A Practical Introduction to `git`

Emanuele Olivetti[1] Rike-Benjamin Schuppner[2]

[1]NeuroInformatics Laboratory (NILab)
Bruno Kessler Foundation (FBK), Trento, Italy
Center for Mind and Brain Sciences (CIMeC), University of Trento, Italy
http://nilab.fbk.eu
olivetti@fbk.eu

[2]HU-Berlin / BCCN Berlin, Germany
http://debilski.de
rikebs@debilski.de

Summer School
"Advanced Scientific Programming in Python"
Christian-Albrechts-Universität, Kiel, Germany
2-7 September 2012

Outline

- Version Control: **`git`**.
- Scenario 1: **single** developer, **local** repository.
  - Demo **single+local**
- Scenario 2: **Team** of developers, **central remote** repository. Minimalistic.
  - Demo **multi+remote**
- Extras: **`git branch`**, how to set up central repo.

Version Control: Naming & Meaning

Wikipedia

"*Revision control*, also known as *version control*, *source control* or *software configuration management* (SCM), is the management of changes to documents, programs, and other information stored as computer files."

Popular Acronyms:

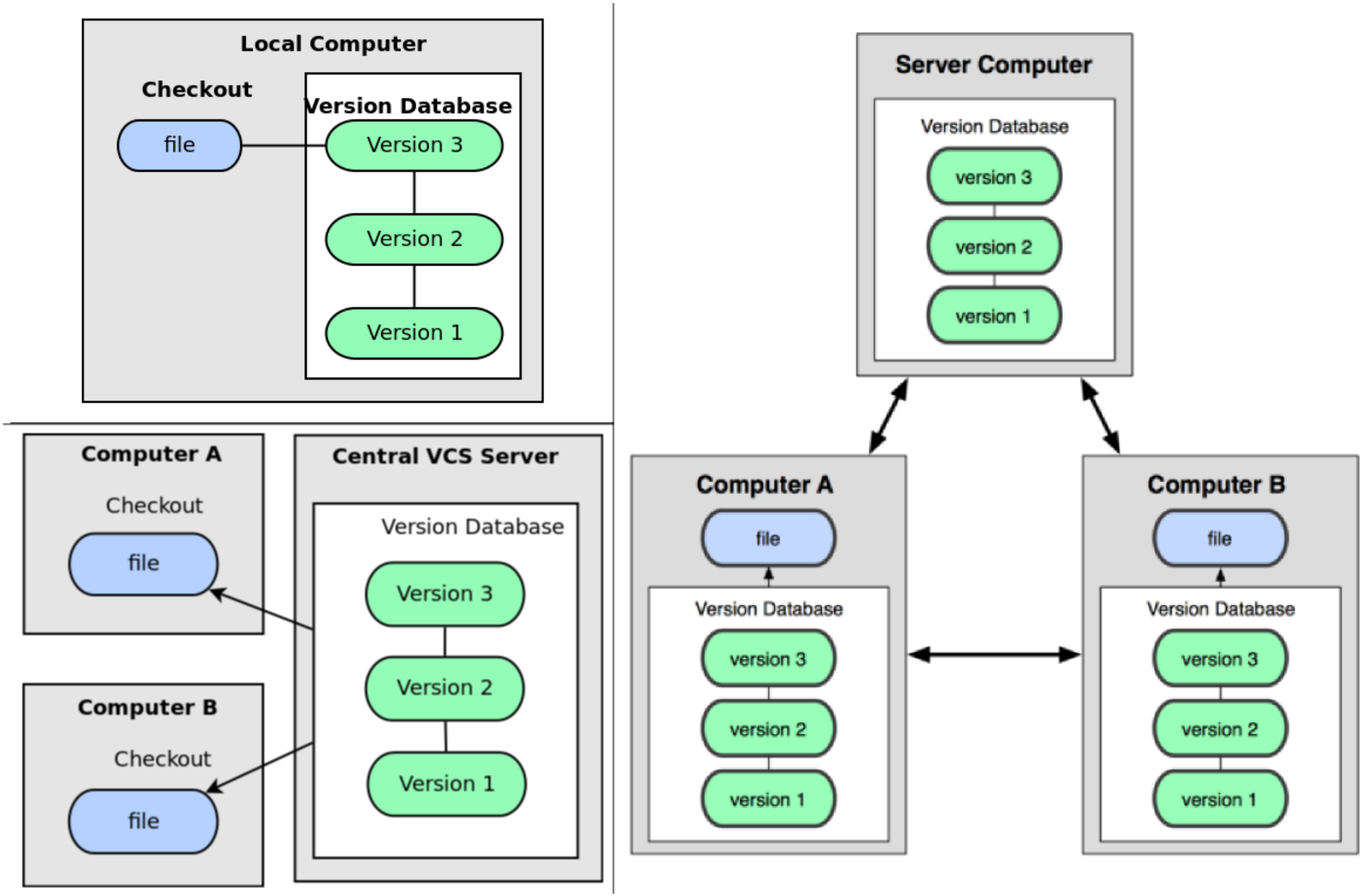
- VC
- SCM

Misnomer:

- ~~Versioning~~

Q: have you ever used VC? (raise your hand = YES)

Version Control: Local, Centralized, Distributed



From *Pro Git*, S.Chacon 2009, CC 3.0 license.

Why `git`?

From: *2010 Python Bootcamp*, Day 3 (Peter Williams).

Why We Like Git

- Fundamental reason: extremely well-engineered, underlying theory is solid. (Turns out Linus knows what he's doing.)
- Rock-solid reliability
- Very, very fast
- Open-source and Free software
- Ergonomic
- Extremely powerful suite of tools
- Decentralized code-sharing model
- Active, committed developer community
- Secure

Survey: `git`

- Q1: Have you heard about **`git`**?
- Q2: Do you use **`git`**?
- Q3: Why the "**`git`**" name? (from **`git`** FAQ)
  1. Random three-letter combination that is pronounceable.
  2. Acronym (global information tracker).
  3. Irony.

`git`? Why "`git`"?

Linus Torvalds: *"I name all my projects after myself. First Linux, now* ***`git`****."*

`http://www.merriam-webster.com/dictionary/git`

[1]git *noun* \'git\

Definition of GIT

British **:** a foolish or worthless person

Examples of GIT

- That *git* of a brother of yours has ruined everything!
- <oh, don't be such a silly *git*, of course your mates want you around>

Origin of GIT

variant of *get,* term of abuse, from [2]*get*

First Known Use: 1929

Related to GIT

Synonyms: berk [*British*], booby, charlie (*also* charley) [*British*], cuckoo, ding-a-ling, dingbat, ding-dong, dipstick, doofus [*slang*], featherhead, fool [*British*], goose, half-wit, jackass, lunatic, mooncalf, nincompoop, ninny, ninnyhammer, nit [*chiefly British*], nitwit, nut, nutcase, simp,

git

```
usage: git [OPTIONS] COMMAND [ARGS]

The most commonly used git commands are:
   add          Add file contents to the index
   commit       Record changes to the repository
   diff         Show changes between commits, commit
   ...
```

git help <command>

git status

Introduce yourself to **git**:

```
git config --global user.name "Emanuele Olivetti"
```

```
git config --global user.email "olivetti@fbk.eu"
```

`git`. Single developer + local repository.

