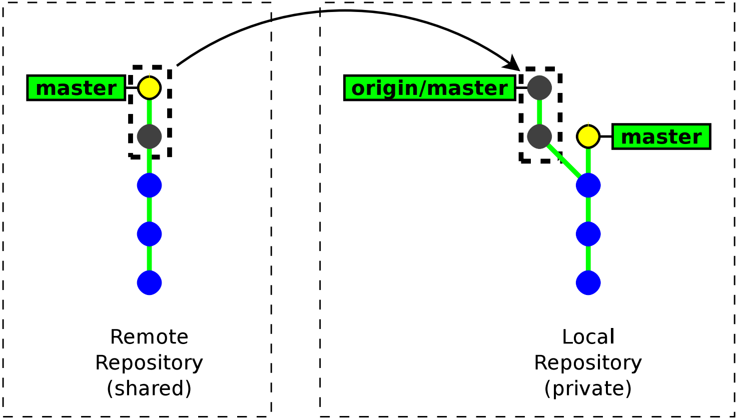
Remote example

```
git push
```



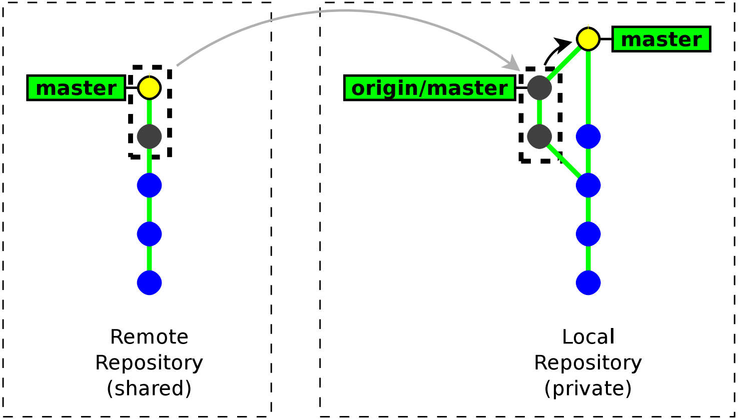
Remote example

`git fetch`



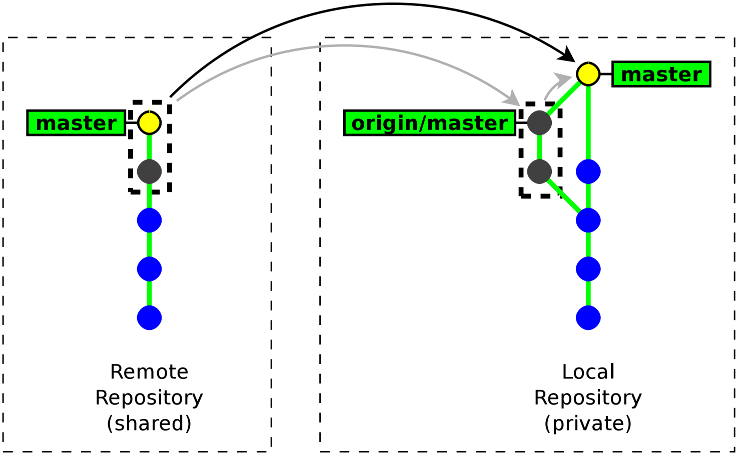
Remote example

```
git merge origin/master
```



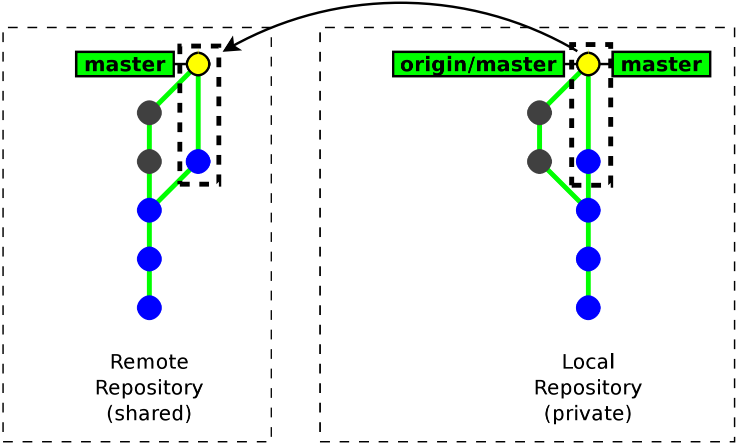
Remote example

```
git pull
```



Remote example

```
git push
```



Importing a new remote branch

`git checkout` *branch_name*

If the *branch_name* does not exist locally, then GIT looks for it in the remote repositories. If it finds it, then it creates the local branch and configures it to track the remote branch.

