Conduite de Projet
Cours 2 — Version Control

Stefano Zacchiroli
zack@irif.fr

Laboratoire IRIF, Université Paris Diderot

2017–2018

URL https://upsilon.cc/zack/teaching/1718/cproj/

Sommaire

Outline

Change

During the life time of a software project, *everything* changes:

- bugs are discovered and have to be fixed (code)
- system requirements change and need to be implemented
- external dependencies change
  - e.g. new version of hardware and software you depend upon
- competitors might catch up

Most software systems can be thought of as a set of evolving versions

- potentially, each of them has to be maintained concurrently with the others

Configuration management

Definition (Configuration Management)

Configuration Management (CM) is concerned with the policies, processes, and tools for managing changing software systems.

(Sommerville)

Why?

- it is easy to lose track of which changes have been incorporated in each version
  - things get even messier with versions which have to be maintained in parallel
- minimize risks of working on the wrong version
- useful for *solo* projects ⇒ backup on steroids + it's easy to forgot which change has been made and why
- useful for team project ⇒ help in praising(, blaming), know who to ask

Configuration management activities

Change management keep track of request for changes (from both customers and developers), evaluate costs/risks/benefits, making committment to change

Version management (or *version control*, revision control, etc.) keeping track of multiple version of (software) components and ensure unrelated changes do not interfere

System building assembling program components, data, and libraries into executable systems

Release management preparing software for external release and keep track of which version is in use at which customer site

Configuration management activities

Version management (or *version control*, revision control, etc.)
keeping track of multiple version of (software) components and ensure unrelated changes do not interfere

Outline

Before version control: diff & patch

The Swiss army knife of change management: diff & patch

diff compute the difference D among a file A and a file B

- can be applied recursively to directories

patch apply a difference D (usually computed using diff) to a file A (possibly producing a new file B)

Demo

diff & patch — discussion

- patches are (were) usually conveyed via email messages to the main software maintainer
- best practices
  - add to emails clear and concise explanations of the purpose of the attached patch
  - do the same in the source code added by the patch
    - nothing new: usual good coding practice; it becomes more important only because the number of software authors grows...)
  - `http://tldp.org/HOWTO/Software-Release-Practice-HOWTO/patching.html`

Poor man's version control

Projects by a license student often look like this:

```
lucien> ls
a.out
projet.ml
projet-save.ml
projet-hier.ml
projet-marche-vraiement.ml
projet-dernier.ml
```

- what are the differences among the 5 source files?
- what are the *relationships* among them?
- hard to answer without specific utilities

Poor men's version control (plural)

Project by *a group* of license students:

```
lucien> ls ~joel/projet
a.out
module.ml
module-de-julien-qui-marche.ml
projet.ml
projet-save.ml
projet-hier.ml
projet-marche-vraiement.ml
projet-dernier.ml
```

```
lucien> ls ~julien/projet
a.out
module.ml
projet.ml
projet-recu-de-joel.ml
module-envoye-a-joel.ml
```

What is the right combination of projet.ml and module.ml to obtain a good grade at the exam?

diff & patch to the rescue

To exchange projet.ml and module.ml a group of students can rely on emails, diff, and patch (a huge improvement!)

Julien

```
lucien> diff -Nurp projet-hier.ml projet.ml > mescorrections
lucien> mail -s "Voici mes modifs" joel@lucien < mescorrection
```

Joel

```
lucien> mail
Mail version 8.1.2 01/15/2001. Type ? for help.
> 1  julien@home Fri Sep 13 20:06   96/4309 voici mes modifs
& s 1 /tmp/changes
& x
lucien> patch < /tmp/changes
```

Julien's changes between projet-hier.ml and projet.ml are now integrated in Joel's copy of projet.ml (hoping no conflicting changes have been made by Joel...)

diff & patch: except that...

Nonetheless, on exam day nothing works, although it worked just the day before. *Panicking*, you'll try to understand:

- what has changed
- who did the change
  - probably you don't care about why, but still...
- when it has been done
- which state, not including that change, works properly
- how to get back to that state

⇒ you (badly) need a *real* Version Control System

Outline

Version Control System (VCS)

A version control system

- manage specific artifacts which form your source code
  - files, directories, their attributes, etc.
- is able to store changes to those artifacts (a VCS implements the notion of version for source code)
  - who has done a change
  - wrt which state
  - why
  - when
- can show the differences among different (stored) states
- can go back in time and restore a previous state
- can manage concurrent work among developers, distributing the changes among them

Basic VCS concepts

A few basic concepts are shared across VCSs: [1]

revision (or **version**) a specific state, or point in time, of the content tracked by the VCS

- granularity and scope vary

history a set of revisions, (partially) ordered

1. although the actual naming changes from system to system; we'll stick to the naming presented here

Basic VCS concepts (cont.)

A few basic concepts are shared across VCSs: [1]

repository (or depot) where the tracked content and all its history, as known to the VCS, is stored

- might be local or remote

working copy a local copy of a revision, which might be acted upon

- where the "real" work happens

checkout (or clone) the action of creating a working copy from a repository

1. although the actual naming changes from system to system; we'll stick to the naming presented here

Basic VCS concepts (cont.)

A few basic concepts are shared across VCSs: [1]

change (or delta) a specific modification to (or *with respect to*) the content tracked by the VCS

- granularity vary

commit (as a verb) the act of writing a change performed in the working copy back to the repository

- = adding a new revision to the history

commit (as a substantive) same as change, for changes that have been committed

diff the act of (or the result of) inspecting the differences among two revisions, or among a revision and the working copy

- inspection format is usually diff

1. although the actual naming changes from system to system; we'll stick to the naming presented here

Branching and merging

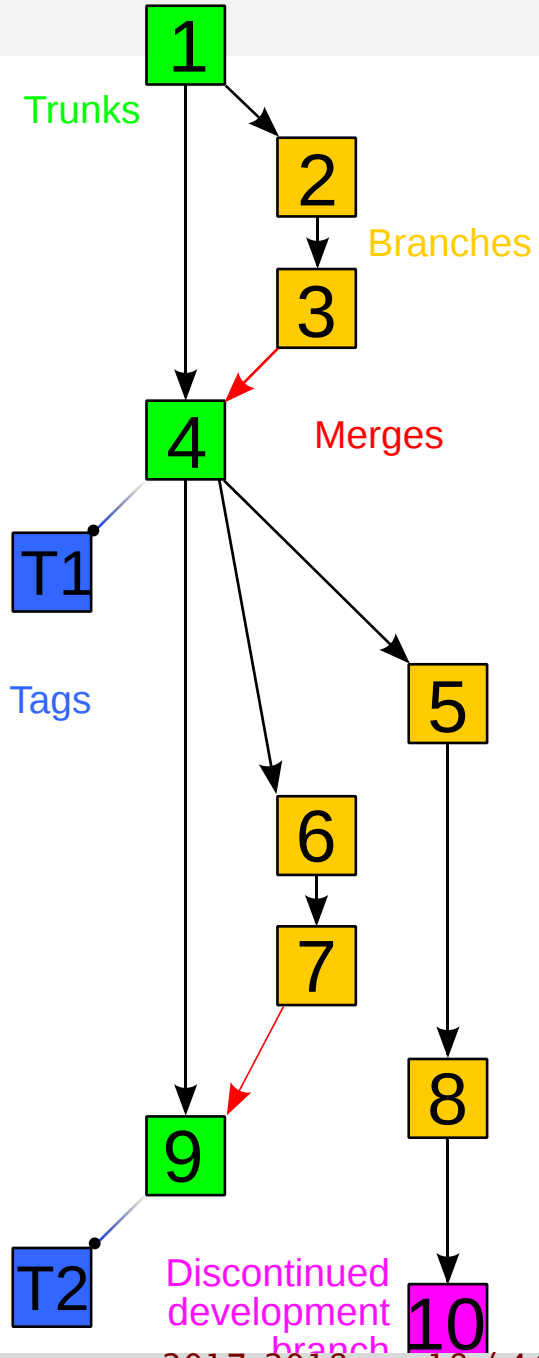
branch (verb) the act of duplicating (or "forking") a specific revision in history, to open up a new line of development

- branches are usually named

branch (substantive) subset of history rooted at a fork point and extending until the next merge point

merge (verb) the act of joining together multiple lines of development, reconciling all their changes together

merge (substantive) the point in history where the merge happens



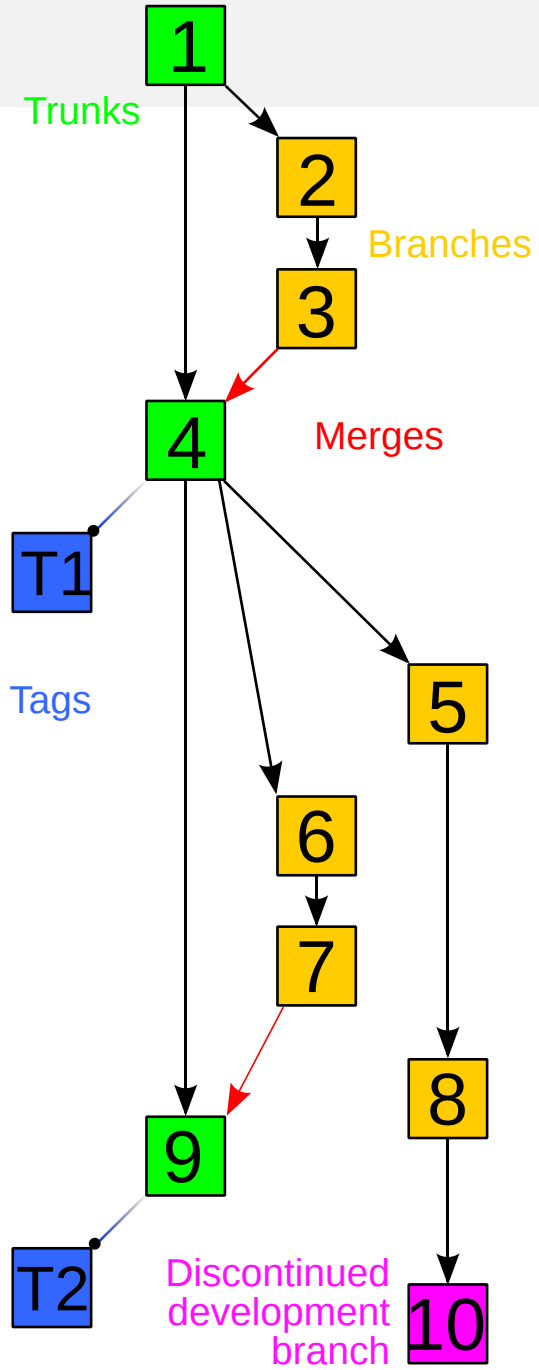
http://en.wikipedia.org/wiki/File:

Branching and merging (cont.)

