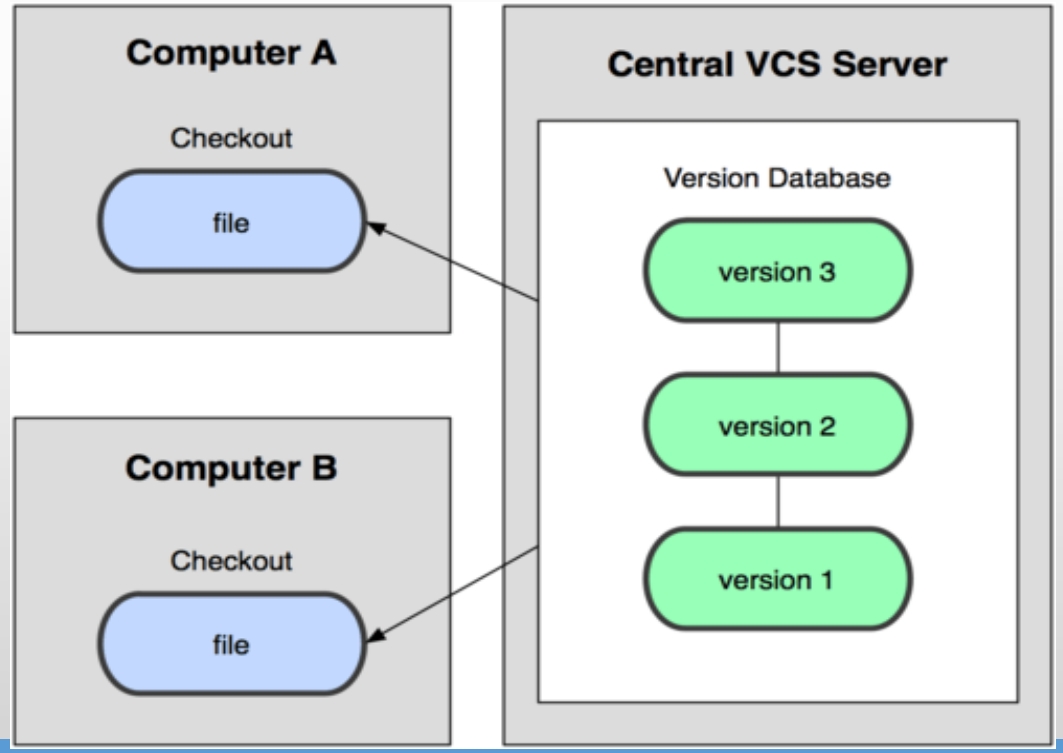
Version Control System - Basics

What is Version Control System?

- Version Control System (VCS) is a software that helps software developers to work together and maintain a complete history of their work



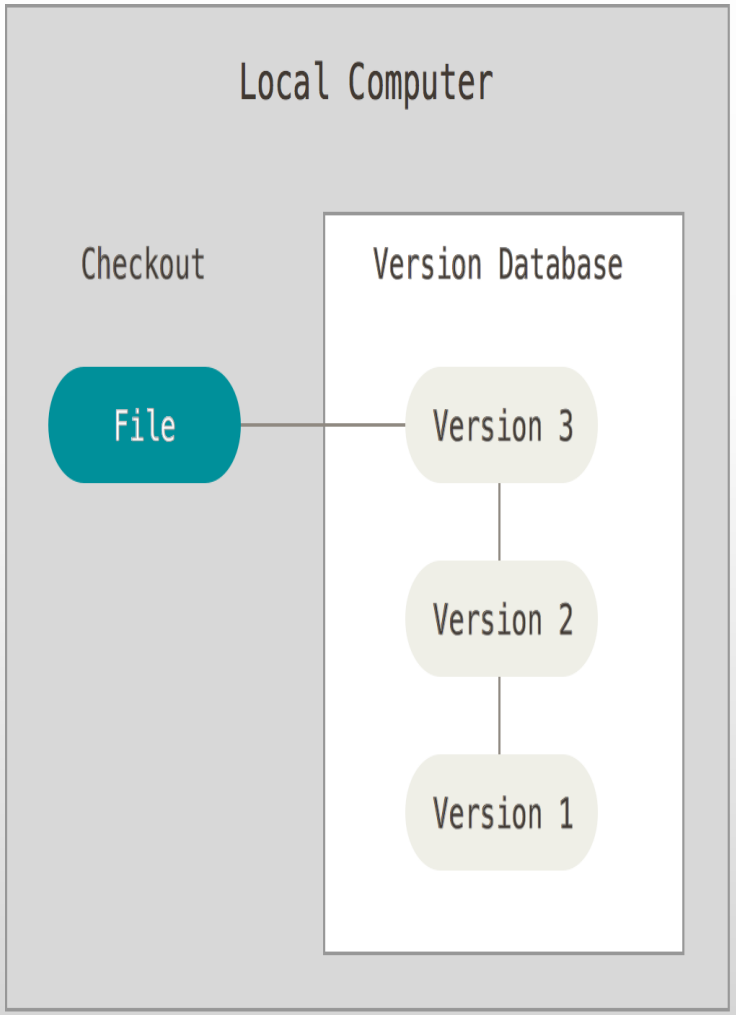
The goals of a Version Control System

- Allow developers to work simultaneously.
- Do not overwrite each other’s changes.
- Maintain history of every version of everything

VCS categories

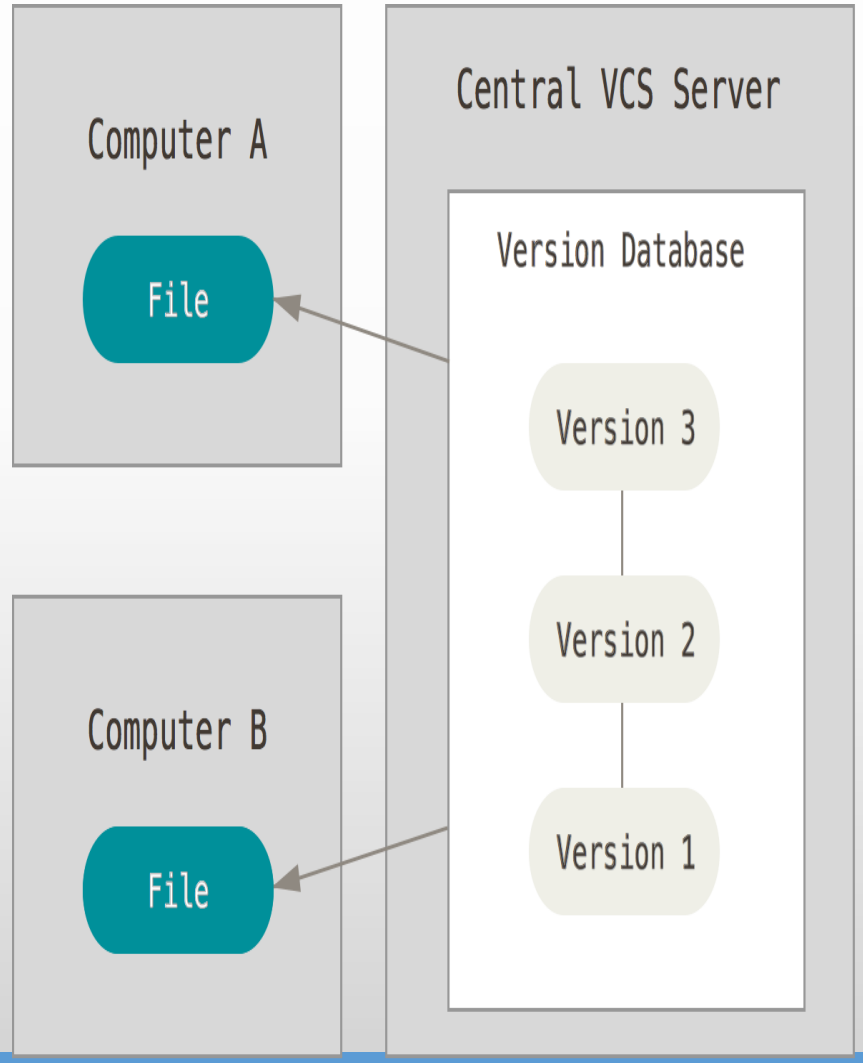
- A VCS is divided into three categories
  - Local Version Control Systems
  - Centralized Version Control System (CVCS)
  - Distributed/Decentralized Version Control System (DVCS)

Local Version Control Systems



- local VCS that had a simple database that kept all the changes to files under revision control.
- RCS is one of the example for local Version control system
- This is used to manage the configuration or any document file with in the single system
- RCS 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.

Centralized Version Control Systems



- CVCS is used when the Documents needs to collaborate with developers on other systems
- Examples are CVS, Subversion, and Perforce.
- Single server that contains all the versioned files, and a number of clients that check out files from that central place.

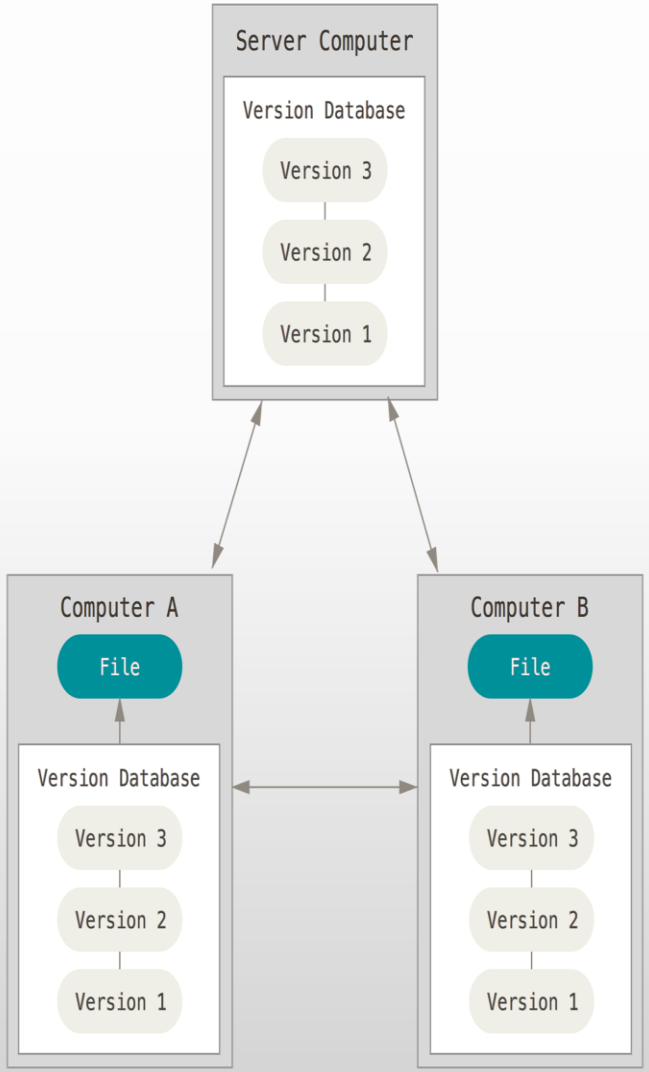
CVCS Advantages

- Everyone knows to a certain degree what everyone else on the project is doing.
- Administrators have fine-grained control over who can do what
- it’s far easier to administer a CVCS than it is to deal with local databases on every client.

Downsides of CVCS

- It is the single point of failure that the centralized server represents.
- If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.
- The hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines.

Distributed Version Control Systems



- In DVCS clients don't just check out the latest snapshot of the files: they fully mirror the repository.
- Example are Git, Mercurial, Bazaar or Darcs
- 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.
- Every clone is really a full backup of all the data.

Advantages of DCVS

- Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.
- Communication is only necessary when sharing changes among other peers.
- Each working copy effectively functions as a remote backup of the codebase and of its change-history, protecting against data loss.
- Allows users to work productively when not connected to a network.
- Makes most operations much faster
- Allows private work, so users can use their changes even for early drafts they do not want to publish.
- Avoids relying on one physical machine as a single point of failure.

Disadvantages of DCVS

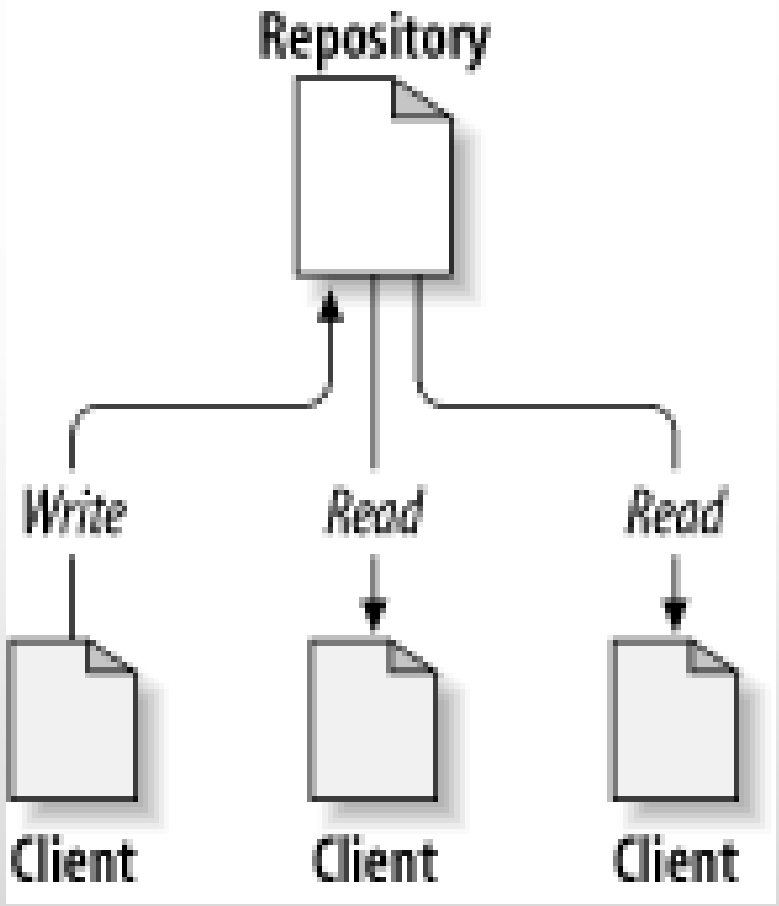
- Initial cloning of a repository is slower as compared to centralized checkout, because all branches and revision history are copied.
- This may be significant if access speed is slow and the repository size is large enough.
- It takes more storage since Data is Stored in multiple places.

GIT History

- Git development began in April 2005 after many developers of the Linux kernel gave up access to BitKeeper
- Linus Torvalds wanted a distributed system that he could use like BitKeeper, but none of the available free systems met his needs, particularly in terms of performance.

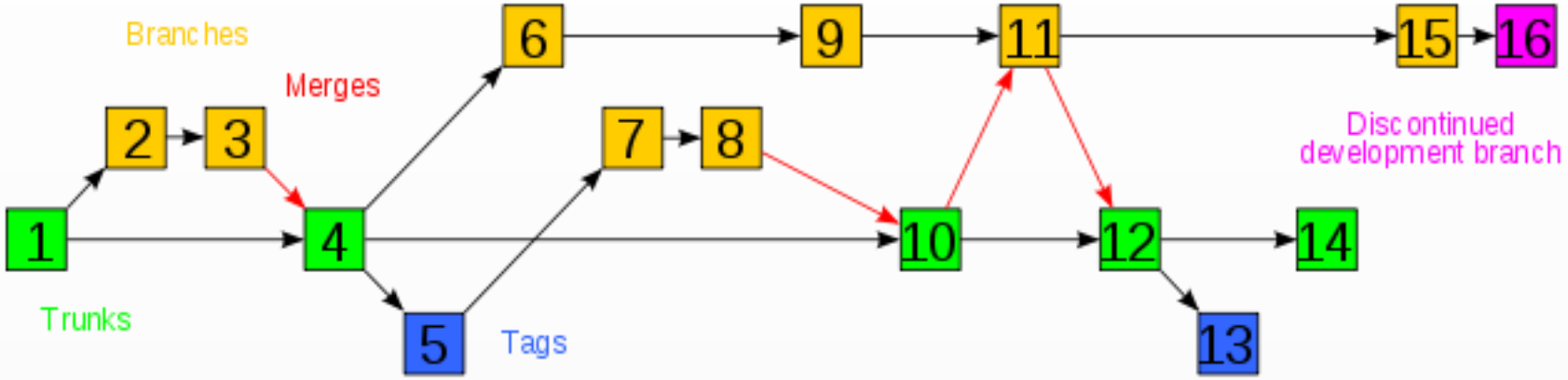
Version Control Terminologies

Repository



- A repository is the heart of any version control system
- It is the central place where developers store all their Document.
- Repository not only stores files but also the history.
- Repository is accessed over a network, acting as a server and version control tool acting as a client.
- Clients can connect to the repository, and then they can store/retrieve their changes to/from repository.
- By storing changes, a client makes these changes available to other people and by retrieving changes, a client takes other people's changes as a working copy.

Trunk



- The trunk is a directory where all the main development happens
- It is usually checked out by developers to work on the project.

Tags

- The tags directory is used to store named snapshots of the project.
- Tag operation allows to give descriptive and memorable names to specific version in the repository.
- For example
- LAST_STABLE_CODE_BEFORE_EMAIL_SUPPORT is more memorable than revision numbers

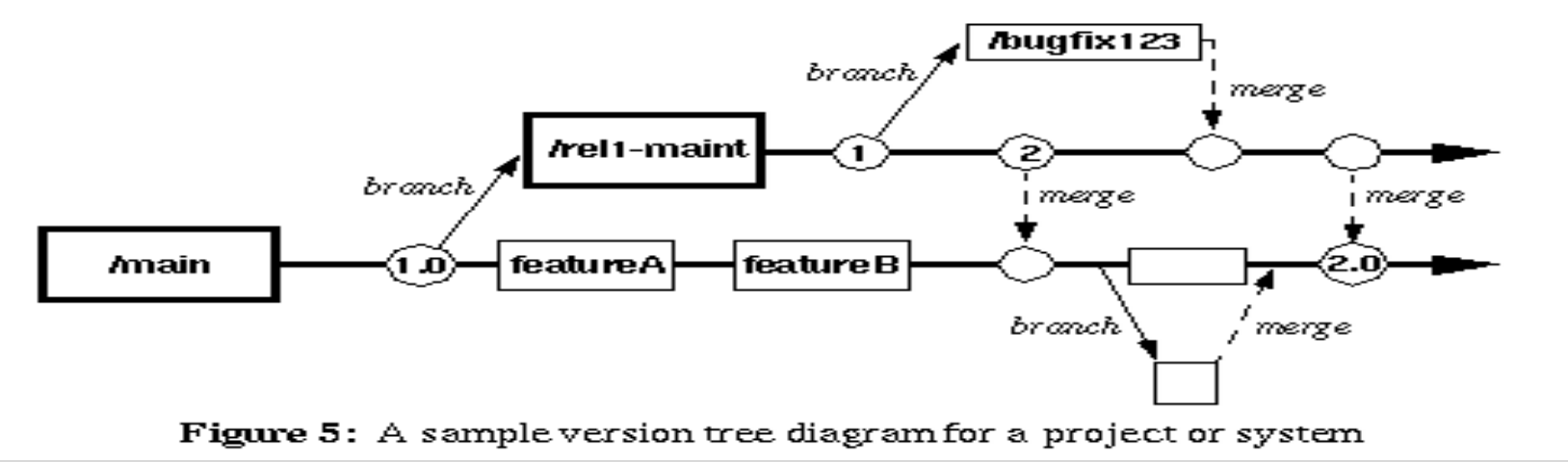
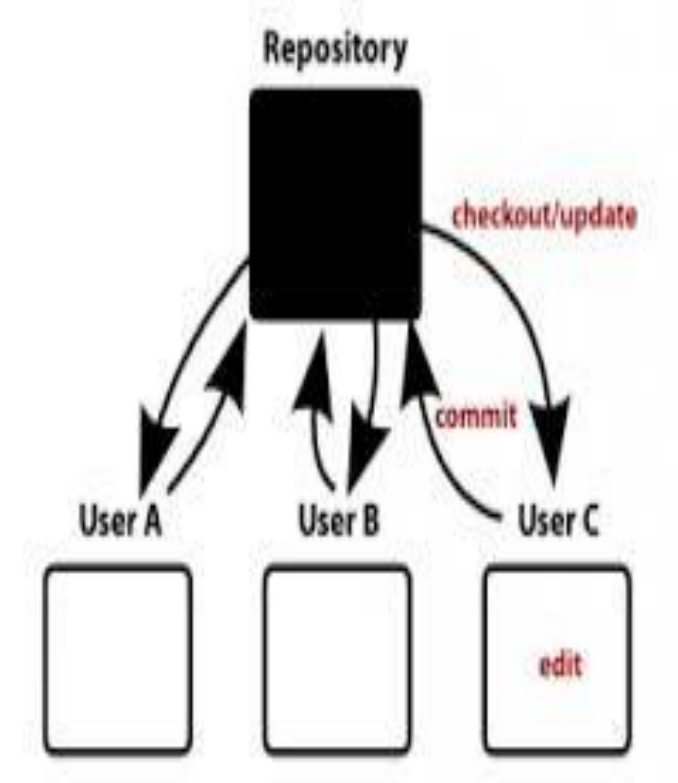


Figure 5: A sample version tree diagram for a project or system

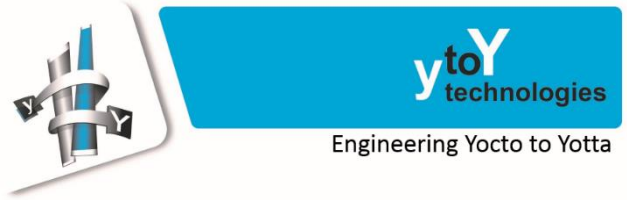
Branches

- Branch operation is used to create another line of development.
- It is useful when you want your development process to fork off into two different directions.
- or example, when you release
- version 5.0, you might want to create a branch so that development of 6.0 features can be kept separate from 5.0 bug-fixes.

Working copy



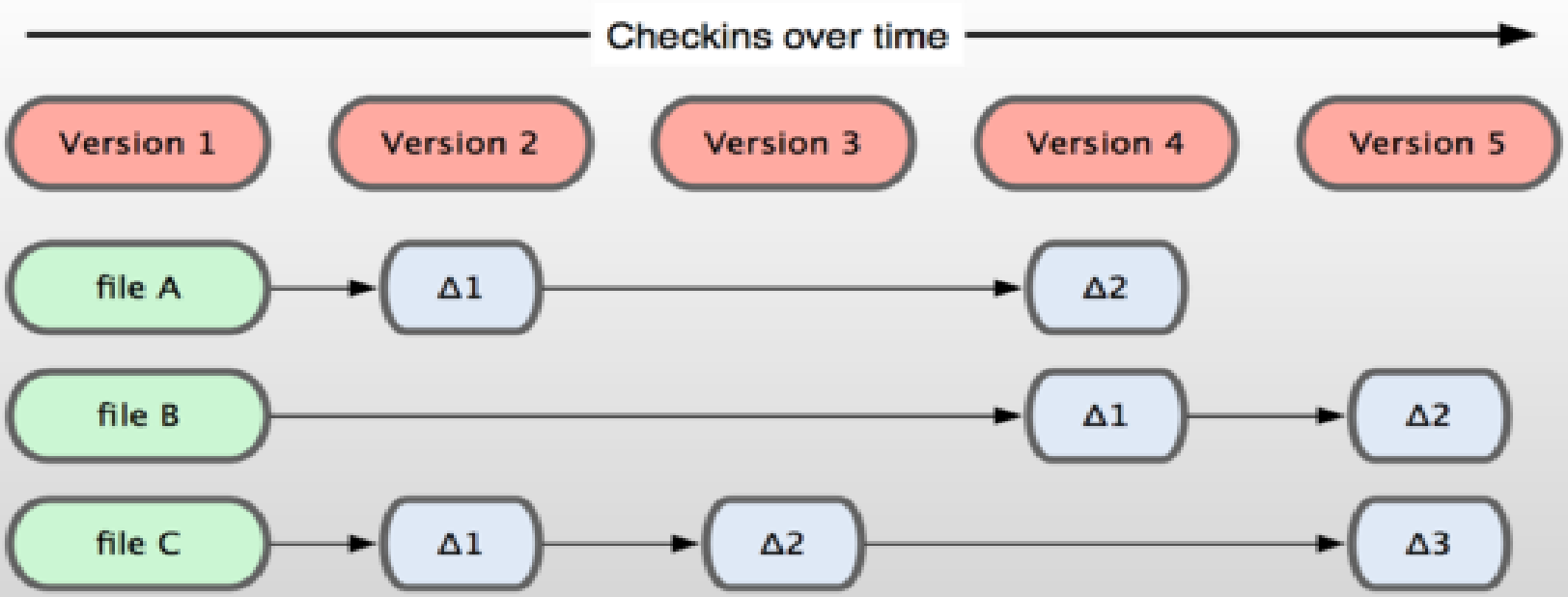
- Working copy is a snapshot of the repository.
- The repository is shared by all the teams, but people do not modify it directly. Instead each developer checks out the working copy.
- The working copy is a private workplace where developers can do their work remaining isolated from the rest of the team.