Scenario 1: single developer + local repository.

Single+Local `git`. Motivations.

- **Q:** do you use VC for local repo?
- Why VC for single developer + local repository?
  - First step towards a shared project.
  - Backup.
  - Keep memory of your work.

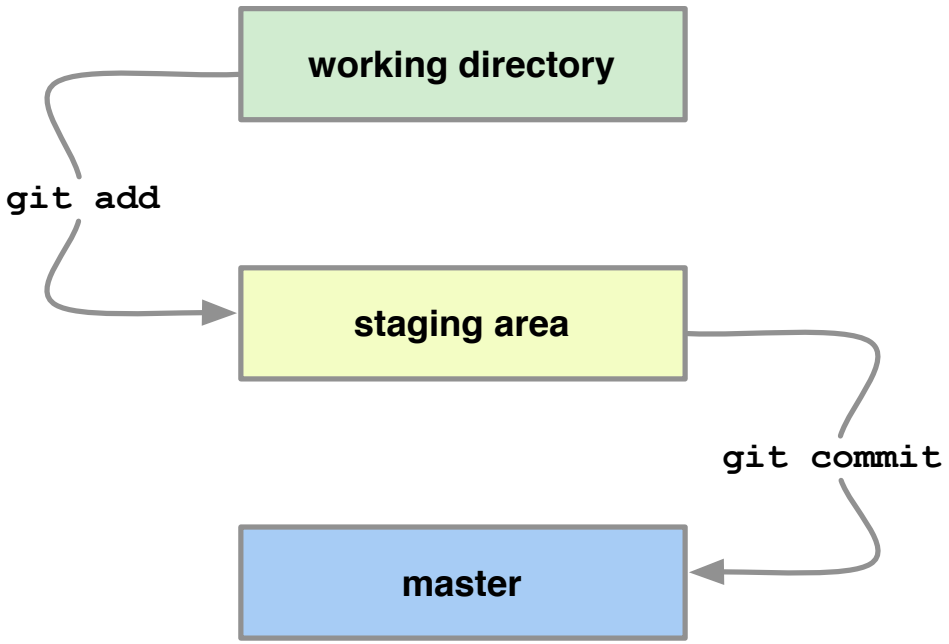
Single+Local `git`. Init.

```
git init
```

- Creates an empty **`git`** repository.
- Creates the git directory: `.git/`

Single+Local `git`. The tracking process.

```
git add <filename>
```




```
git commit -m "Let us begin."
```

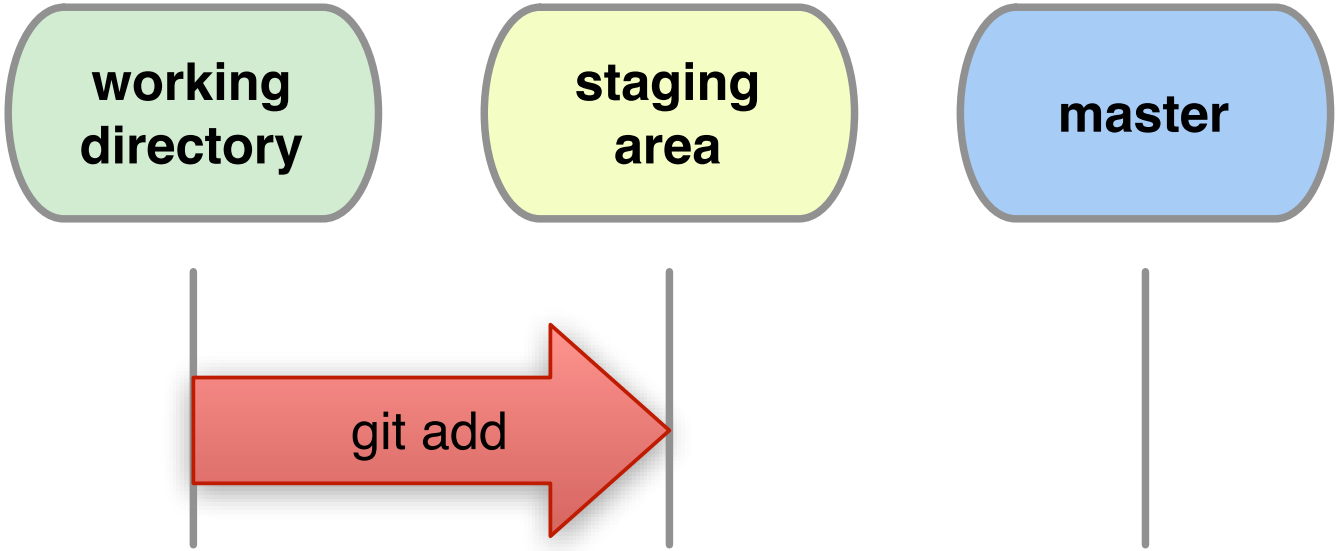
Wikipedia

"A *staging area* is a location where organisms, people, vehicles, equipment or material are assembled before use".

Single+Local `git`. Add.

```
git add file1 [file2 ...]
```

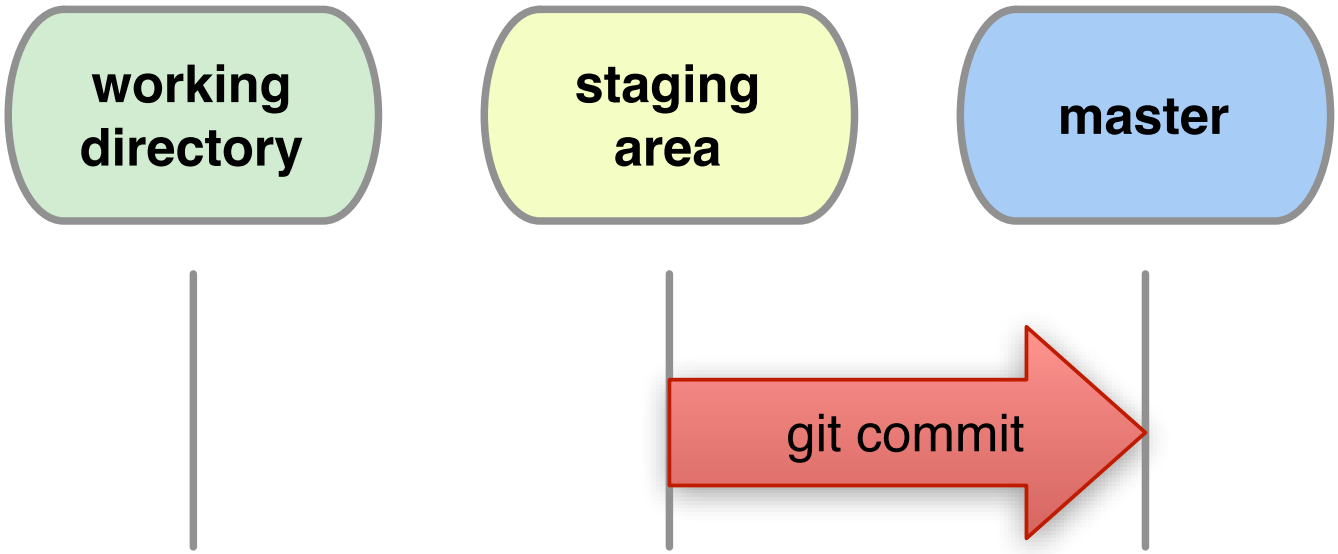
- Adds new files for next commit.
- Adds content from working dir to the staging area (index) for next commit.
- DOES NOT add info on file permissions other than *exec/noexec* (755 / 644).
- DOES not add directories *per se*.