- assuming an idealized purely functional model, content history can then be depicted as a direct acyclic graph
- parallel changes may or may not be compatible. . .

conflict the situation occurring when, upon a merge attempt, changes from involved branches cannot be reconciled

solving a conflict means applying extra changes to combine non (automatically) reconcilable changes or choose a subset of them



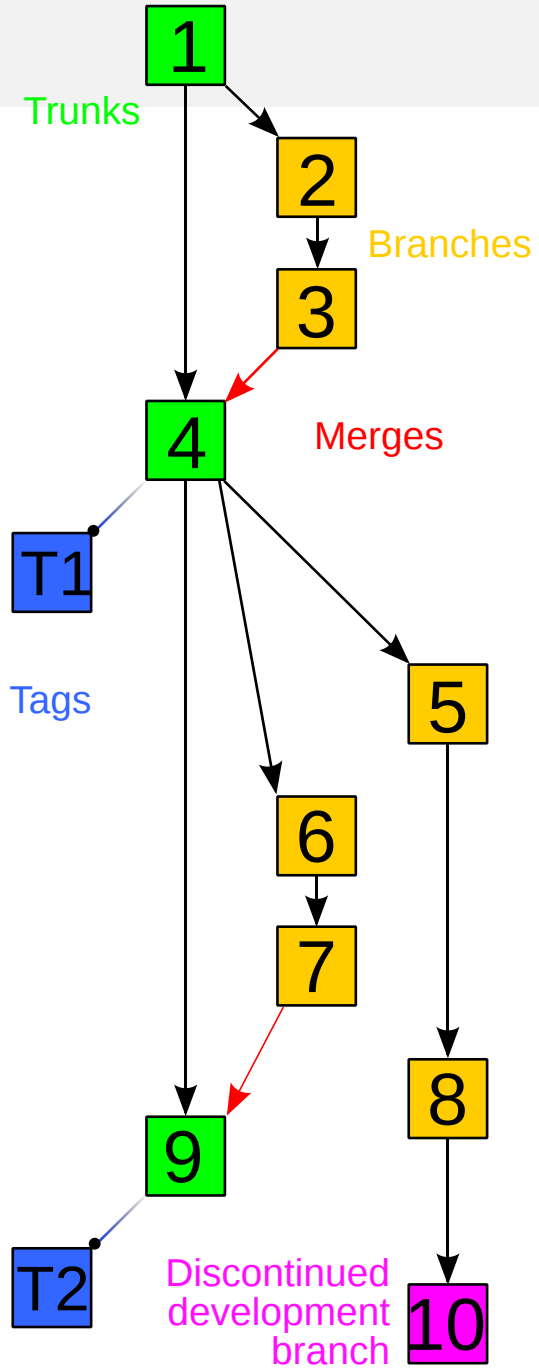
Branching and merging (cont.)

tag (or label) a symbolic name attached to a particular revision in history

head (or tip) the (moving) tag always associated to the most recent commit; might be limited to a specific "special" branch, such as:

trunk (or master) the unique line of development which is not a branch

- peculiar: treating a specific branch as special is not necessary for the idealized model to work



Brief history of VCSs

- 1972 SCCS (Source Code Control System), commercial (AT&T) UNIX-es, part of the Single UNIX Specification; scope: file; modern clone (for compatibility only): `cssc`
- 1982 RCS (Revision Control System) GNU-based UNIX-es; scope: file; Free-er and generally considered more evolved than SCCS, currently maintained by the GNU Project
- 1990 CVS (Concurrent Version System), client-server paradigm; scope: set of files
- late 1990's TeamWare, BitKeeper; early attempt at distributed version control system; proprietary
- 2000 Subversion (SVN); client-server, addressing many defects of CVS
- 2001– Distributed VCS (DVCS) golden age: GNU arch (2001), Darcs (2002), SVK (2003), Monotone (2003), Git (2005), Mercurial (2005), Bazaar (2005) more on this later...

Outline

Revision Control System (RCS)

- one of the oldest system (1982)
- typical of the commercial UNIX era
- scope: single file
  - although the repositories for several files can be stored in the same “shared” place
- repository
  - `file,v` where `file` is the original name of the checked in file
  - changes are stored as incremental reverse diffs
  - minimization of secondary storage: delta compression is triggered by deletion of intermediate revisions
- concurrency model:
  - pessimistic approach: one have to acquire explicit locks before making modification; by default working copies are read-only
  - as the working copy is shared among users, this enforced a rather heavy mutual exclusion discipline

RCS — basic operations

commit	ci FILE	(without lock)
	ci -l FILE	(with lock)
checkout	co FILE	(without lock)
	co -l FILE	(with lock)
diff	rcsdiff -rVERSION1 -rVERSION2 FILE	
history	rlog FILE	
acquire lock	rcs -l FILE	

RCS — basic operations

Demo

RCS — branching and merging

Versions in RCS are trees, where branches are reflected in the syntax of versions. “Minor” version numbers are increased automatically by RCS upon commit; “major” numbers can be specified explicitly by the user upon commit.

- history with single branch

```
(1.1) -> (1.2) -> (1.3) -> (1.4) -> (2.1) -> (2.2)
```

- history with multiple branches

```
(1.1) -> (1.2) -> (1.3) -> (1.4) -> (2.1) -> (2.2)
                      \
                       ----> (1.3.1.1)
```

RCS — branching and merging operations

branch ci -rVERSION FILE
example: ci -r2 foo.ml

branch checkout co -rVERSION FILE

merge rcsmerge -p -rVERSION1 -rVERSION2 FILE > RESULT

- preforms a 3-way diff (a-la `diff3`) among old (common) VERSION1, and the two new versions: VERSION2 and the current state of FILE

Example

```
rcsmerge -p -r1 -r3 foo.ml > foo.ml.new
```

merges the differences among branch 1 and 3 of foo.ml, with differences among branch 1 and the current version of foo.ml; save the result to foo.ml.new

RCS — branching and merging operations

Demo

Outline

Concurrent Versions System (CVS)

- a significant (r)evolution in the history of VCS
- designed to address the (too) constraining mutual exclusion discipline enforced by RCS (hence the emphasis on *concurrent*)
- client-server model
  - enforce decoupling of repository and working copy
  - several working copies exist—generally one for each developer—and can be acted upon independently
  - commands and processes to:
    - "push" local changes from working copies to the repository
    - "pull" changes (made by others) from the repository and merge it with the local (uncommitted) changes
    - deal with conflicts and try to avoid they hit the repository
  - note: the repository is used as the orchestrator and as the sole data storage
    - "local commits" are not possible
    - disconnected operations are heavily limited

CVS — some details

- scope: a project (i.e., a tree of file and directories)
- built as a set of scripts on top of RCS
  - each file has its own ,v file stored in the repository
  - each file has its own set of RCS versions (1.1, 2.3, 1.1.2.4, etc.)
  - a very cool hack, but still a hack
- the repository can be either local (i.e., on the same machine of the working copy) or remote (accessible through the network; common scenario)
- concurrency model:
  - optimistic approach: working copies can be acted upon by default; given that working copies are independent from each other, work is concurrent by default
  - conflicts are noticed upon commit and must be solved locally (i.e., commits prior to merges are forbidden)
  - explicit locks are permitted via dedicated actions
- curiosity: one of the first popular UNIX commands relying on sub-commands

CVS — basic operations

repository setup export CVSROOT=SOME/DIR
cvs init

create a project cd PROJECT-DIR
cvs import -d NAME VENDOR-NAME RELEASE-NAME
example: cvs import -d coolhack zack initial

checkout cvs checkout NAME

status get information about the status of the working copy with respect to (the last contact with) the repository

commit cvs commit [FILE...]
example: cvs commit -m 'fix segmentation fault' foo.c

update (merge changes from the repository in the local copy)
cvs update -d

CVS — basic operations (cont.)

history	cvs log	
diff	cvs diff [FILE...]	(among working copy and last update)
diff	cvs diff -rVERSION1 -rVERSION2 [FILE...]	
remove file	cvs rm FILE	(*schedule* removal; needs commit)
add file	cvs add FILE	(*schedule* addition; needs commit)

CVS — basic operations

Demo

CVS — branching and merging operations

tag

```
cvs tag TAG-NAME
```

branch

```
cvs tag -b BRANCH-NAME
# work on trunk continues
cvs update -r BRANCH-NAME
# work on branch ...
cvs update -A
# go back working on trunk
```

merge

```
cvs update -j BRANCH-NAME     # merge changes (wc)
cvs commit
```

CVS — branching and merging operations

Demo

CVS — discussion

revolutionary for its time
affected by severe limitations nonetheless:

- revisions are per file, i.e., there is no knowledge of repository-wide revisions (they can be emulated by tags, but. . .)
- no knowledge of several common file-system features (e.g., attributes, symlink, file move)
- files are considered textual by default; ad-hoc and limited support for binary content
- branch operations are expensive (underlying assumption: most of the work happens in trunk)
- commits are not atomic operations
- very little support for disconnected operations (e.g., painful when you're hacking on a plane)

Outline

Subversion (SVN) — context

Started in 2000 to overcome CVS limitations.