GIT Basics

Other Version Controller

- Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar .. etc)



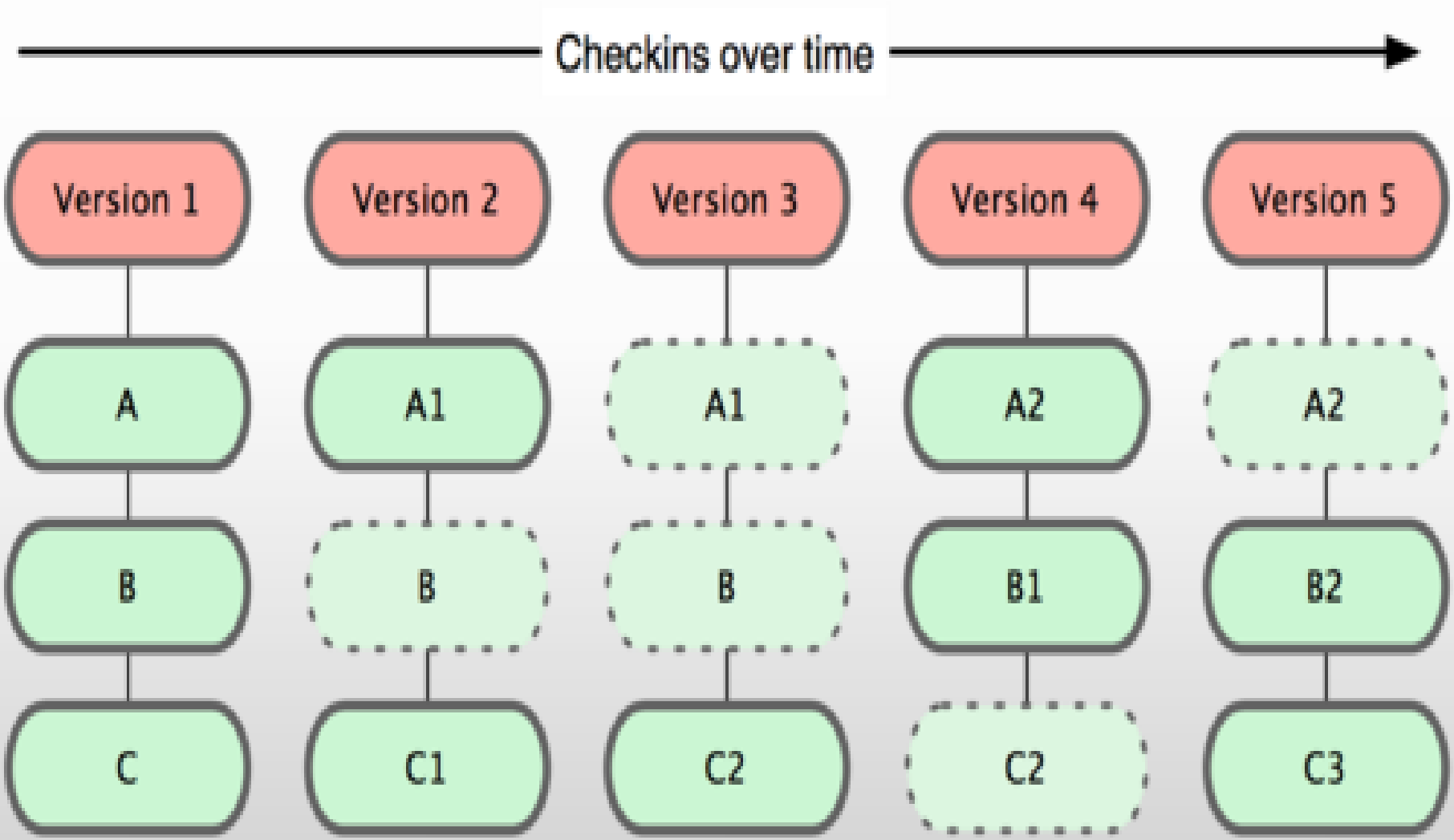
GIT

- **GIT is content-Addressable Storage System**
- GIT Sores the Data in Key Value Format
  - Key: SHA-1 Hash Of Object's Content
  - Value: Compressed Content
- Same Content Never Saved Twice

GIT Basics

- Git thinks of its data more like a set of snapshots of a mini file system
- Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

GIT Basics



Nearly Every Operation Is Local

- Most operations in Git only need local files and resources to operate generally no information is needed from another computer on your network.
- For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you—it simply reads it directly from your local database.
- This enables most of your working with data can happen offline

Git Has Integrity

- Everything in Git is check-summed before it is stored and is then referred to by that checksum.
- It’s impossible to change the contents of any file or directory without Git knowing about it.
- This functionality is built into Git at the lowest levels and is integral to its philosophy.
- You can’t lose information in transit or get file corruption without Git being able to detect it.

GIT Hash

- The mechanism that Git uses for this check summing is called a SHA–1 hash
- This is a 40-character string composed of hexadecimal characters (09 and af) and calculated based on the contents of a file or directory structure in Git.
- A SHA–1 hash looks something like this:
- **24b9da6552252987aa493b52f8696cd6d3b00373**

GIT Hash

- Git stores everything not by file name but in the Git database addressable by the hash value of its contents.

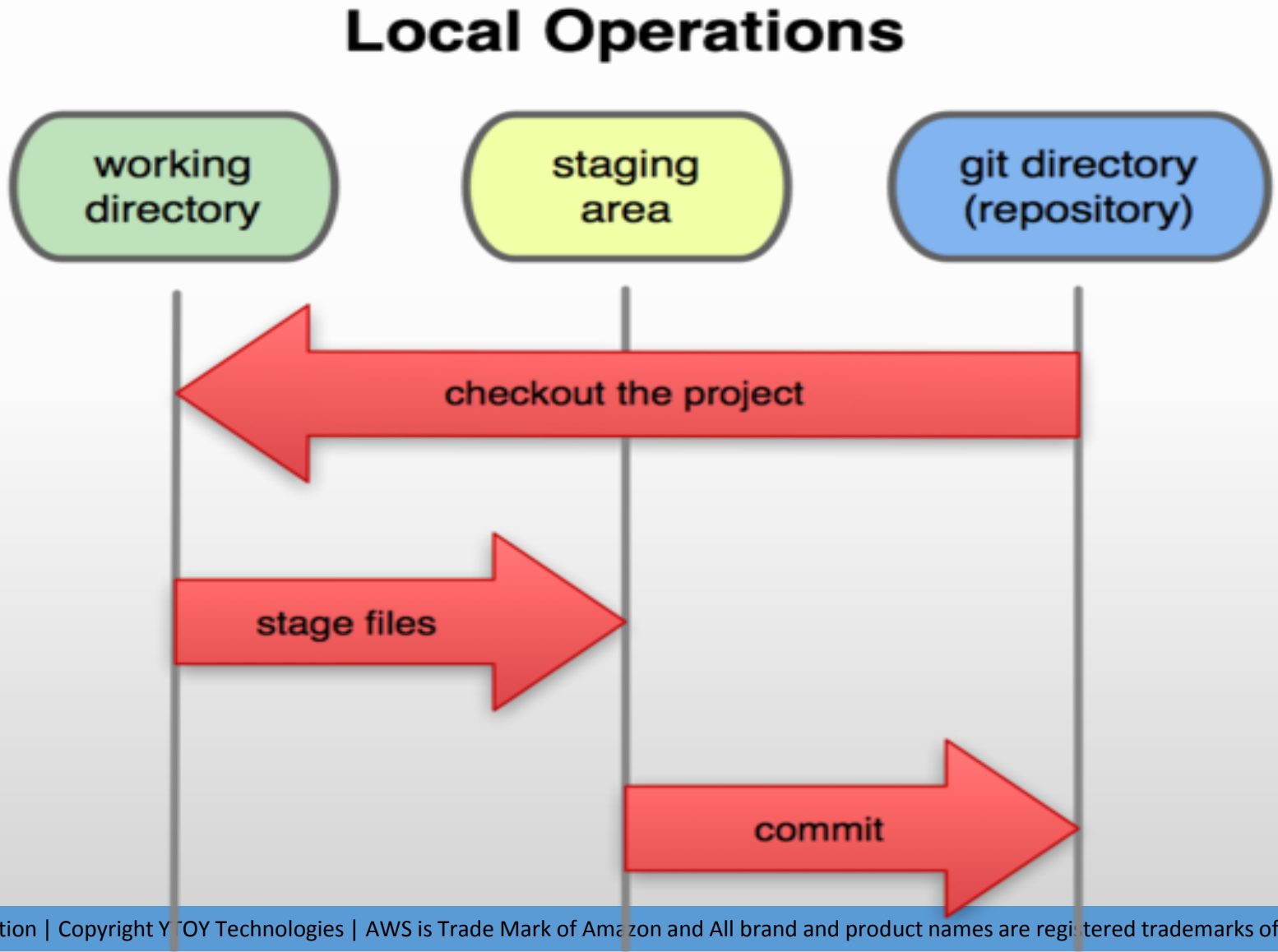
Git Generally Only Adds Data

- All operations in GIT only add data to the Git database.
- It is very difficult to get the system to do anything that is not undoable or to make it erase data in any way.
- After you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

File State in GIT

- Git has three main states that your files can reside in:
  - Committed
  - Modified
  - Staged
- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

sections of a Git project



sections of a Git project

- The three main sections of a Git project:
  - Git directory
  - working directory
  - staging area

GIT Directory

- The Git directory is where Git stores the metadata and object database for your project.
- Git Directories is get copied when you clone a repository from another computer.

Working directory

- 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.

staging area

- staging area is a simple 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

Basic Git workflow

- You modify files in your working directory.
- You stage the files, adding snapshots of them to your staging area.
- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Working With GIT

Creating GIT Repository

- Create Git project using two main approaches:
  - Take an existing project or directory and imports it into Git.
  - clone an existing Git repository from another server.

Initializing a Repository

- Create directory
  - # mkdir /test
- Change Directory
  - #cd /test
  - [root@localhost test]# ls -la
  - total 4
  - drwxr-xr-x. 2 root root   6 Apr 5 08:15 .
  - drwxr-xr-x. 18 root root 4096 Apr 5 08:15 ..

Initializing a Repository

- Intialize the GIT Repository
  - [root@localhost test]# git init
  - Initialized empty Git repository in /test/.git/
  - [root@localhost test]# ls -la
  - total 8
  - drwxr-xr-x. 3 root root   17 Apr 5 08:16 .
  - drwxr-xr-x. 18 root root 4096 Apr 5 08:15 ..
  - drwxr-xr-x. 7 root root 4096 Apr 5 08:16 .git

Cloning an Existing Repository

- If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is git clone.
- Git receives a copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down when you run git clone .
- if your server disk gets corrupted, you can use any of the clones on any client to set the server back to the state it was in when it was cloned

Cloning an Existing Repository

- $ git clone git://github.com/schacon/grit.git
- creates a directory named “grit”, initializes a .git directory inside it, pulls down all the data for that repository.
- checks out a working copy of the latest version.
- If you go into the new grit directory, you’ll see the project files in there, ready to be worked on or used.

Cloning an Existing Repository

- If you want to clone the repository into a directory named something other than repository name
- $ git clone git://github.com/schacon/grit.git mygrit

GIT Access Method

- GIT Protocol
- git://url
- HTTP Ptotocol
- http(s)://url
- SSH Protocol
- user@server:/path.git

Cloning an Existing Repository

- root@y2ysys1:~# git clone root@192.168.56.101:/test
- Cloning into 'test'...
- root@192.168.56.101's password:
- remote: Counting objects: 5, done.
- remote: Compressing objects: 100% (2/2), done.
- remote: Total 5 (delta 0), reused 0 (delta 0)
- Receiving objects: 100% (5/5), done.
- Checking connectivity... done.
- root@y2ysys1:~# ls
- 123 1234563 143 163 firstscript satya test test.tar VirtualBox VMs
- root@y2ysys1:~# cd test/
- root@y2ysys1:~/test# ls -la
- total 24
- drwxr-xr-x 3 root root 4096 Apr 5 09:37 .
- drwx------ 14 root root 4096 Apr 5 09:37 ..
- -rw-r--r-- 1 root root   11 Apr 5 09:37 firstfile
- drwxr-xr-x 8 root root 4096 Apr 5 09:37 .git
- -rw-r--r-- 1 root root   12 Apr 5 09:37 secondfile
- -rw-r--r-- 1 root root   11 Apr 5 09:37 thirdfile

Recording Changes to the Repository

- Git repository and a checkout or working copy of the files for that project.
- Make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.
- each file in your working directory can be in one of two states:
  - Tracked
  - untracked.

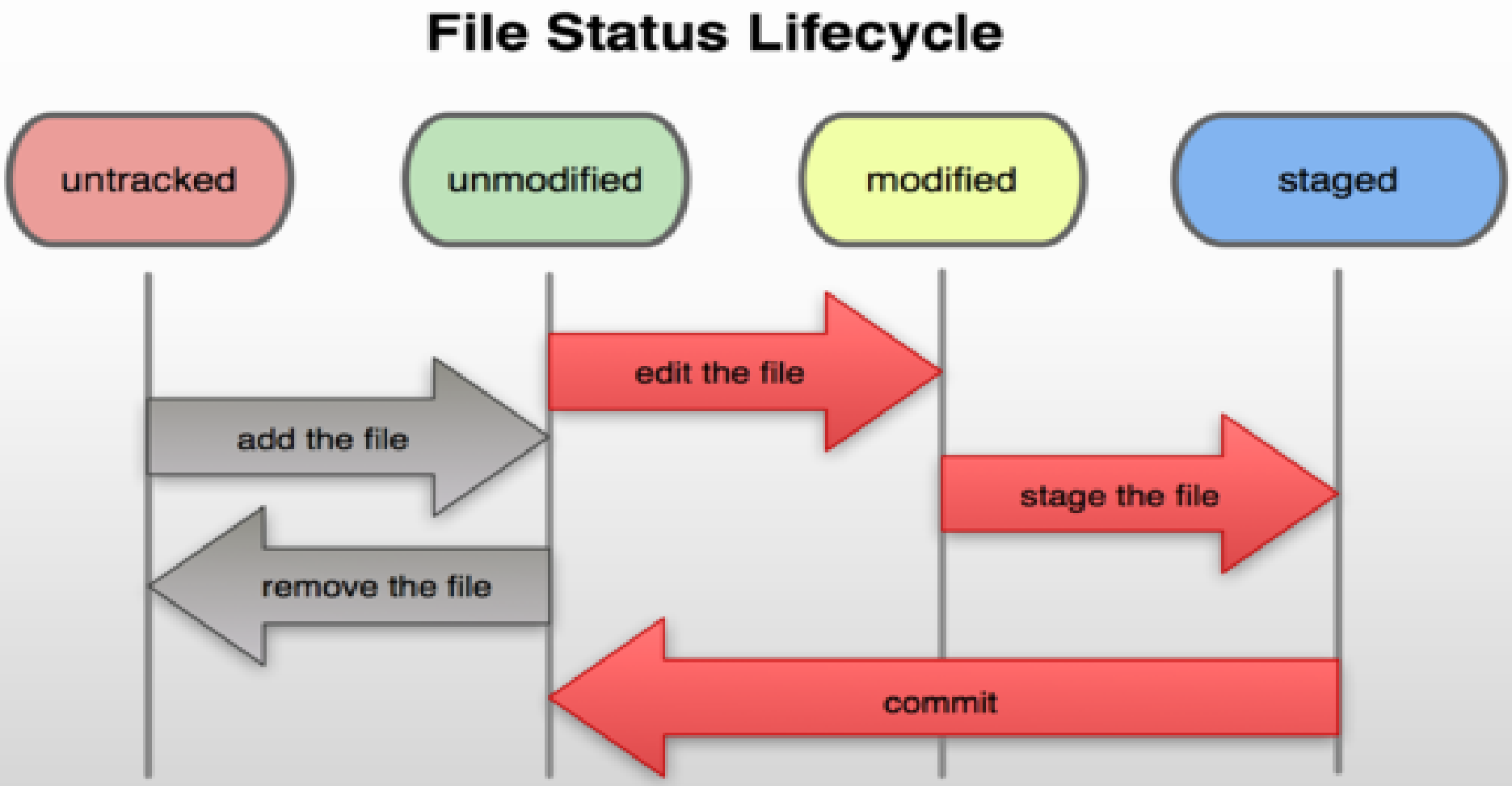
Recording Changes to the Repository

- 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.

Recording Changes to the Repository

- As you edit files, Git sees them as modified, because you've changed them since your last commit.
- Stage these modified files and then commit all your staged changes, and the cycle repeats.

Recording Changes to the Repository



Checking the Status of Your Files

- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- nothing to commit, working directory clean

Checking the Status of Your Files

- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- nothing to commit, working directory clean

Tracking New Files

- root@y2ysys1:~/test# git add fourthfile
- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile

Staging Modified Files

- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile

- 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: secondfile

Staging Modified Files

- root@y2ysys1:~/test# git add secondfile
- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile
- modified: secondfile

Staging Modified Files

- root@y2ysys1:~/test# vim secondfile
- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile
- modified: secondfile

- 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: secondfile

Staging Modified Files

- root@y2ysys1:~/test# vim secondfile
- root@y2ysys1:~/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile
- modified: secondfile

- 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: secondfile

Ignoring Files

- you'll have a class of files that you don't want Git to automatically add or even show you as being untracked.
- These are generally automatically generated files such as log files or files produced by your build system.
- you can create a file listing patterns to match them named .gitignore.

Ignoring Files

- root@y2ysys1:/root/test# echo '*.log' > .gitignore
- root@y2ysys1:/root/test# touch msg.log
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile
- modified: secondfile

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore

Ignoring Files

- root@y2ysys1:/root/test# echo "" > .gitignore
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile
- modified: secondfile

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore
- msg.log

Patterns

- The rules for the patterns you can put in the .gitignore file are as follows:
  - Blank lines or lines starting with # are ignored.
  - Standard glob patterns work.
  - You can end patterns with a forward slash (/) to specify a directory.
  - You can negate a pattern by starting it with an exclamation point (!).

Standard Glob Pattern

- Glob patterns are like simplified regular expressions that shells use.
  - An asterisk (*) matches zero or more characters;
  - [abc] matches any character inside the brackets (in
  - this case a, b, or c);
  - a question mark (?) matches a single character
  - brackets enclosing characters seperated by a hyphen([0-9]) matches any character between them (in this case 0 through 9)

Viewing Your Unstaged Changes

- root@y2ysys1:/root/test# echo "anpther line" >> firstfile
- root@y2ysys1:/root/test# echo "another line" >> secondfile
- root@y2ysys1:/root/test# git diff
- diff --git a/firstfile b/firstfile
- index c9e358c..cd34354 100644
- --- a/firstfile
- +++ b/firstfile
- @@ -1 +1,2 @@
- First File
- +anpther line
- diff --git a/secondfile b/secondfile
- index 537173f..3cbdc3d 100644
- --- a/secondfile
- +++ b/secondfile
- @@ -1,3 +1,4 @@
- Second File
- another line
- third line
- +another line

Viewing Your staged Changes

- root@y2ysys1:/root/test# git diff --staged
- diff --git a/fourthfile b/fourthfile
- new file mode 100644
- index 0000000..ae29abd
- --- /dev/null
- +++ b/fourthfile
- @@ -0,0 +1 @@
- +This is fourth file
- diff --git a/secondfile b/secondfile
- index f686acc..537173f 100644
- --- a/secondfile
- +++ b/secondfile
- @@ -1 +1,3 @@
- Second File
- +another line
- +third line

Committing Your Changes

- commit your changes which are staged
- 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.

Committing Your Changes

- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is up-to-date with 'origin/master'.

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: fourthfile
- modified: secondfile

- 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: firstfile
- modified: secondfile

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore

Committing Your Changes

- root@y2ysys1:/root/test# git commit
- [master b766219] this is first commit
- 2 files changed, 3 insertions(+)
- create mode 100644 fourthfile
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 1 commit.
- (use "git push" to publish your local commits)
- 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: firstfile
- modified: secondfile
- Untracked files:
- (use "git add <file>..." to include in what will be committed)
- .gitignore
- no changes added to commit (use "git add" and/or "git commit -a")

Skipping the Staging Area

- Providing the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 1 commit.
- (use "git push" to publish your local commits)

- 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: firstfile
- modified: secondfile
- Untracked files:
- (use "git add <file>..." to include in what will be committed)
- .gitignore
- no changes added to commit (use "git add" and/or "git commit -a")

Skipping the Staging Area

- root@y2ysys1:/root/test# git commit -a -m "another change"
- [master 9a54799] another change
- 2 files changed, 2 insertions(+)
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 2 commits.
- (use "git push" to publish your local commits)

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore

- nothing added to commit but untracked files present (use "git add" to track)

Removing Files

- 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 next time around.

Removing Files

- root@y2ysys1:/root/test# git rm firstfile
- rm 'firstfile'
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 2 commits.
- (use "git push" to publish your local commits)

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- deleted: firstfile

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore

- root@y2ysys1:/root/test# git commit
- [master 2f2ba97] deleting file
- 1 file changed, 2 deletions(-)
- delete mode 100644 firstfile

Removing Files

- root@y2ysys1:/root/test# git rm --cached thirdfile
- rm 'thirdfile'
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 3 commits.
- (use "git push" to publish your local commits)
- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)
- deleted: thirdfile
- Untracked files:
- (use "git add <file>..." to include in what will be committed)
- .gitignore
- thirdfile
- root@y2ysys1:/root/test# git commit
- [master de50656] removed

Removing Files

- root@y2ysys1:/root/test# ls
- fourthfile msg.log secondfile thirdfile
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 4 commits.
- (use "git push" to publish your local commits)

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore
- thirdfile

- nothing added to commit but untracked files present (use "git add" to track)

Moving Files

- root@y2ysys1:/root/test# git mv fourthfile firstfile
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 4 commits.
- (use "git push" to publish your local commits)

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- renamed: fourthfile -> firstfile

- Untracked files:
- (use "git add <file>..." to include in what will be committed)

- .gitignore
- thirdfile

- root@y2ysys1:/root/test# git commit
- [master 266a80c] moved
- 1 file changed, 0 insertions(+), 0 deletions(-)
- rename fourthfile => firstfile (100%)

Viewing the Commit History

- root@y2ysys1:/root/test# git log
- commit 266a80cfc4dfa5a0289ab459e0313c5de4dc27cd
- Author: mohan <mohan@example.com>
- Date: Sun Apr 5 19:39:35 2015 -0500

- ….

