Change Management

Week 9

Cite Tomer Weiss

What Changes Are We Managing?

Software

- Planned software development
  - team members constantly add new code
- (Un)expected problems
  - bug fixes
- Enhancements
  - Make code more efficient (memory, execution time)

“The only constant in software development is change”

Q: What kind of information you want to keep for changes?

Features Required to Manage Change

- Backups
- Timestamps
- Who made the change?
- Where was the change made?
- A way to communicate changes with team

How to achieve that

- Figure out which parts changed (diff?)
- Communicate changes with team (patch?)
- But diff and patch are not that good

Disadvantages of diff & patch

- Diff requires keeping a copy of old file before changes
- Work with only 2 versions of a file (old & new)
  - Projects will likely be updated more than once
  - store versions of the file to see how it evolved over time

    index.html
    index-2009-04-08.html
    index-2009-06-06.html
    index-2009-08-10.html
    index-2009-11-04.html
    index-2010-01-23.html
    index-2010-09-21.html
- Numbering scheme becomes more complicated if we need to store two versions for the same date

Disadvantages of diff & patch

- Two people may edit the same file on the same date
  - 2 patches need to be sent and merged
- Changes to one file might affect other files (.h & .c)
  - Need to make sure those versions are stored together as a group

Version Control System

- Track changes to code and other files related to software
  - What new files were added?
  - What changes made to files?
  - Which version had what changes?
  - Which user made the changes?
  - Revert to previous version
- Track entire history of software
- Source control software
  - Git, Subversion (SVN), CVS, and others

Version Control System Allows you to

- See who introduced an issue and when
- see who last modified something that might be causing a problem
- compare changes over time
- revert the entire project back to a previous state
- revert files back to a previous state
- And more ...

Version Control System

- Three types
  - Local version control system
  - Centralized Version Control System
  - Distributed Version Control System

Local Version Control Systems

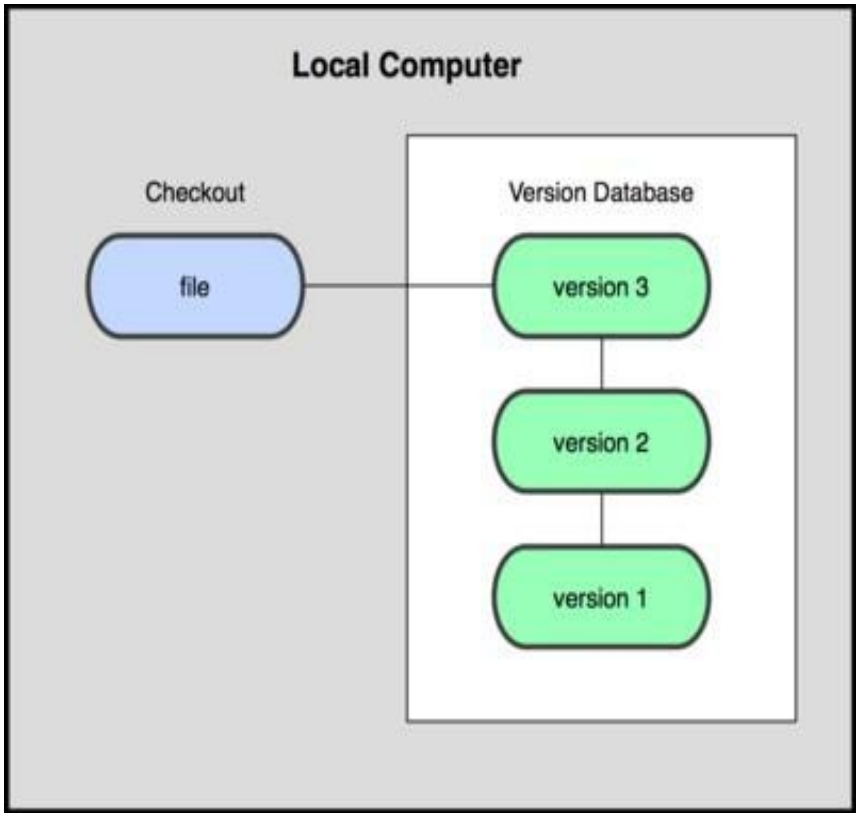


Image Source: git-scm.com

- Organize different versions as folders on the local machine
- No server involved
- Other users copy with disk/network
- Ex: rcs

Centralized Version Control System

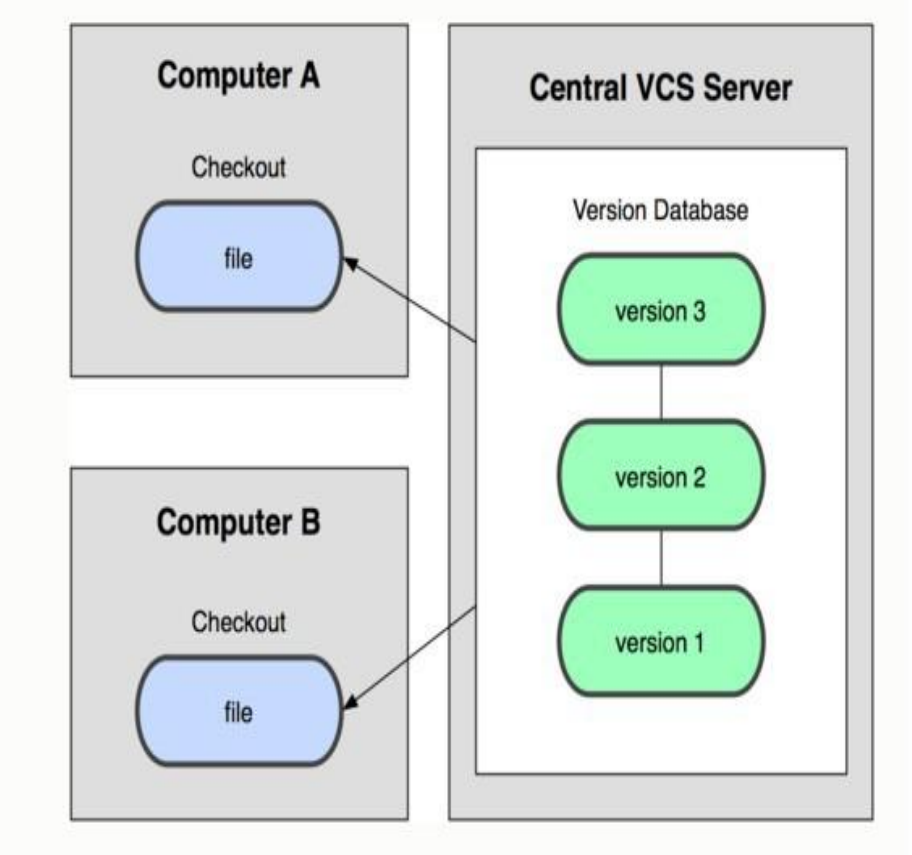


Image Source: git-scm.com

- Version history sits on a central server
- Users will get a working copy of the files
- Changes have to be committed to the server
- All users can get the changes
- Ex: Subversion (SVN)

Centralized: Pros and Cons

"The full project history is only stored in one central place."