Single+Local `git`. Commit.

```
git commit [-m "Commit message."]
```

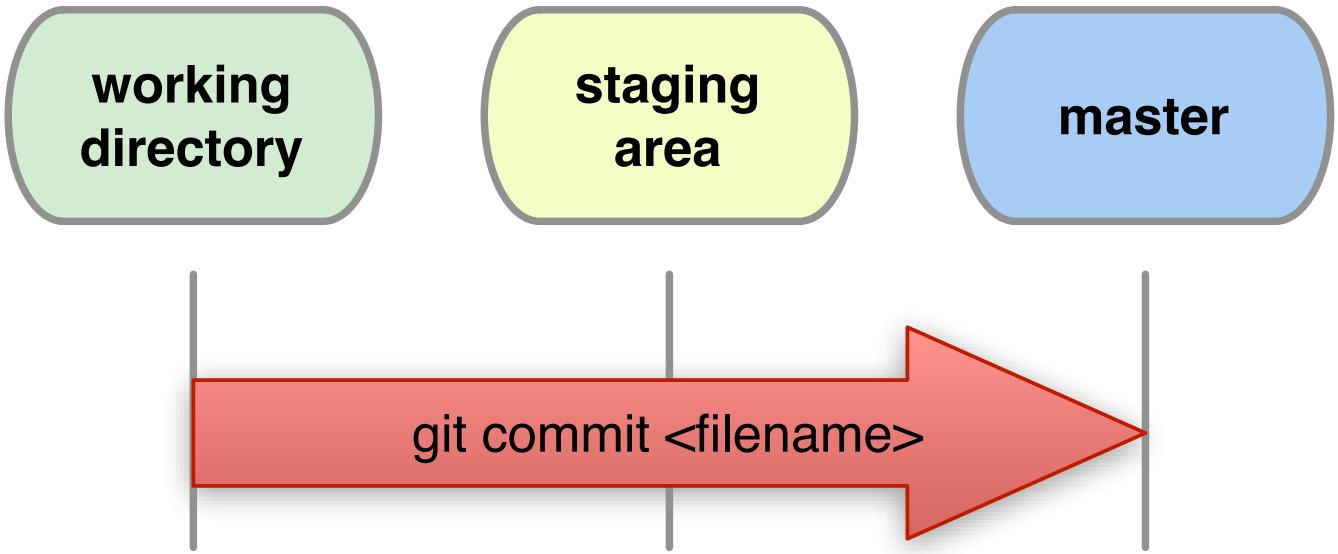
Records changes from the staging area to master.



Single+Local `git`. Commit.

`git commit file1 file2`

Records all changes of **`file1`**, **`file2`** from working dir and staging area to master.



`git commit -a`

Records all changes in working dir and staging area. *Be Careful!*

Single+Local `git`. Commit names. OPTIONAL

- Every *commit* is a **`git`**-object.
- The history of a project is a graph of objects referenced by a 40-digit **`git`**-name: *SHA*1(*object*).
- *SHA*1(*object*) = 160-bit Secure Hash Algorithm.
- Examples:

```
$ git commit README -m "Added README."
[master dbb4929] Added README.
1 files changed, 1 insertions(+), ...
```

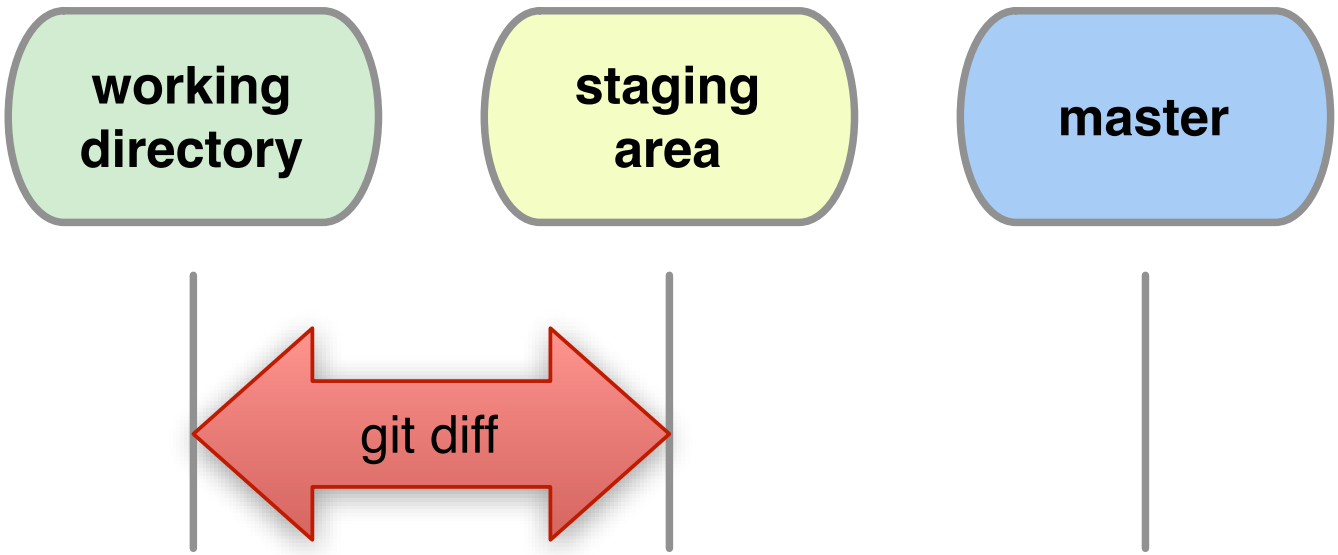
or

```
$ git log
commit dbb49293790b84f0bdcd74fd9fa5cab0...
Author: Emanuele Olivetti <olivetti@fbk.eu>
Date:   Wed Sep 15 00:08:46 2010 +0200
...
```

Single+Local `git`. Diff.