Viewing the Commit History

- root@y2ysys1:/root/test# git log -p -2
- commit 266a80cfc4dfa5a0289ab459e0313c5de4dc27cd
- Author: mohan <mohan@example.com>
- Date: Sun Apr 5 19:39:35 2015 -0500

- moved

Viewing the Commit History

- the --stat option prints 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.
- root@y2ysys1:/root/test# git log --stat -1
- commit 266a80cfc4dfa5a0289ab459e0313c5de4dc27cd
- Author: mohan <mohan@example.com>
- Date: Sun Apr 5 19:39:35 2015 -0500
- moved
- firstfile | 1 +
- fourthfile | 1 -
- 2 files changed, 1 insertion(+), 1 deletion(-)

Viewing the Commit History

- root@y2ysys1:/root/test# git log --pretty=oneline
- 266a80cfc4dfa5a0289ab459e0313c5de4dc27cd moved
- de50656056a87d515fd4c5575fac6257111b8fc2 removed
- 2f2ba972e4ec1f0dabb9ee97c61309a09a5e8d85 deleting file
- 9a5479916dc4b77ce67d71d6b99f2f441fb04aa5 another change
- b766219babf12c8dc8a2561f47c5ab9135e3f431 this is first commit
- 18b11c57017e5c2a2396513920d47ff640648792 Intial Project Version

Viewing the Commit History

- root@y2ysys1:/root/test# git log --pretty=format:"%h - %an, %ar : %s"
- 266a80c - mohan, 24 minutes ago : moved
- de50656 - mohan, 31 minutes ago : removed
- 2f2ba97 - mohan, 39 minutes ago : deleting file
- 9a54799 - mohan, 7 hours ago : another change
- b766219 - mohan, 7 hours ago : this is first commit
- 18b11c5 - root, 13 hours ago : Intial Project Version

Viewing the Commit History

- The placeholders are:
  - %H: commit hash
  - %h: abbreviated commit hash
  - %T: tree hash
  - %t: abbreviated tree hash
  - %P: parent hashes

Viewing the Commit History

- The placeholders are:
- · %p: abbreviated parent hashes
- · %an: author name
- · %aN: author name (respecting .mailmap, see git-shortlog(1) or
- git-blame(1))
- · %ae: author email
- · %aE: author email (respecting .mailmap, see git-shortlog(1) or
- git-blame(1))
- · %ad: author date (format respects --date= option)

· %aD: author date, RFC2822 style

· %ar: author date, relative

· %at: author date, UNIX timestamp

· %ai: author date, ISO 8601 format

· %cn: committer name

· %cN: committer name (respecting .mailmap, see git-shortlog(1)
or git-blame(1))

· %ce: committer email

· %cE: committer email (respecting .mailmap, see git-shortlog(1)
or git-blame(1))

Viewing the Commit History

- %cd: committer date
- %cD: committer date, RFC2822 style
- %cr: committer date, relative
- %ct: committer date, UNIX timestamp
- %ci: committer date, ISO 8601 format
-

Viewing the Commit History

- %d: ref names, like the --decorate option of git-log(1)
- %e: encoding
- %s: subject
- %f: sanitized subject line, suitable for a filename
- %b: body
- %B: raw body (unwrapped subject and body)
- %N: commit notes

Viewing the Commit History

- root@y2ysys1:/root/test# git log --pretty=format:"%h %s" --graph
- * 266a80c moved
- * de50656 removed
- * 2f2ba97 deleting file
- * 9a54799 another change
- * b766219 this is first commit
- * 18b11c5 Intial Project Version

Limiting Log Output

- -<n> , where n is any integer to show the last n commits.
- the time-limiting options such as --since and --until are very useful
- -p Show the patch introduced with each commit.
- --stat Show statistics for files modified in each commit.
- --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.
- --abbrev-commit Show only the first few characters of the SHA-1 checksum instead of all 40.

Limiting Log Output

- --relative-date Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.
- --graph Display an ASCII graph of the branch and merge history beside the log output.
- --pretty Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format).

Limiting Log Output

- --since, --after Limit the commits to those made after the specified date.
- --until, --before Limit the commits to those made before the specified date.
- --author Only show commits in which the author entry matches the specified string.
- --committer Only show commits in which the committer entry matches the specified string.

Undoing Things

- Changing Your Last Commit
  - If you want to try that commit again, you can run commit with the --amend option

- root@y2ysys1:/root/test# git commit -m "added sixth file"
- [master d8005d3] added sixth file
- 1 file changed, 0 insertions(+), 0 deletions(-)
- create mode 100644 seventhfile
- root@y2ysys1:/root/test# git commit --amend -m "added seventh file"
- [master 75c71c7] added seventh file
- 1 file changed, 0 insertions(+), 0 deletions(-)
- create mode 100644 seventhfile
- root@y2ysys1:/root/test# git log -2
- commit 75c71c739fdbcf759572f4d4c2d97b66ffbe49db
- Author: mohan <mohan@example.com>
- Date: Sun Apr 5 20:24:06 2015 -0500

- added seventh file

- commit 266a80cfc4dfa5a0289ab459e0313c5de4dc27cd
- Author: mohan <mohan@example.com>
- Date: Sun Apr 5 19:39:35 2015 -0500

Unstaging a Staged File

- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 6 commits.
- (use "git push" to publish your local commits)

- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)

- new file: eighthfile
- modified: secondfile
- root@y2ysys1:/root/test# git reset HEAD secondfile
- Unstaged changes after reset:
- M secondfile
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 6 commits.
- (use "git push" to publish your local commits)

Unmodifying a Modified File

- **This is a dangerous command: any changes you made to that file are gone**

- root@y2ysys1:/root/test# git status
- 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: secondfile
- root@y2ysys1:/root/test# git status
- On branch master
- Your branch is ahead of 'origin/master' by 6 commits.
- (use "git push" to publish your local commits)
- 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: secondfile

Working with Remotes

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
- 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.

Showing Your Remotes

- git remote command To see which remote servers you have configured
- -v , which shows you the URL that Git has stored for the shortname to be expanded to

- root@y2ysys1:/root/test# git remote
- origin
- root@y2ysys1:/root/test# git remote -v
- origin root@192.168.56.101:/test (fetch)
- origin root@192.168.56.101:/test (push)

Adding Remote Repositories

- To add a new remote Git repository as a shortname you can reference easily, run
- git remote add [shortname] [url]
- git remote add pb git://github.com/paulboone/ticgit.git
- $ git remote -v
- origin git://github.com/schacon/ticgit.git
- pb git://github.com/paulboone/ticgit.git

Adding Remote Repositories

- $ git fetch pb
- remote: Counting objects: 58, done.
- remote: Compressing objects: 100% (41/41), done.
- remote: Total 44 (delta 24), reused 1 (delta 0)
- Unpacking objects: 100% (44/44), done.
- From git://github.com/paulboone/ticgit
- * [new branch]
- master
- -> pb/master
- * [new branch]
- ticgit
- -> pb/ticgit

Fetching and Pulling from Your Remotes

- To get data from your remote projects, you can run
- $ git fetch [remote-name]
- The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet.
- After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

Fetching and Pulling from Your Remotes

- If you cloned a repository, the command automatically adds that remote repository under the name origin.
- So, git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it.
- It’s important to note that the fetch command pulls the data to your local repository
- it doesn’t automati-cally 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.

Fetching and Pulling from Your Remotes

- 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.
- git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

Pushing to Your Remotes

- When you have your project at a point that you want to share, you have to push it upstream.
- git push [remote-name] [branch-name]
- This command 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.

Inspecting a Remote

- root@y2ysys1:/root/test# git remote show origin
- root@192.168.56.101's password:
- * remote origin
- Fetch URL: root@192.168.56.101:/test
- Push URL: root@192.168.56.101:/test
- HEAD branch: master
- Remote branch:
- master tracked
- Local branch configured for 'git pull':
- master merges with remote master
- Local ref configured for 'git push':
- master pushes to master (fast-forwardable)

Renaming Remotes

- **$ git remote rename pb paul**
- **$ git remote**
- **origin**
- **paul**

Removing Remotes

- If you want to remove a reference for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore
- $ git remote rm paul
- $ git remote
- origin

Tagging

- Git has the ability to tag specific points in history as being important.
- Generally use this functionality to mark release points (v1.0, and so on)

Creating Tags

- Git uses two main types of tags
  - Lightweight
  - Annotated
- A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit.
- 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 (GPG).
- It's generally recommended that you create annotated tags so you can have all this information

List Tags

- Listing the available tags in Git
- $ git tag
- v0.1
- v1.3

Show Details of a Tag

- root@y2ysys1:/root/test# git show v1.0
- tag v1.0
- Tagger: mohan <mohan@example.com>
- Date: Sun Apr 5 22:38:43 2015 -0500
- My First Version
- commit 75c71c739fdbcf759572f4d4c2d97b66ffbe49db
- Author: mohan <mohan@example.com>
- Date: Sun Apr 5 20:24:06 2015 -0500
- added seventh file
- diff --git a/seventhfile b/seventhfile
- new file mode 100644
- index 0000000..e69de29

Lightweight Tags

- To create a lightweight tag, don’t supply the -a , -s , or -m option:
- $ git tag v1.4-lw
- $ git tag
- v0.1
- v1.3
- v1.4
- v1.4-lw
- v1.5

Signed Tag

Install and configure GPG

- #[root@localhost ~]# gpg --gen-key
- choose RSA
- key size 1024
- Real Name: <your name>
- Email: your email id
- wait for to finish
- [root@localhost ~]# gpg --list-secret-keys
- /root/.gnupg/secring.gpg
- ----------
- sec 1024R/**8A307674** 2015-04-06
- uid mohan (test) <mohan@example.com>
- ssb 1024R/55717A71 2015-04-06

Signed Tag

- #git config --global user.signingkey **8A307674**
- **# git tag -s v1.1.1 -m “signed tag”**
- **# git show v1.1.1**

- **To verify a signed tag, you use git tag -v [tag-name] .**
- **This command uses GPG to verify the signature. You need the signer’s public key in your keyring for this to work properly:**

Tagging Later

- You can also tag commits after you've moved past them. Suppose your commit history looks like this:
- $ git log –pretty=oneline
- $ git tag -a v1.2 <first 6+ digit of commint key>

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.
- Use git push origin [tagname]

Sharing Tags

- If you have a lot of tags that you want to push up at once, you can also use the
- --tags option to the git push command. This will transfer all of your tags to the
- remote server that are not already there.
- $ git push origin --tags
- Counting objects: 50, done.
- Compressing objects: 100% (38/38), done.
- Writing objects: 100% (44/44), 4.56 KiB, done.
- Total 44 (delta 18), reused 8 (delta 1)
- To git@github.com:schacon/simplegit.git
- * [new tag]
- v0.1 -> v0.1
- * [new tag]
- v1.2 -> v1.2
- * [new tag]
- v1.4 -> v1.4
- * [new tag]
- v1.4-lw -> v1.4-lw
- * [new tag]
- v1.5 -> v1.5

Git Aliases

- set up an alias for each command using git config .
- git config --global alias.unstage ’reset HEAD --’
- git config --global alias.last ’log -1 HEAD’

Git Branching

Git Branching

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.

Git Branching

- Branching model in Git is reffered as its “killer feature,” and it certainly sets Git apart in the VCS community.
- The way Git branches is incredibly lightweight, making branching operations nearly instantaneous and switching back and forth between branches generally just as fast.
- Understanding and mastering this feature gives you a powerful and unique tool and can literally change the way that you develop.

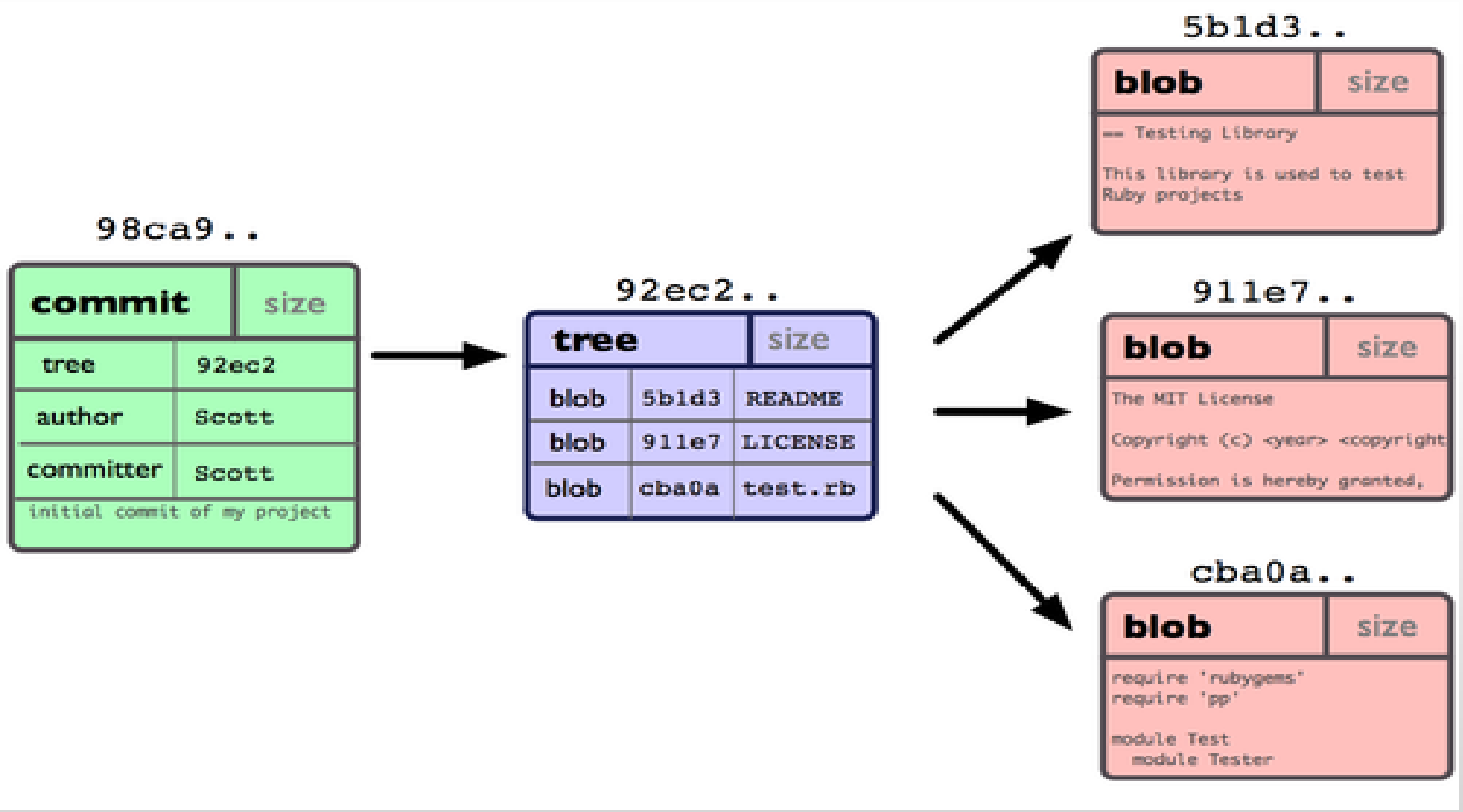
Git Branching

- Git doesn’t store data as a series of change sets or deltas, but instead as a series of snapshots.
- When you commit in Git, Git stores a commit object that contains a pointer to the snapshot of the content you staged
- When you commit in Git, Git stores a commit object that contains a pointer to the snapshot of the content you staged, the author and message metadata, and zero or more pointers to the commit or commits that were the direct parents of this commit:

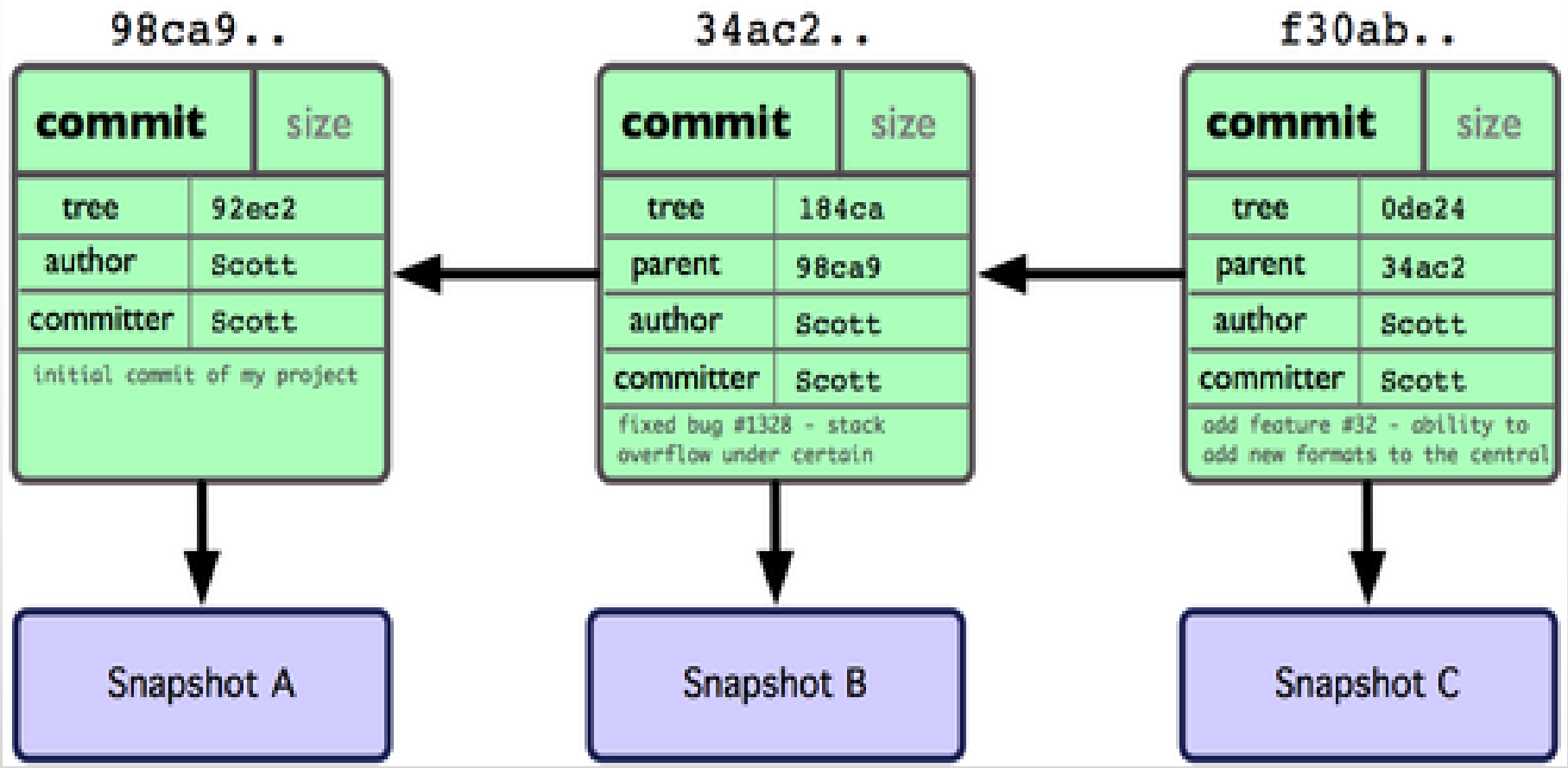
Git Branching

- Zero parents for the first commit,
- one parent for a normal commit,
- and multiple parents for a commit that results from a merge of two or more branches.

Commit Object



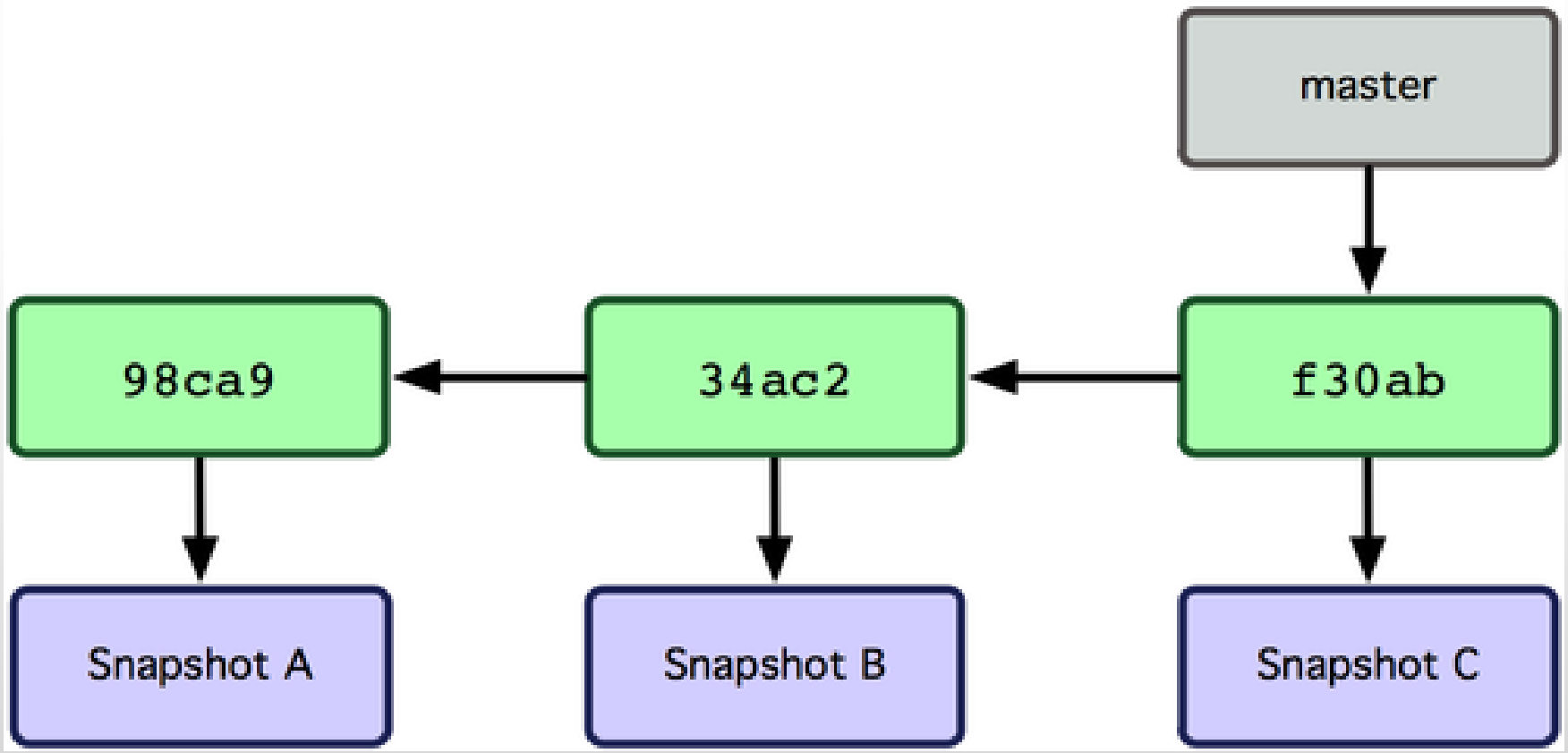
Commit Object



Branch

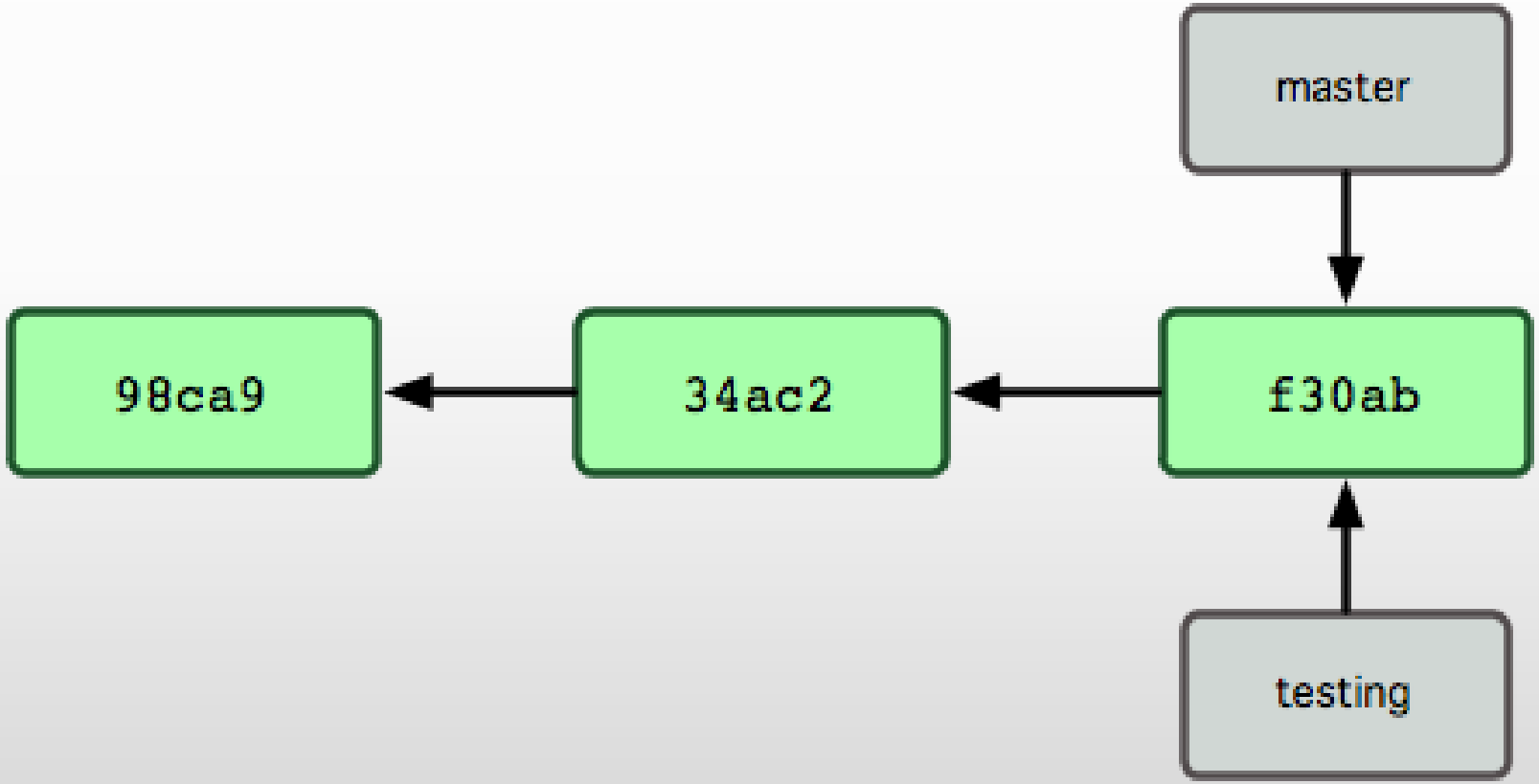
- A branch in Git is simply a lightweight movable pointer to one of these commits
- The default branch name in Git is master.
- As you initially make commits, you're given a master branch that points to the last commit you made.
- Every time you commit, it moves forward automatically.