Historical context:

> *In the world of open source software, the Concurrent Version System (CVS) has long been the tool of choice for version control. And rightly so. CVS itself is free software, and its non-restrictive modus operandi and support for networked operation—which allow dozens of geographically dispersed programmers to share their work—fits the collaborative nature of the open-source world very well. CVS and its semi-chaotic development model have become cornerstones of open-source.*
>
> — *Collins-Sussman*

Subversion (SVN)

- same paradigm of CVS: client-server + independent working copies
- features (i.e., “bug fixes” w.r.t. CVS):
  - atomic commits
  - tracking (in the history!) of file-system level operations (copy, move, remove, etc.)
  - global versioning (rather than per-file versioning)
  - support for symlinks and (some) file-system level metadata
  - cheap (server-side) branches
  - some (but not much) support for disconnected operations (most notably: diff among working copy and HEAD)

SVN — basic operations

To increase adoption chances within CVS circles, SVN command line interface has been designed to be as compatible as possible to CVS interface. The strategy has worked very well! Most commands work as in CVS.

basic operations svn checkout, svn status, svn add, svn remove, svn commit, svn diff, svn log, svn update, ...

repository setup svnadmin create REPO-PATH

create a project (actually: create a directory in a repository)

```
svn checkout REPO-PATH
svn mkdir DIR
svn commit
```

SVN — basic operations

Demo

SVN — branching and merging

- branches in SVN are part of the versioned tree
- to create a branch, one makes a copy of an existing directory to a new path
  - development can then proceed independently in the original and new directory
  - branches are cheap (“shallow copies”) on the repository side (but not on the client side)
  - partial checkouts are possible, to avoid forcing clients to keep all branches at once
- tags work in the same way; the only difference is that a tag doesn’t (i.e., shouldn’t) be committed to
- specific path conventions are suggested. . .

SVN — path conventions

```
project/
    trunk/
        main.ml
        module-foo.ml
        quux/
        ...
    branches/
        feature1/
            main.ml
            module-foo.ml
            quux/
            ...
        feature2/
            main.ml
            module-foo.ml
            quux/
            ...
    tags/
        1.0-rc1/
            main.ml
            module-foo.ml
            quux/
            ...
        1.0/
            main.ml
            module-foo.ml
            quux/
            ...
        1.1/
            main.ml
            module-foo.ml
            quux/
            ...
```

Recommended path conventions.

No strict requirement (but still recommended for uniformity across projects).

SVN — branching and merging example

- branching off trunk

```
$ svn cp -m "Creating branch for feature A" \
  /trunk/component1  /branches/zack-component1-featureA
$ svn log -v /branches/zack-component1-featureA
# prints revision number, say 123
```

- updating the branch

```
$ svn merge -r 123:HEAD /trunk/component1 .
# test new code from master
$ svn commit -m'sync with master'
# Revision 256.
```

- merging into trunk

```
$ cd /trunk/component1
$ svn merge -r 123:HEAD /branches/zack-component1-featureA
$ svn commit -m'integrate feature A by Zack'
$
```

SVN — branching and merging example (better)

```
$ cd /trunk/component1
$ svn merge --reintegrate REPO-PATH/branches/zack-componen
$ svn commit -m'integrate feature A by Zack'
$
```

- keep track of merged revisions via `svn:mergeinfo` property

SVN — branching and merging

Demo

Outline

A Git tutorial

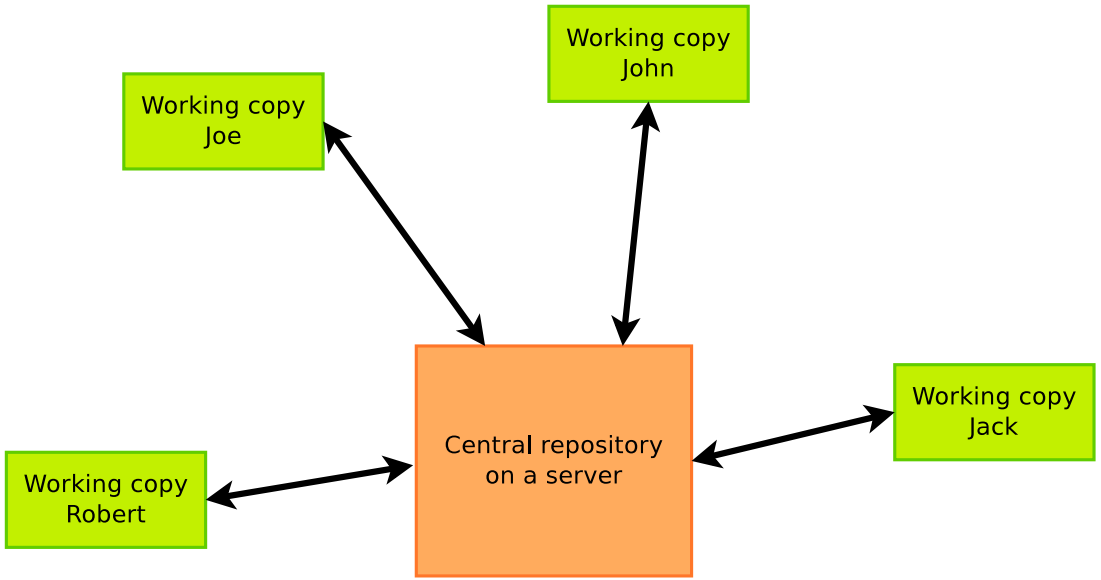
We will follow the excellent tutorial:

Git, a distributed revision control system
Thomas Petazzoni
Free Electrons
thomas.petazzoni@free-electrons.com
http://thomas.enix.org/pub/conf/git2011/presentation.pdf

Agenda

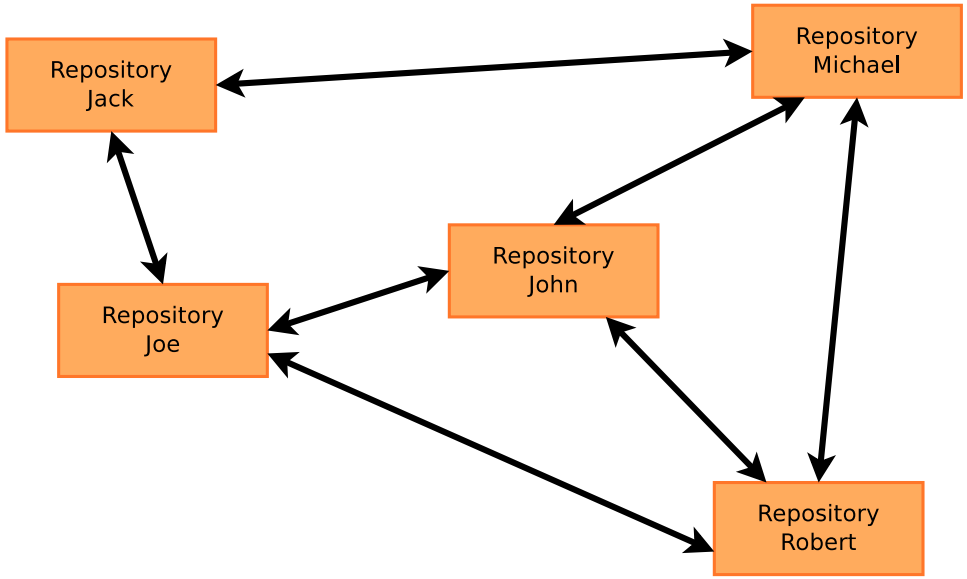
- Version control, centralized vs. distributed
- Git local and basic usage
- Details on Git internals
- Working with branches
- Working with remotes in Git
- Contributing with Git
- Some advanced features

Centralized version control



- The server has all the history of the project
- Users only have a working copy
- All operations involve the central server, no code can be exchanged without this server

Distributed version control



- Each user has a local repository, with the full history of the project
- No server is needed for any operation, everything can be done locally: committing, creating branches and tags, merging, etc.
- Code can be exchanged with others without any central server
- And many more cool features !

A short history of open-source DVCS

- Until 2002, Linus Torvalds and the kernel developers were not using any version control system, because none of the existing one would scale enough for a project as large as Linux.
- In 2002, they started using *BitKeeper*, a proprietary distributed version control system, for which a *free* (as in *free beer*) was available.
- It proved that DVCS were useful for large free software projects

A short history of open-source DVCS

- In 2005, a developer started to reverse engineer the *BitKeeper* protocol in order to create a *free* (as in *free speech*) compatible client
- In reaction, the company developing *BitKeeper* stopped the *free* as in *free beer* client. The kernel community was left without its version control system
- In March 2005, Linus Torvalds starts the development of Git, with the first release published on 7th April 2005

A short history of open-source DVCS

- On 16th June 2005, the first Linux kernel managed by *Git* is released
- When *Git* development started, many other DVCS were started: *Monotone*, *SVK*, *Mercurial*, *Darcs*, *Arch*, *Bazaar*
- Six years later, only two free DVCS are still widely used: *Git* and *Mercurial*, with *Git* having probably more than 90% "market share" in the free software community
- Projects such as Gnome, Eclipse, KDE, the Linux kernel, X.org, Freedesktop, Qt and many other open-source projects
- **Nowadays, knowing Git is mandatory to contribute to most open-source projects**

Creating a repository

```
$ mkdir myproject
$ cd myproject
```

```
$ git init
Initialized empty Git repository in .../myproject/.git/
```

The .git directory

```
$ ls -al
drwxr-xr-x 7 thomas thomas 4096 2011-03-17 20:31 .git
```

```
$ ls -al .git
```

```
drwxr-xr-x 2 thomas thomas 4096 2011-03-17 20:31 branches
-rw-r--r-- 1 thomas thomas   92 2011-03-17 20:31 config
-rw-r--r-- 1 thomas thomas   73 2011-03-17 20:31 description
-rw-r--r-- 1 thomas thomas   23 2011-03-17 20:31 HEAD
drwxr-xr-x 2 thomas thomas 4096 2011-03-17 20:31 hooks
drwxr-xr-x 2 thomas thomas 4096 2011-03-17 20:31 info
drwxr-xr-x 4 thomas thomas 4096 2011-03-17 20:31 objects
drwxr-xr-x 4 thomas thomas 4096 2011-03-17 20:31 refs
```

Working copy and repository

- The project directory is the **working copy**: it will contain directly the files of the project, that we can edit, modify, commit, etc.. At any given time, the working directory allows to edit one particular version of the project.
- The `.git` directory is the **repository**: it will contain the complete history of the project.