Pros

- Everyone can see changes at the same time
- Simple to design

Cons

- Single point of failure (no backups!)

Distributed Version Control System

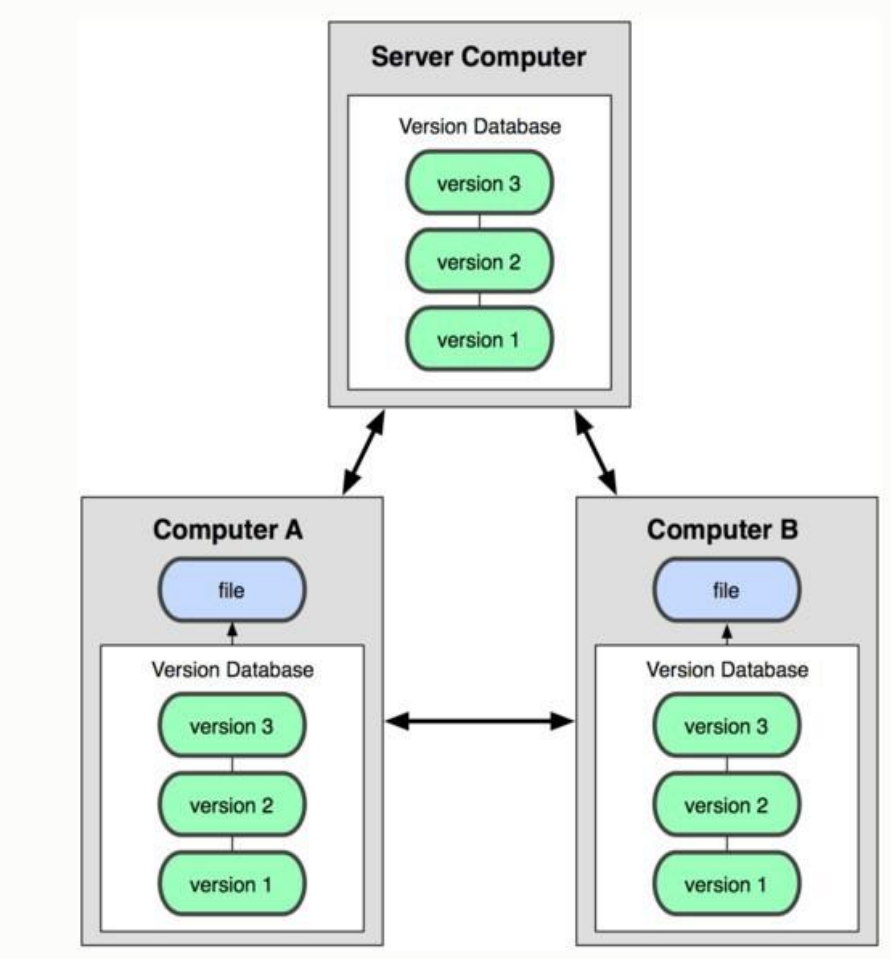


Image Source: git-scm.com

- Version history is replicated on every user's machine
- Users have version control all the time
- Changes can be communicated between users
- Ex: Git

Distributed: Pros and Cons

"The entire project history is downloaded to the hard drive"

Pros

- Commit changes/revert to an old version while offline
- Commands run extremely fast because tool accesses the hard drive and not a remote server
- Share changes with a few people before showing changes to everyone

Cons

- long time to download
- A lot of disk space to store all versions

Centralized vs. Distributed VCS

- Single central copy of the project history on a server
- Changes are uploaded to the server
- Other programmers can get changes from the server
- Examples: **SVN**, CVS

- Each developer gets the full history of a project on their own hard drive
- Developers can communicate changes between each other without going through a central server
- Examples: **Git**, Mercurial, Bazaar, Bitkeeper

SVN vs Git

- Git uses distributed version control
- Svn uses centralized version control

- Git uses a stream of snapshots to store data
- Svn uses files and a list of file-based changes

