****

**Git Version Control**

**Submitted By:**

**Shaikh Sameer**

### Git Introduction

History

• before 2005: Linux sources were managed with Bitkeeper

(proprietary DVCS tool) 5

• 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

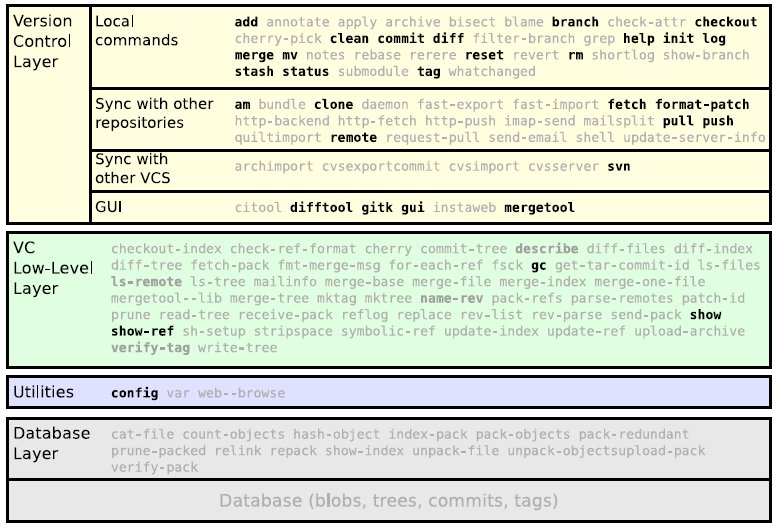
• pluggable merge strategies

• staging area (index)

• Performance

• no delta encoding

Git Commands



**Local Git**

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**6

6unless 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**

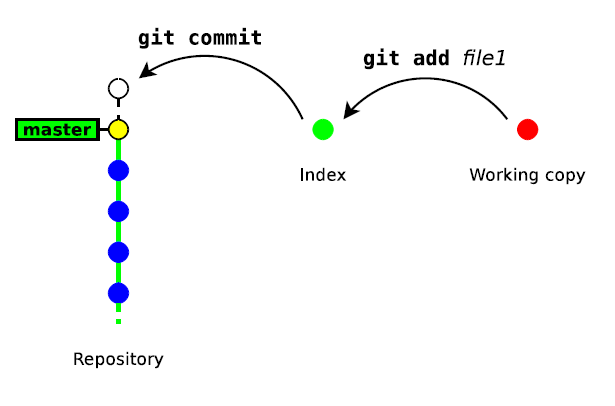
(Also called **index**)

The index stores the files scheduled for the next commit:

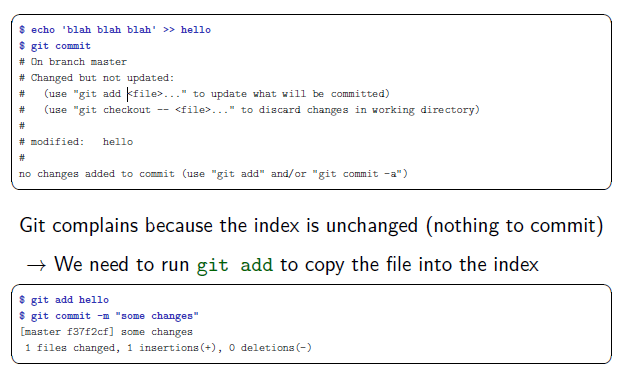
• git add *files* ! copy files into the index

• git commit ! commits the content of the index

**The staging area (aka the “index”)**



Update a file



**Bypassing the index7**

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

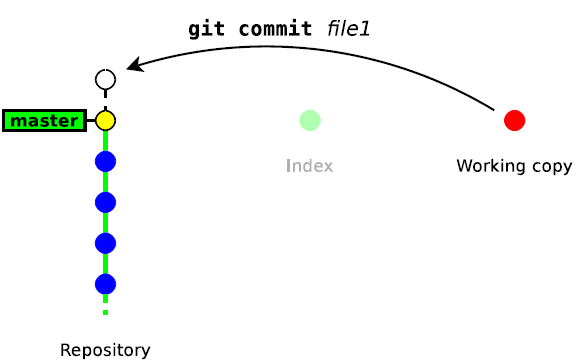
**Note:** when bypassing the index, GIT ignores new files:

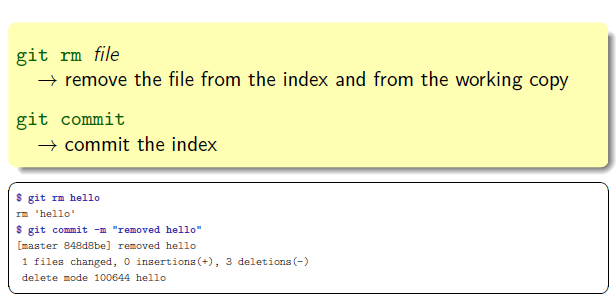
• “git commit .” commits only files that were present in the last commit

(updated files)

• “git add . && git commit” commits everything in the working tree

(including new files)



**Deleting Files**

**Showing Difference**

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**

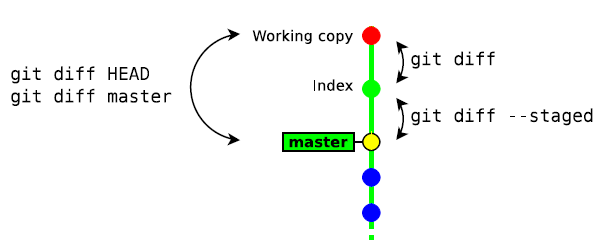
git diff --staged [ *rev a* ] [ -- *path . . .* ]

! 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**





**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 index8, but the

working copy is left intact

• git reset --hard drops all the changes in the index **and** in

the working copy

**Resetting changes in the working copy**

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)

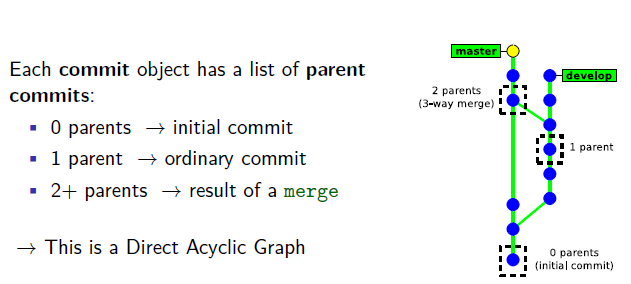
• git log : show the history

• git mv : move/rename a file9

• git tag : creating/deleting tags (to identify a particular

revision)

**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 committed 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**

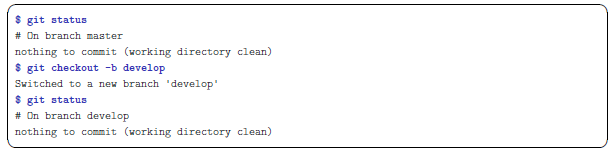
git checkout -b *new branch* [ *starting point* ]

• *new branch* is the name of the new branch

• *starting point* is the starting location of the branch (possibly a

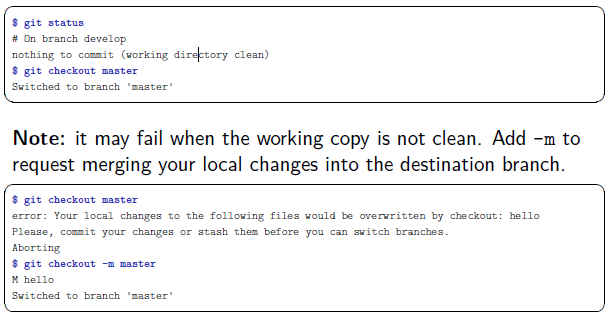
commit id, a tag, a branch, . . . ). If not present, git will use

the current location.