Configuration

Git maintains configuration at two levels:

- At the **project** level, in the `.git/config` file
- At the **user** level, in the `~/.gitconfig` file
- At the **system** level, in the `/etc/gitconfig` file (rarely used)

The configuration can be edited:

- With the `git config` command. By default at the **project** level, with `--global` at the **user** level
- Manually by editing the configuration files

Basic configuration

At the minimum, one must set its name and e-mail through the *git* configuration: these informations are used to identify each commit.

```
git config --global user.name \
    "Thomas Petazzoni"

git config --global user.email \
    thomas.petazzoni@free-electrons.com
```

Resulting `.gitconfig` file:

```
[user]
        name = Thomas Petazzoni
        email = thomas.petazzoni@free-electrons.com
```

Add a file

```
$ cat > README
This is a wonderful project
Ctrl+D
```

```
$ git add README
```

Status of the working copy

```
$ git status

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

Committing the changes

```
$ git commit
```

Starts a text editor to define the commit message

```
[master (root-commit) 6ea2ec1] First commit.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```

or, alternatively:

```
$ git commit -m ‘‘My commit message’’
```

Rules for the commit message

A commit message should be composed of:

- A first line that briefly summarizes the change. Should be less than $\approx$ 80 characters. *Git* will use it as the short description of the commit.
- An empty line, separating the short description and the long description.
- A long description, of arbitrary size. Most projects want this to be wrapped at $\approx$ 80 characters.
- A `Signed-off-by: Foobar <foobar@company.com>` line, as requested by various projects. `git commit -s` adds the *SoB* automatically.

An example commit message

```
net: fix rds_iovec page count overflow

As reported by Thomas Pollet, the rdma page counting can overflow.  We
get the rdma sizes in 64-bit unsigned entities, but then limit it to
UINT_MAX bytes and shift them down to pages (so with a possible "+1" for
an unaligned address).

So each individual page count fits comfortably in an 'unsigned int' (not
even close to overflowing into signed), but as they are added up, they
might end up resulting in a signed return value. Which would be wrong.

Catch the case of tot_pages turning negative, and return the appropriate
error code.

Reported-by: Thomas Pollet <thomas.pollet@gmail.com>
Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
Signed-off-by: Andy Grover <andy.grover@oracle.com>
Signed-off-by: David S. Miller <davem@davemloft.net>
```

Add a few more files

```
$ cat > foo.c

#include <stdio.h>
int main(void) {
   printf("Hello world\n");
   return 0;
}
Ctrl+D

$ git add foo.c
$ git commit -s -m ‘‘foo.c:  new program’’
```

Exploring history: git log

```
$ git log

commit 43ed997a01891a4bfe2cd9c5d41d23e7099068cf
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Tue Mar 29 20:17:39 2011 +0200

    foo.c: new program

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

commit f01765d134d897ff373e70c4f1df7610b810392e
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Tue Mar 29 20:17:33 2011 +0200

    Documentation for project

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
```

Shows the history in reverse chronological order. Other orderings are possible.

Exploring history: more `git log`

- $ `git log -p` to show the patch together with each commit
- $ `git log foo.c` to show the changes affecting a particular file or directory
- $ `git log commit1..commit2` to show the changes between two specific commits
- $ `git show somecommit` to show the change done by a particular commit

Commit identification

- The distributed nature of Git makes it impossible to provide a linear revision number that monotonically increments over time, as is done in *Subversion* or *CVS*
- Each commit is uniquely identified by a *SHA1* hash of its contents
- For example:
  `f01765d134d897ff373e70c4f1df7610b810392e`
- One can also refer to it in a shorter-form, as long as it is unique:
  - `$ git show f017`
  - `$ git show f01765d1`

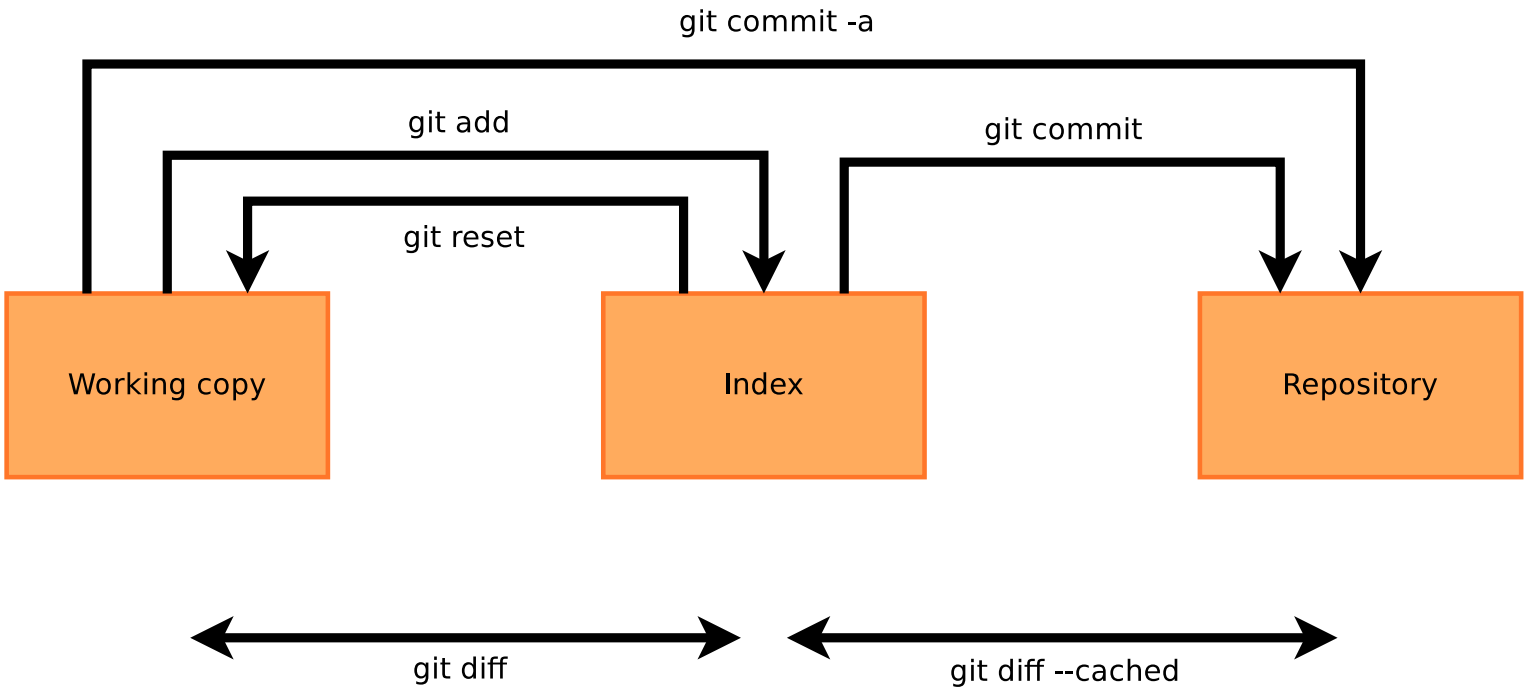
Commit identification: example

```
commit f01765d134d897ff373e70c4f1df7610b810392e
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Tue Mar 29 20:17:33 2011 +0200

    Documentation for project

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

diff --git a/README b/README
new file mode 100644
index 0000000..3803bca
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+This is a wonderful project
```

The index: principle

- Git does not directly commit all the changes you have your working directory, as *Subversion* or *CVS* do
- Instead, Git requires you to **stage** the changes you would like to commit, before doing the actual commit
- This is done through a special *space*, confusingly called the **index**
- When used with partial-file staging (seen later), it is a very powerful feature

The index: principle



The index: make various changes

After adding one line to the README file and changing the message in foo.c, we have:

$ git diff

```
diff --git a/README b/README
index 3803bca..bbdf5e3 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 This is a wonderful project
+really wonderful!
diff --git a/foo.c b/foo.c
index 0e58fa9..0518d69 100644
--- a/foo.c
+++ b/foo.c
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(void) {
-   printf("Hello world\n");
+   printf("Bonjour Monde\n");
    return 0;
 }
```

The index: stage one change

Now, we stage the changes of `foo.c` into the index:

```
$ git add foo.c
```

```
$ git diff --cached
```

```
diff --git a/foo.c b/foo.c
index 0e58fa9..0518d69 100644
--- a/foo.c
+++ b/foo.c
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(void) {
-   printf("Hello world\n");
+   printf("Bonjour Monde\n");
    return 0;
 }
```

These are the changes inside the index, which will be committed if I do `git commit`

The index: what's left in the working copy ?

```
$ git diff
```

```
diff --git a/README b/README
index 3803bca..bbdf5e3 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 This is a wonderful project
+really wonderful!
```

These are the changes inside the working copy, left to be committed in a later commit.

The index: what's in the index ?