Data in SVN

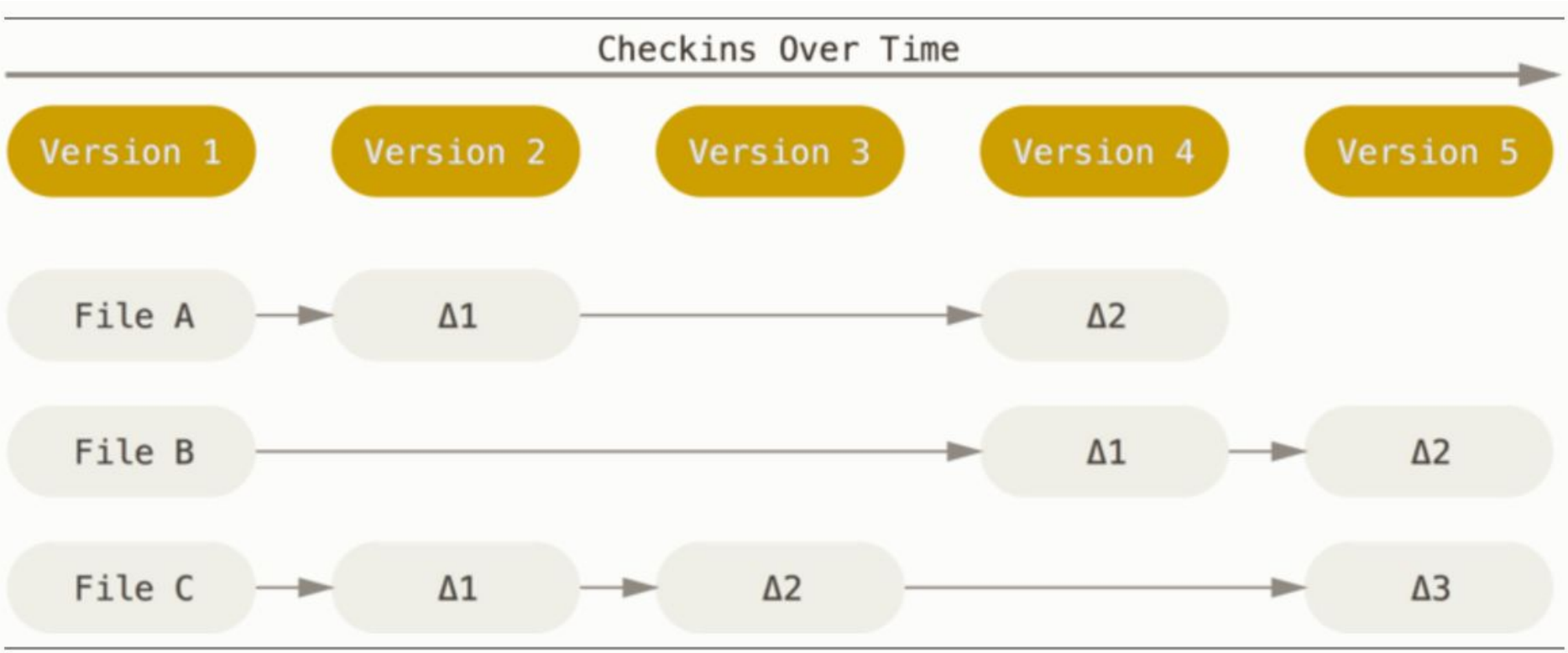


Figure 1-4. Storing data as changes to a base version of each file.

Cite: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Data in Git

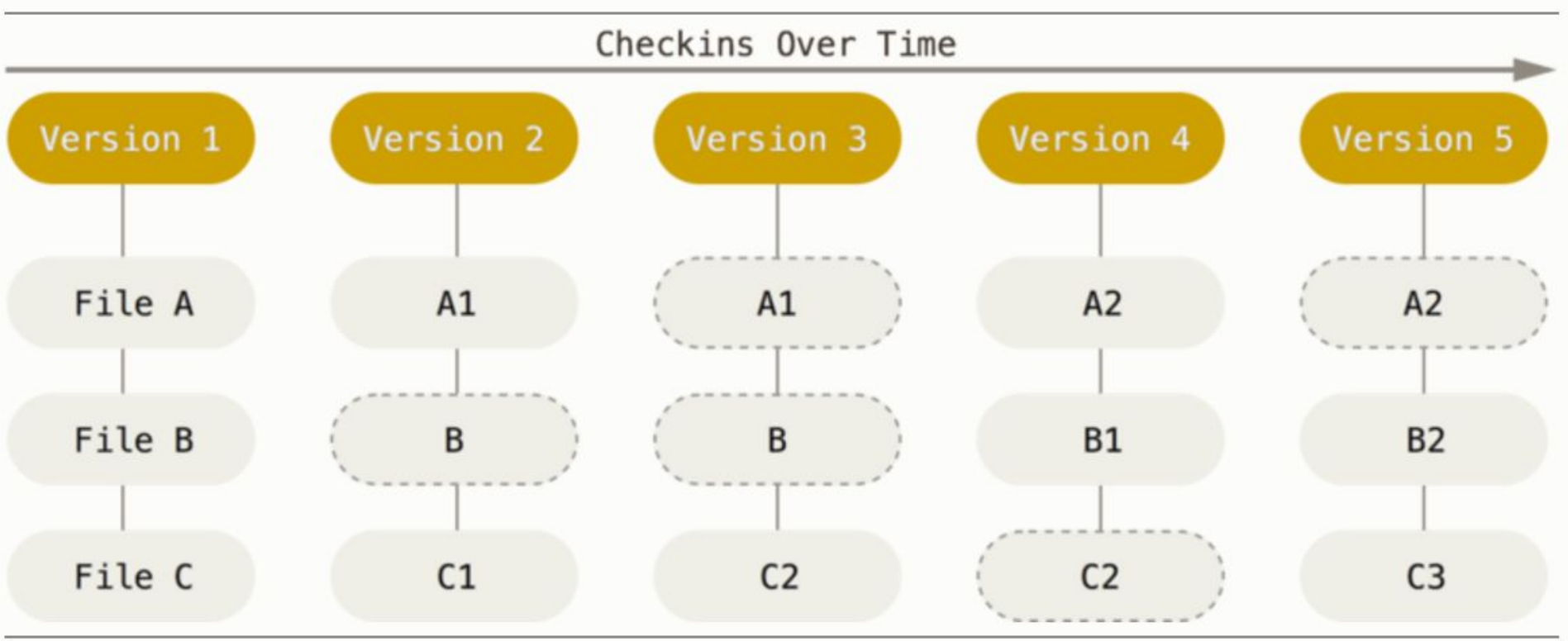


Figure 1-5. Storing data as snapshots of the project over time.

