



Version Control Systems.

GIT



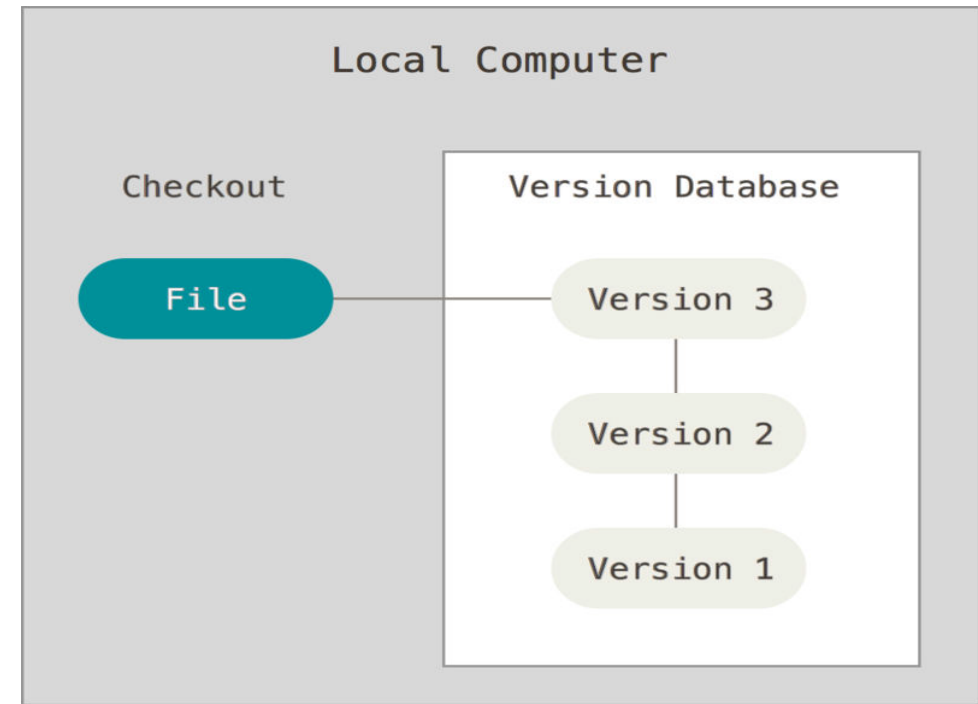
1. What is a Version Control System?

- It is a system(software) that records changes to a file or set of files over time.
- You can recall specific versions later.
- Types:
 - Local Version Control Systems
 - Centralized Version Control Systems
 - Distributed Version Control Systems



1.1 Local Versions Control Systems

- It is a simple database that kept all the changes to files under revision control.
- Example: RCS, it works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.



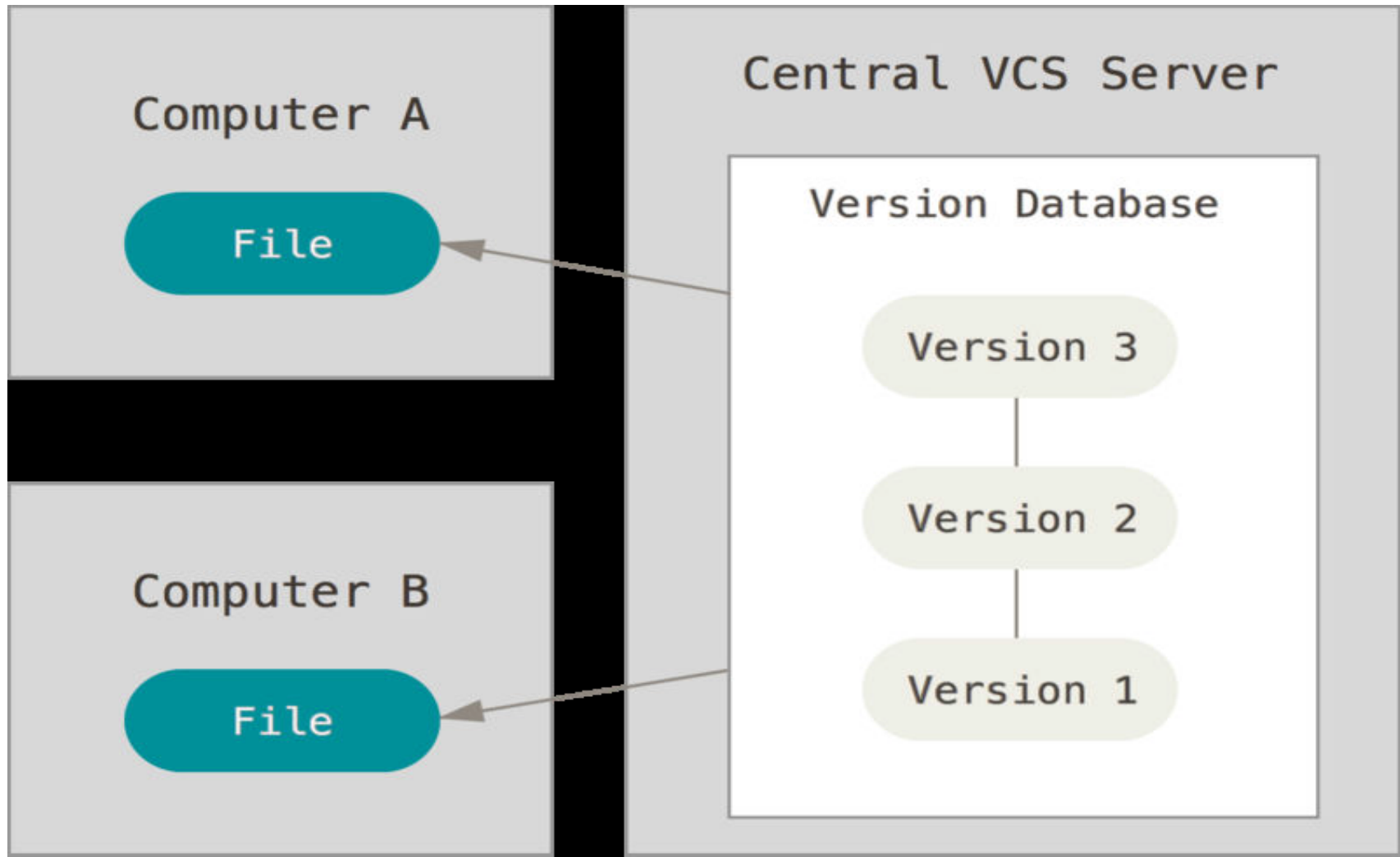


1.2 Centralized Version Control Systems

- If developers need to collaborate with each other they will need a Centralized Version Control Systems.
- It consist in a single server that contains all the versioned files, and a number of clients that check out files from that central place.
- For many years, this has been the standard for version control.
- Examples: CVS, Subversion, and Perforce,