Branch



Create New Branch

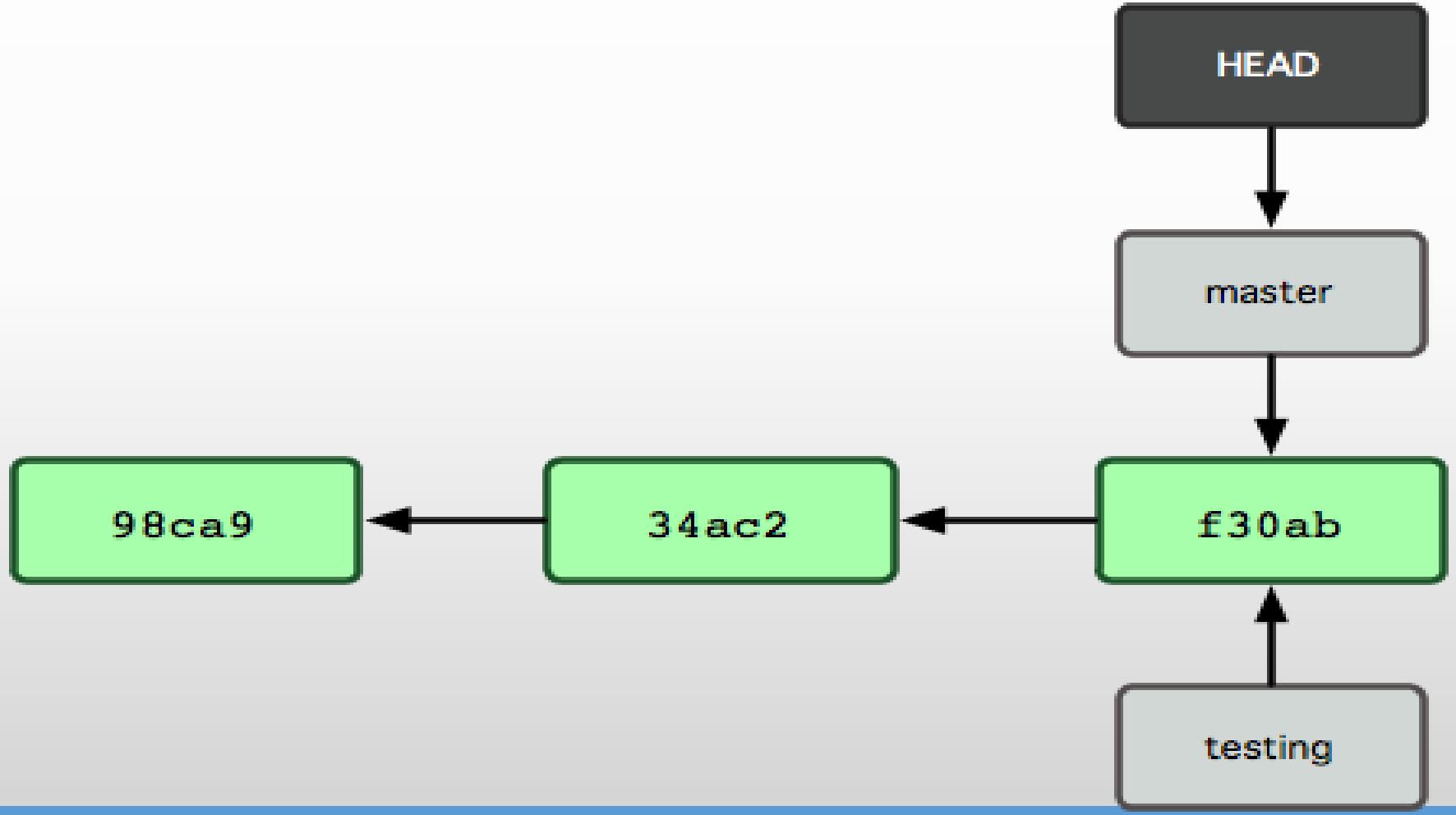
- $ git branch testing



HEAD Branch

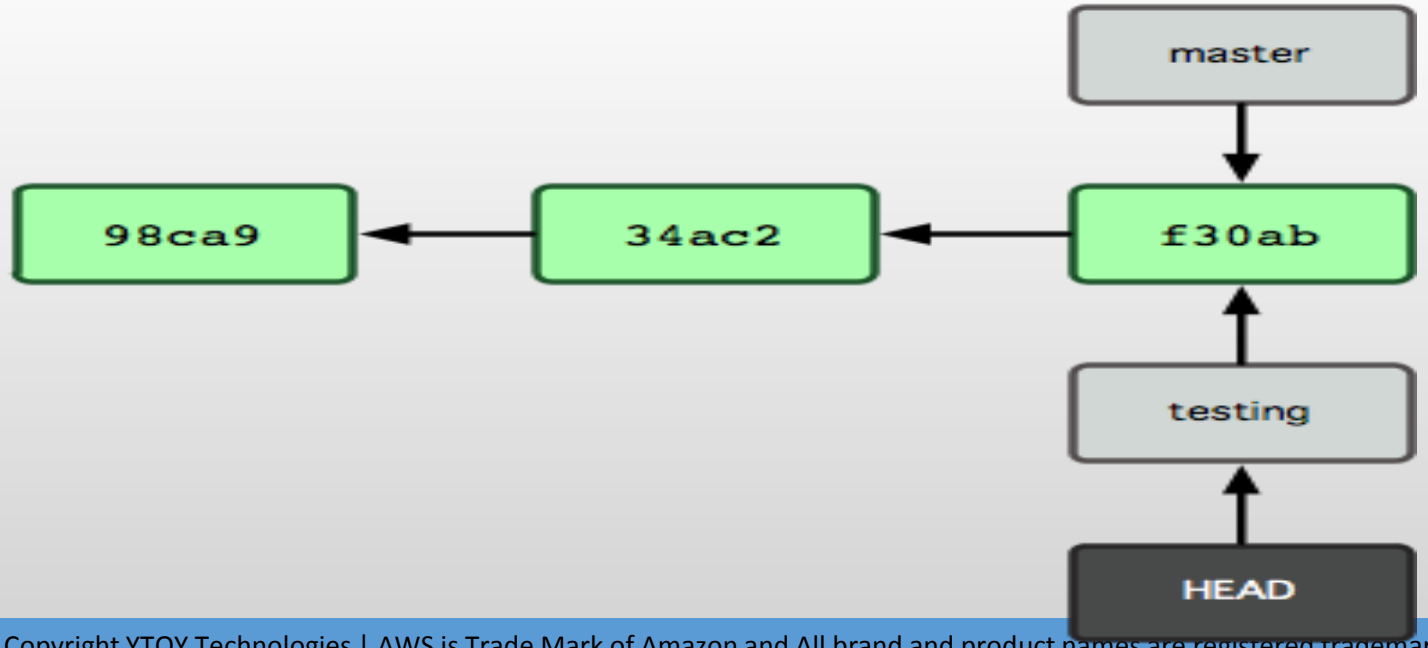
- GIT Identifies what branch you're currently on keeps a special pointer called HEAD.
- this is a pointer to the local branch you're currently on.
- n this case, you're still on master. The git branch command only created a new branch — it didn't switch to that branch

HEAD Branch



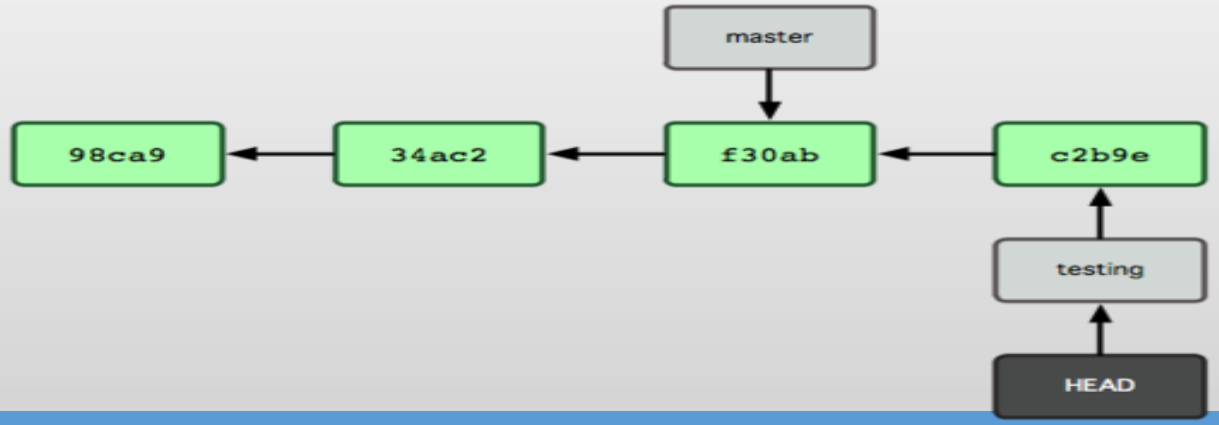
Switch Branches

- To switch to an existing branch, you run the git checkout command.
- $ git checkout testing
- This moves HEAD to point to the testing branch



Switch Branches

- Lets do commit on Test Branch
- $ vim test.rb
- $ git commit -a -m ’made a change’
- The branch that HEAD points to moves forward with each commit.

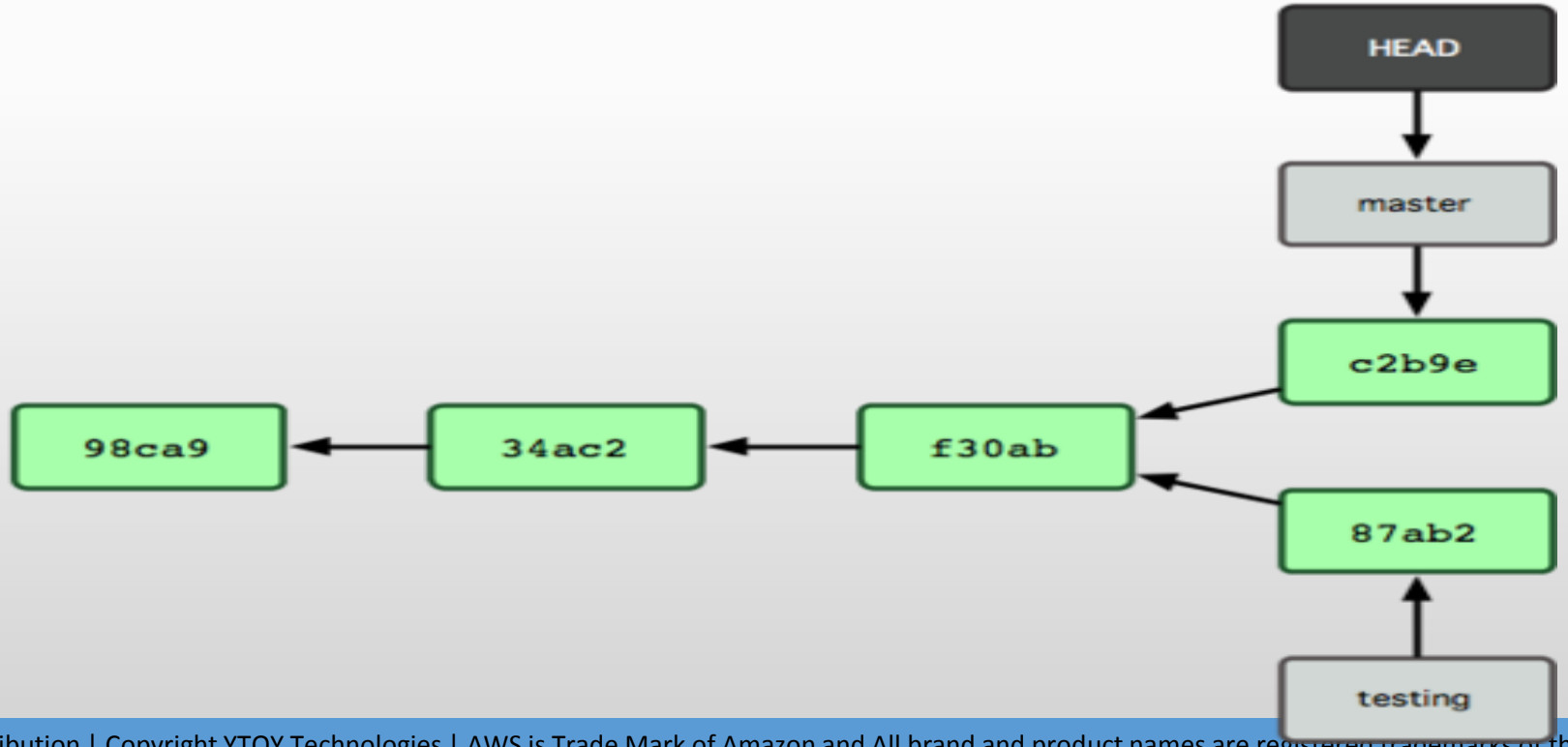


Switch Branches

- Let's switch back to the master branch
  - $ git checkout master
- That command did two things.
  - It moved the HEAD pointer back to point to the master branch
  - it reverted the files in your working directory back to the snapshot that master points to.
- This also means the changes you make from this point forward will diverge from an older version of the project.

Switch Branches

- Let's make a few changes and commit again:
  - $ vim test.rb
  - $ git commit -a -m 'made other changes'



Basic Branching and Merging

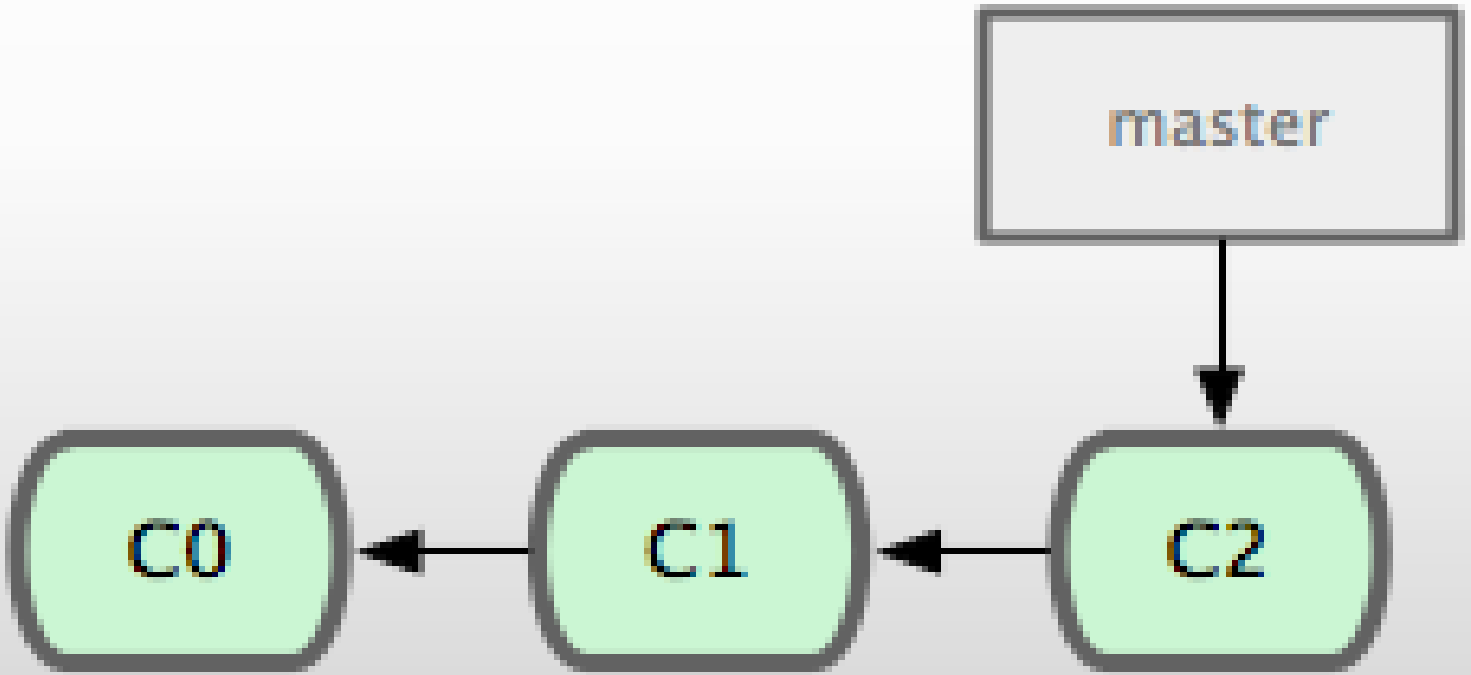
- Let’s go through a simple example of branching and merging with a workflow
- You’ll follow these steps:
- 1. Do work on a web site.
- 2. Create a branch for a new story you’re working on.
- 3. Do some work in that branch.

Basic Branching and Merging

- At this stage, you'll receive a call that another issue is critical and you need a hotfix.
- You'll do the following:
- 1. Revert back to your production branch.
- 2. Create a branch to add the hotfix.
- 3. After it's tested, merge the hotfix branch, and push to production.
- 4. Switch back to your original story and continue working.

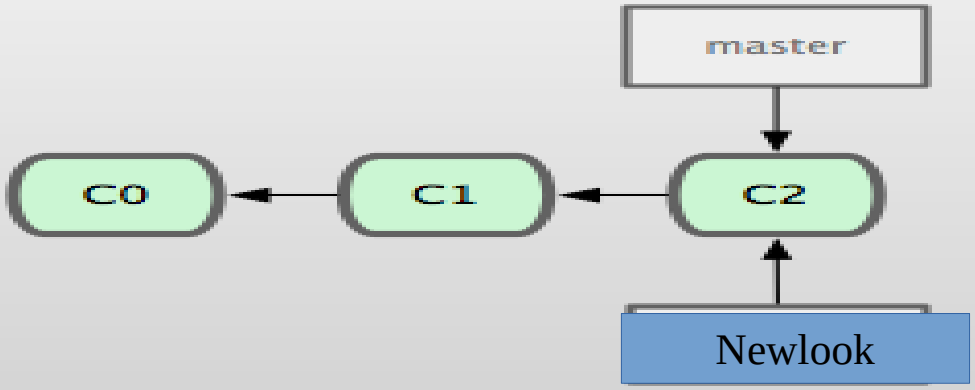
Basic Branching

- Create a repository
- Do a couple of commits



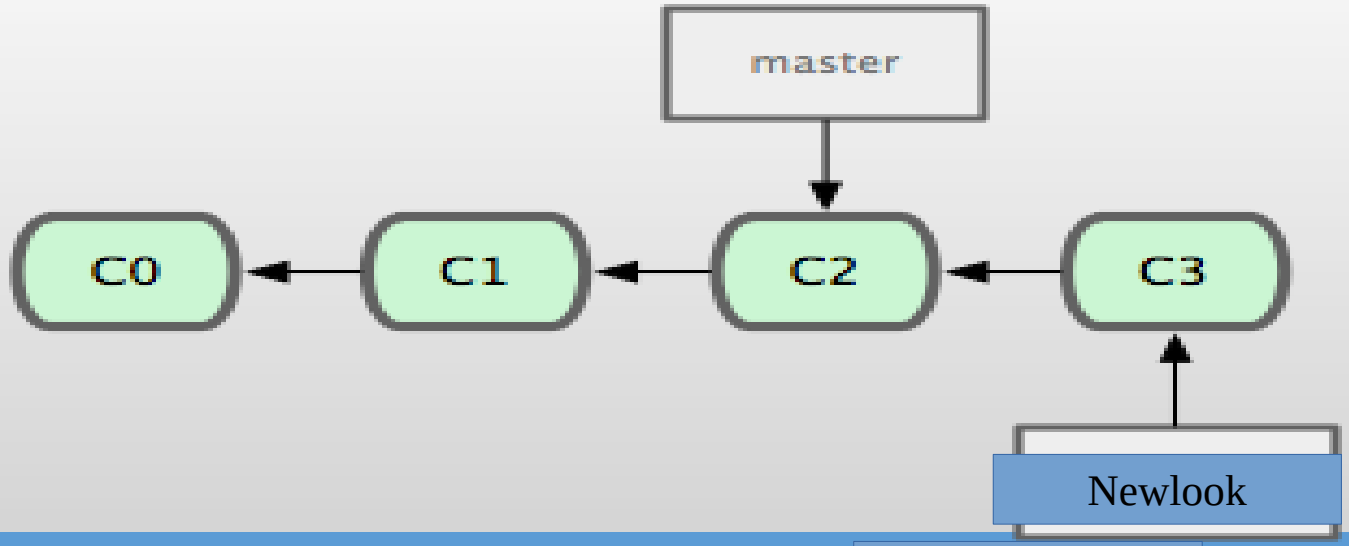
Basic Branching

- Now I wanted to revamp my website so I create a branch newlook
- git checkout -b newlook
- Which is equivalent to :
  - Git branch newlook
  - Git checkout newlook



Basic Branching

- Do some commits on Branch newlook
- $ vim index.html
- $ git commit -a -m ’added a new footer [newlook]’



Basic Branching

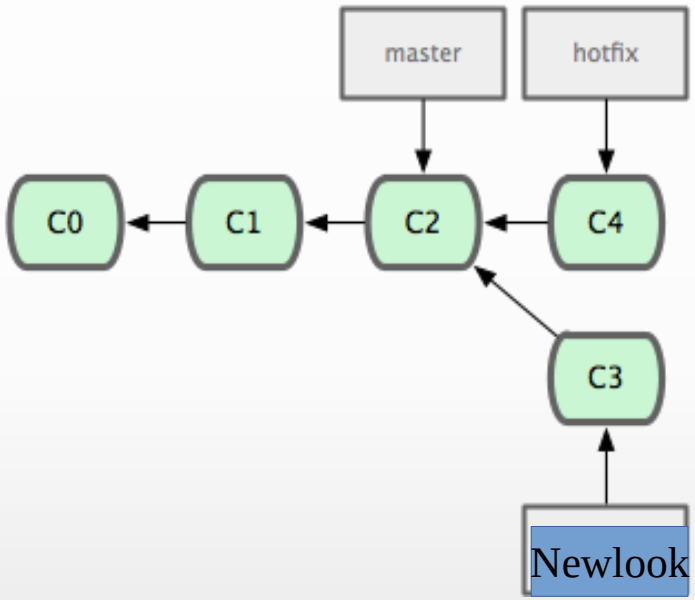
- Now you have to create a hotfix for the production issue
  - switch back to your master branch.
  - Create a Hotfix Branch
- Before you do that
  - if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches.
  - It's best to have a clean working state when you switch branches.
  - There are ways to get around this (namely, stashing and commit amending) that we'll cover later.
  - For now, you've committed all your changes, so you can switch back to your master branch
- $ git checkout master

Basic Branching

- At this point, your project working directory is exactly the way it was before youvstarted working on newlook
- Git resets your working directory to look like the snapshot of the commit that the branch you check out points to. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Basic Branching

- Let’s create a hotfix1 branch on which to work until it’s completed
- $ git checkout -b ’hotfix’
- Switched to a new branch "hotfix"
- $ vim index.html
- $ git commit -a -m ’fixed the broken email address’
- [hotfix]: created 3a0874c: "fixed the broken email address"
- 1 files changed, 0 insertions(+), 1 deletions(-)