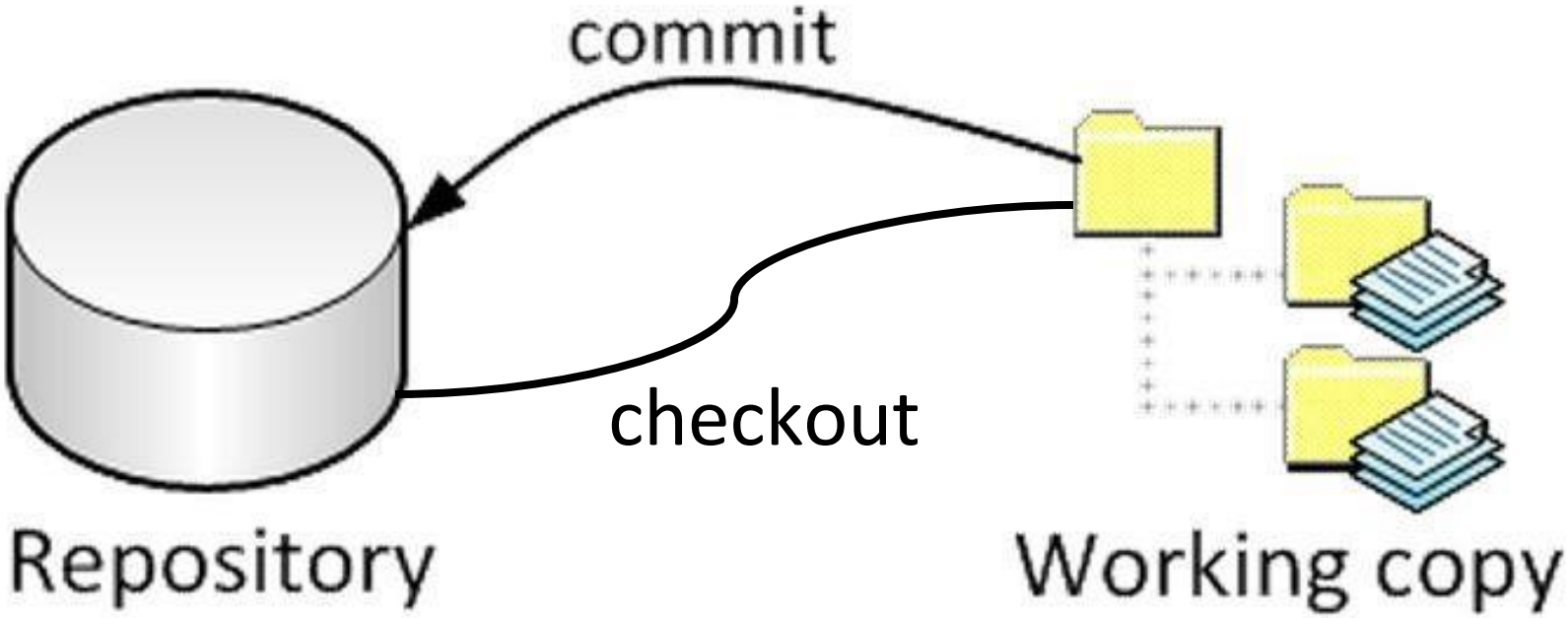
Cite: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Git source control

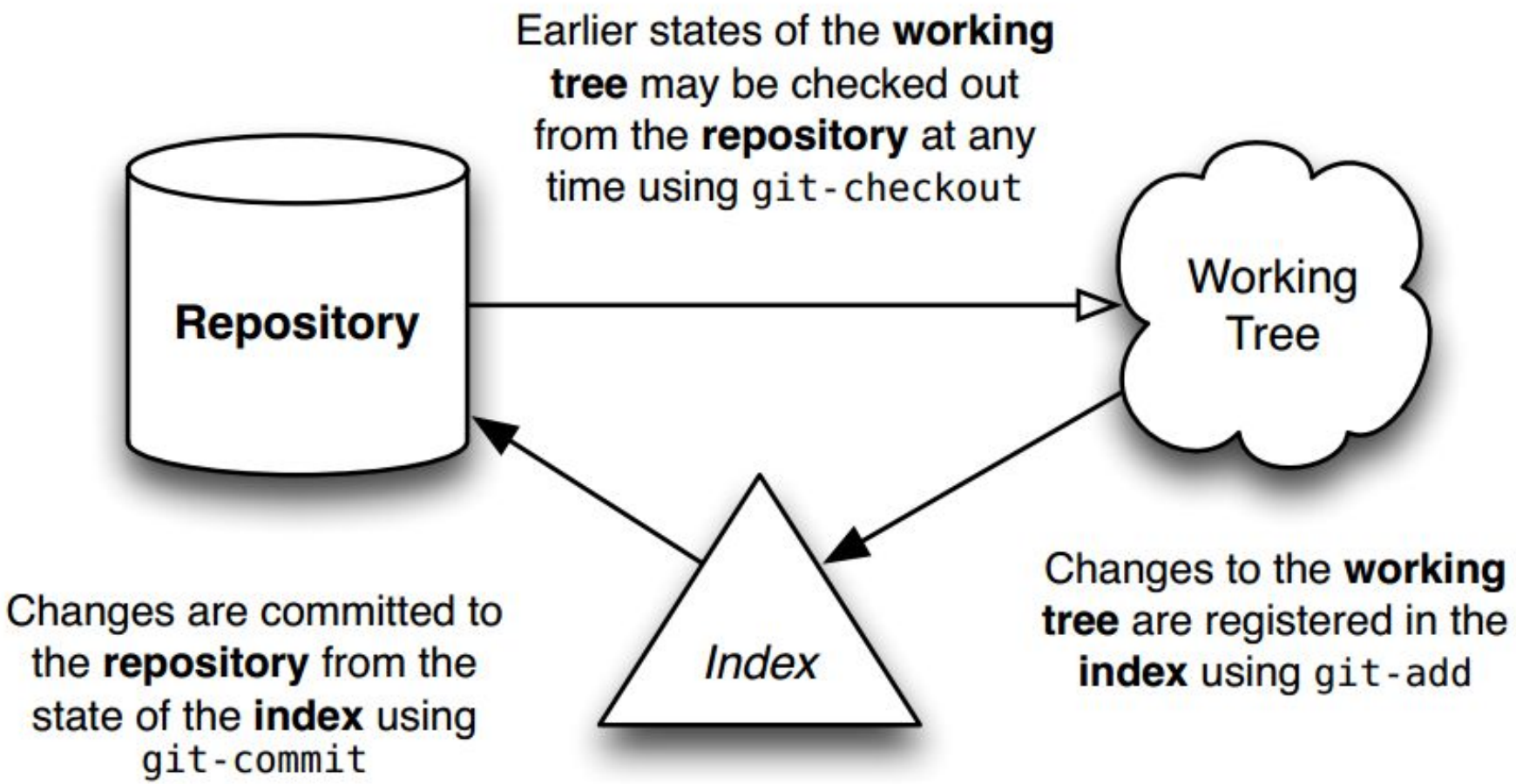
Terminology

- **Repository**
  - Files and folders related to the software code
  - Full history of the software
- **Working copy**
  - Copy of software's files in the repository
- **Check-out**
  - To create a working copy of the repository
- **Check-in/Commit**
  - Write the changes made in the working copy to the repository
  - Commits are recorded by the SCS