1.2 Centralized Version Control Systems



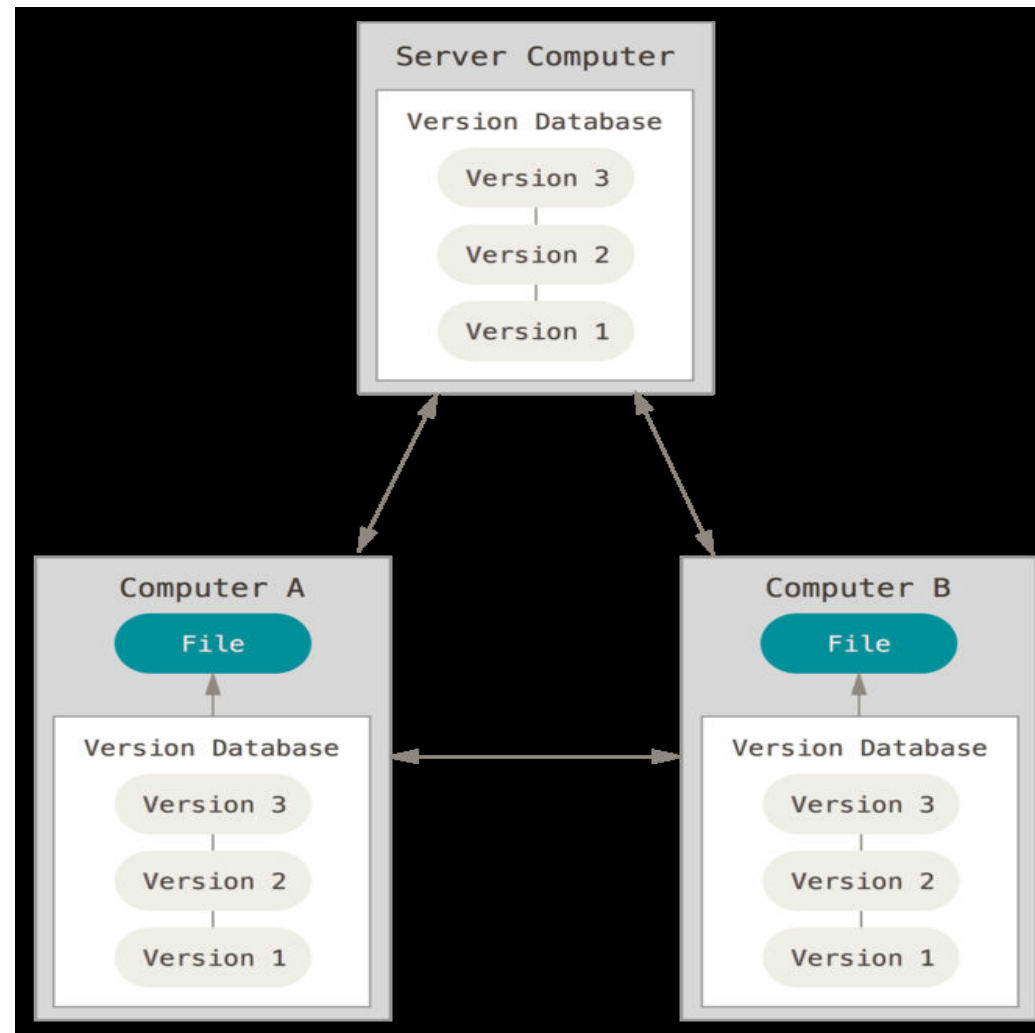


1.3 Distributed Version Control Systems

- Centralized Version Control Systems have a problem: what happen if the server goes down for an hour, or the hard disk the central database is on becomes corrupted ?
- This problem can be solved with Distributed Version control systems.
- Clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it.



1.3 Distributed Version Control Systems



Web Applications Development

Web Application Deployment

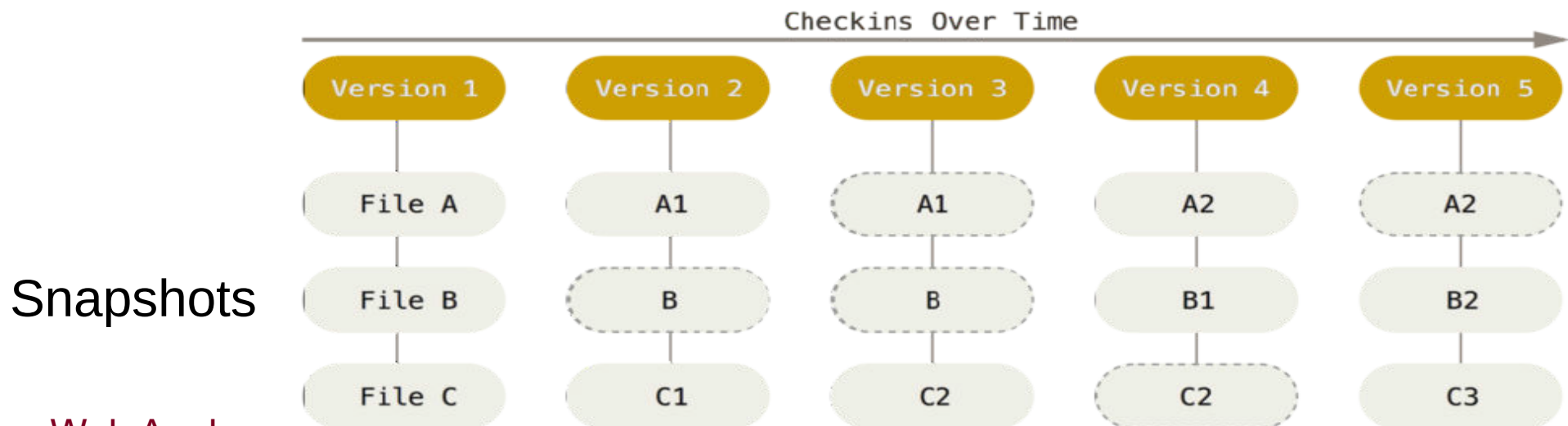
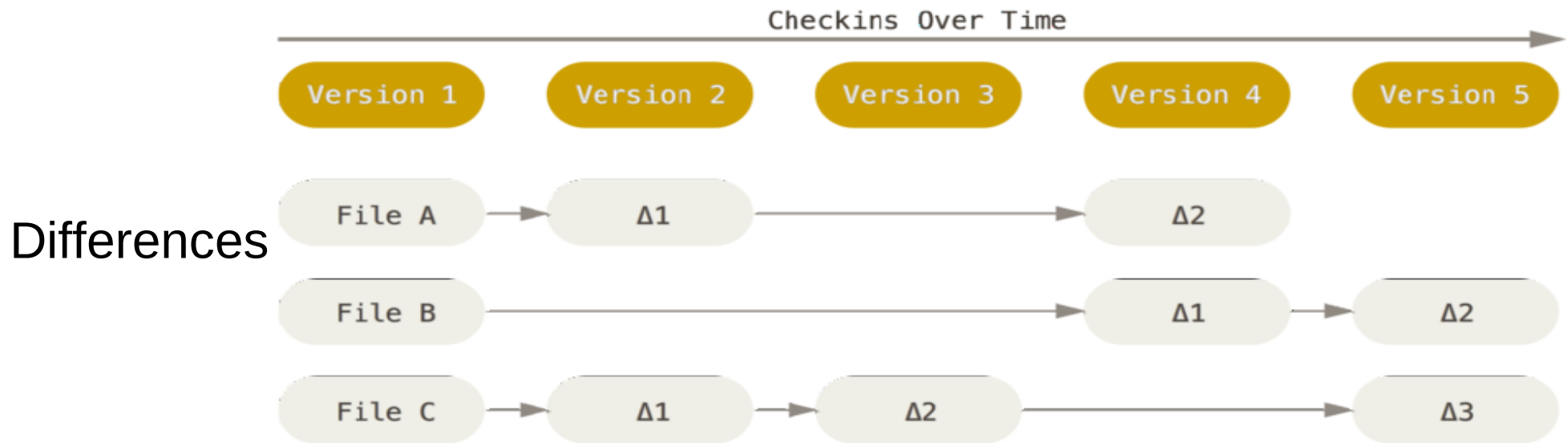