`git diff`

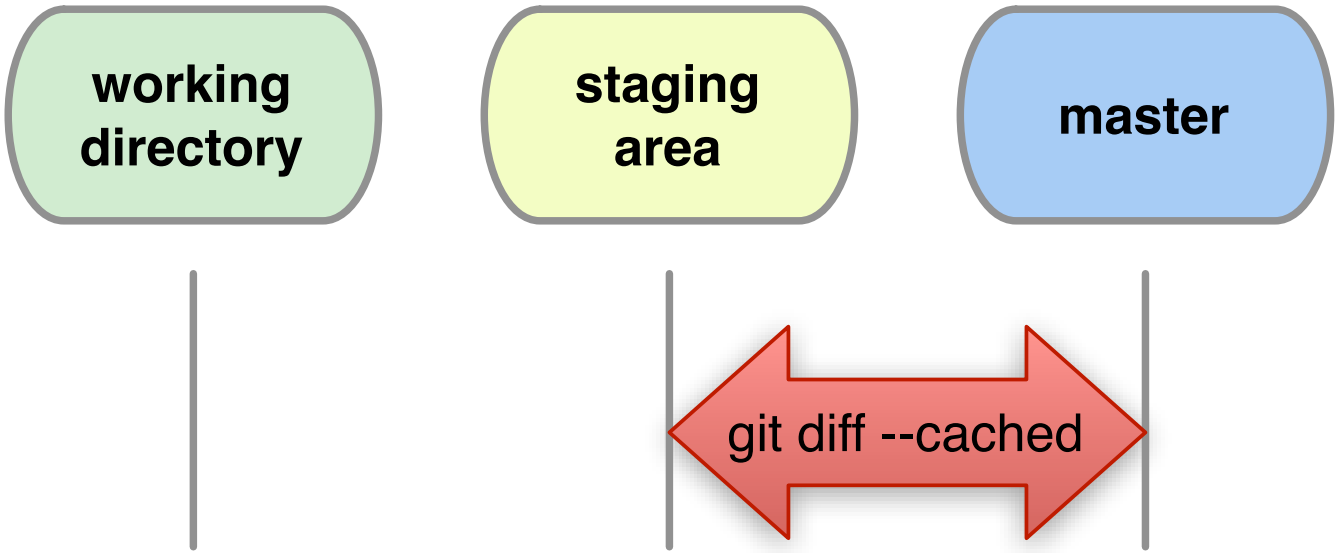
Shows what changes between *working directory* and *staging area* (*index*).



Single+Local `git`. Diff. OPTIONAL

Q: "`git add`" then "`git diff`". What output?

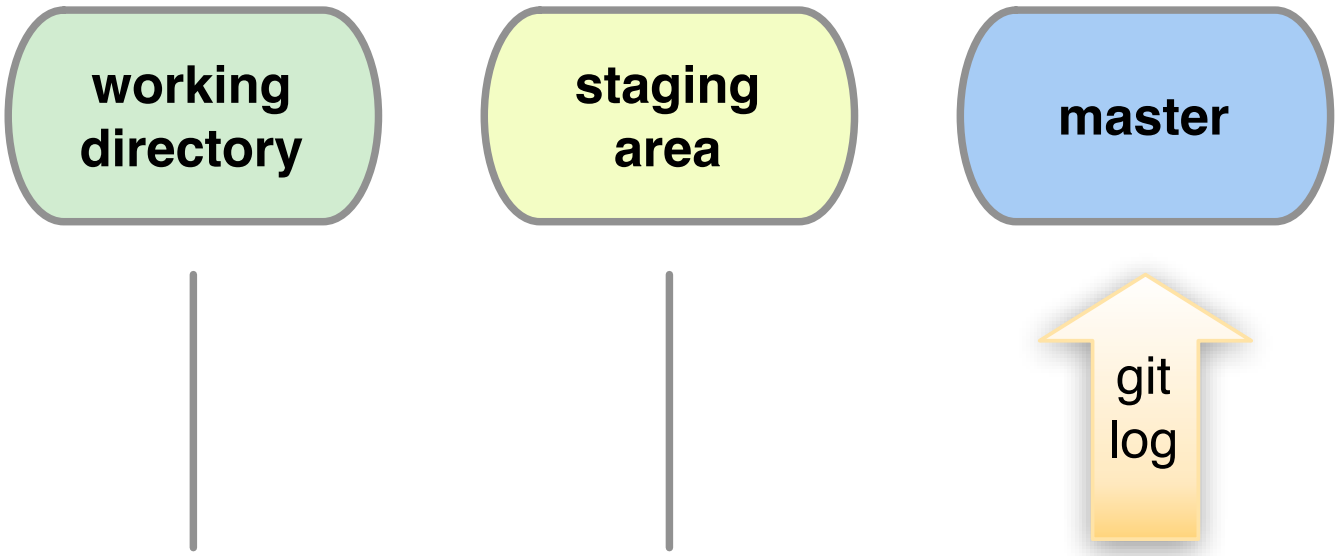
`git diff --cached` shows differences between index and last commit (HEAD).



Single+Local `git`. Logs.

`git log`

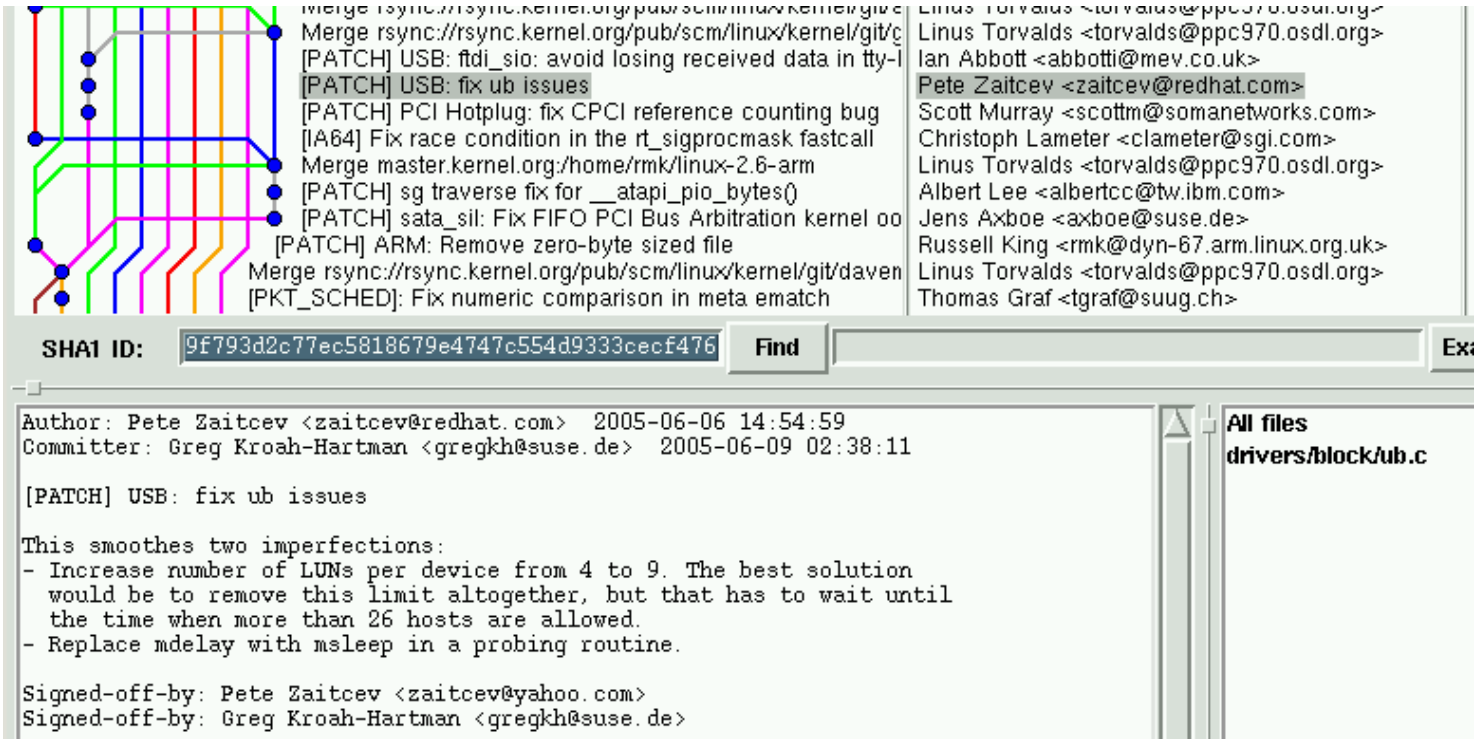
Shows details of the commits.