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   foo.c
#
# 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:   README
#
```

The index: commit the contents of the index

We commit the contents of the index:

```
$ git commit -m ‘‘foo.c:  translate to french’’
```

```
[master 8f1fab2] foo.c: translate to french
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The index: after the commit

```
$ git show

commit 8f1fab278c876f8677b3b644bbb5403c11a676ea
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Tue Mar 29 21:22:41 2011 +0200

    foo.c: translate to french

diff --git a/foo.c b/foo.c
index 0e58fa9..0518d69 100644
--- a/foo.c
+++ b/foo.c
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(void) {
-   printf("Hello world\n");
+   printf("Bonjour Monde\n");
    return 0;
 }
```

The index: what's left in the working copy ?

```
$ git diff

diff --git a/README b/README
index 3803bca..bbdf5e3 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 This is a wonderful project
+really wonderful!
```

Committing everything ?

- What if I want to commit all the changes I have in my working directory, without bothering to stage them in the index ?
- You can use `git commit -a`

Moving and removing files

- `git mv f1 f2` will move or rename one file or directory. History is preserved accross renames.
- `git rm f1` will remove one file or directory.
- In both cases, the change is done in the index and needs to be committed.

Git basics summary

- `git init`, initialize a repository
- `git add`, stage a file for commit
- `git commit`, commit changes in the index
- `git log`, explore history
- `git show`, show one commit
- `git reset`, reset changes from the index to the working directory
- `git mv`, move files
- `git rm`, remove files

Git internals: object types

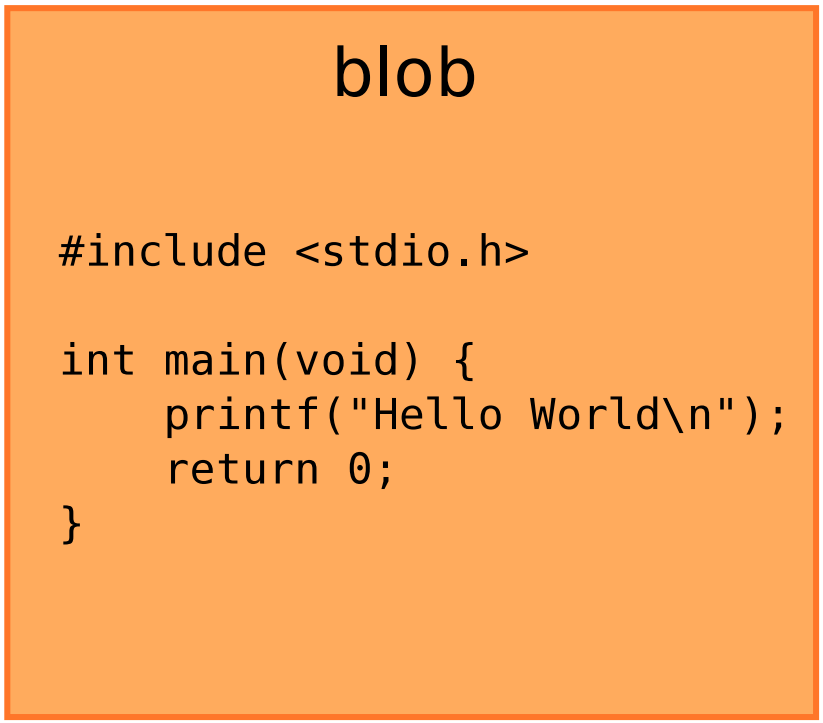
There are three major object types in Git:

- The **blob**
- The **tree**
- The **commit**

All objects are identified by their SHA1.

Git internals: the blob

A **blob** simply allows to store the contents of a particular version of a file, without its name. Just a chunk of binary data.



Git internals: the tree

A **tree** represents a directory, with pointers to blobs for the files and pointers to trees for the subdirectories.

Git internals: the commit

A **commit** represents a particular commit, which associates a particular state of a tree with an author, a committer and a message, and also points to a parent commit.

Exploring commit objects

Raw informations about the latest commit:

$ git show --format=raw

```
commit 92179e1ea0ba3d62bc2f12463370c3f998ba7d62
tree c5241c1c8a626e7e7b7e8a457eebd6e6e2393aa0
parent fc8911d4b0da304ca6ff9b1fc93ce3f2fbdd1008
author Thomas Petazzoni <thomas.petazzoni@free-electrons.com> 1301427783 +0200
committer Thomas Petazzoni <thomas.petazzoni@free-electrons.com> 1301427783 +02

    Update documentation

diff --git a/README b/README
index 3803bca..bbdf5e3 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 This is a wonderful project
+really wonderful!
```

Exploring tree objects

Let's look at the tree:

```
$ git ls-tree
c5241c1c8a626e7e7b7e8a457eebd6e6e2393aa0

100644 blob bbdf5e3c38e09706b6cb9ca0d87af9d4940e58b1 README
100644 blob 0518d6958a90b7ae45530e93632967826b0ee3d4 foo.c
040000 tree 7751df8a2c450e0860c311fedeff797dd912bda1 src
```

Exploring blob objects

Let's look at one of the blobs:

```
$ git show
bbdf5e3c38e09706b6cb9ca0d87af9d4940e58b1

This is a wonderful project
really wonderful!
```

Exploring another commit object

Let's look at the parent commit:

```
$ git show --format=raw
fc8911d4b0da304ca6ff9b1fc93ce3f2fbdd1008

commit fc8911d4b0da304ca6ff9b1fc93ce3f2fbdd1008
tree fed70e7fea02547c4ebb74122c98a3c268586377
parent 4fc1910b790f6bba82b9eafa5297146cd0c9e2f5
author Thomas Petazzoni <thomas.petazzoni@free-electrons.com> 1301427777 +0200
committer Thomas Petazzoni <thomas.petazzoni@free-electrons.com> 1301427777 +02

    bar.c: new source file

diff --git a/src/bar.c b/src/bar.c
new file mode 100644
index 0000000..c5853ff
--- /dev/null
+++ b/src/bar.c
@@ -0,0 +1,3 @@
+int bar(void) {
+    return 42;
+}
```

Exploring another tree object

Let's look at the root tree for this commit:

$ `git ls-tree fed70e7fea02547c4ebb74122c98a3c268586377`

```
$ git ls-tree fed70e7fea02547c4ebb74122c98a3c268586377
100644 blob 3803bca12517a0974a6eb979b7b17e6f0941d550 README
100644 blob 0518d6958a90b7ae45530e93632967826b0ee3d4 foo.c
040000 tree 7751df8a2c450e0860c311fedeff797dd912bda1 src
```

We have the same blob for `foo.c`, the same tree for `src`, but a different blob for `README`, this is because the commit changed the `README` file.

Exploring another blob object

Let's look at the state of the README file at this commit:

```
$ git show
3803bca12517a0974a6eb979b7b17e6f0941d550

This is a wonderful project
```

Branches

- Branches are probably one of the most **powerful** and **useful** feature of Git
- While traditional VCS make branches difficult to create and manage, Git makes it **very easy**
- Branches are **kept completely local**, allowing each developer to organize its work in has many branches as he wants
- Branches are **cheap**, so typically a developer would create a branch even for a very small work (fixing a bug, etc.)
- Branches can be merged together, or exchanged with other developers

Listing branches and default branch

- One can list all local branches using `git branch`
- By default, there is a `master` branch
- The current branch is highlighted with a star

```
$ git branch

* master
```

Creating and switching between branches

- To create a branch: `git branch branchname`. The branch is created from where you are as the starting point.
- To switch to a branch: `git checkout branchname`
- To do both at once: `git checkout -b branchname`
- When you are in a branch, all commits you do end up in the current branch

Creating and switching between branches

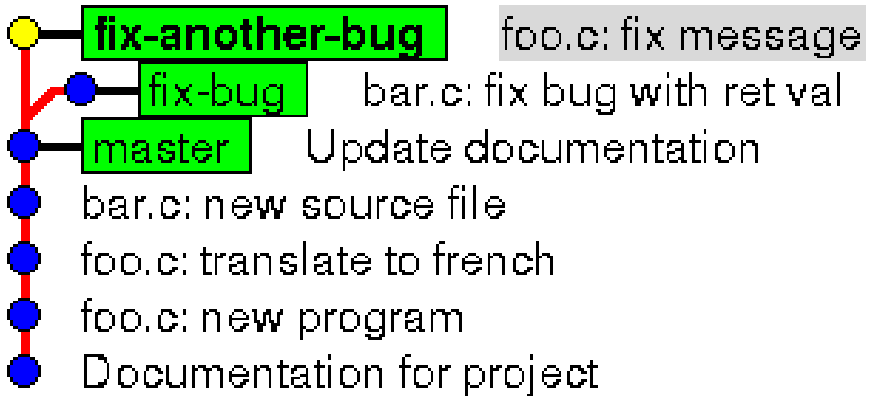
```
$ git branch

* master

$ git branch fix-bug
$ git branch

  fix-bug
* master

$ git checkout fix-bug

Switched to branch 'fix-bug'

$ git branch

* fix-bug
  master
```

Making a change in a branch

```
$ emacs src/bar.c
$ git commit -a -m ‘‘bar.c:  fix bug with ret
val’’
$ git log master..

commit ac4d966da54b24d784854cabb0c72855aa4b44f5
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Tue Mar 29 22:20:22 2011 +0200

    bar.c: fix bug with ret val
```

Branches in `gitk`

Gitk is a Git history visualizer, started with `gitk --all`, it shows the history for all branches:



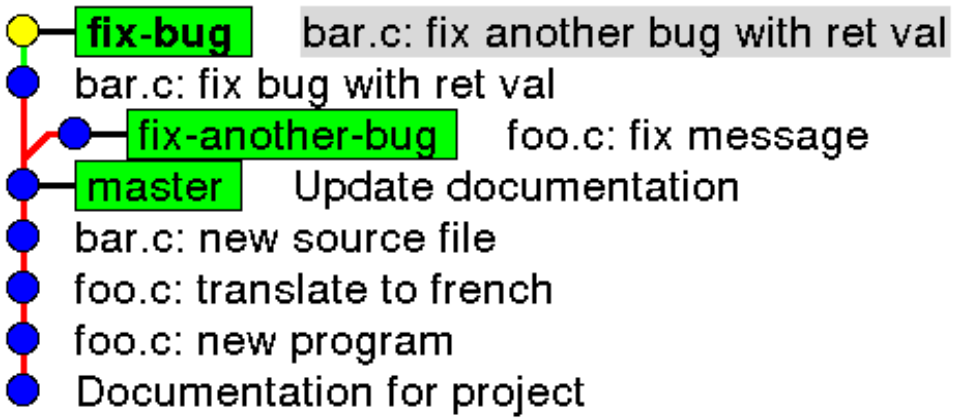
Branches in Git

- Branches are just pointers to the latest commit in this particular branch
- Thanks to the *parent* pointer in each commit, Git can go back inside the history
- The branches are described as SHA1 stored in simple text files in `.git/refs/heads`
- `HEAD` is a special pointer that always points to the latest commit in the current branch

Merging a branch

- If you have split your work in several branches and want to merge them together.
- Go to the destination branch (where things should be merged)
- Use the `git merge branchname` command
- Contrary to Subversion, all the branch history will be preserved, even if the branch gets deleted.