2. How does GIT work?

- Snapshots, Not Differences:
 - Most other systems store information as a list of file-based changes, differences
 - GIT every time a change is committed, takes a picture of what all files look like at that moment and stores a reference to that snapshot.
 - If files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.



2. How does GIT work?



Web Applications Development

Web Application Deployment



2. How does GIT work?

- Most operations in Git only need local files and resources to operate because you have the entire history of the project on your local disk.
- Git has three main states that files can reside in:
 - Committed: the data is safely stored in your local database
 - Modified: you have changed the file but have not committed it to your database yet
 - Staged: you have marked a modified file in its current version to go into your next commit snapshot.



2. How does GIT work?

- There are three main sections in a Git project:
 - The Git directory: is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
 - The working directory: is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.



2. How does GIT work?

- The staging area: is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It's sometimes referred to as the “index”, but it's also common to refer to it as the staging area.



2. How does GIT work?

- The basic Git workflow goes something like this:
 1. You modify files in your working directory.
 2. You stage the files, adding snapshots of them to your staging area.
 3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.



3. Installing GIT

- Installing on Linux
 - To install run the following command “sudo apt-get install git”
 - git help -- all, all available commands are printed on the standard output.

3. Installing GIT

Customizing GIT

- git config is a tool to set configuration variables that control all aspects of how Git looks and operates.
- These variables can be stored in three different places:
 - /etc/gitconfig file(-- system option): Contains values for every user on the system and all their repositories.
 - ~/.gitconfig or ~/.config/git/config file(-- global option): Specific to your user.
 - config file in the Git directory (that is, .git/config) of whatever repository you're currently using (without any option).

Each level overrides values in the previous level, so values in .git/config trump those in /etc/gitconfig.

3. Installing GIT

- Customizing GIT

- An user name and an email address are needed:
 - `git config --global user.name "mi nombre"`
 - `git config --global user.email minombre@abastos.org`
- You can also set a default text editor associated with git
 - `git config --global core.editor gedit`



4. Getting a GIT Repository

- There are two main approaches
 - Take an existing directory and import it to GIT
 - Clone an existing repository from another GIT server



4. Getting a GIT Repository

- Take an existing directory and import it to GIT
 - Go to the directory
 - Type *git init*. This creates a new subdirectory named `.git` that contains all of your necessary repository files (a Git repository skeleton)
 - Type *git add file-name* in order to indicate the files you want to track. Examples:
 - `git add LICENSE`
 - `git add *.c`
 - Type *git commit* to confirm the changes. Example:
 - `git commit -m 'initial project version'`



4. Getting a GIT Repository

- Clone an existing repository from another GIT server

Type `git clone [url]`. Then Git receives a full copy of nearly all data that the server has. Examples:

- `git clone https://github.com/libgit2/libgit2`
- `git clone https://github.com/libgit2/libgit2 mylibgit`

That creates a directory named “libgit2”, initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2 directory, you’ll see the project files in there, ready to be worked on or used.

The second command does the same thing as the previous one, but the target directory is called mylibgit.



4.1 .git Directory Structure

- A .git directory has the following structure

```
cliente@cliente-VirtualBox:~/Documents/prueba1/.git$ ls -R
.:
branches  config  description  HEAD  hooks  info  objects  refs

./branches:

./hooks:
applypatch-msg.sample  pre-applypatch.sample  pre-push.sample
commit-msg.sample      pre-commit.sample      pre-rebase.sample
post-update.sample      prepare-commit-msg.sample  update.sample

./info:
exclude

./objects:
info  pack

./objects/info:

./objects/pack:

./refs:
heads  tags

./refs/heads:

./refs/tags:
cliente@cliente-VirtualBox:~/Documents/prueba1/.git$
```



4.1 .git Directory Structure

Where

- config is the repository specific configuration file.
- hooks are customization scripts used by various Git commands.
- info/exclude contain files you want to ignore from git it affects only local repositories
- Objects/ is the directory where the data of your Git objects is stored – all the contents of the files you have ever checked in, your commits, trees and tag objects.



4.2 .gitignore file

- You can create a *.gitignore* file to list patterns, the matched files will be ignored by git.
- We usually want Git to ignore files that are generally automatically such as:
 - a log files
 - files produced by your build system.



4.2 .gitignore file

- These are some rules for the patterns you can put in the .gitignore file
 - Blank lines or lines starting with # are ignored.
 - You can start patterns with a forward slash (/) to avoid recursivity.
 - You can end patterns with a forward slash (/) to specify a directory.
 - An asterisk (*) matches zero or more characters
 - You can negate a pattern by starting it with an exclamation point (!)