Single+Local `git`. Logs.

`gitk`

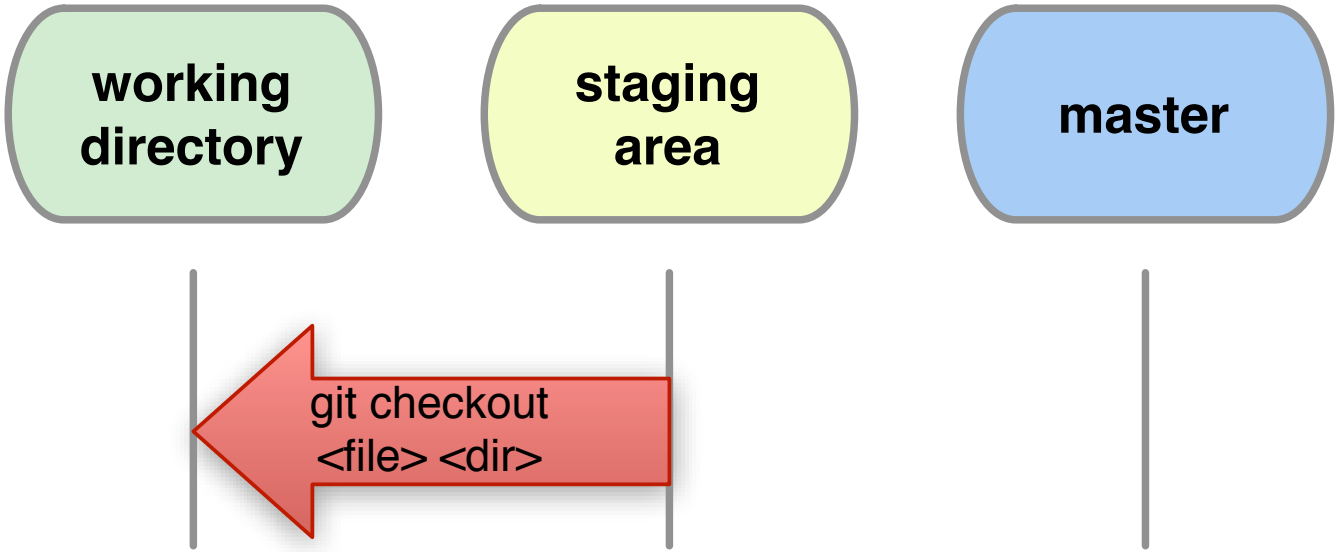
GUI to browse the `git` repository.



Single+Local `git`. “How to clean this mess??” OPT.

```
git checkout <filename>
```

Get rid of what changed in **`<filename>`** (between working dir and staging area).



Single+Local `git`. Time travelling. OPTIONAL

Back to the past when you did commit `dbb49293790b84...`

`git checkout dbb4929`

...and now, *back to the present!*

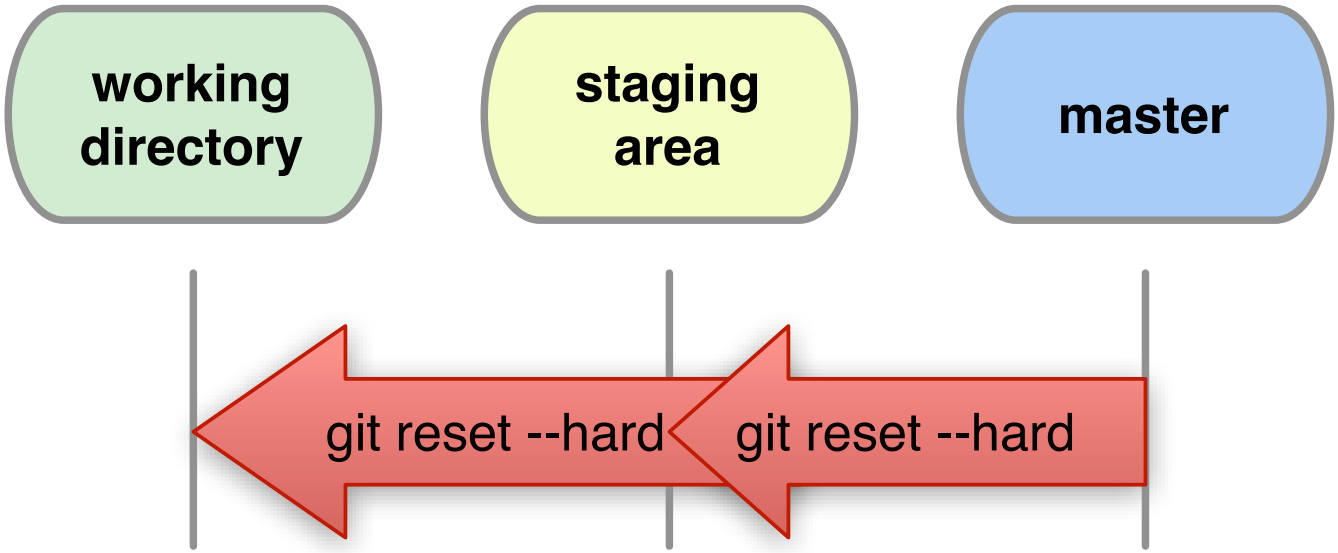
`git checkout master`

Single+Local `git`. “How to clean this mess??”. OPT.

First read *carefully* **`git status`**. If you panic:

```
git reset --hard HEAD
```

Restore all files as in the last commit.



Warning: `reset` can destroy history!

Single+Local `git`. (Re)move. OPTIONAL

Warning: whenever you want to *remove*, *move* or *rename* a tracked file use **`git`**:

```
git rm <filename>
```

```
git mv <oldname> <newname>
```