**Switching between branches**

git checkout [-m] *branch name*

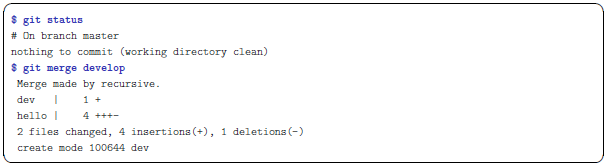


**Merging a branch**

git merge *other branch*

This will merge the changes in *other branch* into the current

branch.



**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 other branch.

! if the branch was already merged previously, then only the

Changes since the last merge will be merged.

**Branching example**



**How Git merges files ?**

If the same file was independently modified in the two branches,

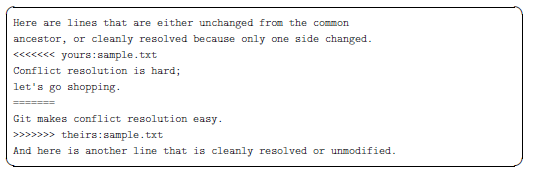
then Git needs to merge these two variants

• **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 <<<<<<< >>>>>>>



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

**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

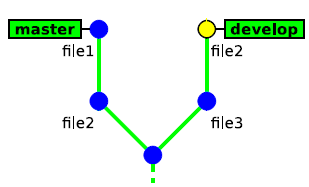
• or with a conflict resolution tool(xxdiff, kdiff3, emerge, ...)

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.**



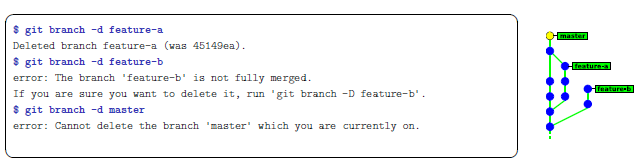
**Deleting branches**

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 is safe

unlike git branch -D which **deletes unconditionnally** ( ) the branch

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”

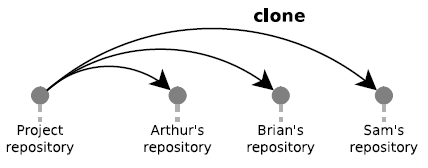
5. make a new commit in each branch so as to generate a conflict (edit the

same part of a file)

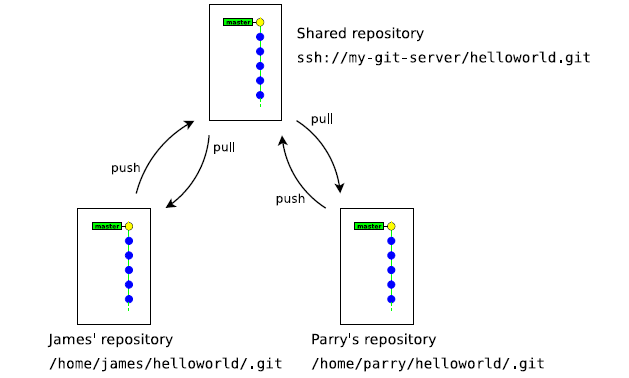
6. merge branch “develop” into “master”, and fix the conflict

7. merge “master” into “develop”

**Team Workflow**



**Simple workflow (Centralised)**



**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**12

• 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

**Pushing (uploading) local changes to the remote repository**

git push [ --tags ]

• git push examines the current branch, then:

• if the branch is tracking an upstream branch, then the local

changes (commits) are propagated to the remote branch

• if not, then nothing is uploaded

*(new local branches are considered private by default)*

• In case of conflict git push will fail and require to run git

pull first

**Pushing a new branch to the remote repository**

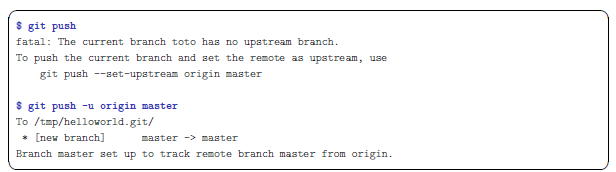
git push -u *destination repository ref* [*ref . . .* ]