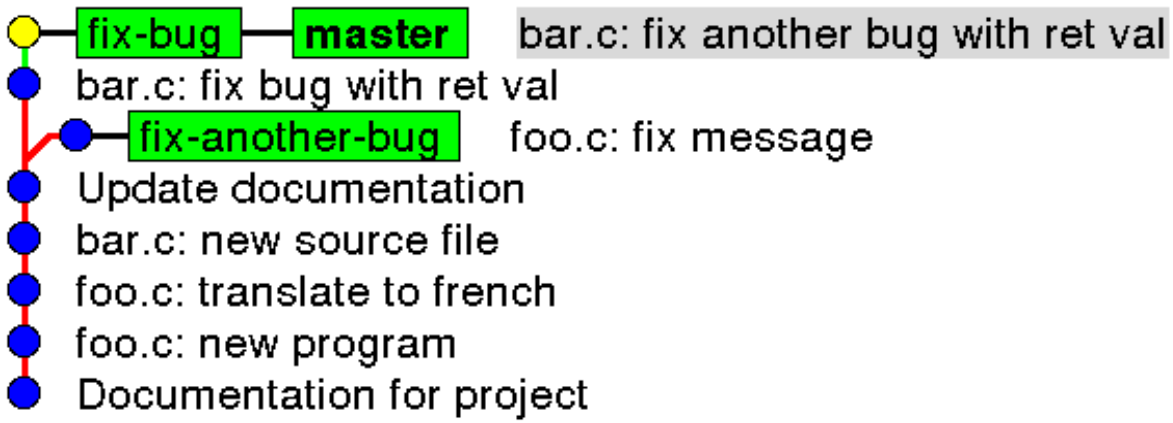
Merging: state before



Merging: doing the merge

```
$ git checkout master

Switched to branch 'master'

$ git merge fix-bug

Updating 92179e1..10e8da2
Fast-forward
 src/bar.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Merging: state after the merge



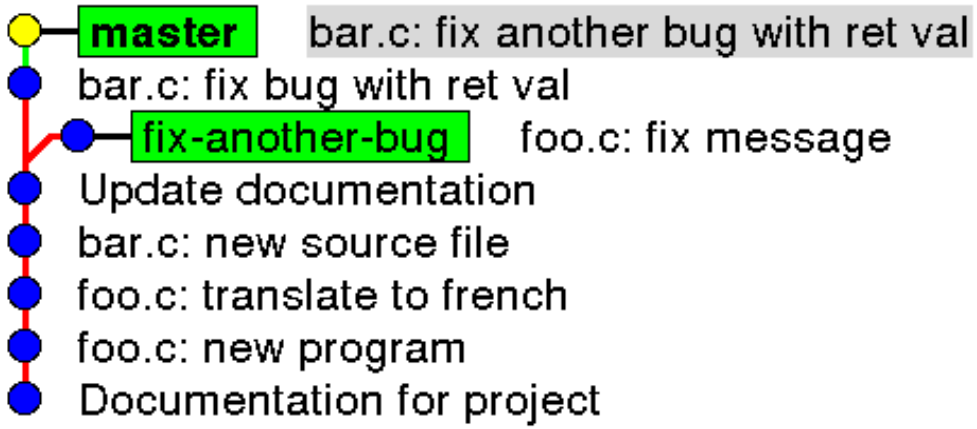
Merging: deleting the branch

```
$ git branch -d fix-bug
```

```
Deleted branch fix-bug (was 10e8da2).
```

Note: `git branch -d` only works with completely merged branches. If you want to remove a non-merged branch, you need to use `git branch -D`.

Merging: state after the merge and deletion



Refering to commits

There are many ways to refer to commits, or range of commits:

- `master..mybranch`, all commits between `master` and `mybranch`
- `master..`, all commits from `master` to where you are
- `HEAD` is the latest commit
- `HEAD^` is the parent of the latest commit
- `HEAD~3` is the grand-grand-parent of the latest commit

Working with remote servers

- Obviously, since Git is distributed, you can communicate with remote servers.
- This is typically done using the following commands
  - `git clone`
  - `git push`
  - `git pull`

Cloning

- **Cloning** is the operation that consists in *cloning a complete repository* to your local machine and *creating a working copy* for the master branch.
- Done with `git clone server-url`
- Useful when you're not starting a project from scratch, but want to contribute to an existing project.
- `git clone` is needed only once, just like `svn checkout` with *Subversion* (not to be confused with `git checkout` !)
- It will setup an initial *remote* named **origin** which points to the server you have cloned from.

Git protocols

Git repositories can be accessed:

- Through the `git://` protocol, which is the native git protocol, offering the best performance. For read-only access only. Usually the best solution to *clone* a project.
- Through the `http://` protocol. For read-only access, lower performance than `git://`. Mostly useful if you are behind a firewall that doesn't allow `git://`.
- Through the `ssh://` protocol. For read-write access.

Cloning: example

```
$ git clone git://git.busybox.net/buildroot

Initialized empty Git repository in /tmp/buildroot/.git/
remote: Counting objects: 68156, done.
remote: Compressing objects: 100% (25281/25281), done.
remote: Total 68156 (delta 46316), reused 64299 (delta 42543)
Receiving objects: 100% (68156/68156), 25.22 MiB | 185 KiB/s, done.
Resolving deltas: 100% (46316/46316), done.
```

Pulling

- **Pulling** is the operation that consists in *fetching changes* from a remote repository and *merging them* into the current local branch
  - Under the hood, `git pull` does `git fetch` and then `git merge`
- Typically, one keeps the *master* branch free of any local change, and updates it with a simple *git pull* command. Equivalent to `svn update` in the *Subversion* world.
- By default, pulls from the `master` branch of the `origin` repository.
- Can also be used to merge contents from other remote repository, if you're integrating the work of other developers.

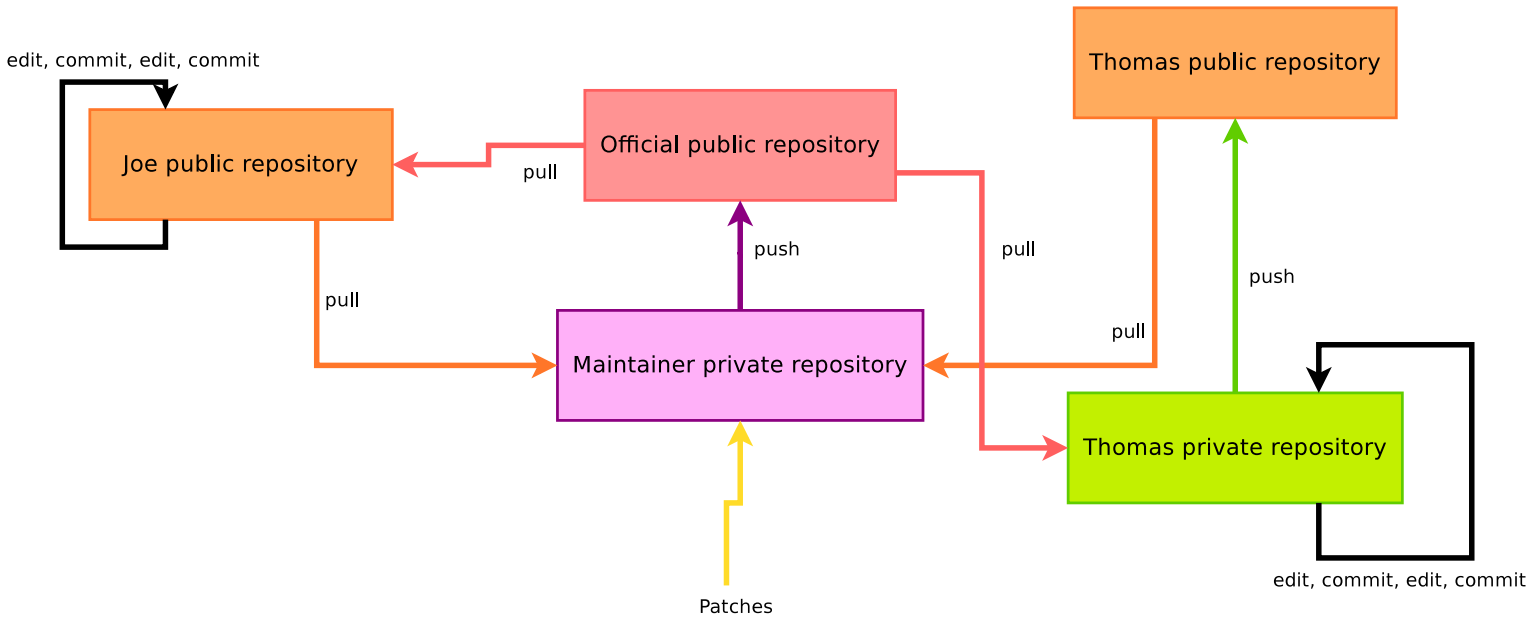
Pulling: example