4.2 .gitignore file

- `[abc]` matches any character inside the brackets (in this case a, b, or c)
- A question mark (`?`) matches a single character;
- Brackets enclosing characters separated by a hyphen (`[0-9]`) matches any character between them (in this case 0 through 9).
- You can also use two asterisks to match nested directories; `a/**/z` would match `a/z`, `a/b/z`, `a/b/c/z`



4.2 .gitignore file

- Examples:

- Ignore all .a files

`*.a`

- But do track lib.a, even though you're ignoring .a files above

`!lib.a`

- Only ignore the TODO file in the current directory, not subdir/TODO

`/TODO`



4.2 .gitignore file

Examples:

- Ignore all files in any directory named build
build/
- Ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
- Ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf



4.2 .gitignore file

- You can find a helpful set templates of .gitignore files in

<https://github.com/github/gitignore>

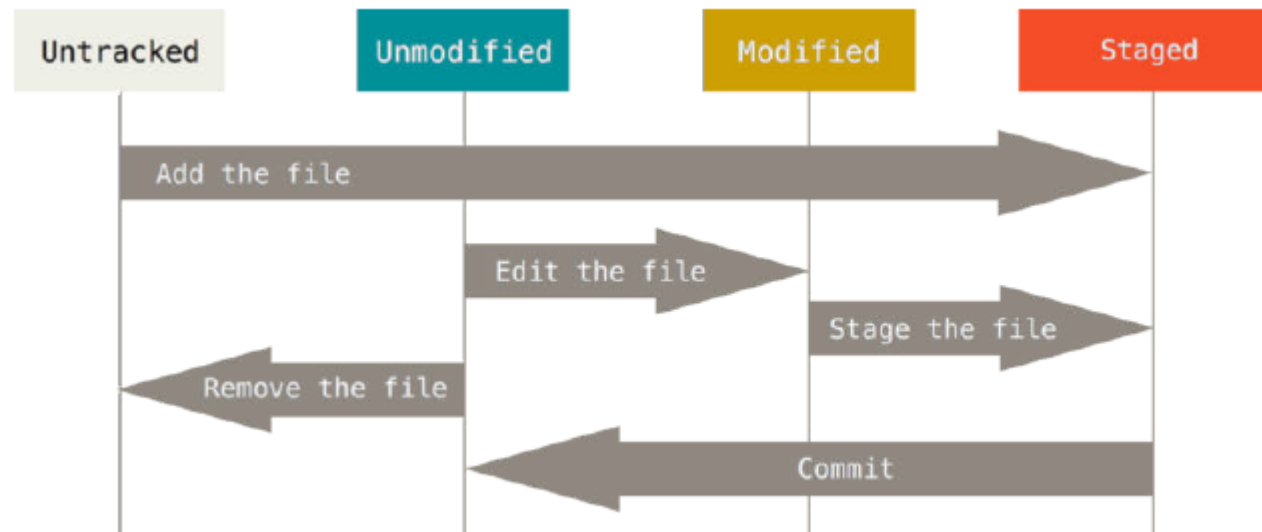


5. Recording changes to the Repository

- Files in your working directory can be in one of two states: tracked or untracked
 - Tracked files are files that were in the last snapshot. They can be unmodified, modified, or staged
 - Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area.
- When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven't edited anything.
- When you edit files, Git sees them as modified, because you've changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats



5. Recording changes to the Repository



The lifecycle of the status of your files.



5. Recording changes to the Repository

■ Checking status of files

- The git status command determine which files are in which state.

- Example n° 1:

If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
nothing| to commit, working directory clean
```

That means there are no tracked and modified files.



5. Recording changes to the Repository

Example n° 2

If you add a new file called README, when you run the command you should get

```
$ git status
On branch master
Untracked files:
(use "git add <file>..." to include in what will be committed)
    README
nothing added to commit but untracked files present (use "git add" to track)
```

Your new README file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so.

In order to begin tracking a new file, you use the command `git add`. To begin tracking the README file, you can run this

```
$ git add README
```

If you run the status command again, you can see that your README file is now tracked and staged to be committed.

It's staged because it's under the "Changes to be committed" heading

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



5. Recording changes to the Repository

Example nº 3

If you change a previously tracked file called “CONTRIBUTING.md” and then run your git status command again, you get something that looks like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

It appears under a section named “Changed but not staged for commit”. To stage it, you run the git add command(git add is a multipurpose command).Then, both files are staged and will go into your next commit

Web Applications Development

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

    new file:   README
    modified:   CONTRIBUTING.md
```




5. Recording changes to the Repository

Example nº 3 (cont)

At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it. If you run git status again after changing the file, you will see

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README
    modified:   CONTRIBUTING.md
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Now CONTRIBUTING.md is listed as both staged and unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the git add command. If you commit now, the version of CONTRIBUTING.md as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file.



5. Recording changes to the Repository

- Git diff show changes between commits, commit and working tree, etc.
 - Git diff show differences between the working directory and the staged area
 - Git diff --staged show differences between the staged area and the last commit.



6. Committing Your Changes

- You can commit the changes with `$ git commit`
- Remember that anything that is still unstaged – any files you have created or modified that you haven't run `git add` on since you edited them – won't go into this commit. They will stay as modified files on your disk.
- When you run ***git commit*** the editor of your choice is launched.
Example

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   CONTRIBUTING.md
~
".git/COMMIT_EDITMSG" 9L, 283C
```
- You can remove these comments and type your commit message, or you can leave them there to help you remember what you're committing. When you exit the editor, Git creates your commit with that commit message



6. Committing Your Changes

- Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
2 files changed, 2 insertions(+)
create mode 100644 README
```

- Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.



7. Skipping the staging area

- The staging area amazingly useful for crafting commits exactly how you want them.
- But, if you only want to commit every file that have been changed, you can skip the staging area. How can yo do this? Adding the option `-a` to the `git commit` command. Then Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
1 file changed, 5 insertions(+), 0 deletions(-)
```

Notice how you don't have to run `git add` on the "CONTRIBUTING.md" file in this case before you commit.



8. Removing files

- If you remove a file from your working directory, it shows up under the “Changed but not updated” (that is, unstaged) area of your git status output.

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        deleted:    PROJECTS.md
no changes added to commit (use "git add" and/or "git commit -a")
```

- To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The **git rm** command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.