• explicit variant of git push: the local reference *ref* (a branch

or a tag) is pushed to the remote *destination repository*

• -u/--set-upstream configures the local branch to track the

remote branch13 (this is usually what you want)



**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

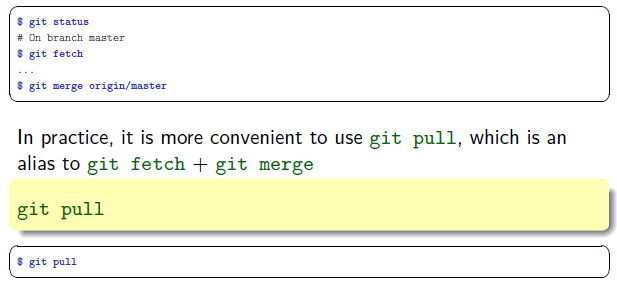
repository is updated to match the new position of the branch

master in the remote repository

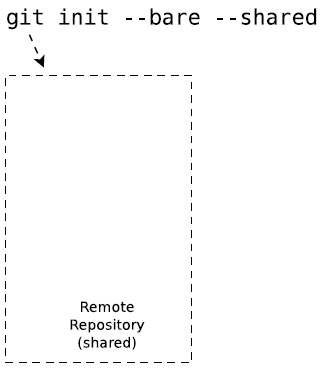
**Merging remote changes into the current local branch**

Changes in the remote repository can be merged explicitely into

the local branch by running git merge



**Remote example**



**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

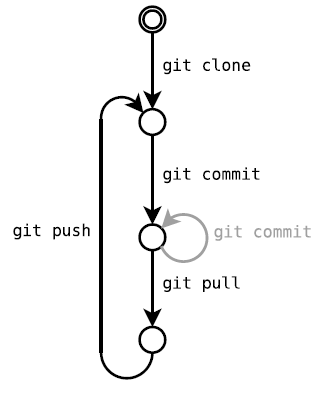
5. git checkout master

• 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**



**Server**

**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)

**Admin Considerations**

Administrating a GIT server is relatively simple14

• 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 git commands are local)*

• server outages are much less disruptive

*(user can collaborate by other means)*

• only core developers need write access

**How to publish a GIT repository (1/2)**

• Native protocol (git daemon) on tcp port 9418

• public access only, no authentication

! git://server.name.org/path/to/the/repository.git

• GIT over SSH

• 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)**

• HTTP/HTTPS server

• Firewall friendly

• Many authentication methods (provided by the HTTP server)

• can provide SSL encryption, even for anonymous users

\*http://username@server.name.org/path/to/the/repository.git

• *Dumb server* (repository published as static files)

• Very easy to set up (in read-only mode)

• Less efficient

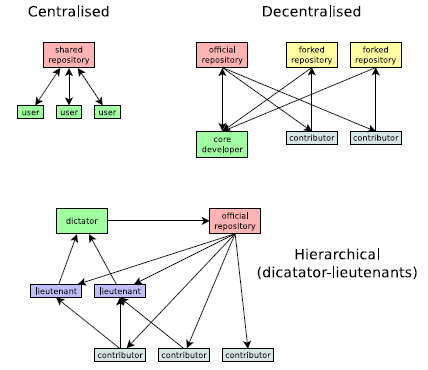
• Read-write mode requires webdav

• *Smart server* (git http-backend)

• cgi script running the native daemon over HTTP

• Backward-compatible with the dumb client

**Common workflows**



### Explicit pull/push

push and pull can work on any arbitrary repository

git push *url local branch*

! push the *local branch* to the repository *url*

git pull *url remote branch*

\*merge the *remote branch* from the repository *url* into the current local branch