Remember to **`commit`** these changes!

```
git commit -m "File (re)moved."
```

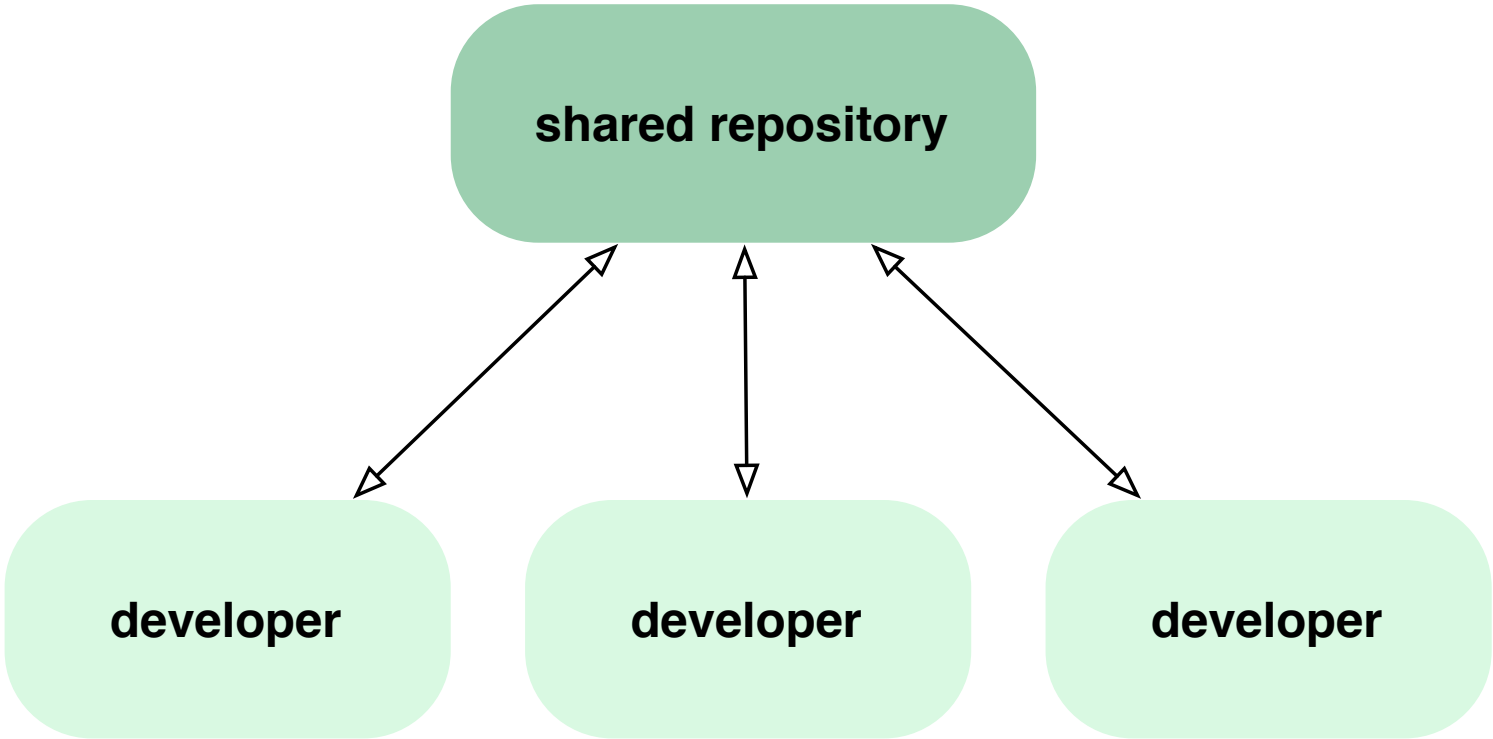
Single+Local `git`. Demo.

Demo: **`demo_git_single_local.txt`**

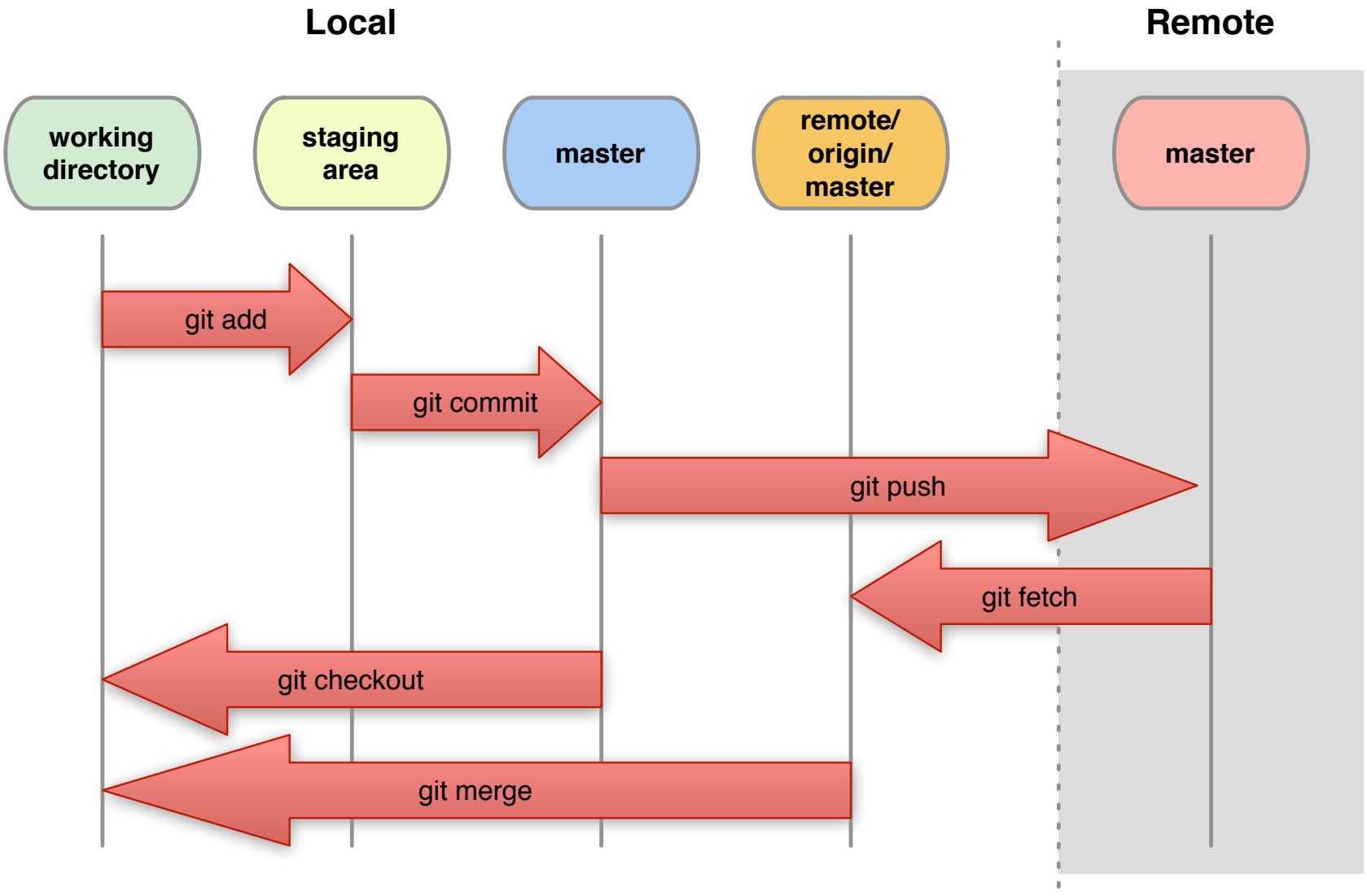
multi+remote/shared `git`.

Scenario 2: multiple developers + remote central repository.

multi+remote/shared `git`.