Big Picture in local machine



Git Local Workflow



Git States

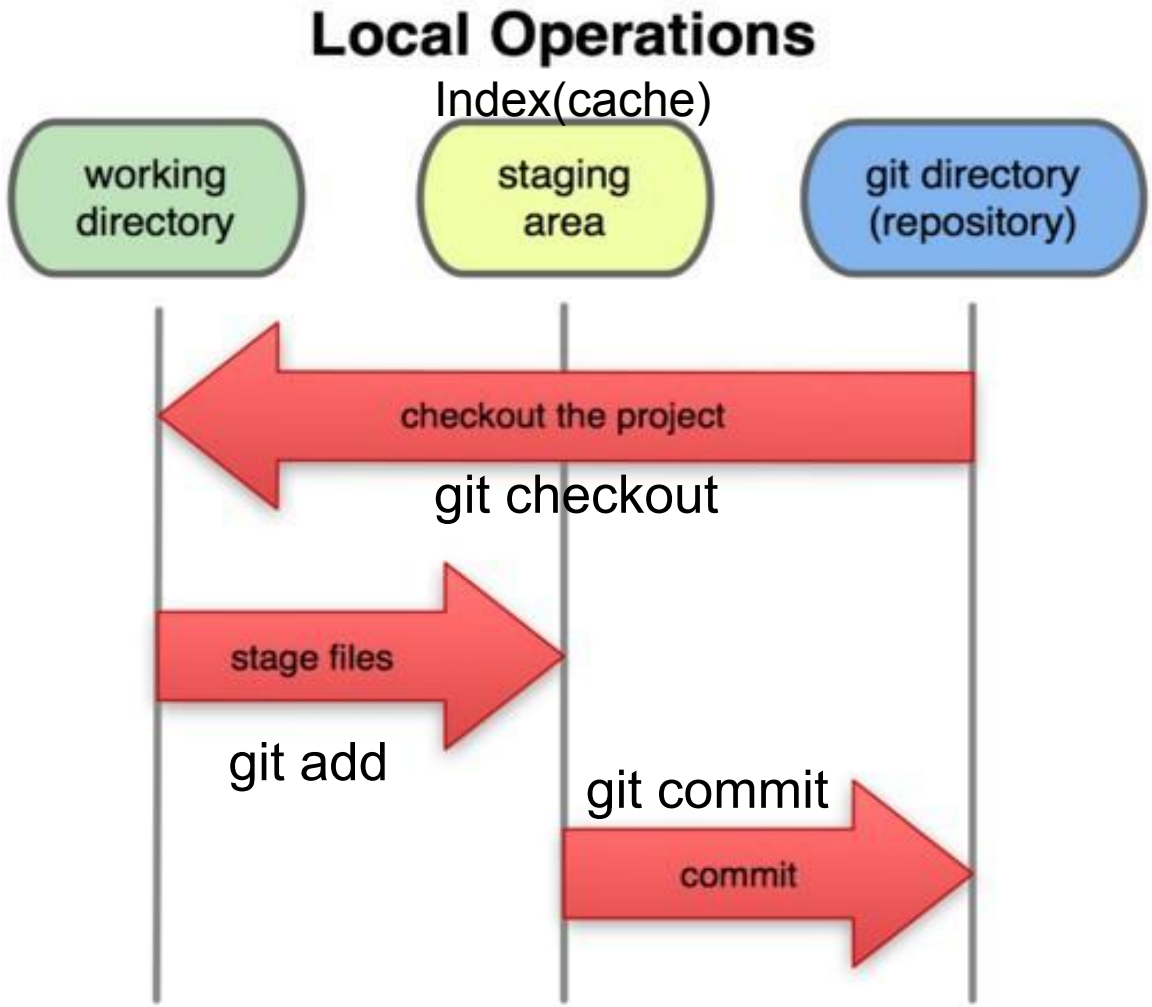


Image Source: git-scm.com

First Git Repository

- `$ mkdir gitroot`
- `$ cd gitroot`
- `$ git init`
  - creates an empty git repo (.git directory with all necessary subdirectories)
- `$ echo "Hello World" > hello.txt`
- `$ git add .`
  - Adds content to the index
  - Must be run prior to a commit
- `$ git commit –m 'Check in number one'`

Git commands

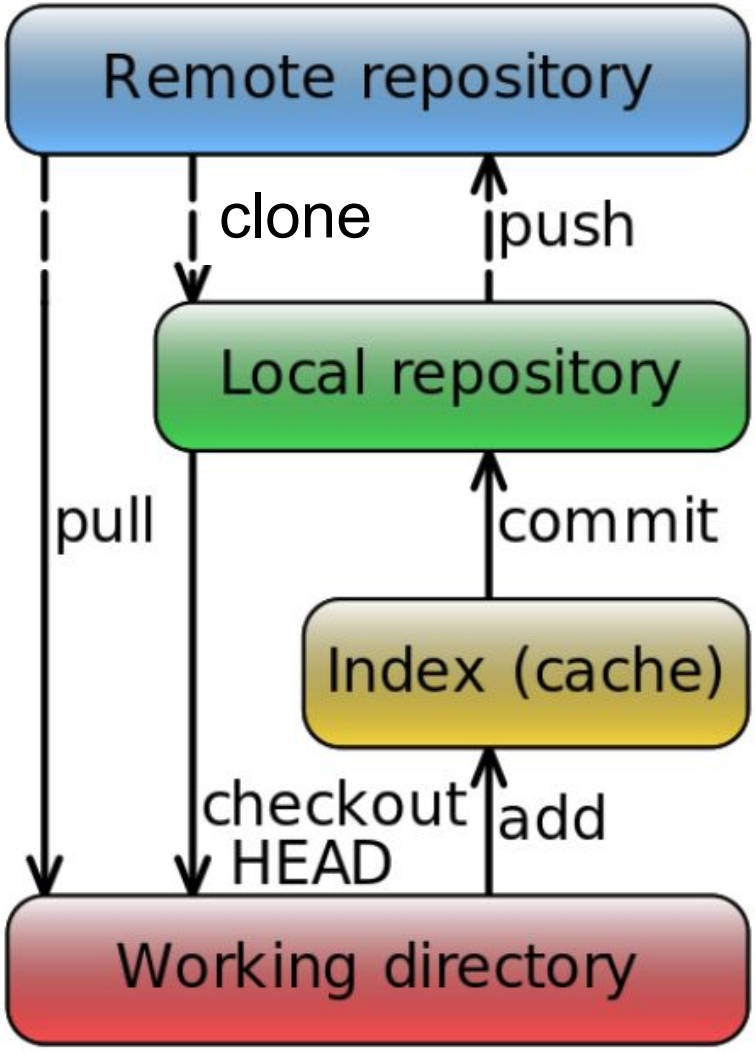
- Repository creation
  - **git init** (start a new repository)
  - **git clone** (create a copy of an existing repository)
- Branching
  - **git checkout <tag/commit> -b <new_branch_name>**