```
$ git pull

Updating 2c97608..187ca32
Fast-forward
 package/Makefile.autotools.in         |    2 +
 package/buildroot-libtool-v2.4.patch  |   47 ++++++++++++++++++++++++++++++++++++++++++++++++
 package/qt/qt.mk                      |    2 +-
 3 files changed, 50 insertions(+), 1 deletions(-)
 create mode 100644 package/buildroot-libtool-v2.4.patch
```

Two main possible workflows

- A **centralized** workflow, which is similar to Subversion workflow. There is a central Git repository on a public server, and all project participants have write access to it. They can simply `git push` their changes to it.
- A **distributed** workflow, where only the project maintainer has write access to the official Git server. This is the workflow used by many free software projects, such as the Linux kernel.

Distributed workflow



Setting up a public repository

- Do a bare clone of your repository (a clone without a working copy)

```
git clone --bare /home/thomas/myproject
```

- Make it work through HTTP

```
cd myproject.git
git --bare update-server-info
mv hooks/post-update.sample hooks/post-update
chmod a+x hooks/post-update
```

- Transfer the bare clone to a remote location, publicly accessible, on which you have write access:

```
scp -r myproject.git
login@somewhere.com:~/public_html
```

Accessing the repository

- You can access your repo at
  `login@somewhere.com:~/public_html/project.git`
- Others can access your repo at
  `http://somewhere.com/~login/project.git`

Pushing changes to your repository

- To push the current branch:
  ```
  git push
  login@somewhere.com:~/public_html/project.git
  ```
- To push the branch named `foobar` to a branch named `barfoo` on your remote repository:
  ```
  git push
  login@somewhere.com:~/public_html/project.git
  foobar:barfoo
  ```
- To delete the remote branch `barfoo`:
  ```
  git push
  login@somewhere.com:~/public_html/project.git
  :barfoo
  ```

Remote branches

With `git branch -a` you can list all branches, both local and remote ones.

Short names for remotes

- Typing the complete URL of a remote is painful.
- Git offers a command, `git remote`, to manage *aliases* for remotes. These aliases can then be used with all Git commands, especially `git pull` and `git push`
- `git remote add thealias theurl` adds a remote
- `git remote rm thealias` removes a remote
- `git remote` shows all remotes
- `git remote show thealias` gives details about a remote
- `git remote prune thealias` to delete all branches that no longer exist remotely

Short names for remotes

- A remote `origin` is created by `git clone`, it refers to the server from which the repository was cloned, and is used as the default for pull/push.
- All remote aliases are stored in `.git/config`
- Typically useful for your public repository, but also for the public repositories of the developers you're working with.

Short names for remotes

- Add a remote for my public repository:
  ```
  git remote add public
  login@somewhere.com:~/public_html/project.git
  ```
- Push the current branch to it:
  ```
  git push public
  ```

Typical workflow

1. Clone
2. Create branch for feature development or bug fix
3. Make changes, make one or more commit
4. Either
   - Push the branch to a public repository
   - Tell the project maintainer to pull your branch
5. or
   - Send patches by e-mail
6. Once the changes are merged: remove the branch and `git pull` your master branch
7. Goto step 2

Sending a pull request

- Helps in sending an e-mail to ask the maintainer to pull one of your publicly visible branch.
- Make sure your branch is publicly visible and up-to-date:
  `git push public mybranch`
- Prepare the text for the pull request:
  `git request-pull master http://somewhere.com/~login/project.git`
  - The `master` in the command is the starting point of the interval of commits for which the pull request is generated.
- Send the text by e-mail to the maintainer.

Pull request example

```
$ git request-pull master
http://somewhere.com/~login/project.git

The following changes since commit 10e8da2b115bab3419a28e9af52a5d67c3f797cc:

  bar.c: fix another bug with ret val (2011-03-29 22:28:58 +0200)

are available in the git repository at:
  http://thomas.enix.org/pub/demo.git fix-another-bug

Thomas Petazzoni (2):
      foo.c: fix message
      foo.c: more messages

 foo.c |    3 ++-
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Sending patches

- Another way of contributing is to *send patches* to a mailing list. It allows other to review and comment your patches.
- Patches are generated using `git format-patch`, the short description is used as the title, the long description as the changelog of the patch
- Patches are sent using `git send-email`
- `git send-email` requires a properly configured SMTP setup:
  - `git config --global sendemail.smtpserver foobar.com`
  - `git config --global sendemail.smtpuser user`
  - `git config --global sendemail.smtppass pass`

Sending patches

```
$ git format-patch master

0001-foo.c-fix-message.patch
0002-foo.c-more-messages.patch

$ git send-email --to mailing@project.org *.patch

To: thomas@enix.org
Subject: [PATCH 1/2] foo.c: fix message
Date: Tue, 29 Mar 2011 23:57:10 +0200
...
To: thomas@enix.org
Subject: [PATCH 2/2] foo.c: more messages
Date: Tue, 29 Mar 2011 23:57:11 +0200
```

git send-email tips

- `git config --global sendemail.confirm auto` avoids the need to confirm the sending of each message
- `git config --global sendemail.chainreplyto false` avoids to have each e-mail being a reply to the previous one: all patches are attached directly to the main mail
- `git send-email --compose` opens a text editor to write a special text for the introduction message
- Patches formatted with `git format-patch` are better than normal patches, as they properly handle binary files and file renames/removal.
- A maintainer can integrate a patch sent by `git send-email` using the `git am` command.

git send-email output

▽	[Buildroot] [pull request] Pull request for branch for-2011.02/python-bump	Thomas Petazzon	24/01/11
	[Buildroot] [PATCH 01/10] python: Move to version 2.7	Thomas Petazzon	24/01/11
	[Buildroot] [PATCH 02/10] python: Port the python2.4 patches to 2.7	Thomas Petazzon	24/01/11
	[Buildroot] [PATCH 03/10] python: Add the needed patches to compile python2.7 in buildroot.	Thomas Petazzon	24/01/11
	[Buildroot] [PATCH 04/10] python: Fix make install (Workaround python's bug #1669349)	Thomas Petazzon	24/01/11
	[Buildroot] [PATCH 05/10] libffi: new package	Thomas Petazzon	24/01/11
▽	[Buildroot] [PATCH 06/10] autotools: allow host package to use <pkg>_MAKE_ENV and <pkg>	Thomas Petazzon	24/01/11
▽	Re: [Buildroot] [PATCH 06/10] autotools: allow host package to use <pkg>_MAKE_ENV and <	Peter Korsgaard	24/01/11
▽	Re: [Buildroot] [PATCH 06/10] autotools: allow host package to use <pkg>_MAKE_ENV and	Thomas Petazzon	25/01/11
	Re: [Buildroot] [PATCH 06/10] autotools: allow host package to use <pkg>_MAKE_ENV anc	Peter Korsgaard	25/01/11
	[Buildroot] [PATCH 07/10] python: convert to autotargets, bump to 2.7.1, many improvements	Thomas Petazzon	24/01/11
▽	[Buildroot] [PATCH 08/10] Remove .py or .pyc depending on Python configuration	Thomas Petazzon	24/01/11
	Re: [Buildroot] [PATCH 08/10] Remove .py or .pyc depending on Python configuration	Peter Korsgaard	24/01/11
▽	[Buildroot] [PATCH 09/10] python-serial: new package	Thomas Petazzon	24/01/11
	Re: [Buildroot] [PATCH 09/10] python-serial: new package	Peter Korsgaard	24/01/11
▽	[Buildroot] [PATCH 10/10] python-mad: new package	Thomas Petazzon	24/01/11
	Re: [Buildroot] [PATCH 10/10] python-mad: new package	Peter Korsgaard	24/01/11
▽	Re: [Buildroot] [pull request] Pull request for branch for-2011.02/python-bump	Thomas Petazzon	25/01/11
	Re: [Buildroot] [pull request] Pull request for branch for-2011.02/python-bump	Peter Korsgaard	25/01/11

Reviewing changes from others

To review changes made by other developers that are visible through their public repository, one can do:

```
$ git fetch thedeveloper
```

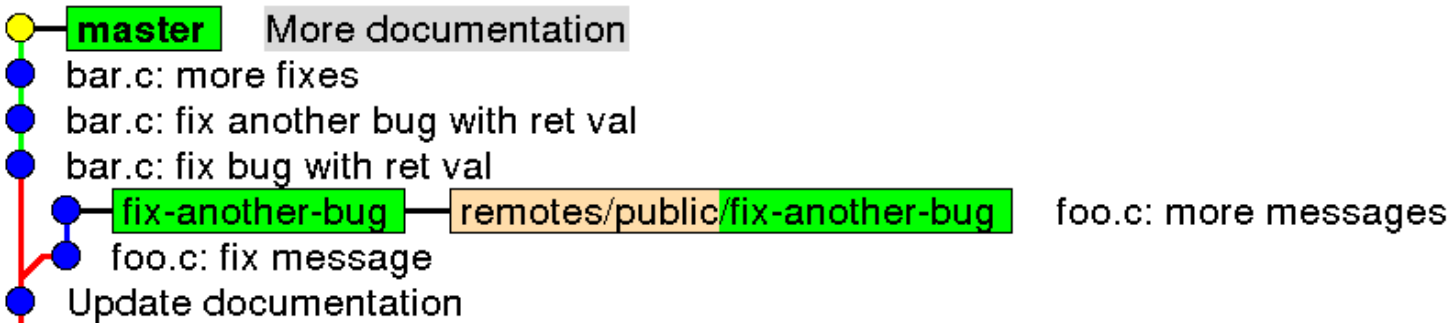
then all branches made by the other developer are accessibles as `remotes/thedeveloper/branchname`

```
$ git log -p
master..remotes/thedeveloper/somebugfix
```