8. Removing files

Example

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted:    PROJECTS.md
no changes added to commit (use "git add" and/or "git commit -a")
```

- The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the index already, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.
- Perhaps, you may want to keep the file on your hard drive but not have Git track it anymore. To do this, use the `--cached` option:

```
$ git rm --cached README
```



9. Renaming files

- If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

- This is equivalent to running something like this:

```
$ mv README.md README  
$ git rm README.md  
$ git add README
```

- Pay attention, ***git mv*** command doesn't move file, it renames files. Git doesn't explicitly track file movement.



10. Viewing the commit history

- After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.

Example

```
$ git clone https://github.com/schacon/simplegit-progit
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
    changed the version number
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
    removed unnecessary test
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 10:31:28 2008 -0700
    first commit
```

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and e-mail, the date written, and the commit message



10. Viewing the commit history

- Options:
 - p: show the difference introduced in each commit
 - <n>: limit the output to only the last “n” entries
 - stat: print below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.
 - shortstat: display only the changed/insertions/deletions line from the --stat command
 - name-only: show the list of files modified after the commit information
 - name-status: show the list of files affected with added/modified/deleted information as well
 - pretty: show commits in an alternate format. Options include one line, short, full, fuller, and format (where you specify your own format).



11 Undoing things

Undoing commits

- One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the --amend option:

```
$ git commit --amend
```

- This command takes your staging area and uses it for the commit
- Example: If you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'initial commit'  
$ git add forgotten_file  
$ git commit --amend
```



11 Undoing things

- Unstaging a staged file, the command to do this is ***git reset HEAD <file>***. Example:

```
git reset HEAD README
```

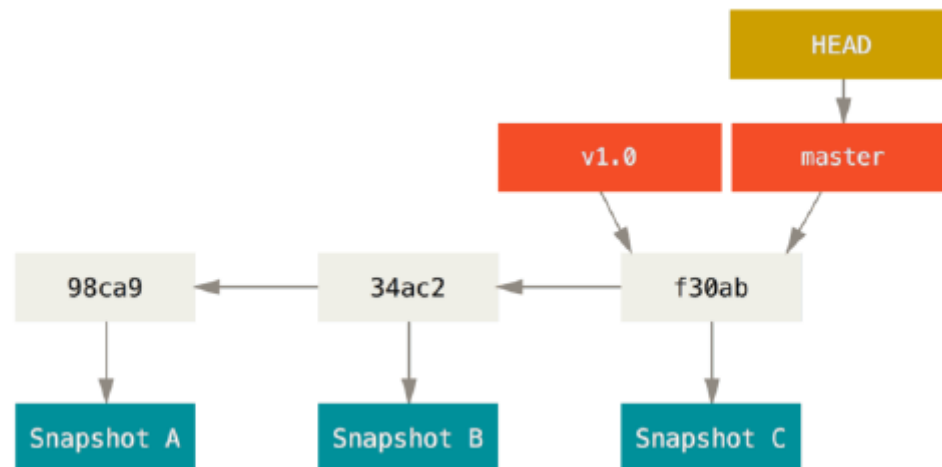
- Unmodifying a modified file, the command to do this is ***git checkout --<file>***. Example:

```
git checkout -- README
```



12 Git branching

- Branching means to diverge from the main line of development and continue to do work without messing with that main line.
- The default branch name in Git is **master**
- When commits are made a master branch is given, that master branch points to the last commit made.



Web Applications Development

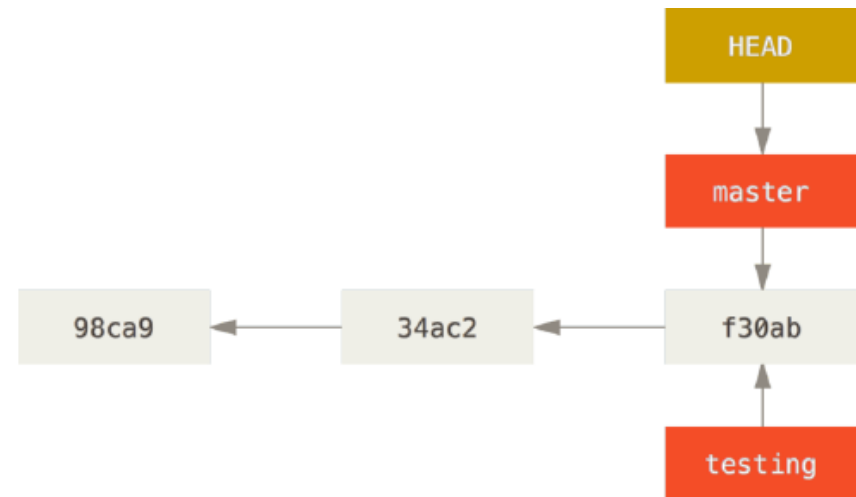
Web Application Deployment



12 Git branching

- HEAD is a special pointer that points to the local branch you are currently on.
- To add a New branch we use the *git branch* command. This creates a new pointer to the same commit we are currently on(HEAD). It doesn't change to the new branch. Example:

git branch testing





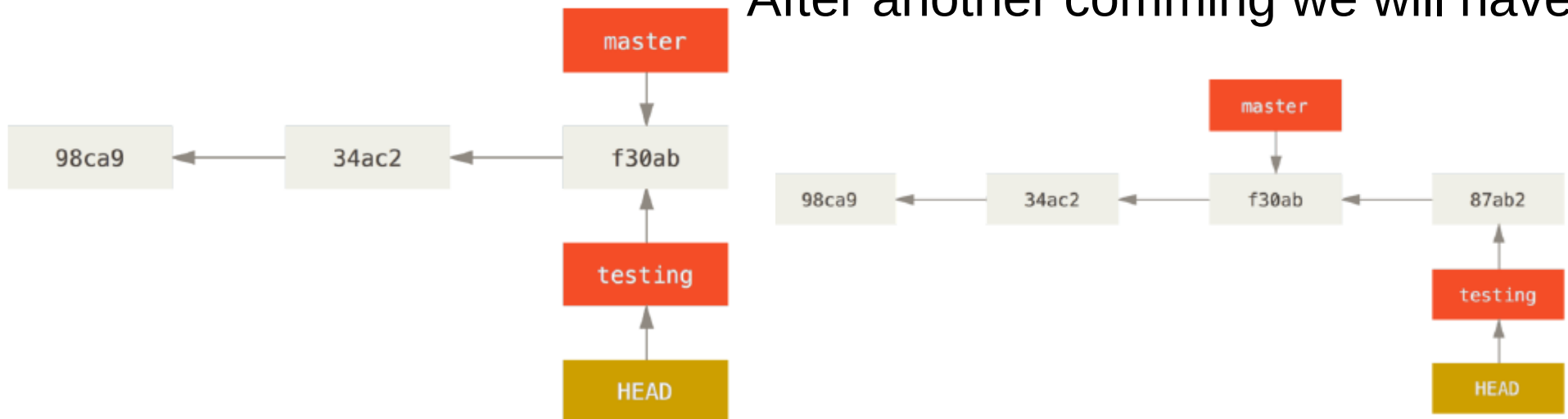
12 Git branching

- To switch to an existing branch, you run the git checkout command.

git checkout testing

This moves HEAD to point the testing branch.