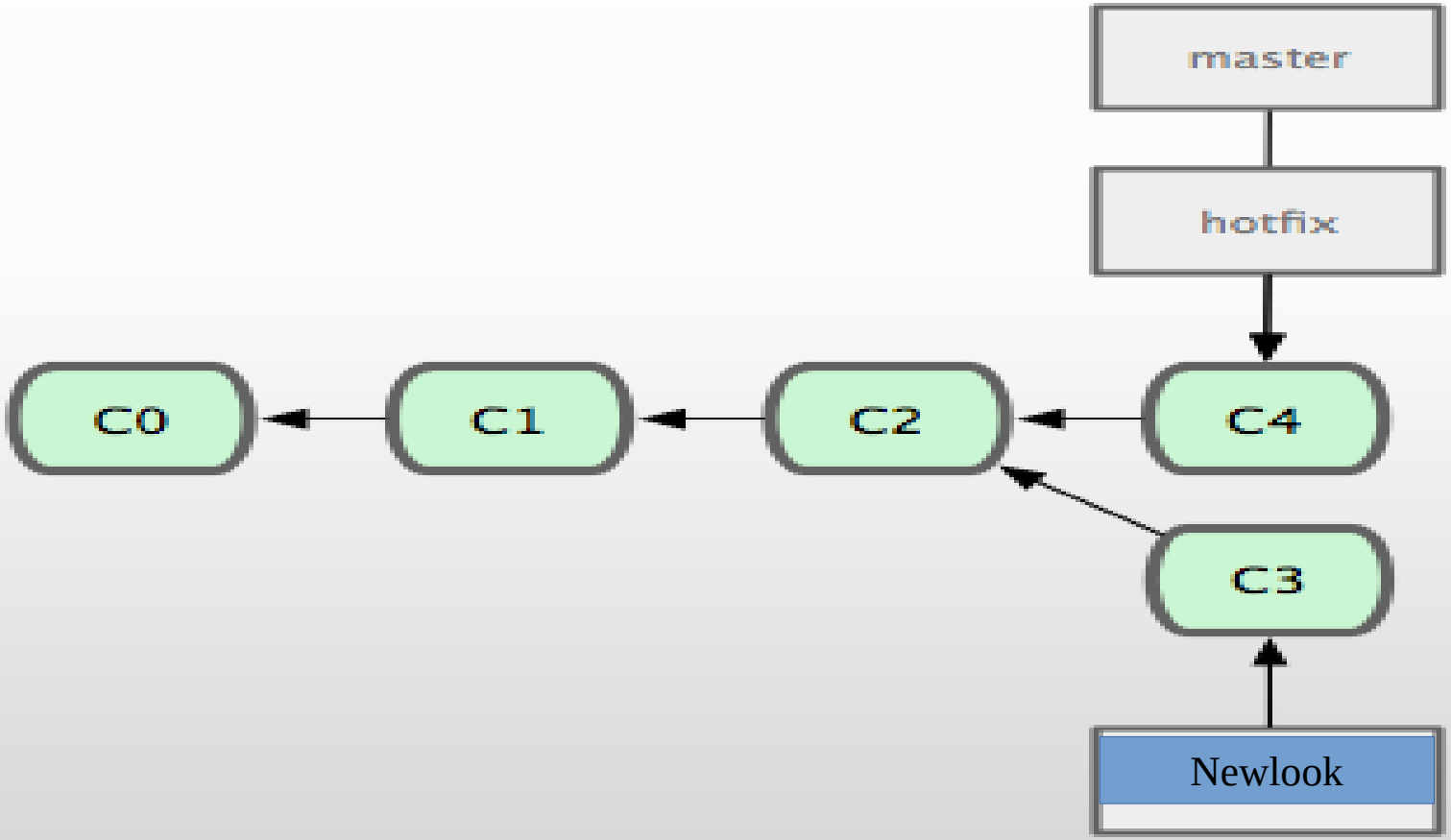
Basic Merging

- You can run your tests, make sure the hotfix is what you want, and merge it back into your master branch to deploy to production. You do this with the git merge command
- $ git checkout master
- $ git merge hotfix
- Updating f42c576..3a0874c
- Fast forward
- README |
- 1 -
- 1 files changed, 0 insertions(+), 1 deletions(-)

Fast forward

- commit pointed to by the branch you merged in was directly upstream of the commit you're on, Git moves the pointer forward.
- when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast forward".

Basic Merging



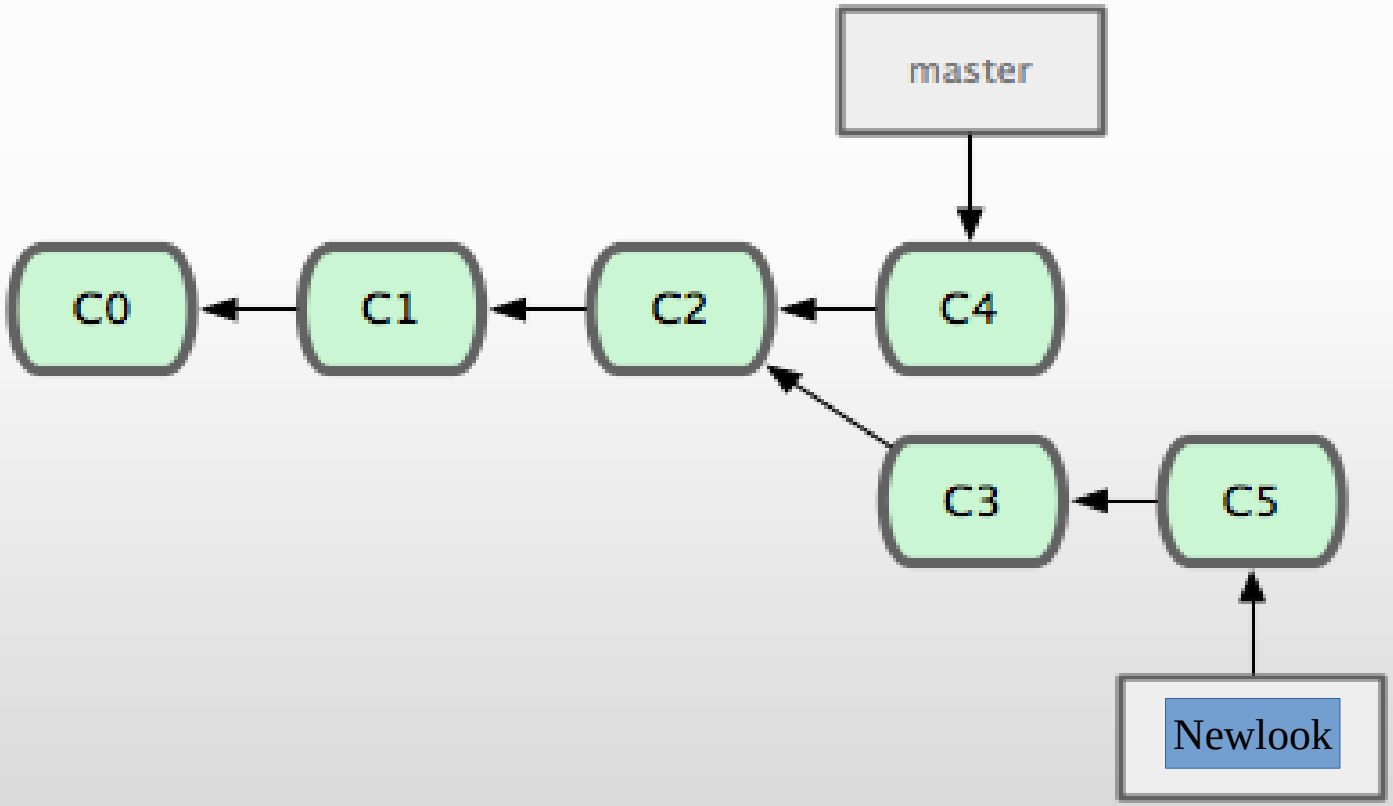
Delete a Branch

- $ git branch -d hotfix
- Deleted branch hotfix (3a0874c)

Basic Merging

- Now we will move to new look project and start working on it
- $ git checkout newlook
- Switched to branch "newlook"
- $ vim index.html
- $ git commit -a -m ’finished the new footer [newlook]’
- [newlook]: created ad82d7a: "finished the new footer [issue 53]"
- 1 files changed, 1 insertions(+), 0 deletions(-)

Basic Merging



Basic Merging

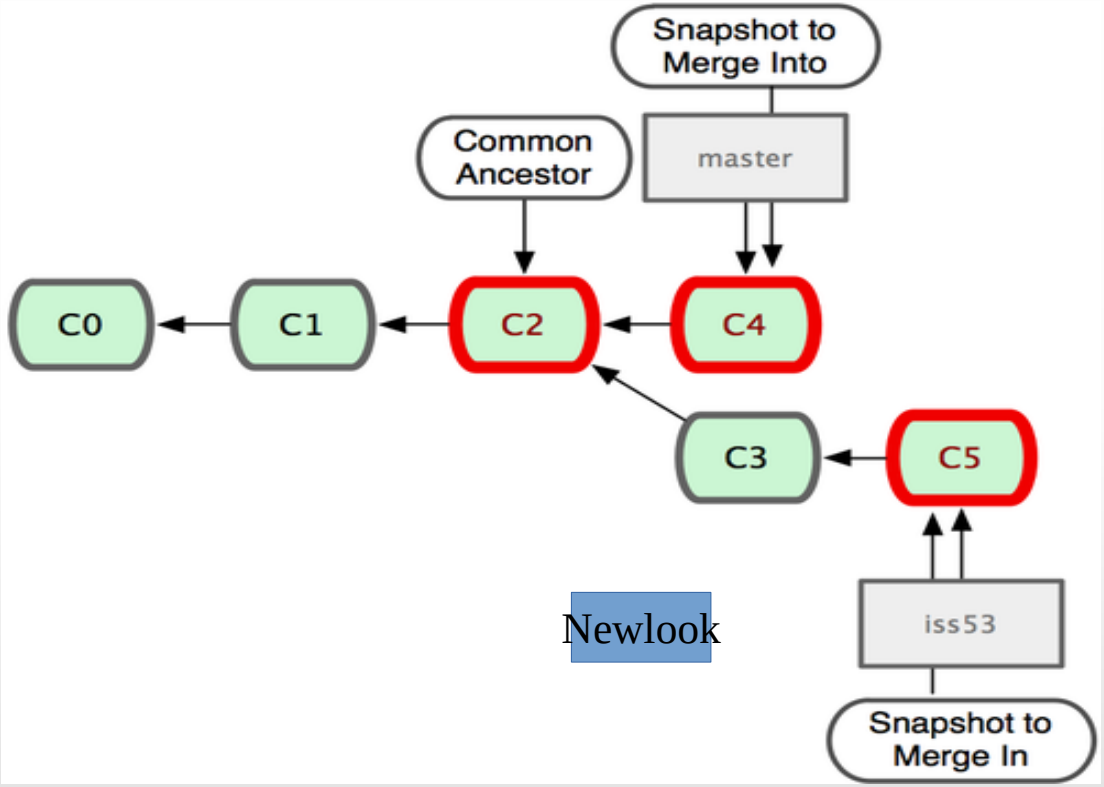
- Work is complete and ready to be merged into your master branch.
- $ git checkout master
- $ git merge newlook
- Merge made by recursive.
- README |
- 1 +
- 1 files changed, 1 insertions(+), 0 deletions(-)

three-way merge

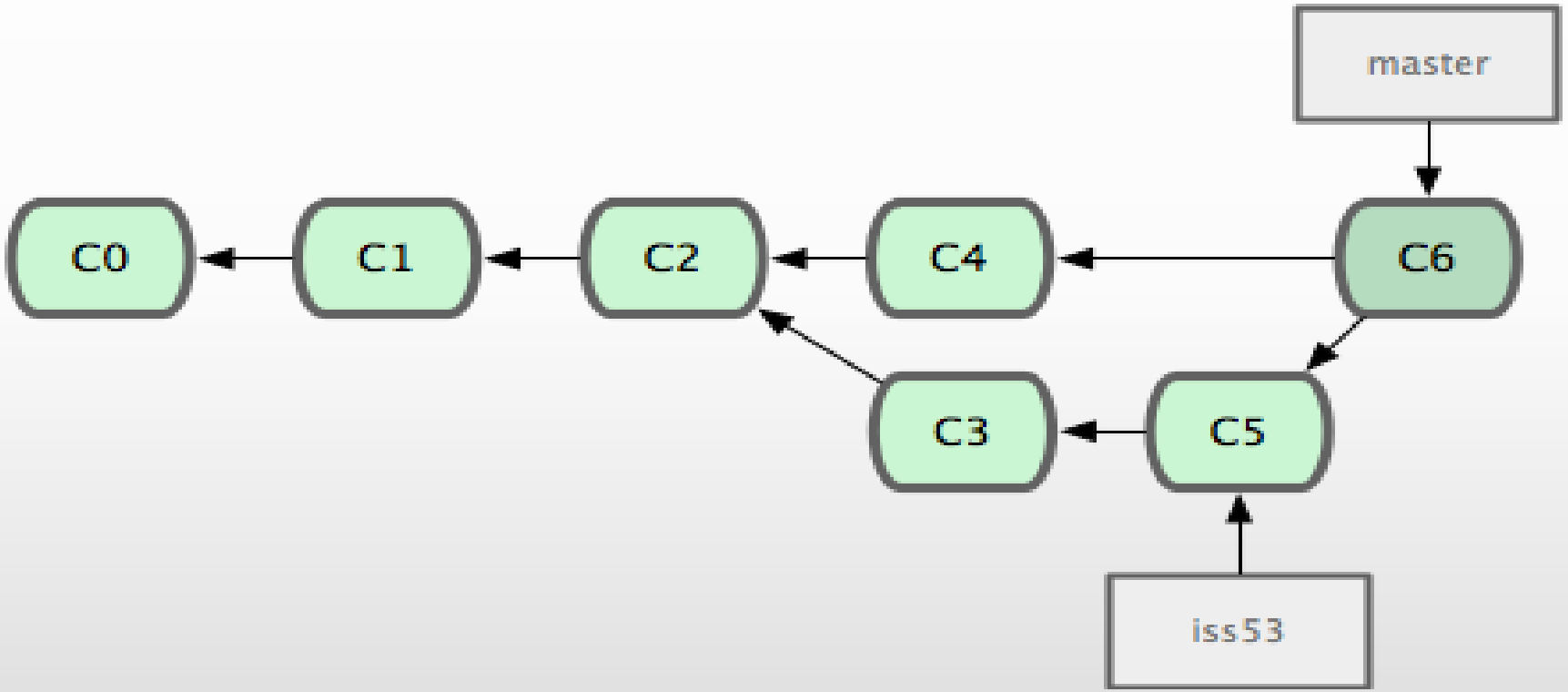
- Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work.
- In this case, Git does a simple **three-way merge**, using the two snapshots pointed to by the branch tips and the common ancestor of the two.
- Git automatically identifies the best common-ancestor merge base for branch merging.

three-way merge

- Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it
- This is referred to as a merge commit and is special in that it has more than one parent.
- Git determines the best common ancestor to use for its merge base;



three-way merge



git branch -d iss53

Basic Merge Conflicts

- If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly.
- $ git merge newlook
- Auto-merging index.html
- CONFLICT (content): Merge conflict in index.html
- Automatic merge failed; fix conflicts and then commit the result.

Basic Merge Conflicts

- Git hasn’t automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status
- [master*]$ git status
- index.html: needs merge
- # On branch master
- # Changed but not updated:
- #
- (use "git add <file>..." to update what will be committed)
- #
- (use "git checkout -- <file>..." to discard changes in working directory)
- #
- # unmerged:
- index.html
- #

Basic Merge Conflicts

- Anything that has merge conflicts and hasn’t been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts
- You can use git mergetool to fix the issue
- $ git mergetool
- merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
- Merging the files: index.html
- Normal merge conflict for ’index.html’:
- {local}: modified
- {remote}: modified
- Hit return to start merge resolution tool (opendiff):

Basic Merge Conflicts

- You can run git status again to verify that all conflicts have been resolved:
- $git status
- On branch master
- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)
- modified:index.html
- you can type git commit to finalize the merge commit.

Branch Management

- listing of your current branches:
- $ git branch
- iss53
- * master
- Testing
- Notice the * character that prefixes the master branch: it indicates the branch that you currently have checked out.

Branch Management

- To see the last commit on each branch, you can run git branch v :
- $ git branch -v
- iss53
- 93b412c fix javascript issue
- * master 7a98805 Merge branch ’iss53’
- testing 782fd34 add scott to the author list in the readmes

Branch Management

- To see which branches are already merged into the branch you're on, you can run git branch merged :
- $ git branch --merged
- iss53
- * master
- Because you already merged in iss53 earlier, you see it in your list. Branches of this list without the * in front of them are generally fine to delete with git branch -d you've already incorporated their work into another branch, so you're not going to lose anything.

Branch Management

- To see all the branches that contain work you haven’t yet merged in
- $ git branch --no-merged
- Testing
- Trying to delete it with git branch -d will fail:
- $ git branch -d testing
- error: The branch ’testing’ is not an ancestor of your current HEAD.
- If you are sure you want to delete it, run git branch -D testing

Branching Workflows

- we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate it into your own development cycle.
  - Long-Running Branches
  - Topic Branches

Long-Running Branches

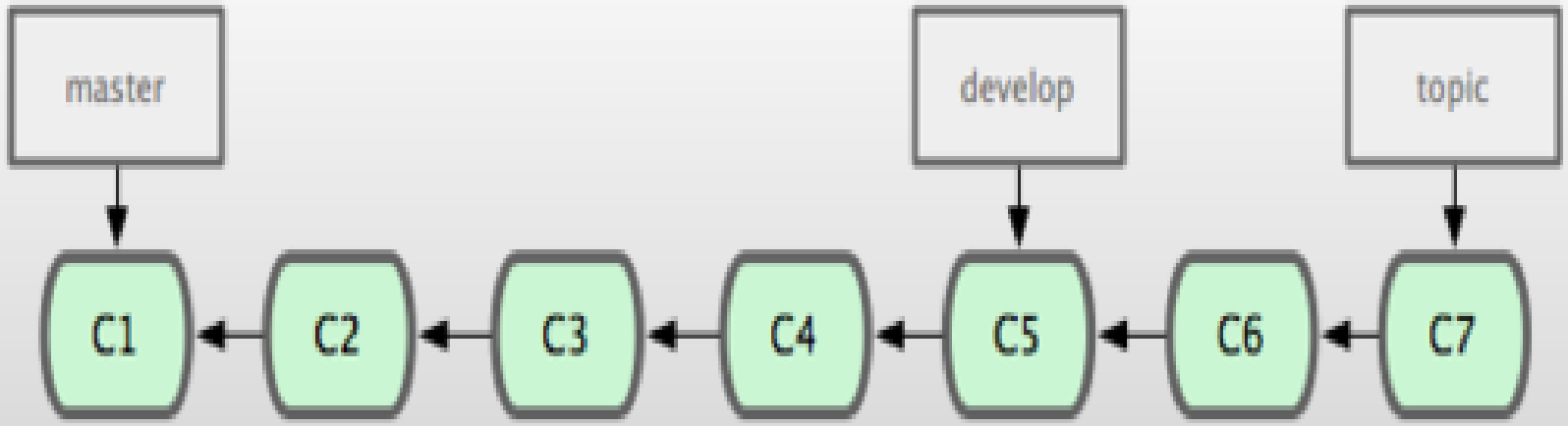
- Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do.
- This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Long-Running Branches

- Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their master branch — possibly only code that has been or will be released.
- They have another parallel branch named develop or next that they work from or use to test stability — it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into master .

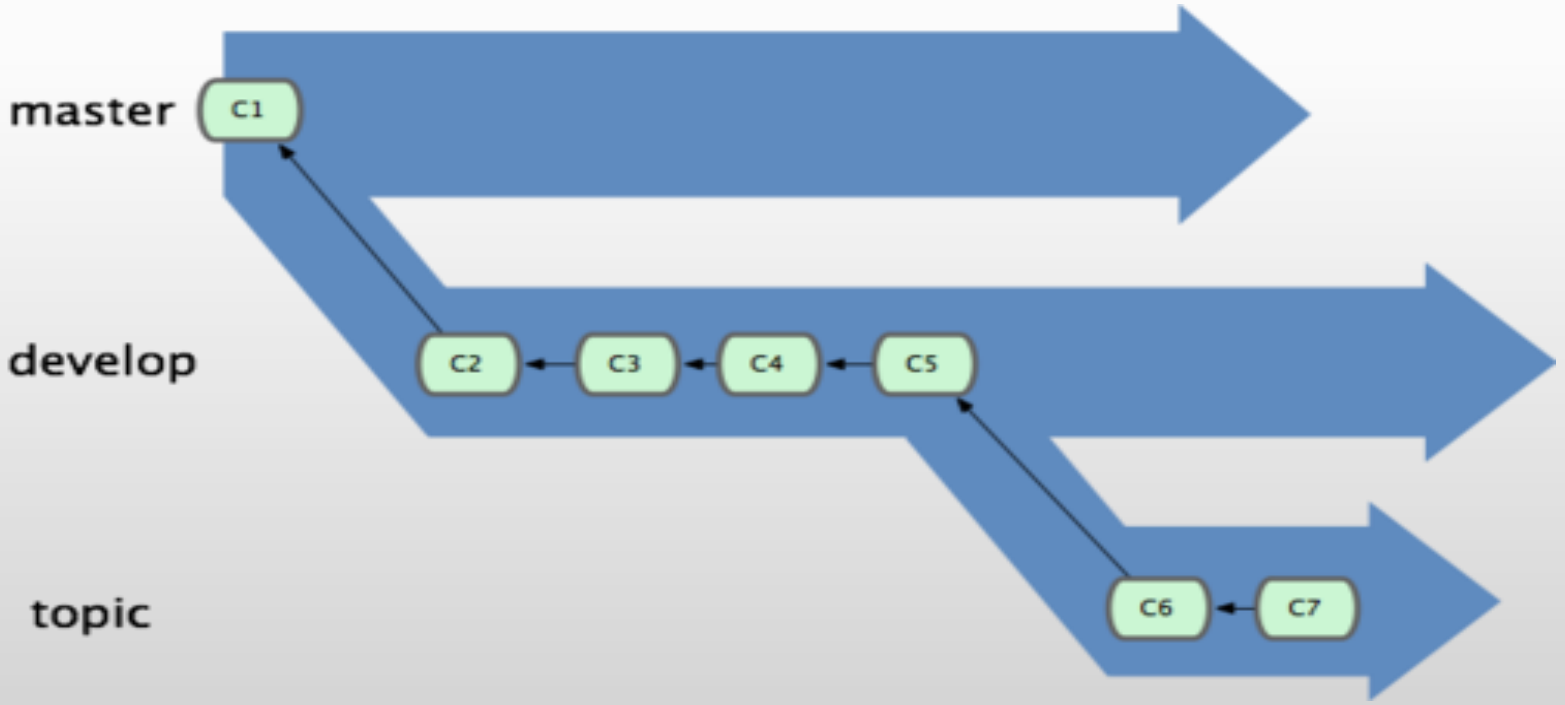
Long-Running Branches

- In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history



Long-Running Branches

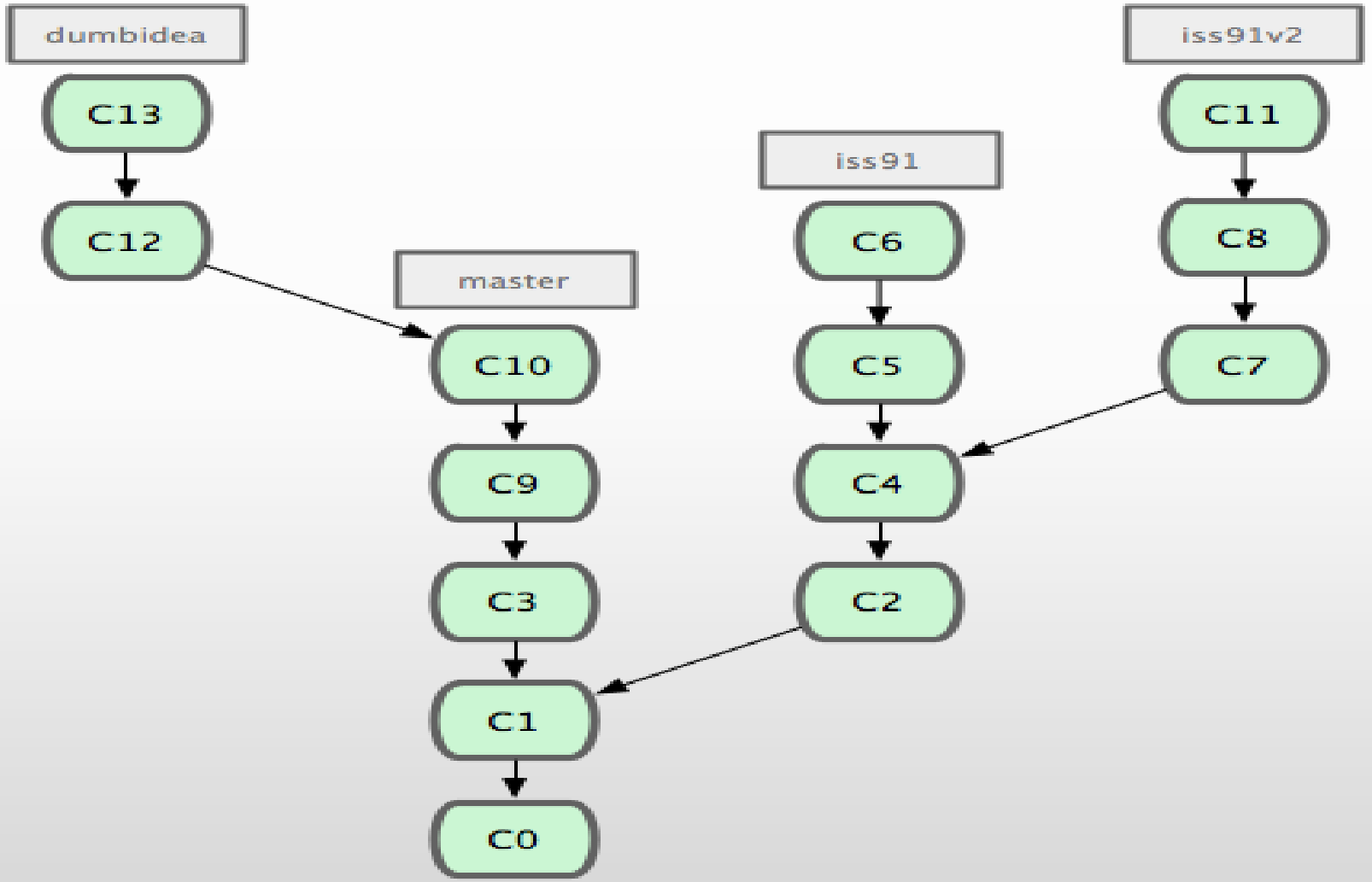
- It’s generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they’re fully tested



Topic Branches

- A topic branch is a shortlived branch that you create and use for a single particular feature or related work.
- Consider an example of doing some work (on master), branching off for an issue(iss91), working on it for a bit, branching off the second branch to try another way of handling the same thing (iss91v2), going back to your master branch and working there for a while, and then branching off there to do some work that you’re not sure is a good idea (dumbidea branch).