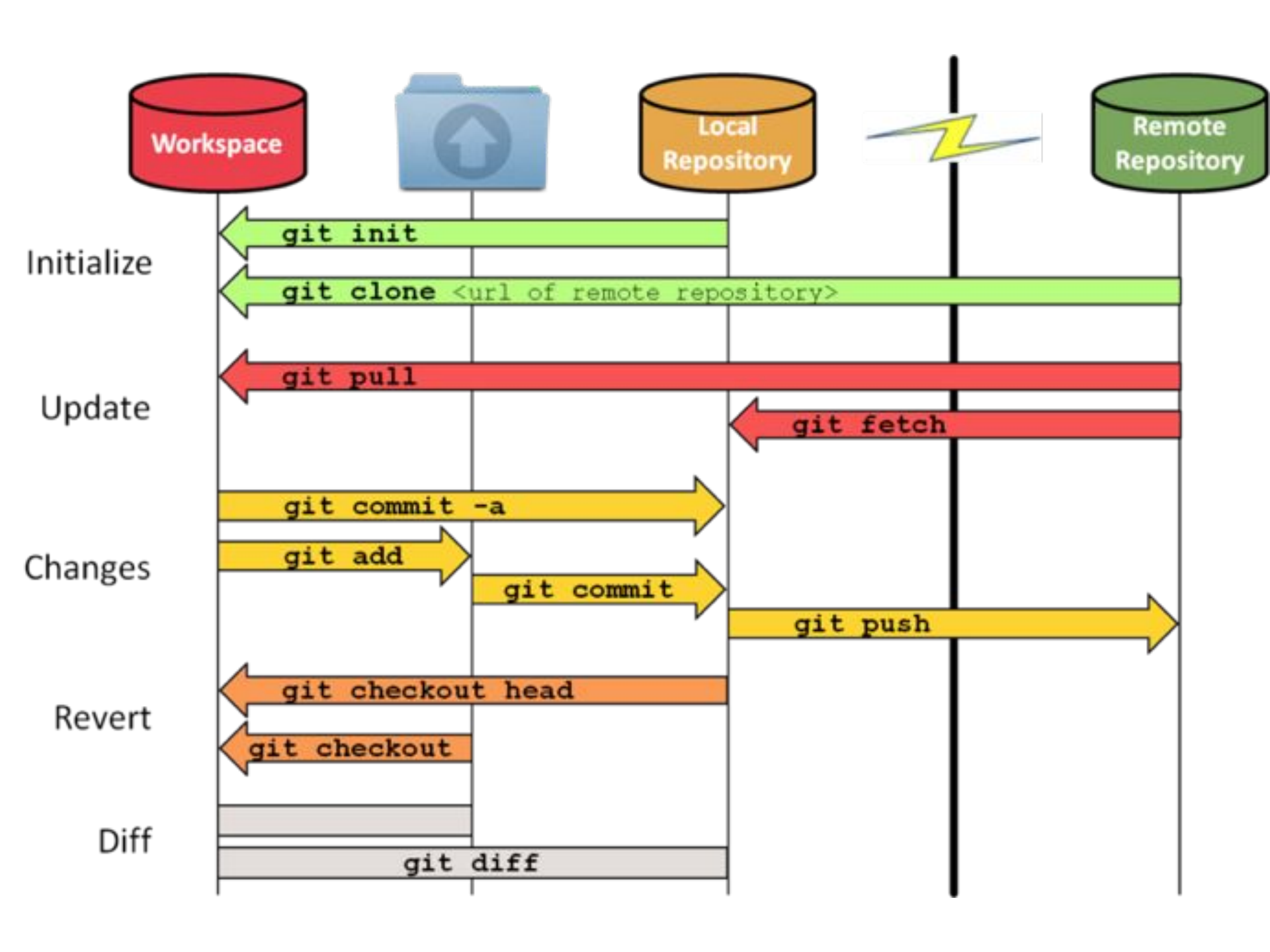
    (creates a new branch)
- Commits
  - **git add** (stage modified files)
  - **git commit** (check-in changes to the repository)
- Getting info
  - **git status** (shows modified files, new files, etc)
  - **git diff** (compares working copy with staged files)
  - **git log** (shows history of commits)
  - **git show** (show a certain object in the repository)
- Getting help
  - **git help**

Working With Git

- `$ echo "I love Git" >> hello.txt`
- `$ git status`
  - Shows list of modified files
  - hello.txt
- `$ git diff`
  - Shows changes we made compared to index
- `$ git add hello.txt`
- `$ git diff`
  - No changes shown as diff compares to the index
- `$ git diff HEAD`
  - Now we can see changes in working version
- `$git commit –m 'Second commit'`

Overview from whole system




Workspace
Local Repository
Remote Repository
Initialize
git init
git clone <url of remote repository>
Update
git pull
git fetch
Changes
git commit -a
git add
git commit
git push
Revert
git checkout head
git checkout
Diff
git diff

Second Git repository

- Git clone <repo>
- initializes a .git directory inside it
- pulls down all the data for that repository
- checks out a working copy of the latest version

- Try the example in the Lab

Lab 4

- GNU Diffutils uses " ` " in diagnostics
  - Example: diff . –
  - Output: diff: cannot compare - to a directory
  - Want to use apostrophes only
- Diffutils maintainers have a patch for this problem

  maint: quote 'like this' or "like this", not `like this'
- Problem: You are using Diffutils version 3.0, and the patch is for a newer version

Lab 4

- Task: Fix an issue with the diff diagnostic
- Crucial Steps: first create a new work directory
  - (4) Generate a patch
    - First get the hash in the log file
    - Then use git format-patch -1 [hash code] \ --stdout > [the patch file]
  - (9) learn the detailed usage of vc-diff and vc-revert
  - (10) consider changing ` to ‘

Useful Links

- Git tutorial
- Git Beginner's Tutorial with testing terminal
- Git Cheat Sheet
- Git from the bottom up
- Putty X11 forwarding