After another comming we will have:






12 Git branching

- To create a new branch and switch to it, you run the following git command.

git checkout -b testing

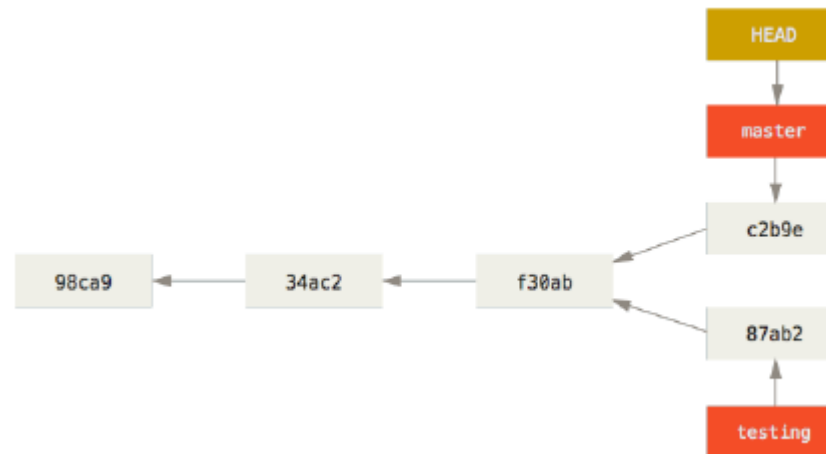
That is,

git checkout -b testing  {
git branch testing
+
git checkout testing



12 Git branching

- Switching branches changes files in working directory. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch.





12 Git branching

- The command `git branch` with no arguments shows a list of current branches.

git branch

```
$ git branch
lss53
* master
testing
```

Notice the *character that prefixes the master branch, it indicates the branch that you are currently have checked out(the branch that HEAD points to)

- To see the last commit on each branch, the option is `-v`

git branch -v

```
$ git branch -v
lss53 93b412c fix javascript issue
* master 7a98805 Merge branch 'lss53'
testing 782fd34 add scott to the author list in the readmes
```



12 Git branching

- To delete a branch run

git branch -d testbranch

- To get some information about branches that have been merged into the branch you are currently on, you can run

git branch --merged

```
$ git branch --merged  
lss53  
* master
```

Branches on this list without the * in front of them are generally fine to delete, you have already incorporated their work into another branch, so you are not going to loose anything



12 Git branching

- To get some information about branches that have not been merged into the branch you are currently on you, can run

git branch --no-merged

```
$ git branch --no-merged  
testing
```

This branch contains work that isn't merged in yet, trying to delete it with `git branch -d` will fail

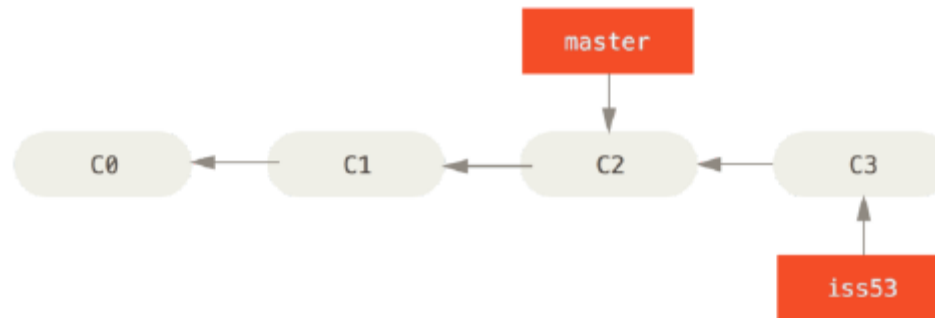
```
$ git branch -d testing  
error: The branch 'testing' is not fully merged.  
If you are sure you want to delete it, run 'git branch -D testing'
```

If you really do want to delete the branch and lose that work, you can force it with `-D`, as the helpful message points out



13 Basic Branching and Merging. Example

- Imagine that
 - You work on a web site.
 - Create a branch for a new story you are working on(iss53)
 - Do some work in that branch

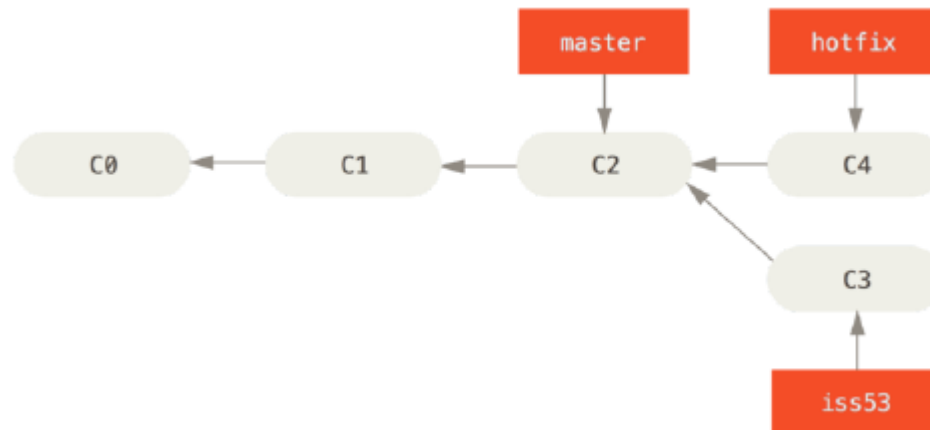




13 Basic Branching and Merging. Example

At this point, you'll receive a call that another issue is critical and you need a hotfix, then you will do the following