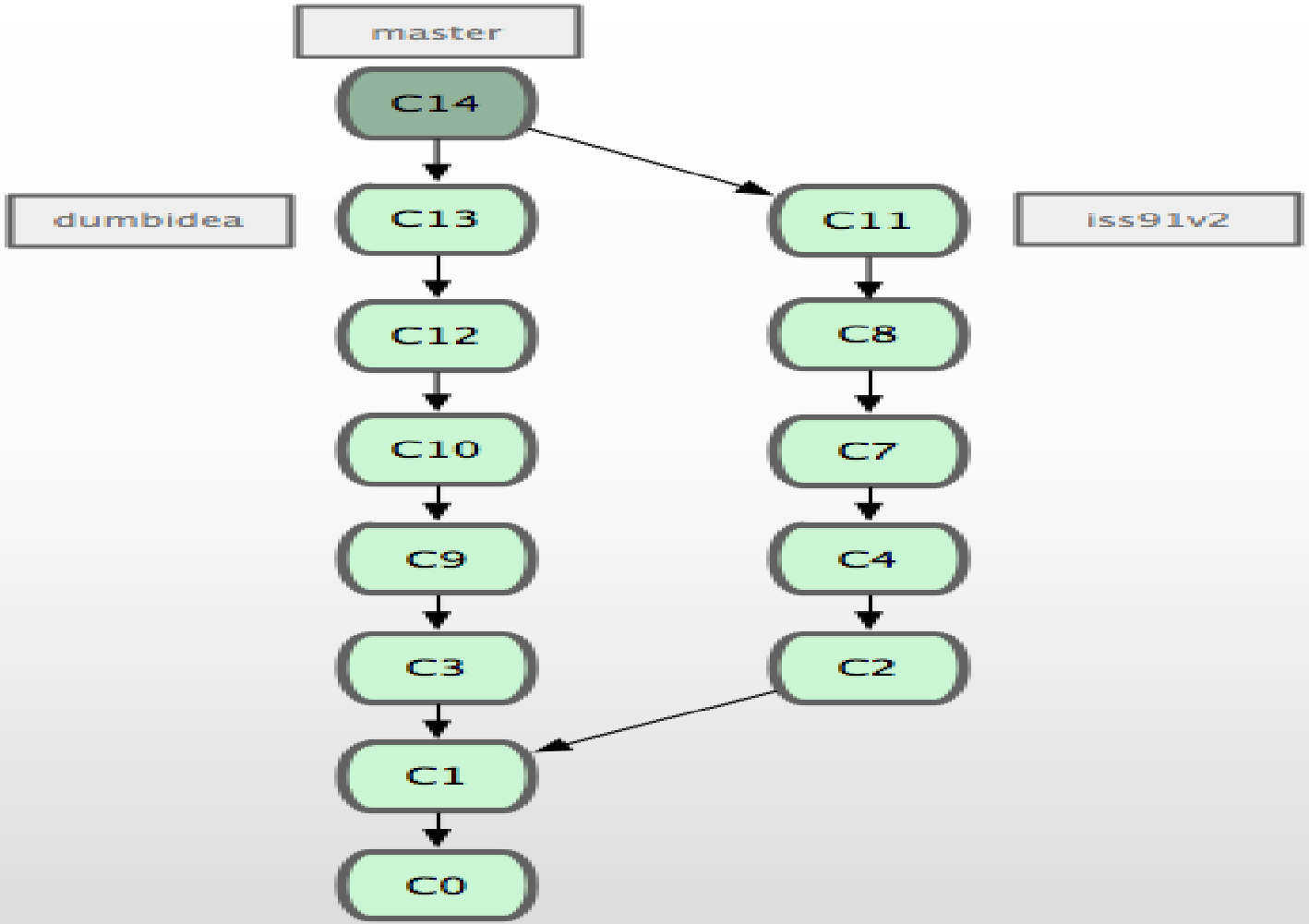
Topic Branches



Topic Branches

- Now, let's say you decide you like the second solution to your issue best (iss91v2); and you showed the dumbidea branch to your coworkers, and it turns out to be genius.
- You can throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two. Your history then looks like

Topic Branches



Remote Branches

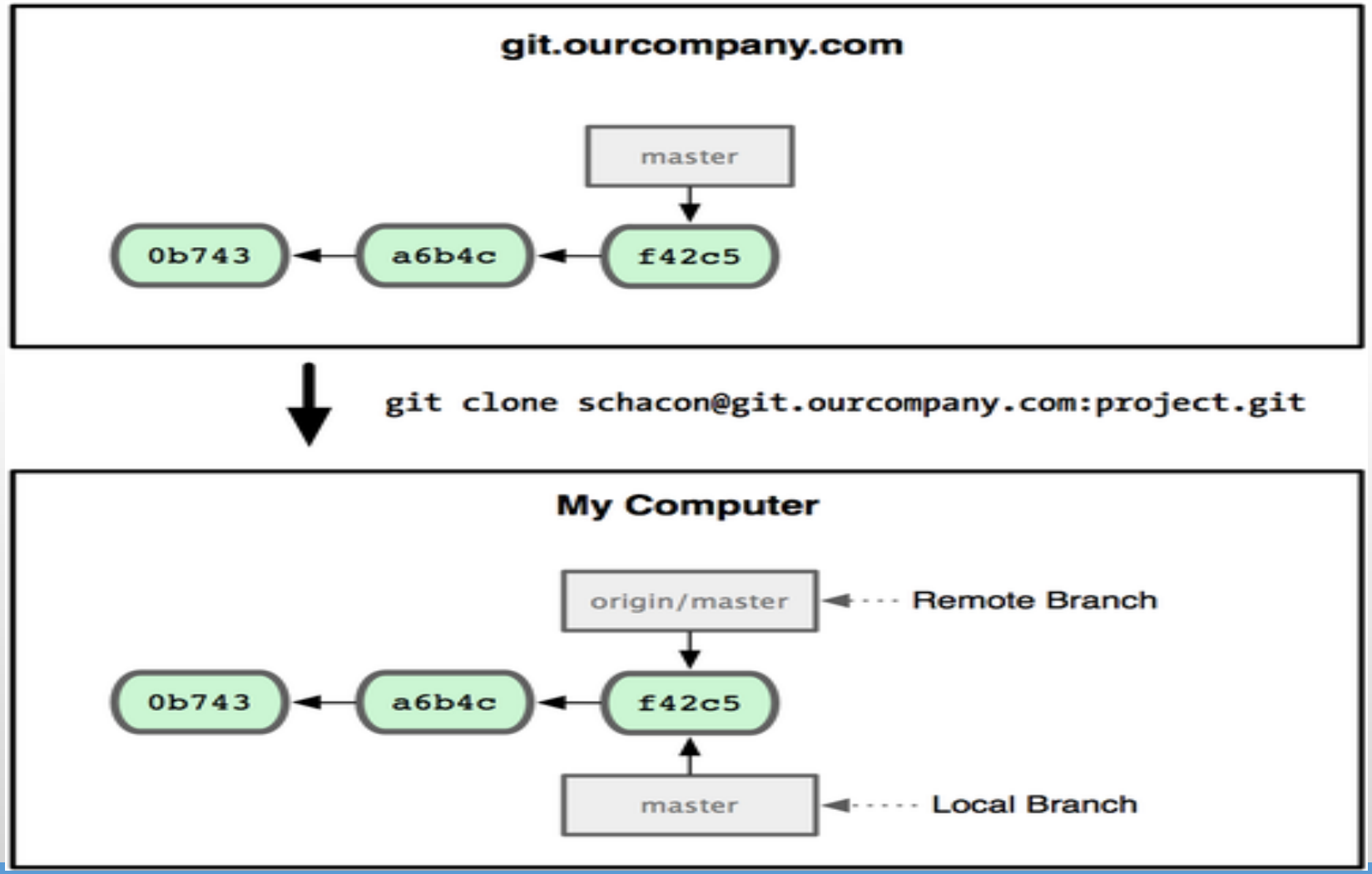
Remote Branches

- Remote branches are references to the state of branches on your remote repositories.
- They’re local branches that you can’t move; they’re moved automatically whenever you do any network communication.
- Remote branches act as bookmarks to remind you where the branches on your remote repositories were the last time you connected to them.

Remote Branches

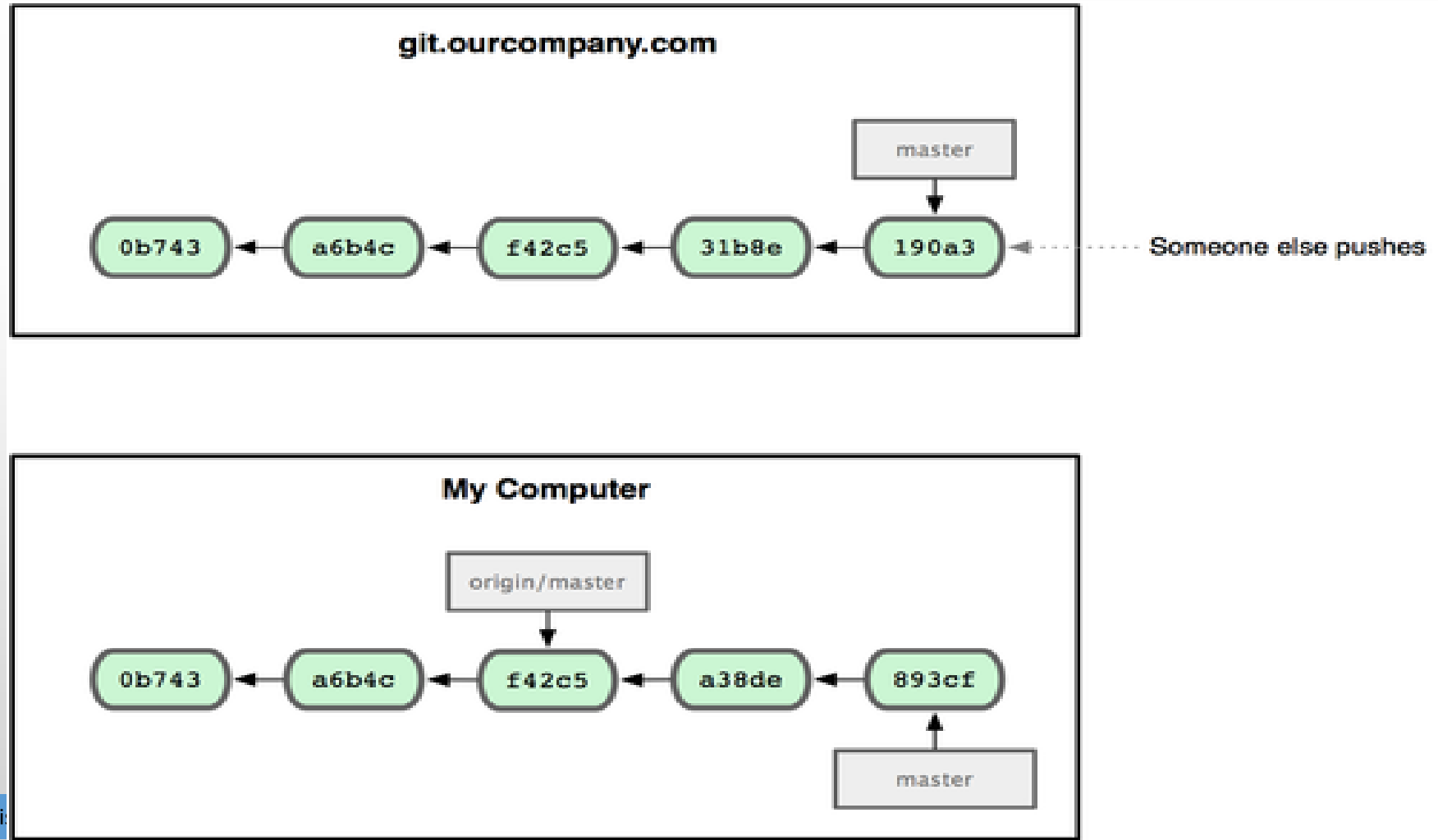
- They take the form (remote)/(branch) . For instance, if you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it, you would check the origin/master branch.

Remote Branches

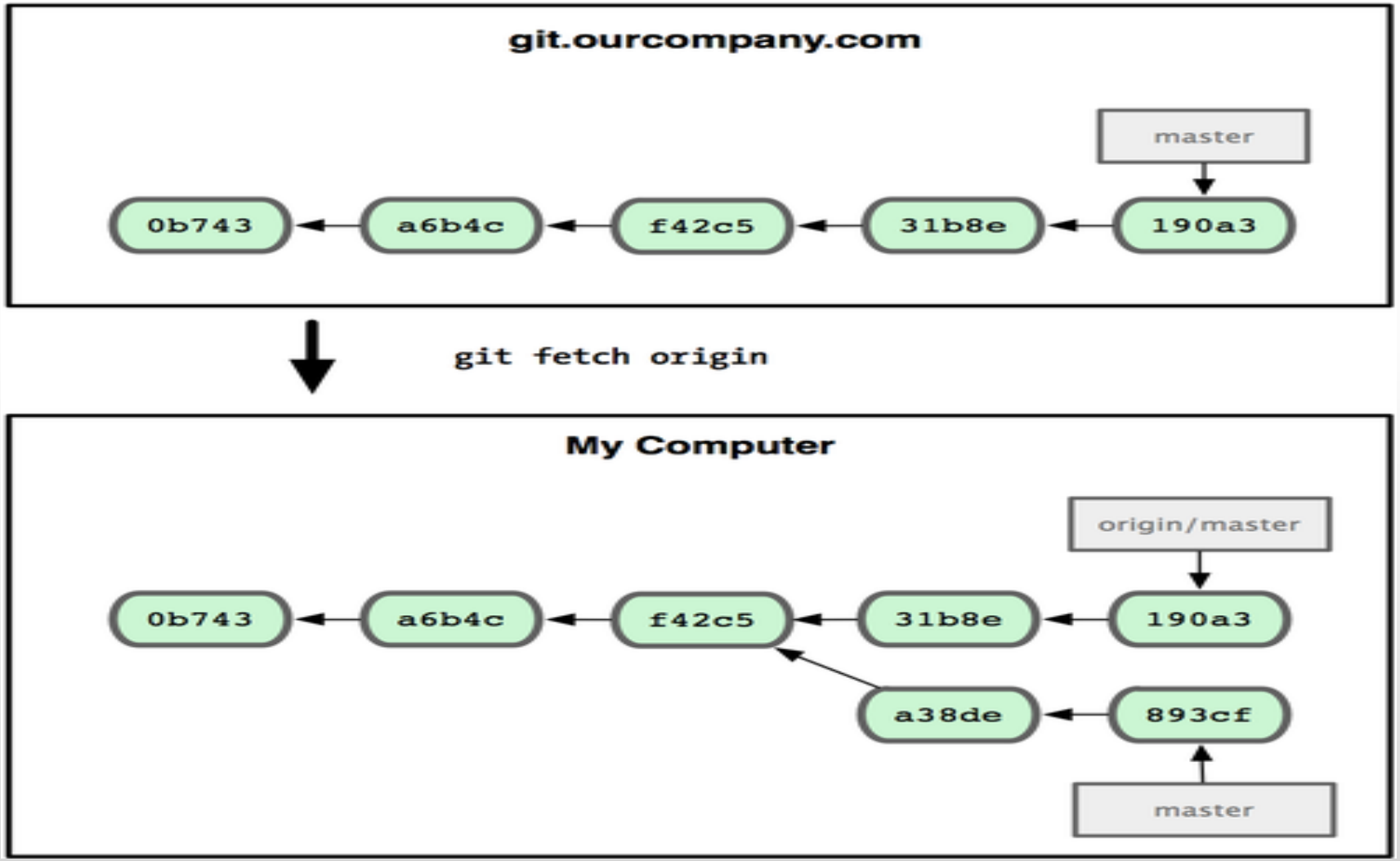


Remote Branches

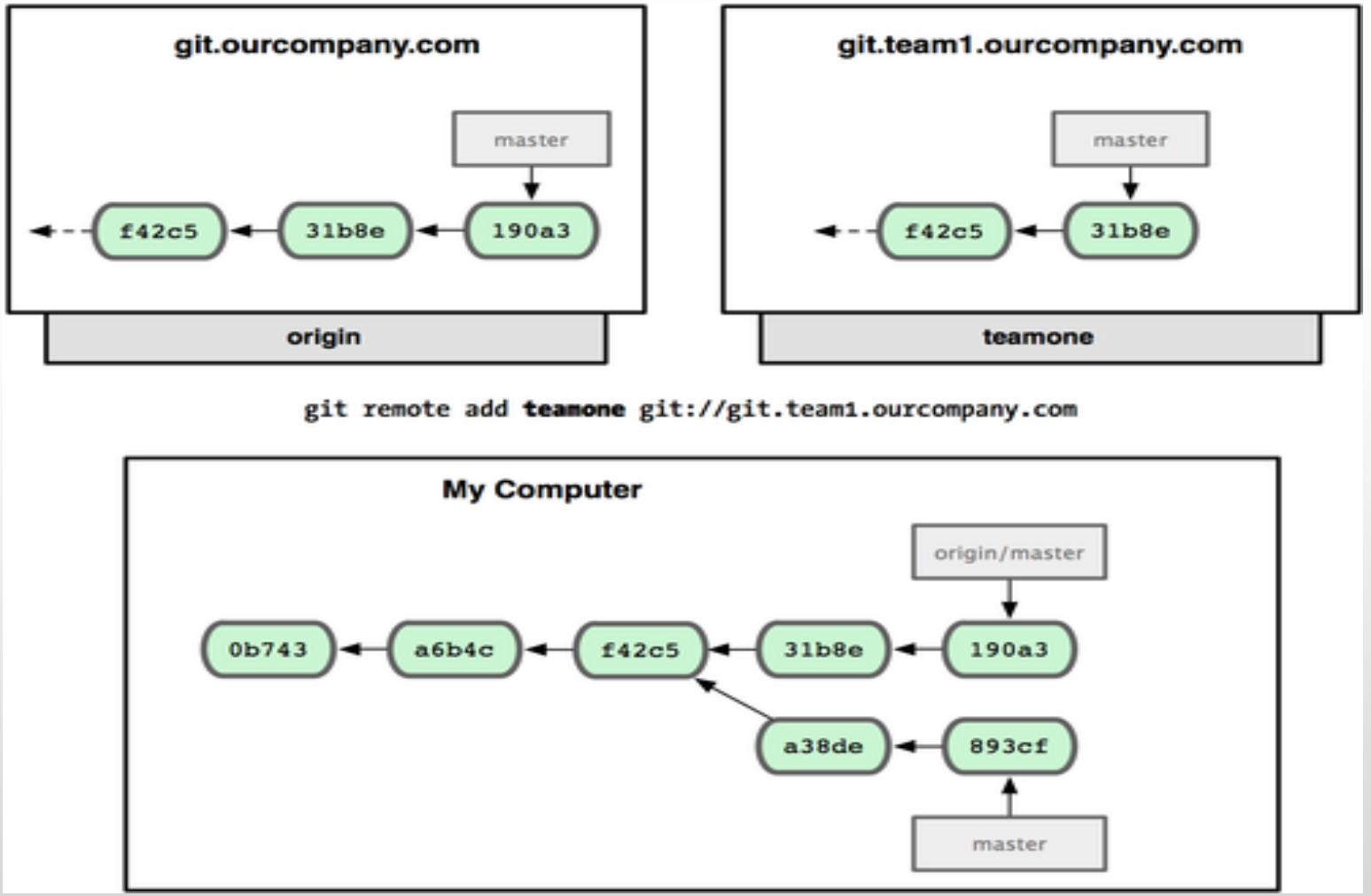
Working locally and having someone push to your remote server makes each history move forward differently.



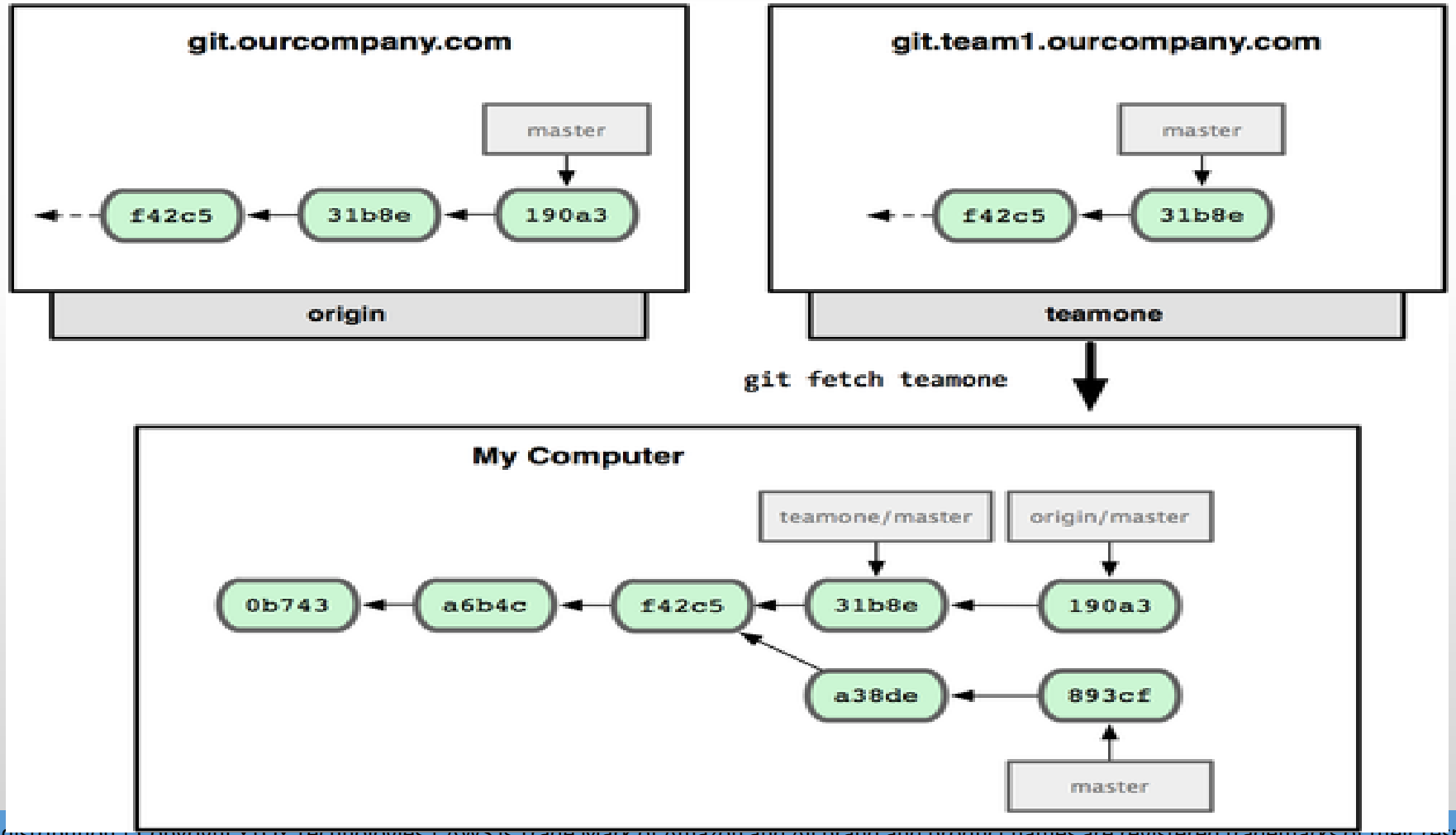
Remote Branches



Multiple Remotes



Multiple Remotes



Pushing

- When you want to share a branch with the world, you need to push it up to a remote that you have write access to.
- Your local branches aren't automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share.
- That way, you can use private branches do work you don't want to share, and push up only the topic branches you want to collaborate on.