  (you'll need this for gitk)
- Use gitk to understand git

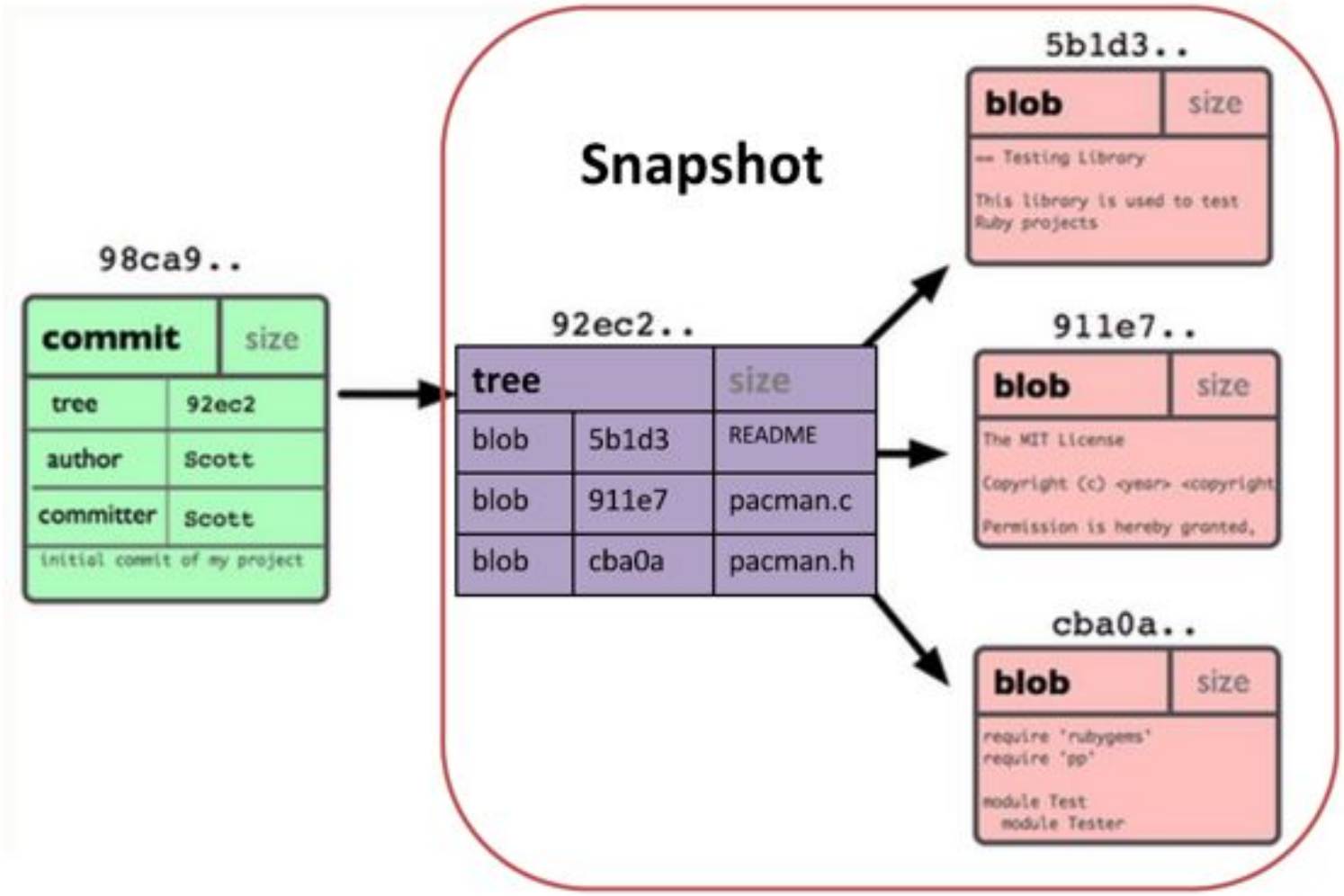
Change Management

Week 5

Git Repository Objects

- Objects used by Git to implement source control
  - **Blobs**
    - Sequence of bytes
  - **Trees**
    - Groups blobs/trees together
  - **Commit**
    - Refers to a particular “git commit”
    - Contains all information about the commit
  - **Tags**
    - A named commit object for convenience (e.g. versions of software)
- Objects uniquely identified with **hashes**