Rebasing

- You create a branch
- You do some work on this branch, with several commits
- The development goes on in the official project, with several changes being made
- How can you update your changes on top of all the improvements done by the other developers ?
- Solution: `git rebase`

Before the rebase

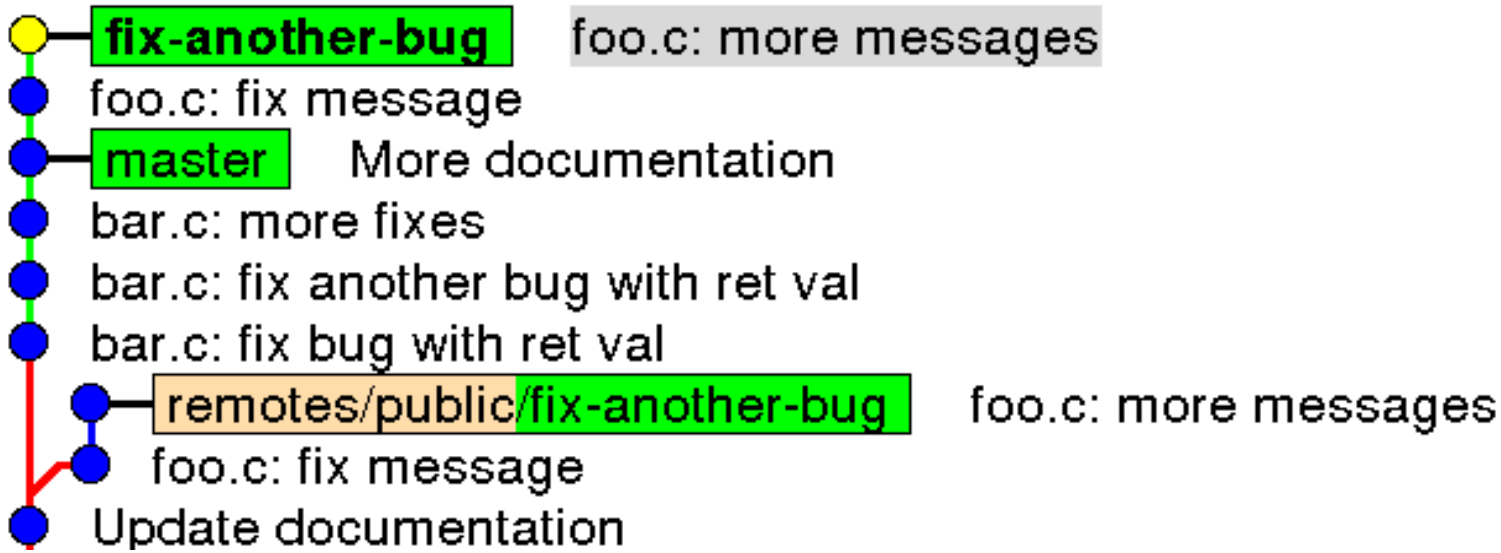


The branch `fix-another-bug` is behind `master`

Doing the rebase

```
$ git checkout fix-another-bug
$ git rebase master

First, rewinding head to replay your work on top of it...
Applying: foo.c: fix message
Applying: foo.c: more messages
```

After the rebase



The branch `fix-another-bug` is on top of `master`

Interactive rebasing

- You develop
- You commit
- You develop
- You commit
- Oops, I forgot this, you fix, you commit
- $\rightarrow$ you have an ugly history, which means ugly patches that show how stupid you are to the rest of the project members
- Git allows you to hide your stupidity!
- Your friend is **interactive rebasing**, using `git rebase -i`

Interactive rebasing

- You are in branch `fix-another-bug`, which has been started on top of the `master` branch
- You run `git rebase -i master`
- A text editor will open, with one line per commit in your branch. Each line is prefixed with `pick`. With each line you can:
  - Keep the `pick`, the commit will be kept
  - Remove the line, which will completely remove the commit from the history
  - Change `pick` to `edit`, which will stop the rebase at the given commit, which allows to make further modifications to it
  - Change `pick` to `reword`, which allows to rephase the commit log
  - Change `pick` to `fixup`, which merges the commit into the previous one
  - Change `pick` to `squash`, which merges the commit into the previous one and edits the commit message

git log master..

```
commit 2644e423d9b3f5514284f49f207cd4f7a8e8a764
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:50:41 2011 +0200

    Really fix the return value

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

commit 874a6a0e3f0058a84dd857d2ef68f8b71cb3aeb5
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:50:27 2011 +0200

    Fix return value

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

commit 87b3b0c7e8905d9c0328508050c5e0b596b873cf
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:50:13 2011 +0200

    foo.c: add englich

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

commit 969387105cfcc70562a88dd8505c57418bd4354f
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:49:52 2011 +0200

    foo.c: add spanish

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
```

Git rebasing example

What I want is:

- Fix the commit message in 87b3b0c7e8905d9c0328508050c5e0b596b873cf
- Merge 2644e423d9b3f5514284f49f207cd4f7a8e8a764 into 874a6a0e3f0058a84dd857d2ef68f8b71cb3aeb5

Git rebasing example

$ `git rebase -i master`

It opens a text editor with:

```
pick 9693871 foo.c: add spanish
pick 87b3b0c foo.c: add englich
pick 874a6a0 Fix return value
pick 2644e42 Really fix the return value

# Rebase 974cd34..2644e42 onto 974cd34
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

Git rebasing example

You change it to:

```
pick 9693871 foo.c: add spanish
reword 87b3b0c foo.c: add englich
pick 874a6a0 Fix return value
fixup 2644e42 Really fix the return value
```

After rebase, `git log master..`

```
commit c332772f74bbc808105e4076bbb821d762c9653f
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:50:27 2011 +0200

    Fix return value

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

commit 06a174ee4426cb83004614cf64a33c12c42670b0
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:50:13 2011 +0200

    foo.c: add english

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>

commit 969387105cfcc70562a88dd8505c57418bd4354f
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Mar 30 15:49:52 2011 +0200

    foo.c: add spanish

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
```

Rebase: history modification

- `git rebase` is a tool that rewrites the history: it merges, splits, and changes commits
- Therefore, if other people had access to a branch you created, you should never rebase on this branch, unless others are well informed that the branch might get rebased
- Rebasing is only useful to make modifications before showing your work to others.

Partial staging

- Wondered why the *index* thing is so cool ?
- Because you can stage only parts of the modification made to a given file to the *index* for committing!
- `git add --patch myfile`
  will ask for each chunk of the patch:
  - if you want to stage it, and possibly all other chunks
  - if you don't want to stage it, and possibly quit now
  - if you want to split the chunk into smaller chunks
- Another possibility is `git add -i`, for interactive adding

Git partial adding example

```
$ git diff

diff --git a/foo.c b/foo.c
index 0518d69..6371bb4 100644
--- a/foo.c
+++ b/foo.c
@@ -1,5 +1,6 @@
 #include <stdio.h>
+#include <stdlib.h>
 int main(void) {
     printf("Bonjour Monde\n");
-    return 0;
+    return 1;
 }
```

Git partial adding example

```
$ git add --patch foo.c

diff --git a/foo.c b/foo.c
index 0518d69..6371bb4 100644
--- a/foo.c
+++ b/foo.c
@@ -1,5 +1,6 @@
 #include <stdio.h>
+#include <stdlib.h>
 int main(void) {
     printf("Bonjour Monde\n");
-    return 0;
+    return 1;
 }
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

Git partial adding example

Let's split the commit:

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
```

stage the first chunk:

```
Split into 2 hunks.
@@ -1,3 +1,4 @@
 #include <stdio.h>
+#include <stdlib.h>
 int main(void) {
    printf("Bonjour Monde\n");
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
```

and skip the last chunk:

```
@@ -2,4 +3,4 @@
 int main(void) {
    printf("Bonjour Monde\n");
-   return 0;
+   return 1;
 }
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? q
```

Git partial adding example

```
$ git diff --cached

diff --git a/foo.c b/foo.c
index 0518d69..41d6359 100644
--- a/foo.c
+++ b/foo.c
@@ -1,4 +1,5 @@
 #include <stdio.h>
+#include <stdlib.h>
 int main(void) {
    printf("Bonjour Monde\n");
    return 0;
```

Git partial adding example

```
$ git diff

diff --git a/foo.c b/foo.c
index 41d6359..6371bb4 100644
--- a/foo.c
+++ b/foo.c
@@ -2,5 +2,5 @@
 #include <stdlib.h>
 int main(void) {
    printf("Bonjour Monde\n");
-   return 0;
+   return 1;
 }
```

.gitignore

- Version control systems can usually be told to ignore certain file names or file name patterns.
- With *Git*, this is done in `.gitignore` files
- Each directory can have a `.gitignore` file, but *Git* travels back through the parent directories to find other `.gitignore` files
- For example, you can have a global `.gitignore` file at the root of your project that ignores `*.o`, and another `.gitignore` file in a sub-directory to ignore a particular file.

Other topics

- **Bisecting**, to quickly find the commit that introduced a regression
- `git reset` has many more features to reset the state of the index or the working copy
- `git stash` to put uncommitted changes on the side while doing something else
- `git blame` to check which of your colleague introduce this stupid bug
- **Tags**, with `git tag`
- `git filter-branch`, to rewrite a complete branch to make modifications on several commits
- `git svn`, for integration with *Subversion* repositories
- Graphical interfaces: Giggle in Gtk, QGit in Qt, EGit in Eclipse, etc.
- Git forges, such as *Gitorious* or *Github*

References

Aiello, Sachs
Configuration Management Best Practices: Practical Methods that Work in the Real World
Addison-Wesley, 1st edition, 2010

Chacon
Pro Git
Apress, 2005. http://progit.org/book/

Credits

- diff/patch slides content adapted from Roberto Di Cosmo's slides at `http://www.dicosmo.org/CourseNotes/LogicielLibre/`
- Git slides are by Thomas Petazzoni `http://thomas.enix.org/pub/conf/git2011/presentation.pdf`