Pushing particular branch

- If you have a branch named serverfix that you want to work on with others:
  - git push (remote) (branch) :

- $ git push origin serverfix
- Counting objects: 20, done.
- Compressing objects: 100% (14/14), done.
- Writing objects: 100% (15/15), 1.74 KiB, done.
- Total 15 (delta 5), reused 0 (delta 0)
- To git@github.com:schacon/simplegit.git
- * [new branch]
- serverfix -> serverfix

Pushing particular branch

- Git automatically expands the serverfix branchname out to refs/heads/serverfix:refs/heads/serverfix ,
- which means, “Take my serverfix local branch and push it to update the remote’s serverfix branch.”
- You can also do git push origin serverfix:coolfix

Pushing particular branch

- The next time one of your collaborators fetches from the server, they will get a reference to where the server's version of serverfix is under the remote branch origin/serverfix :
- $ git fetch origin
- remote: Counting objects: 20, done.
- remote: Compressing objects: 100% (14/14), done.
- remote: Total 15 (delta 5), reused 0 (delta 0)
- Unpacking objects: 100% (15/15), done.
- From git@github.com:schacon/simplegit
- * [new branch]
- serverfix
- -> origin/serverfix

Merging Remote Branches

- It’s important to note that when you do a fetch that brings down new remote branches,
- you don’t automatically have local, editable copies of them.
- you don’t have a new serverfix branch — you only have an origin/serverfix pointer that you can’t modify.
- To merge this work into your current working branch, you can run git merge origin/serverfix .

Merging Remote Branches

- If you want your own serverfix branch that you can work on, you can base it off your remote branch:
- $ git checkout -b serverfix origin/serverfix
- Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
- Switched to a new branch "serverfix"
- This gives you a local branch that you can work on that starts where origin/serverfix is.

Tracking Branches

- Checking out a local branch from a remote branch automatically creates what is called a tracking branch.
- Tracking branches are local branches that have a direct relationship to a remote branch.
- If you're on a tracking branch and type git push, Git automatically knows which server and branch to push to.
- Also, running git pull while on one of these branches fetches all the remote references and then automatically merges in the corresponding remote branch.

Tracking Branches

- When you clone a repository, it generally automatically creates a master branch that tracks origin/master . That's why git push and git pull work out of the box with no other arguments.
- you can set up other tracking branches if you wish — ones that don't track branches on origin and don't track the master branch.
- git checkout -b [branch] [remotename]/[branch] .

Tracking Branches

- $ git checkout --track origin/serverfix
- Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
- Switched to a new branch "serverfix"
- To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:
- $ git checkout -b sf origin/serverfix
- Branch sf set up to track remote branch refs/remotes/origin/serverfix.
- Switched to a new branch "sf"

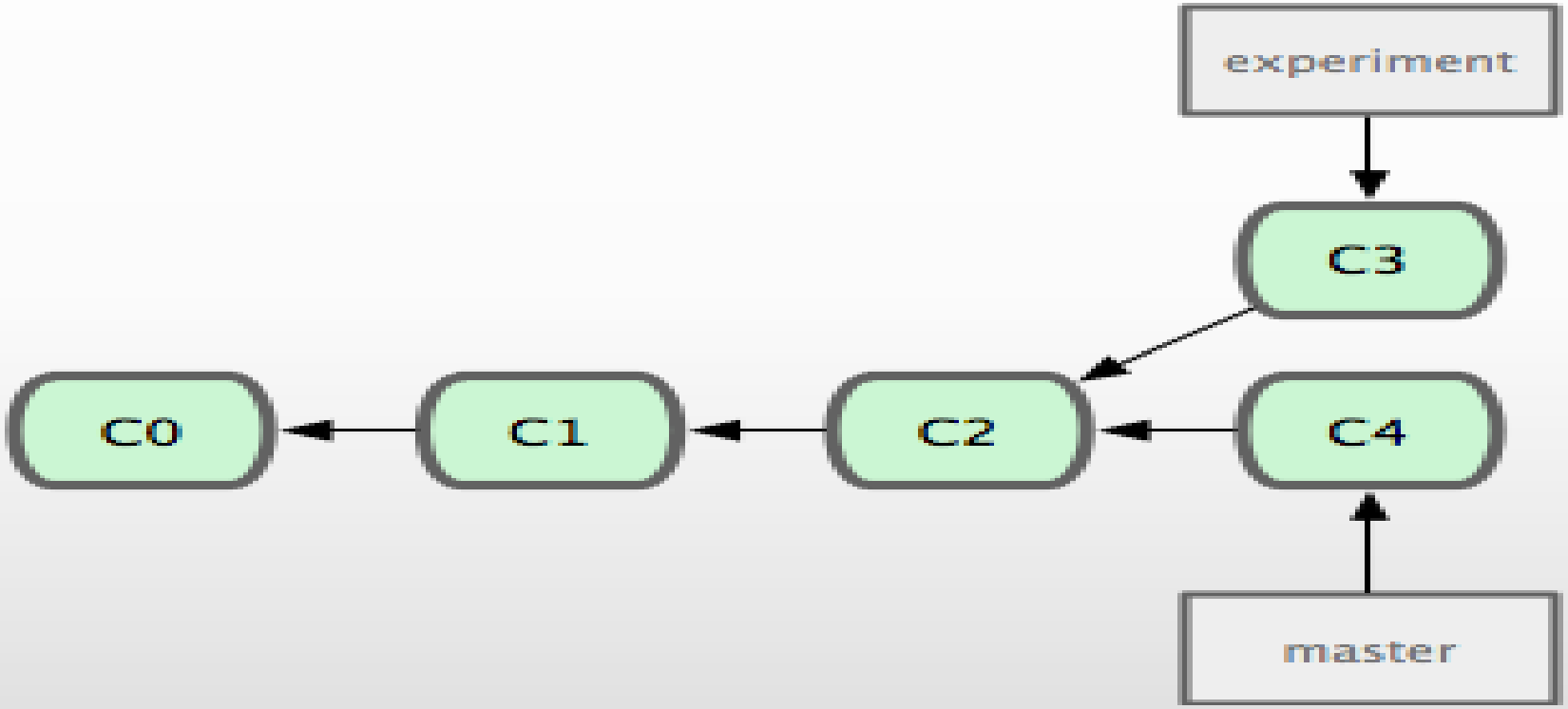
Deleting Remote Branches

- Suppose you're done with a remote branch — say, you and your collaborators are finished with a feature and have merged it into your remote's master branch
- You can delete a remote branch using the rather obtuse syntax
- git push [remotename] :[branch] .
- If you want to delete your serverfix branch from the server, you run the following:
- $ git push origin :serverfix
- To git@github.com:schacon/simplegit.git
- - [deleted] serverfix

Rebasing

- In Git, there are two main ways to integrate changes from one branch into another:
  - Merge
  - rebase

The Basic Rebase



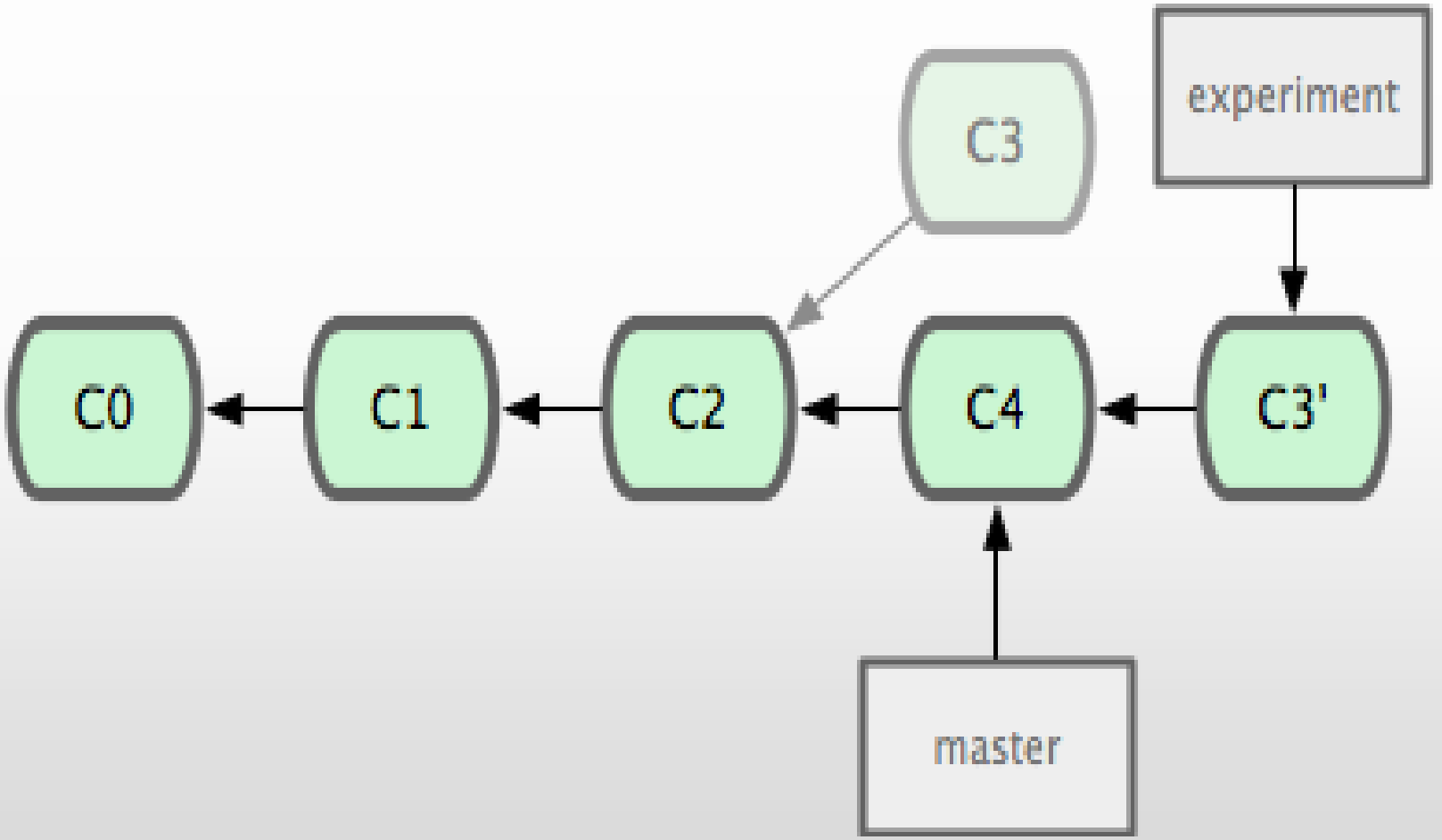
The Basic Rebase

- you can take the patch of the change that was introduced in C3 and reapply it on top of C4.
- In Git, this is called rebasing. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.
- In this example, you'd run the following:
- $ git checkout experiment
- $ git rebase master
- First, rewinding head to replay your work on top of it...
- Applying: added staged command

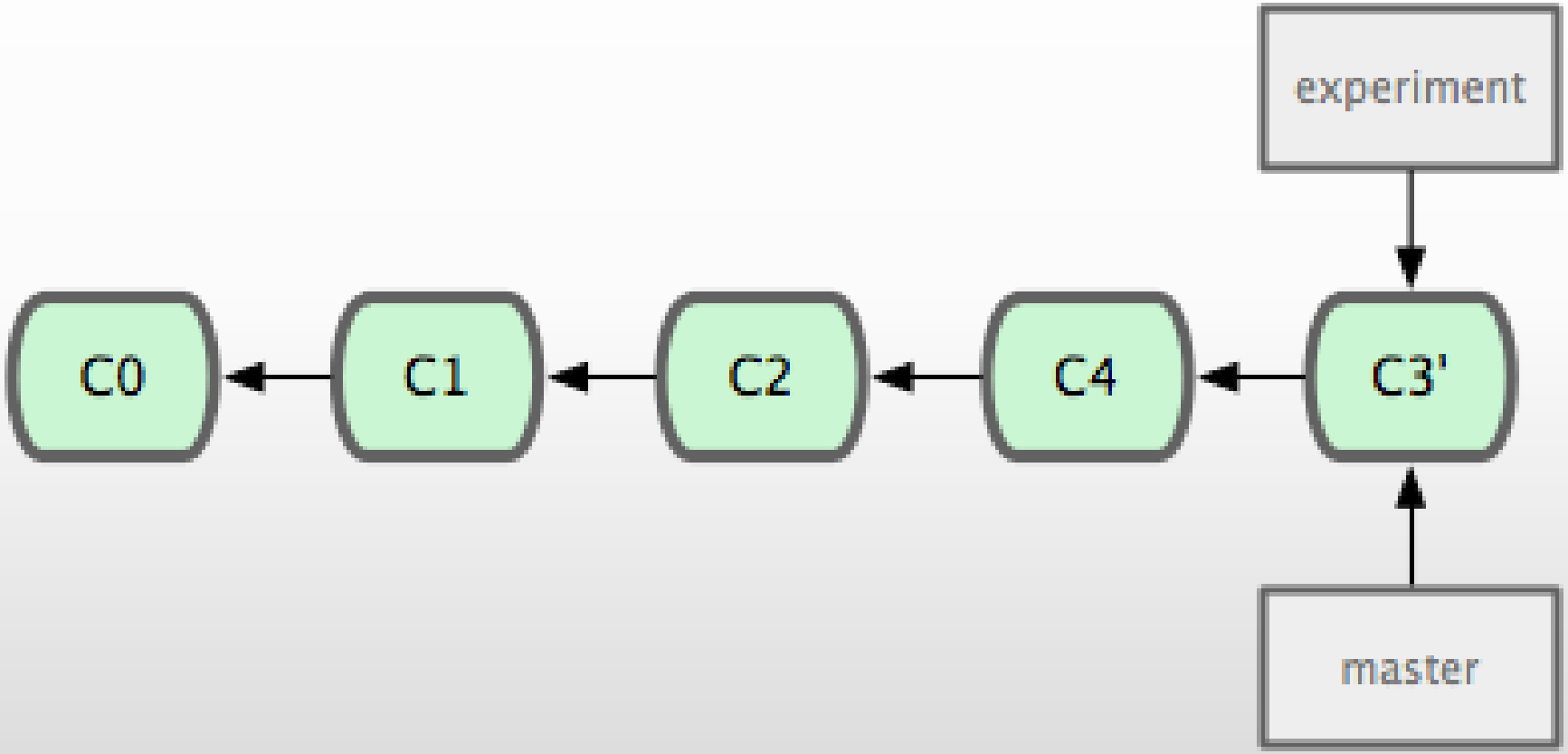
The Basic Rebase

- It works by going to the common ancestor of the two branches, getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files,resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.
- There is no difference in the end product of the integration between merge and rebase, but rebasing makes for a cleaner history.
- If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

The Basic Rebase



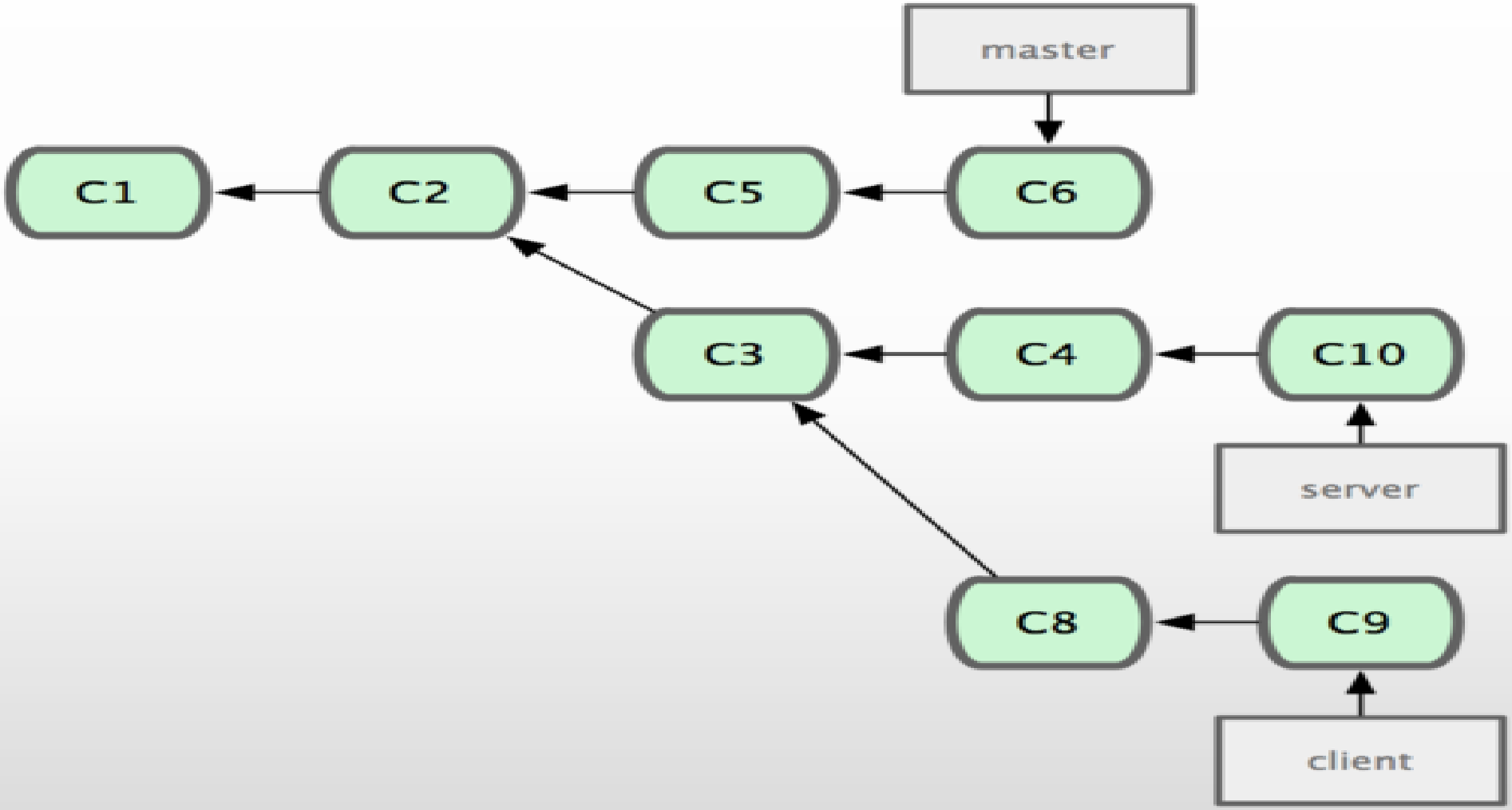
The Basic Rebase



The Basic Rebase

- Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

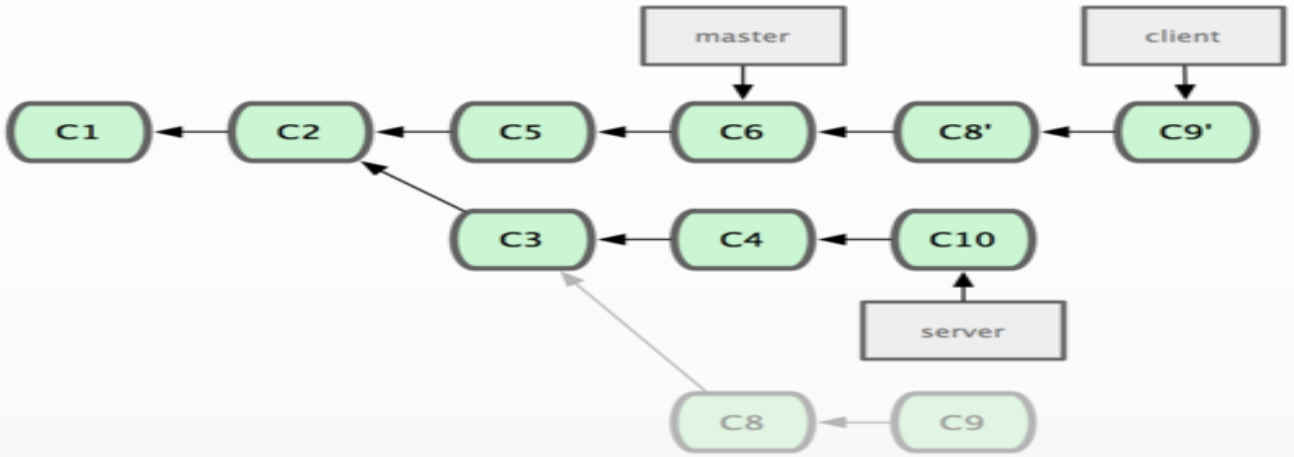
Complex rebasing



Complex rebasing

- $ git rebase --onto master server client
- “Check out the client branch, figure out the patches from the common ancestor of the client and server branches, and then replay them onto master .”