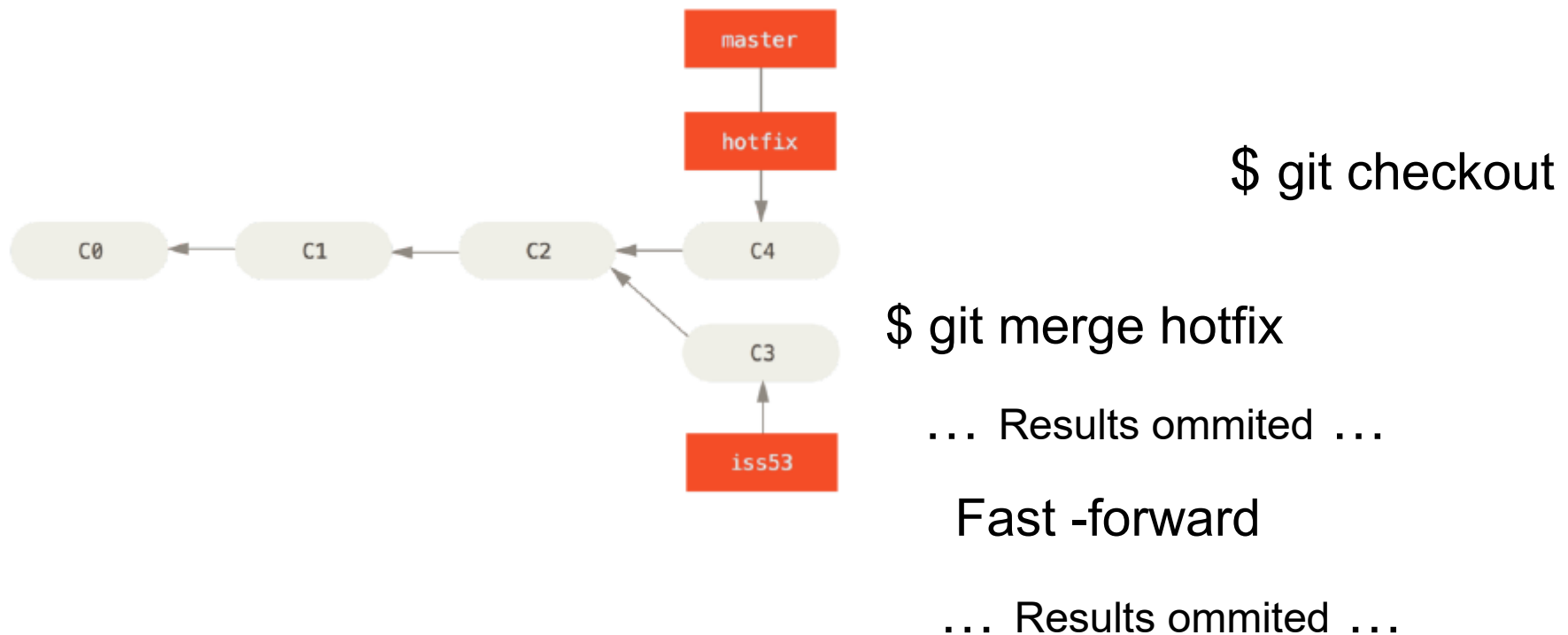
- Before switching branches, it is best to have a clean working state, so you should commit the changes.
- Switch to your production branch(master)
- Create a branch to add the hotfix (hotfix)
- Switch to hotfix branch and work





13 Basic Branching and Merging. Example

- After it's tested, you should change to master branch and merged the hotfix branch and push to production.



Notice the phrase “**Fast-forward**”, it means that Git simplifies things by moving the pointer forward because there is no divergent work to merge together.



13 Basic Branching and Merging. Example

• We continue working on branch iss53

• Now we decide to merge master and iss53

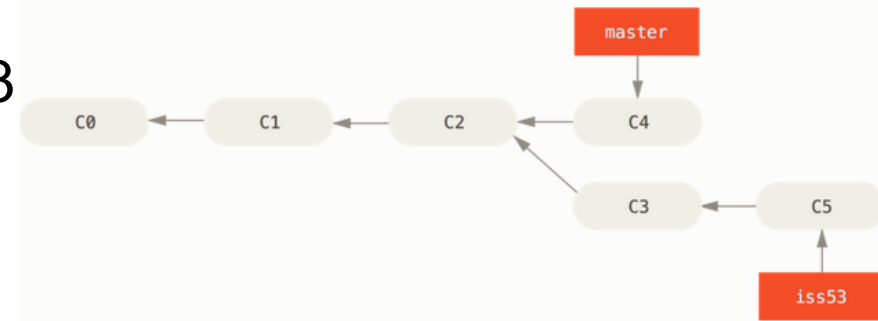
\$git checkout master

\$git merge iss53

Merge made by the 'recursive' strategy

.....

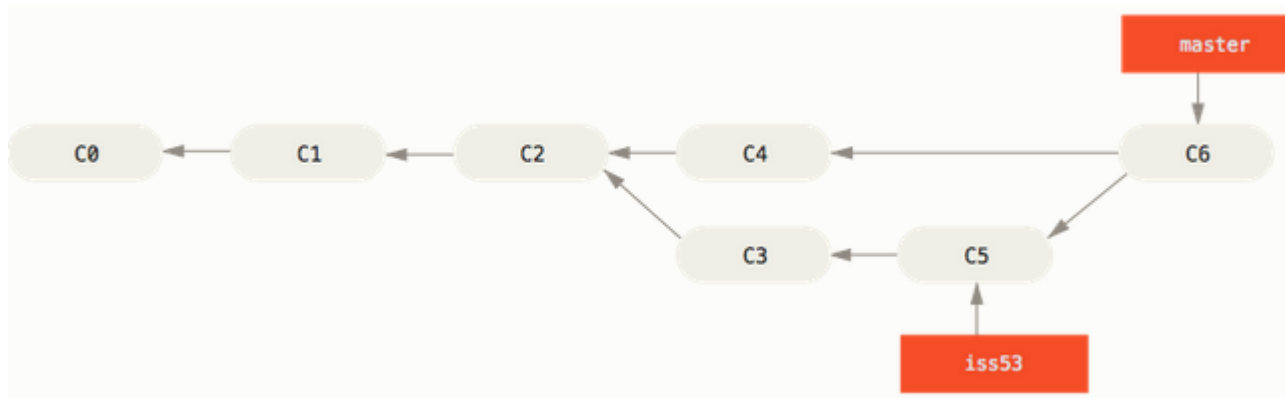
In this case, the issue history diverges from some older point, that is, the branch we are on isn't a direct ancestor of the branch we are trying to merge in. Git does a simple three-way merge, using master(C4) and iss53(C5) snapshots and the common ancestor of the two(C2).





13 Basic Branching and Merging. Example

- Git creates a new snapshot that has more than one parent.