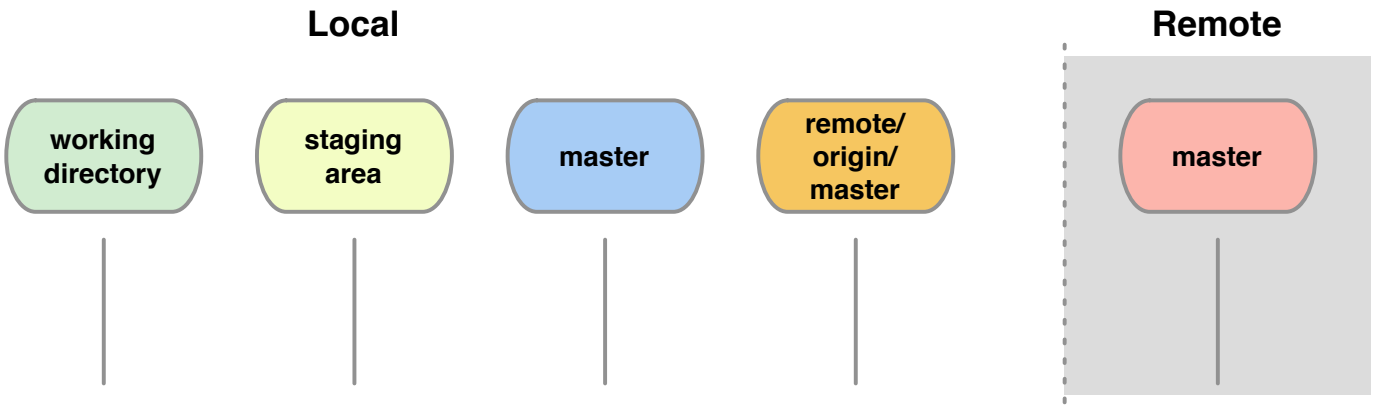
multi+remote/shared `git`.



multi+remote/shared `git`.

`git clone <URL>`

Creates *two* local copies of the whole remote repository.



Available transport protocols:

- `ssh://`, `git://`, `http://`, `https://`, `file://`

Ex.: `git clone git://github.com/hanke/PyMVPA`

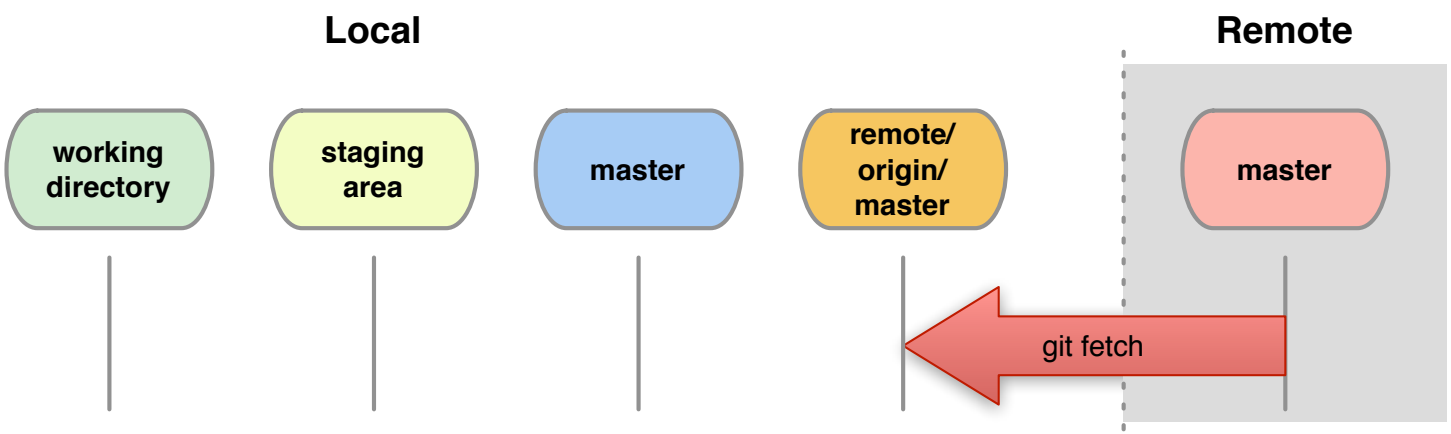
`git remote -v`

Shows name and `URL` of the remote repository.

multi+remote/shared `git`. Fetch.

`git fetch`

- Downloads updates from remote master to local remote master.
- The local master, staging area and working directory do not change.

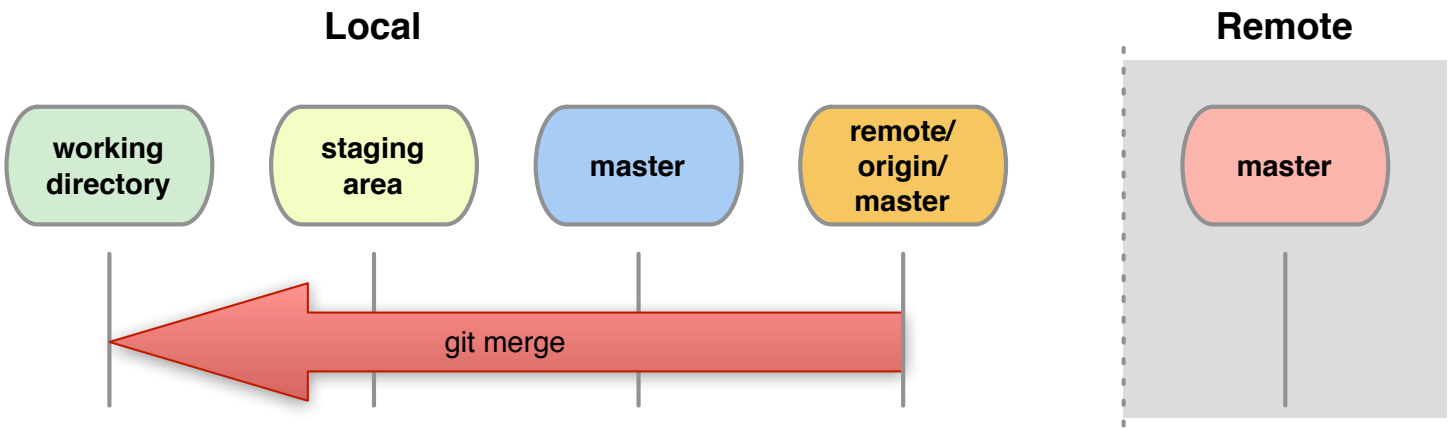


Q: Why **origin**?

A: Just a label for **Remote**. Choose the one you like.

`git merge`

- Joins development histories together.
- Warning: can generate *conflicts*!
- **Hint**: merge only when all changes are committed.



`git fetch` + `git merge` = `git pull`

Conflict!

```
...
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
...
```

multi+remote/shared `git`.

How to resolve conflicts.