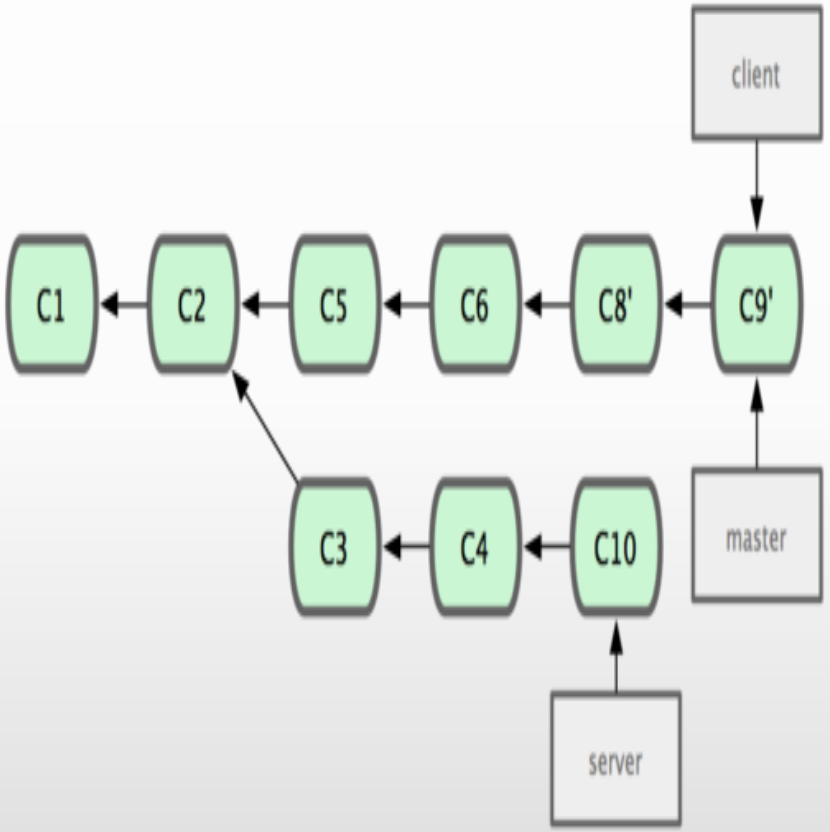
Complex rebasing



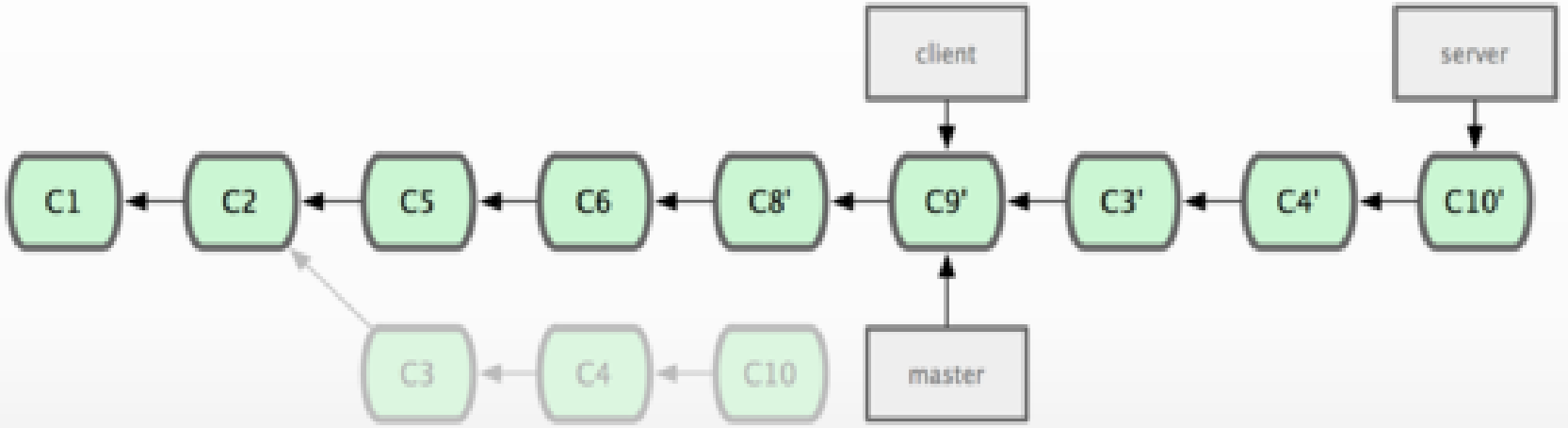
- Now you can fast-forward your master branch (see Figure 3.33):
- $ git checkout master
- $ git merge client

Complex rebasing

- Let’s say you decide to pull in your server branch as well.
- You can rebase the server branch onto the master branch without having to check it out first by running
- git rebase [basebranch] [topicbranch]
- $ git rebase master server

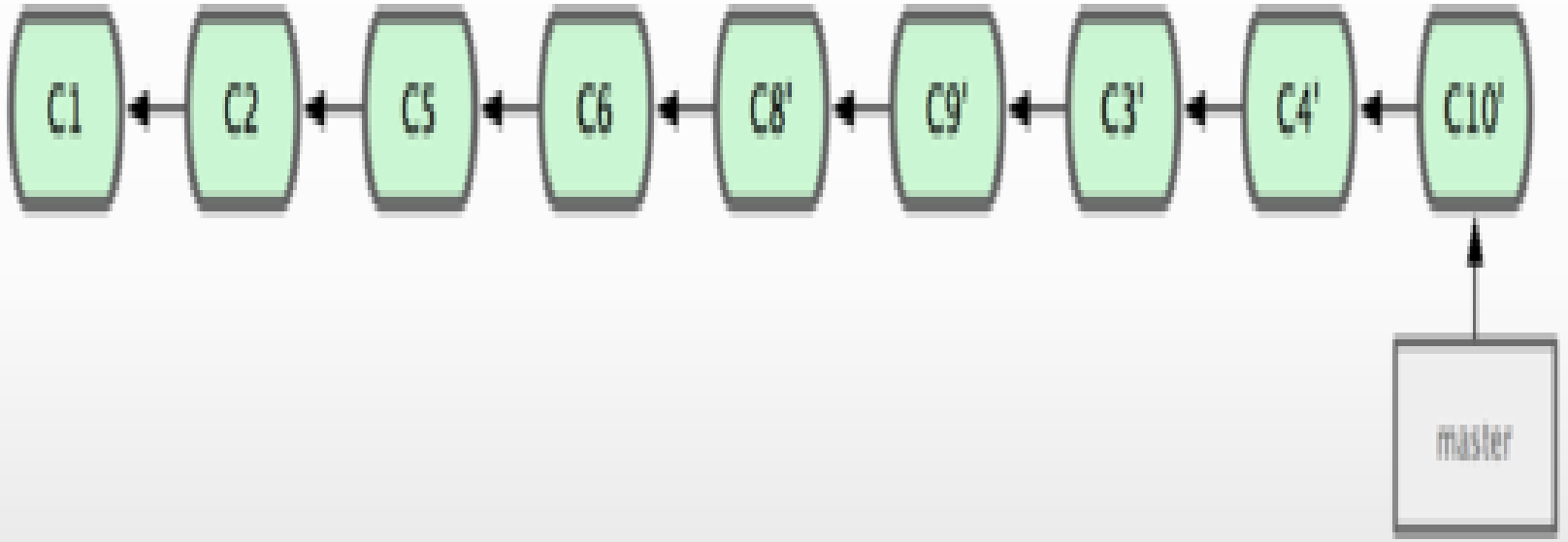


Complex rebasing



- Then, you can fast-forward the base branch (master):
  - $ git checkout master
  - $ git merge serve
- You can remove the client and server branches because all the work is integrated
  - $ git branch -d client
  - $ git branch -d server

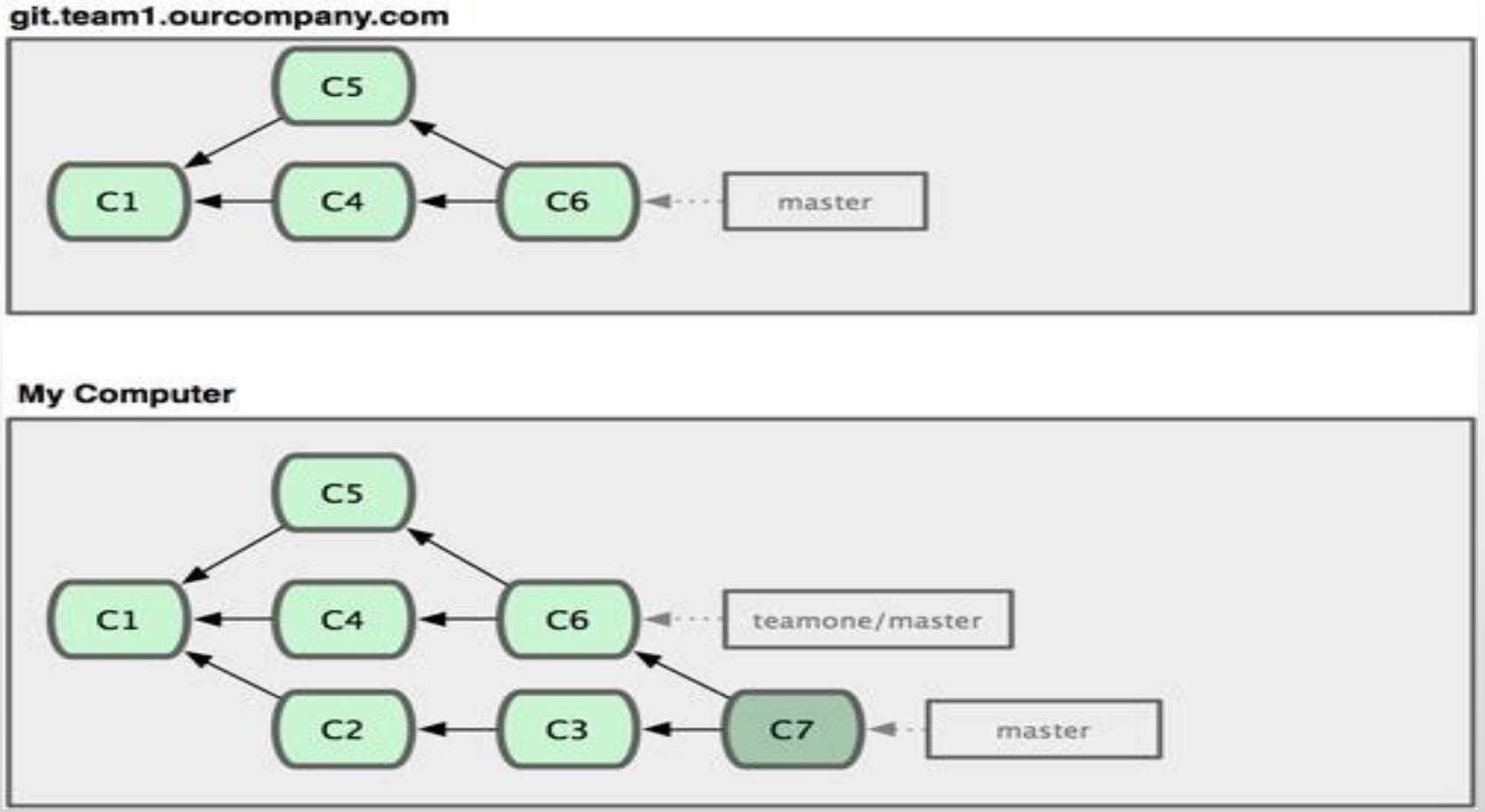
Complex rebasing



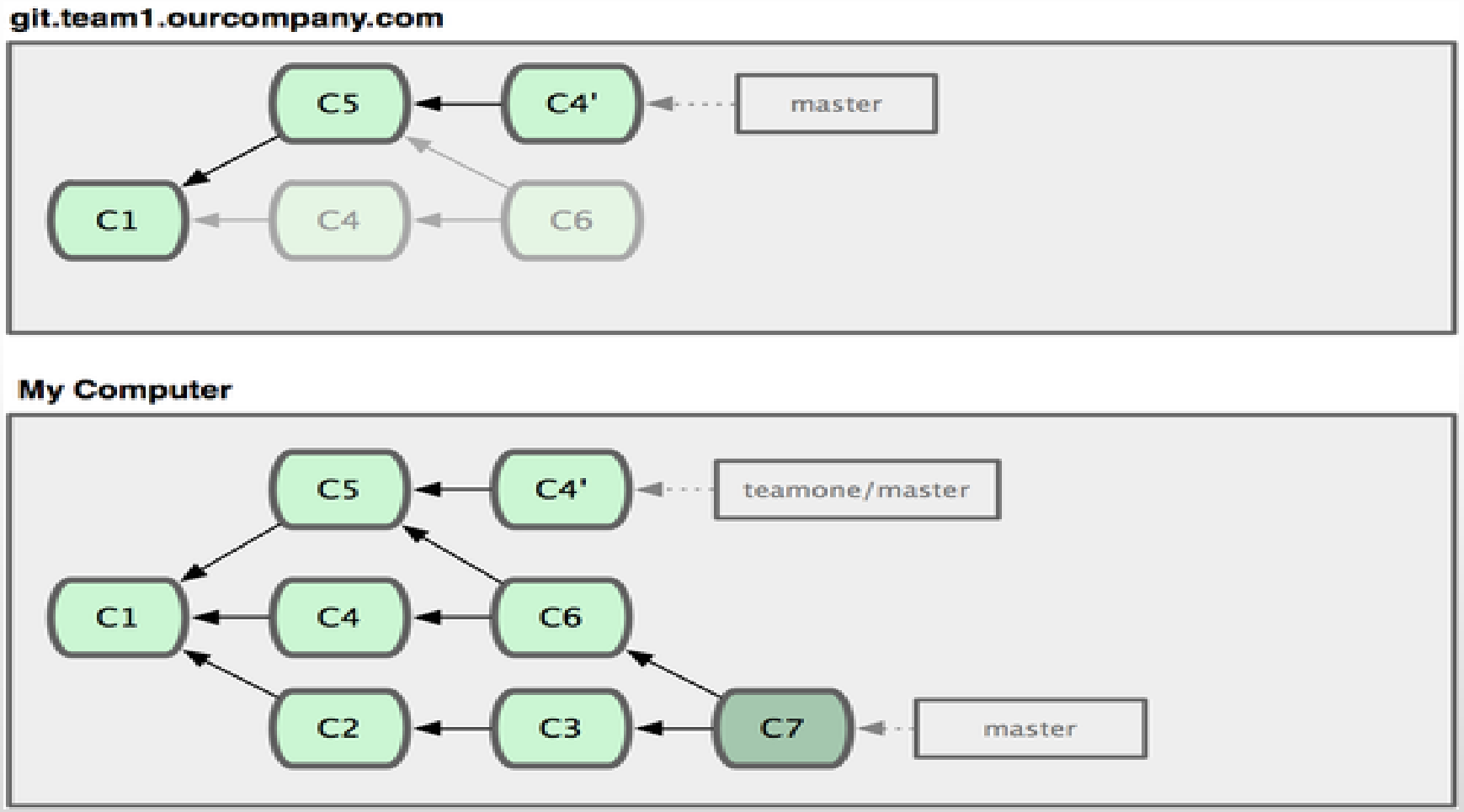
The Perils of Rebasing

- **Do not rebase commits that you have pushed to a public repository.**
- If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with git rebase and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

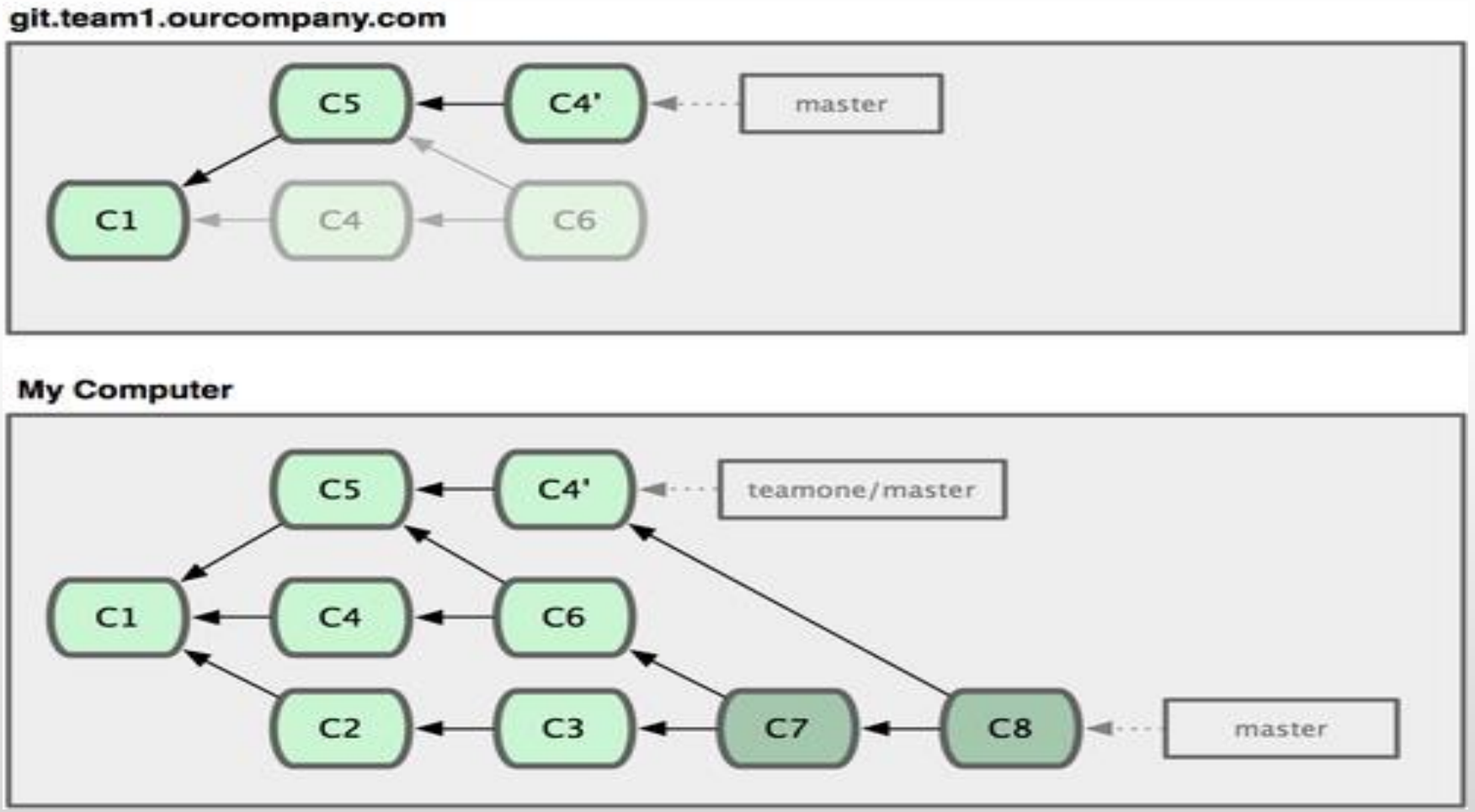
The Perils of Rebasing



The Perils of Rebasing



The Perils of Rebasing



Short SHA

- $ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
- $ git show 1c002dd4b536e7479f
- $ git show 1c002d

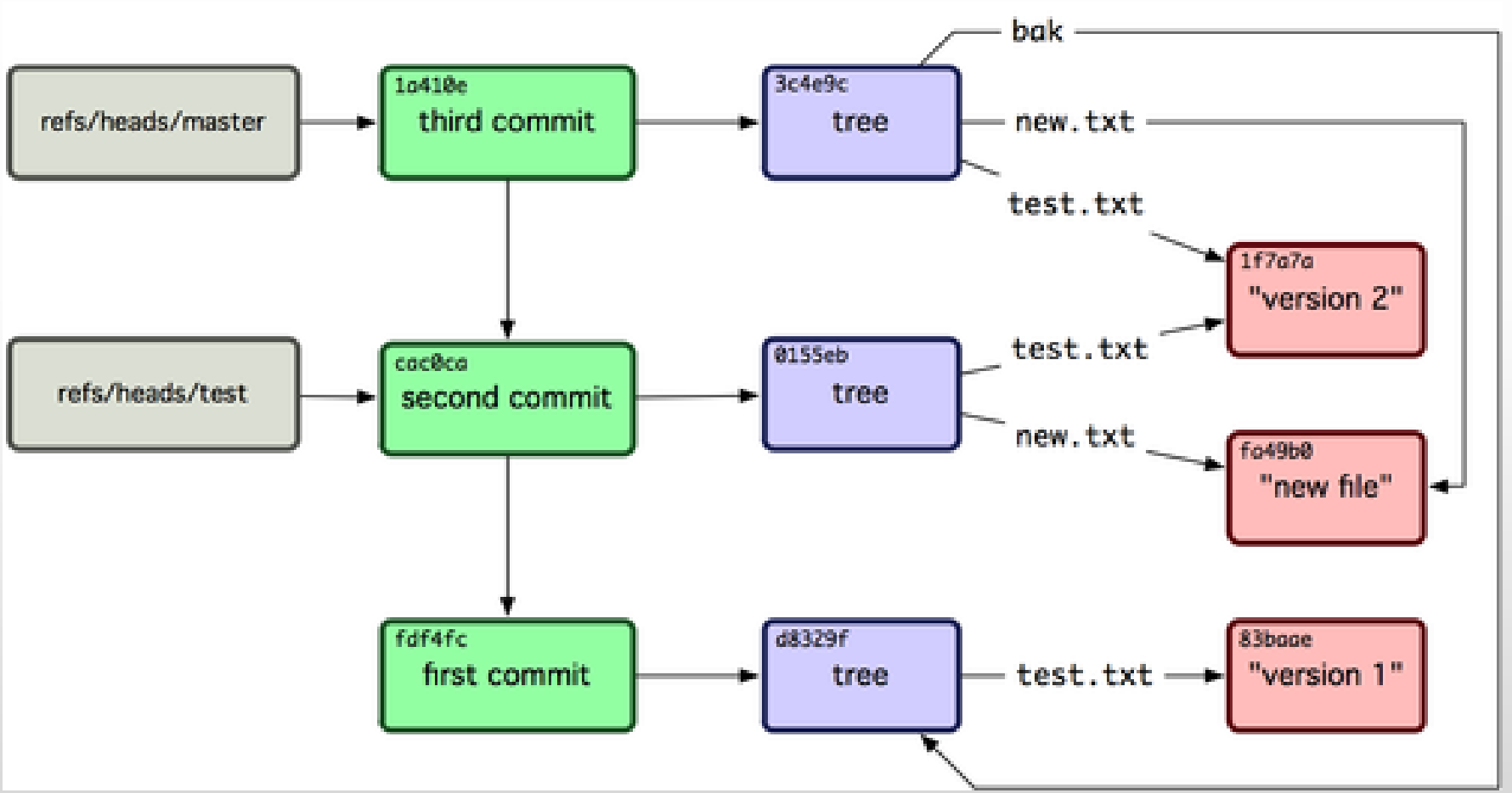
Branch References

- assuming that the topic1 branch points to ca82a6d :
- $ git show ca82a6dff817ec66f44342007202690a93763949
- $ git show topic1
- $ git rev-parse topic1
- ca82a6dff817ec66f44342007202690a93763949

Git References

- “references” or “refs”; you can find the files that contain the SHA–1 values in the .git/refs directory.
- $ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
- That’s basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit

Git References



The HEAD

- The HEAD file is a symbolic reference to the branch you're currently on.
- By symbolic reference, I mean that unlike a normal reference, it doesn't generally contain a SHA–1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:
- $ cat .git/HEAD
- ref: refs/heads/master
- $ git symbolic-ref HEAD
- refs/heads/master

Tags

- The tag object is very much like a commit object — it contains a tagger, a date, a message, and a pointer.
- The main difference is that a tag object points to a commit rather than a tree. It’s like a branch reference, but it never moves — it always points to the same commit but gives it a friendlier name.
- $ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 m ’test tag’
- $ cat .git/refs/tags/v1.1
- 9585191f37f7b0fb9444f35a9bf50de191beadc2

Remotes

- If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the refs/remotes directory.
- $ git remote add origin git@github.com:schacon/simplegit-progit.git
- It adds a section to your .git/config file, specifying the name of the remote (origin), the URL of the remote repository, and the refspec for fetching:
- [remote "origin"]
- url = git@github.com:schacon/simplegit-progit.git
- fetch = +refs/heads/*:refs/remotes/origin/*

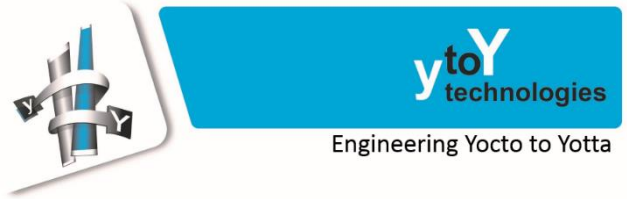
Remotes

- The format of the refspec is an optional + , followed by <src>:<dst> ,
- where <src>is the pattern for references on the remote side
- <dst> is where those references will be written locally.
- The + tells Git to update the reference even if it isn’t a fast-forward.

Remotes

- [remote "origin"]
- url = git@github.com:schacon/simplegit-progit.git
- fetch = +refs/heads/master:refs/remotes/origin/master
- fetch = +refs/heads/experiment:refs/remotes/origin/experiment

- [remote "origin"]
- url = git@github.com:schacon/simplegit-progit.git
- fetch = +refs/heads/master:refs/remotes/origin/master
- fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*



Remotes

- [remote "origin"]
- url = git@github.com:schacon/simplegit-progit.git
- fetch = +refs/heads/*:refs/remotes/origin/*
- push = refs/heads/master:refs/heads/qa/master

Deleting References

- You can also use the refspec to delete references from the remote server by running something like this:
- $ git push origin :topic
- Because the refspec is <src>:<dst> , by leaving off the <src> part, this basically says to make the topic branch on the remote nothing, which deletes it.

Reflog

- **$ git reflog**
- 734713b... HEAD@{0}:
- d921970... HEAD@{1}:
- 1c002dd... HEAD@{2}:
- 1c36188... HEAD@{3}:
- 95df984... HEAD@{4}:
- 1c36188... HEAD@{5}:
- 7e05da5... HEAD@{6}:
- commit: fixed refs handling, added gc auto, updated
- merge phedders/rdocs: Merge made by recursive.
- commit: added some blame and merge stuff
- rebase -i (squash): updating HEAD
- commit: # This is a combination of two commits.
- rebase -i (squash): updating HEAD
- rebase -i (pick): updating HEAD