- Finally, we can delete branch iss53

`$ git branch -d iss53`



13.1 MergeConflicts

- **MERGE CONFLICTS** happen when the two branches have changed the same part of the same file. Something like that is going to be shown

CONFLICT (content): Merge conflict in XXXXX.YYY

Automatic merge failed; fix conflicts and then commit the result

- Git adds standard conflict-resolution markers to the files that have conflicts

```
<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>> iss53:index.html
```

- The version in HEAD is the top part of that block (everything above the =====), while the version in your iss53 branch looks like everything in the bottom part.
- To resolve the conflict, you have to either choose one side or the other or merge the contents yourself.
- Finally, you must run the commands **git add...** and **git commit** again.



14 GIT on the Server

- It is the preferred method for collaborating with someone.
- You would have an intermediate repository that you and your collaborators have access to, and push to and pull from that.
- This intermediate or remote repository is generally a *bare repository*; a Git repository that has no working directory. It is the contents of your project's .git directory and nothing else.



14 GIT on the Server

- Git can use three major protocols to transfer data: HTTP, SSH and Git protocol.
 - SSH: access over SSH is secure, it is an authenticated network protocol and all data transfer is encrypted and as compacted as possible,
 - HTTP: It is easy to use, however it allows a read-only access
 - Git protocol: it is often the fastest network transfer protocol available. It uses the same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead. It's also probably the most difficult protocol to set up.



14 GIT on the Server

■ To set up a Git server you have to follow several steps:

1) It is assumed that you have a Local Git, and a repository.

- Export an existing repository into a new bare repository. By convention, bare repository directories end in *.git*
 - Go to the parent directory that you have previously create a repository
 - Run

git clone --bare directory-name directory-name.git

- Example

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

You should now have a copy of the Git directory data in your my_project.git directory



14 GIT on the Server

2) It is assumed that you have a remote server where git, apache2 and openssh-server have already been installed and set up:

- `sudo apt-get install git`
- `sudo apt-get install openssh-server`
- `sudo apt-get install apache2`
- `sudo a2enmod cgi rewrite`

3) In the remote server the repositories are going to store under a shared directory, for instance:

- `sudo mkdir /opt/git`



14 GIT on the Server

- 4) Put the bare repository on the server. We will use the ssh to copy the bare repository on the server:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

- 5) Other users who have SSH access to the same server which has read-access to the /opt/git directory can clone your repository by running

```
$ git clone user@git.example.com:/opt/git/my_project.git
```



15 Working with remotes

- To be able to collaborate on any Git project, you need to know how to manage your remote repositories
- Remote repositories are versions of your project that are hosted on the Internet or network somewhere
- You can have several of them, each of which generally is either read-only or read/write for you.
- Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.
- Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not



15 Working with remotes

● Showing remotes.

- you can run the **git remote** command to see which remote servers you have configured

```
$ git remote  
origin
```
- If you've cloned your repository, you should at least see origin – that is the default name Git gives to the server you cloned from
- With -v option, Git shows the URL's that Git has stored for the shortname to be used when reading and writing to that remote

```
$ git remote -v  
bakkdoor https://github.com/bakkdoor/grit (fetch)  
bakkdoor https://github.com/bakkdoor/grit (push)  
koke git://github.com/koke/grit.git (fetch)  
koke git://github.com/koke/grit.git (push)  
origin git@github.com:mojombo/grit.git (fetch)  
origin git@github.com:mojombo/grit.git (push)
```



15 Working with remotes

● Adding remotes.

- To add a new remote Git repository as a shortname you can reference easily, run ***git remote add [shortname] [url]***

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb      https://github.com/paulboone/ticgit (fetch)
pb      https://github.com/paulboone/ticgit (push)
```

- if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run ***git fetch pb***:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
* [new branch]      master    -> pb/master
* [new branch]      ticgit    -> pb/ticgit
```

Paul's master branch is now accessible locally as pb/master – you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it



15 Working with remotes

● Fetching and pulling from remotes

- It's important to note that the *git fetch* command pulls the data to your local repository – it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.
- If you have a branch set up to track a remote branch, you can use the *git pull* command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the *git clone* command automatically sets up your local master branch to track the remote master branch



15 Working with remotes

Pushing to remotes

- When you have your project at a point that you want to share, you have to push it upstream
- The command to push your master branch to your origin server is ***git push [remote-name] [branch-name]***. This push any commits you've done back up to the server.
- Git push*** works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to pull down their work first and incorporate it into yours before you'll be allowed to push



15 Working with remotes

- **Inspecting remotes:** to see more information about a particular remote, you can use the `git remote show <remote>` command and you will get something like this:

```
$ git remote show origin
```

```
* remote origin
```

```
Fetch URL: https://github.com/schacon/ticgit
```

```
Push URL: https://github.com/schacon/ticgit
```

```
HEAD branch: master
```

```
Remote branches:
```

```
master                tracked
```

```
dev-branch            tracked
```

```
Local branch configured for 'git pull':
```

```
master merges with remote master
```

```
Local ref configured for 'git push':
```

```
master pushes to master (up to date)
```



15 Working with remotes

Renaming and removing remotes:

- You can run **git remote rename** to change a remote's shortname. For instance, if you want to rename pb to paul, you can do so with git remote rename:

```
$ git remote rename pb paul
```

```
$ git remote
```

```
origin
```

```
Paul
```

- To remove a remote for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can either use git remote remove or git remote rm:

```
$ git remote remove paul
```

```
$ git remote
```

```
origin
```



15 Working with remotes

● Tagging

- Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points
- Git uses two main types of tags: lightweight and annotated
 - A lightweight tag is just a pointer to a specific commit.

```
$ git tag v1.4-lw
$ git tag
v0.1
```

- Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard. The easiest way is to specify -a when you run the tag command.

The -m specifies a tagging message

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```



15 Working with remotes

● Tagging latter

- You can also tag commits after you've moved past them. Suppose your commit history looks like this:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
6d52a271eda8725415634dd79daabb4d9b6008e Merge branch 'experiment'
```

- To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

```
$ git tag -a v1.2 9fceb02
```




15 Working with remotes

● Tagging

- ***Git tag*** lists the tags in alphabetical order
- You can also search for tags with a particular pattern

● Example

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
```



15 Working with remotes

■ Sharing Tags:

- By default, the `git push` command doesn't transfer tags to remote servers.
- You will have to explicitly push tags to a shared server after you have created them:
 - `git push origin v1.5`
- If you have a lot of tags that you want to push up at once, you can run:
 - `git push origin --tags`
- After that, when someone else clones or pulls from your repository, they will get all your tags as well.



Bibliography

- Pro Git

Scott Chacon and Ben Straub

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.