1. See where conflicts are:

```
git diff
```

2. Edit conflicting lines.
3. Add changes to the staging area:

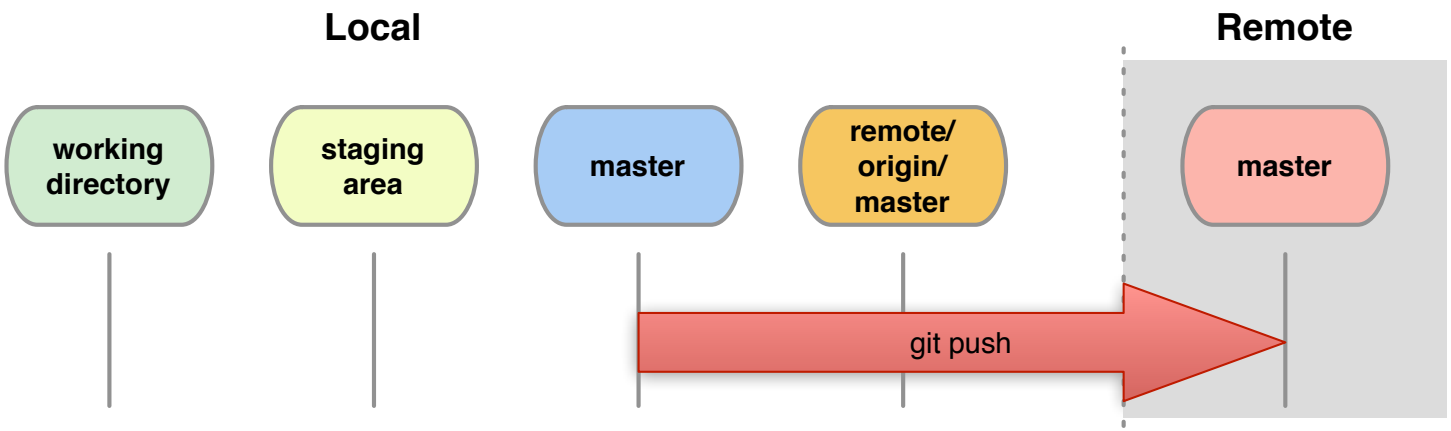
```
git add file1 [...]
```

4. Commit changes:

```
git commit -m "Conflicts solved."
```

`git push`

- Updates *remote masters* (both Local and Remote).
- Requires **`fetch+merge`** first.



multi+remote/shared `git`.

Demo: **`demo_git_multi_remote.txt`**.

Other related files:

- **`create_remote_repo_sn.sh`**
- **`collaborator1.sh`**
- **`collaborator2.sh`**
- **`collaborator2.sh`**

Extras. OPTIONAL

- Local branching + demo.
- Setting up a remote shared repository + demo.

Local branching. OPTIONAL

```
git branch
```

Shows names of local branches.

```
git branch new_feature
```

Creates a new branch named **new_feature**.

```
git checkout new_feature
```

Switches to branch **new_feature**.

```
git checkout master
```

Switches back to the **master** branch.

```
git merge new_feature
```

Merge **new_feature** changes into **master**.

Local branching. OPTIONAL

Demo: **`demo_git_branching_local.txt`**.

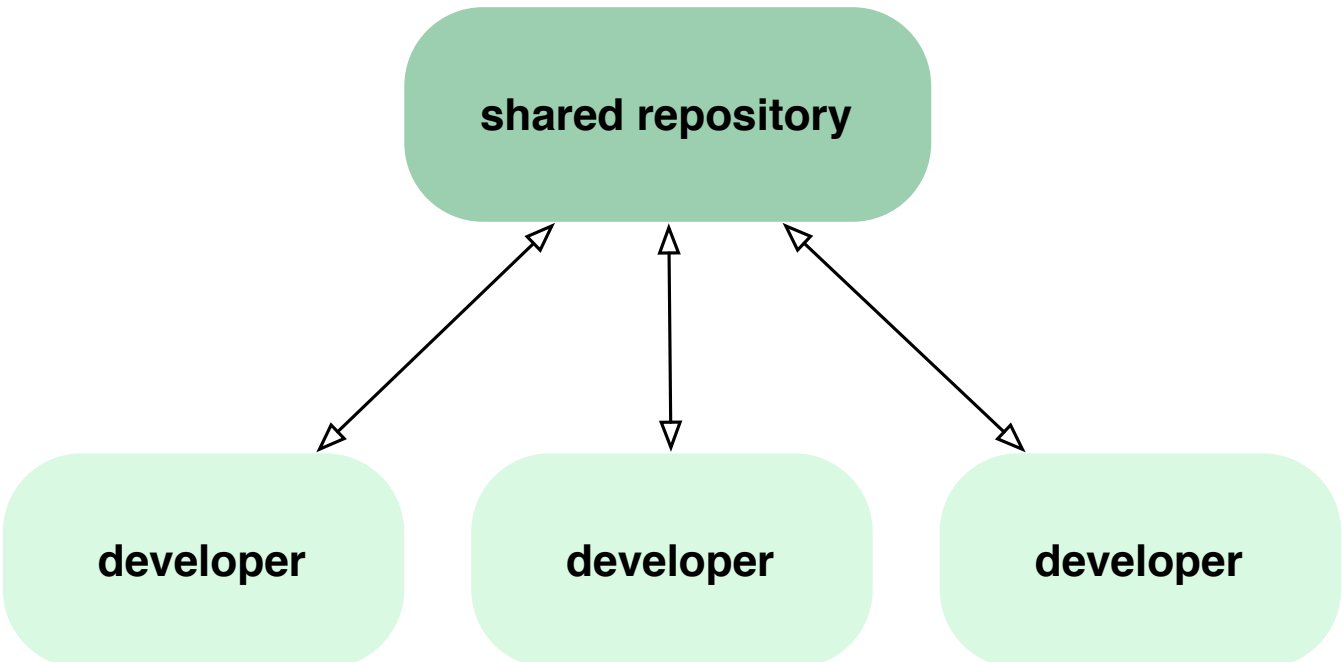
Setting up a remote+shared repository. OPTIONAL

GOAL: I want to share my local repository so others can **`push`**.

"Why can't I just *extend permissions* in my **local** repo?"

- Yes you can...
- ...but your colleagues will not push (read-only).

To have it read-write: set up a **remote** *shared* repository.



Setting up a remote+shared repository. OPTIONAL

You have a local repository and want to share it (**ssh**) from a remote server.

On *remote* server create **bare+shared** repository:

- `mkdir newproject`
- set up proper *group* permissions: `chmod g+rws newproject`
- `cd newproject`
- `git --bare init --shared=group`

On *local* machine push your repository to remote:

- `git remote add origin`
  `ssh://<user>@remote.com/path/newproject`
- `git push -u origin master`

Everybody clones the shared repository:

```
git clone ssh://<user>@remote.com/path/newproject
```

Setting up a remote+shared repository. OPTIONAL

Demo: **`demo_git_setup_remote.txt`**.

Repositories available for you

```
git clone ...
```

PacMan!

```
git://github.com/Debilski/pelita.git
```

Your ***personal*** `git` repository (empty):

```
<username>@python.g-node.org:/git/<username>
```

Your ***group<X>*** `git` repository (empty):

```
<username>@python.g-node.org:/git/group<X>
```

Q1: Why "`<repo>.git`"?

Just a reminder about the repository being **`bare`**.

Q2: Why "`ssh://<URL>/`" vs. "`<URL>:`" ?

absolute vs. relative (to *home*) path.

Credits

- Zbigniew Jędrzejewski-Szmek
- Tiziano Zito
- Bastian Venthur
- `http://progit.com`
- `apcmag.com`
- `lwn.net`
- `http://www.markus-gattol.name/ws/scm.html`

I want to know more about `git`!

Understanding how `git` works:

- `git` foundations, by Matthew Brett: `http://matthew-brett.github.com/pydagogue/foundation.html`
- The `git` parable, by Tom Preston-Werner: `http://tom.preston-werner.com/2009/05/19/the-git-parable.html`

Excellent guides:

- "Pro Git" book: `http://git-scm.com/book` (FREE)
- `git` magic: `http://www-cs-students.stanford.edu/~blynn/gitmagic/`

Cool Stuff

Code Swarm on NumPy & SciPy:

http://il.youtube.com/watch?v=wnaF24aVWTE&feature=related

Gource:

http://code.google.com/p/gource/