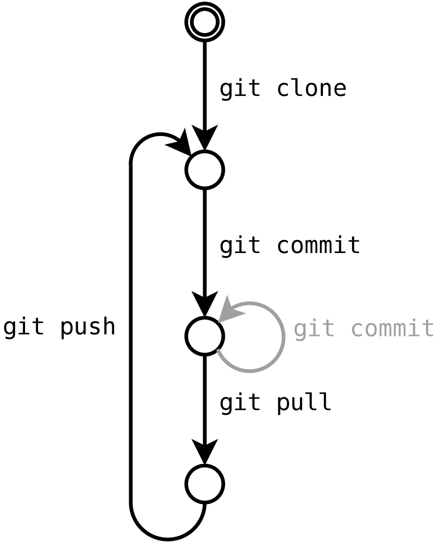
```
$ git branch --all
* master
  remotes/origin/master
  remotes/origin/new-fancy-feature
$ git checkout new-fancy-feature
Branch new-fancy-feature set up to track remote branch new-fancy-feature from
origin.
Switched to a new branch 'new-fancy-feature'
$ git branch  --all
  master
* new-fancy-feature
  remotes/origin/master
  remotes/origin/new-fancy-feature
```

Cloning a repository

```
git clone url [ directory ]
```

- `git clone` makes a local copy of a remote repository and configures it as its `origin` remote repository.
- `git clone` is a shortcut for the following sequence:
  1. `git init` *directory*
  2. `cd` *directory*
  3. `git remote add origin` *url*
  4. `git fetch`
  5. `git checkout master`
- In practice you will rarely use `git init`, `git remote` and `git fetch` directly, but rather use higher-level commands: `git clone` and `git pull`.

Typical Workflow



Exercises

0. *(remember to visualise your operations with "*`gitk --all`*" → hit F5)*
1. clone the following repository
   `ssh://`*username*`@senslab2.irisa.fr/git/helloworld.git`
2. use `gitk --all` (to display remote branches too)
3. make some commits and synchronise (`pull`/`push`) with the origin repository
4. do it again so as to experience and resolve a conflict
5. use `git fetch` to review remote commits **before** merging them
6. create a new branch, make a commit and publish it to the shared repository
7. check out a branch created by another participant

Part 6.
Administrating a server

- Creating a shared repository
- Protocols
- Common workflows

Creating a shared repository

`git init --bare --shared` *my-shared-repository*`.git`

- A bare repository (`--bare`) is a repository without any working copy.
  - by convention bare repositories use the `.git` extension
  - bare repository are updated by importing changes from another repository (push operation)
- `--shared` is meant to make this repository group-writable (unix group)

```
$ git init --bare --shared helloworld.git
Initialized empty shared Git repository in /tmp/helloworld.git/
$ ls helloworld.git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Admin Considerations

Administrating a GIT server is relatively simple[12]

- no partial access
  (access is granted to the full repository)
- no access policies in GIT itself
  (access control to be handled by the HTTP/SSH server)
- low server load
  (most operations are local)
- server outages are much less disruptive
  (user can collaborate by other means)
- only core developers need write access

[12] compared to centralised Version Control systems

How to publish a GIT repository (1/2)

- GIT daemon (TCP port 9418)
  - very efficient
  - public access only, no authentication

→ `git://server.name.org/path/to/the/repository.git`

- GIT over SSH
  - very efficient
  - strong authentication & encryption
  - restricted shell possible with `git-shell`

→ `ssh://username@server.name.org/path/to/the/repository.git`

- Local access

→ `/path/to/the/repository.git`

How to publish a GIT repository (2/2)

- Dumb HTTP/HTTPS server (read-only)
  - very easy to set up (static content[13] only)
  - less efficient
  - firewall friendly
  - can provide SSL authentication/encryption, even for anonymous users

→ `http://username@server.name.org/path/to/the/repository.git`

- Webdav HTTP/HTTPS server (read-write)
  - same as the dumb server, but with write access
  - more complex to set up

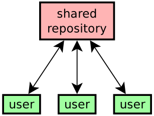
[13] Note that it requires generating some index files with `git update-server-info` after every update, this can be automated in the **post-update** hook

GIT-centric forges

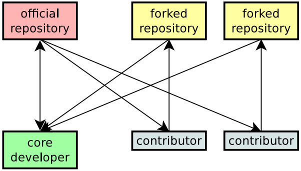
- Hosting only
  - GitHub
    `https://github.com/`
  - BitBucket
    `https://bitbucket.com/`
  - Google Code
    `https://code.google.com/`
- Open source software
  - Gitlab
    `http://gitlab.org`
  - Gitorious
    `http://gitorious.org`

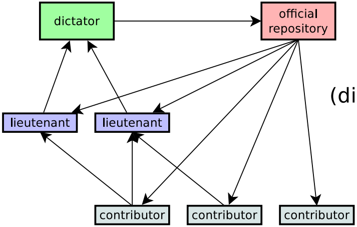
Common workflows

Centralised



Decentralised





Hierarchical
(dicatator-lieutenants)

more about workflows at: https://www.atlassian.com/git/workflows

Part 7.
Extras

- Some advices
- Common traps
- Documentation
- Next tutorial

Some advices (1/2)

- commit as often as you can (keep independent changes in separate commits)
- run `git diff` before preparing a commit
- in commit messages, describe the rationale behind of your changes (it is often more important than the change itself)
- do not forget to run `git push`

Some advices (2/2)

- don't be fully desynchronised → run `git pull` enough often to avoid accumulating conflicts
- idem for feature branches
(merge from the mainstream branch enough often)
- when creating complex patches (as an external contributor) prefer using one branch per patch
- keep a `gitk` instance open when doing fancy things

Common traps (1/2)

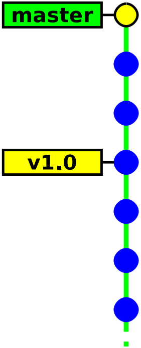
- `git diff` without arguments shows the difference with the index → run `git diff HEAD` to show the differences with the last commit

- `git reset` reverts the index, but keeps the working copy unchanged
  → do `git reset --hard` if you need to revert the working copy too

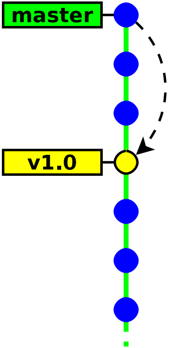
Common traps (2/2)

- GIT is not forgiving, do not ignore its warnings and do not use `--force` unless you have a clear idea of what you are doing
- GIT's history is not immutable
- `git checkout` on an arbitrary commit or a tag (anything that is not a branch) puts your in "detached HEAD" state. You can commit, but your history be lost if you don't create any branch (or tag) to reference them.

Detached head state

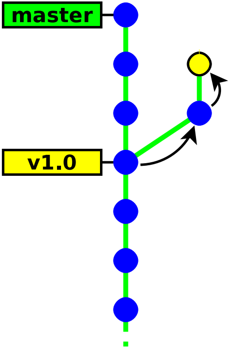


Detached head state



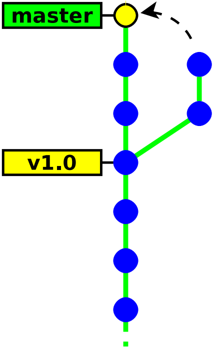
```
git checkout v1.0
```

Detached head state



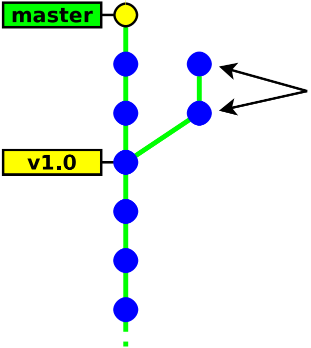
```
git commit
git commit
```

Detached head state



```
git checkout master
```

Detached head state



Git's garbage collector will discard these two commits

(they are not referenced by any branch/tag)

Other useful utility commands

- `git gc` → garbage collector (run it when the `/.git/` directory takes too much space)
- `git stash` → save/restore the state of the working copy and index (useful when in need to commit an urgent fix)
- `git clean` → clean the working tree (⚠you must ensure that all your code is committed)
- `git bisect` → locating which commit introduced a bug
- `git cherry-pick` → merging a single commit
- `git revert` → cancelling a previous commit

Further documentation

- `man git` *cmd* (tough & exhaustive)
- `man gitglossary`
- The Git book

  `http://git-scm.com/book`
- The Git community book

  `http://www.scribd.com/doc/7502572/The-Git-Community-Book`
- Github learning materials

  `http://learn.github.com/`
- Atlassian learning materials

  `https://www.atlassian.com/git/tutorial`

  `https://www.atlassian.com/git/workflows`

Next tutorial

Next tutorial sessions: “Git for advanced users”

- git internals
- rewriting the history
- playing with your index
- handling dependencies between repositories
- maintaining a set of patches
- interacting with other tools (SVN, Mercurial)