Git Repo Structure



Basic branching

- Git commit will create a commit object

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

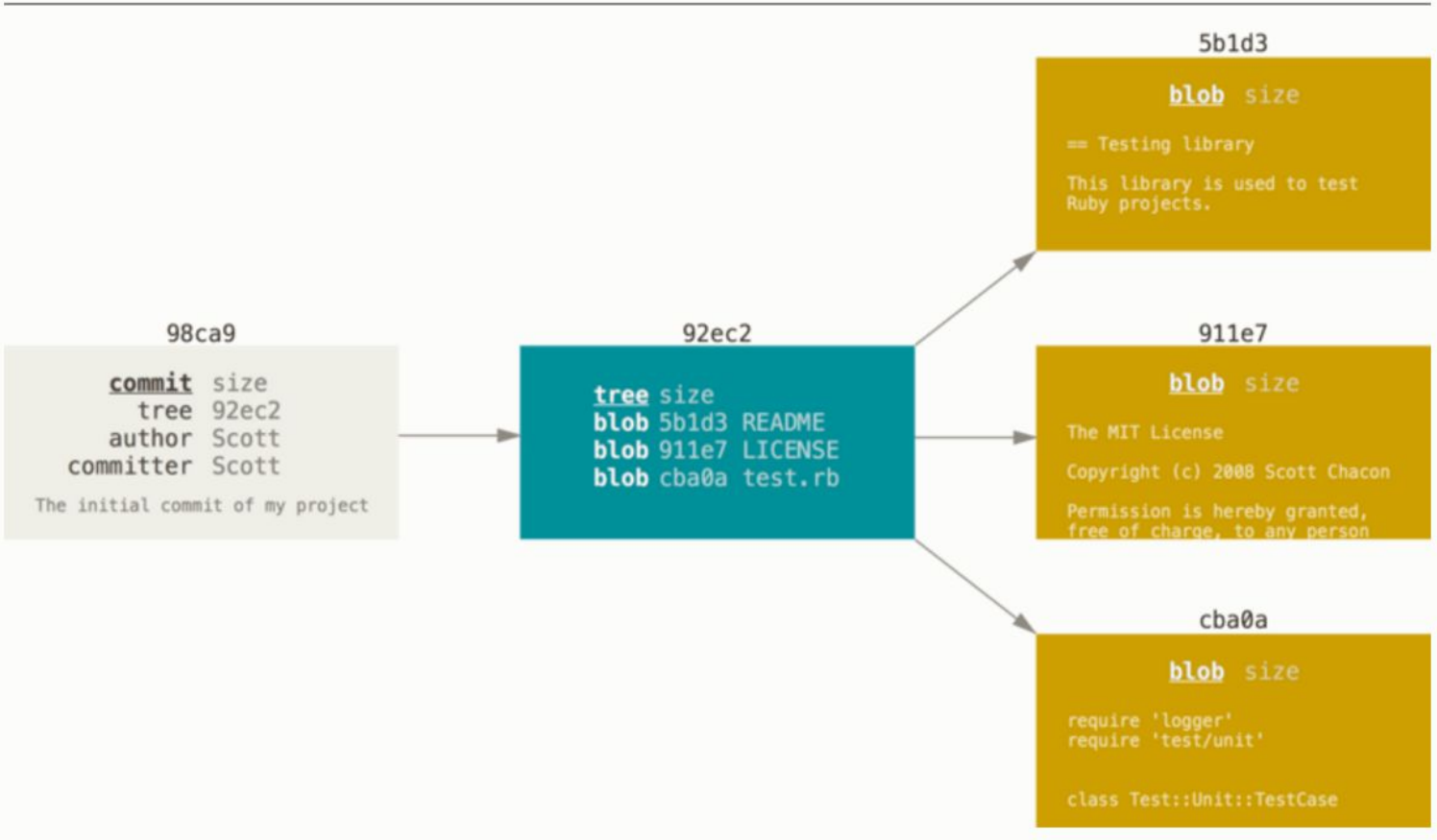


Figure 3-1. A commit and its tree

- While you have more commits

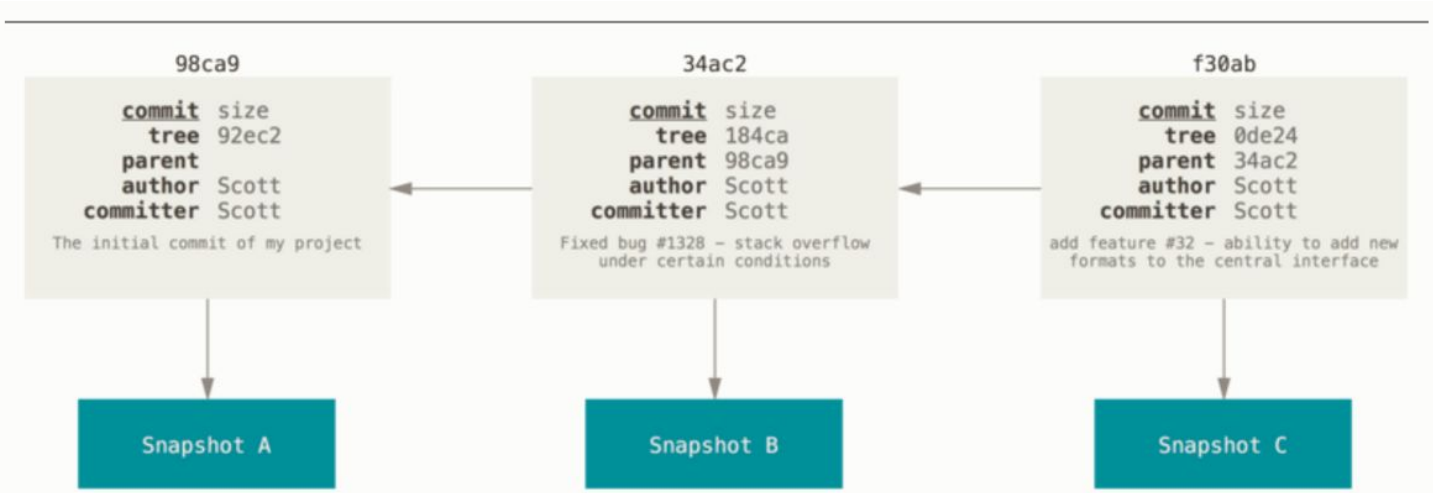
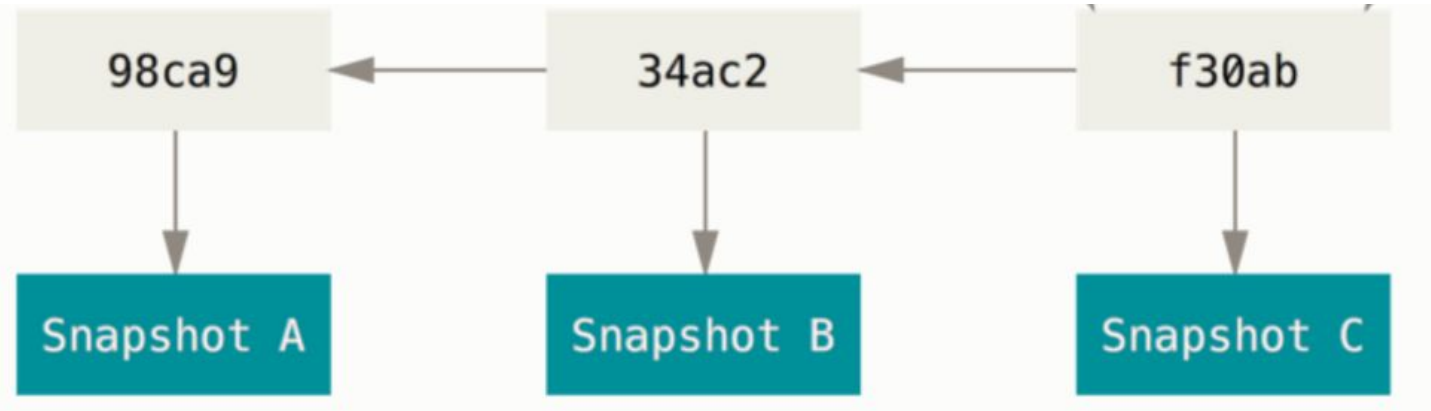


Figure 3-2. Commits and their parents



- branch: a lightweight movable pointer to one of these commits and all ancestor commit
- Master branch: the default branch name in Git. Every time you commit, it moves forward automatically.
- To create/delete a new branch

  git branch <new_branchname>

  git branch -d <branch_name>

```
$ git branch testing
```

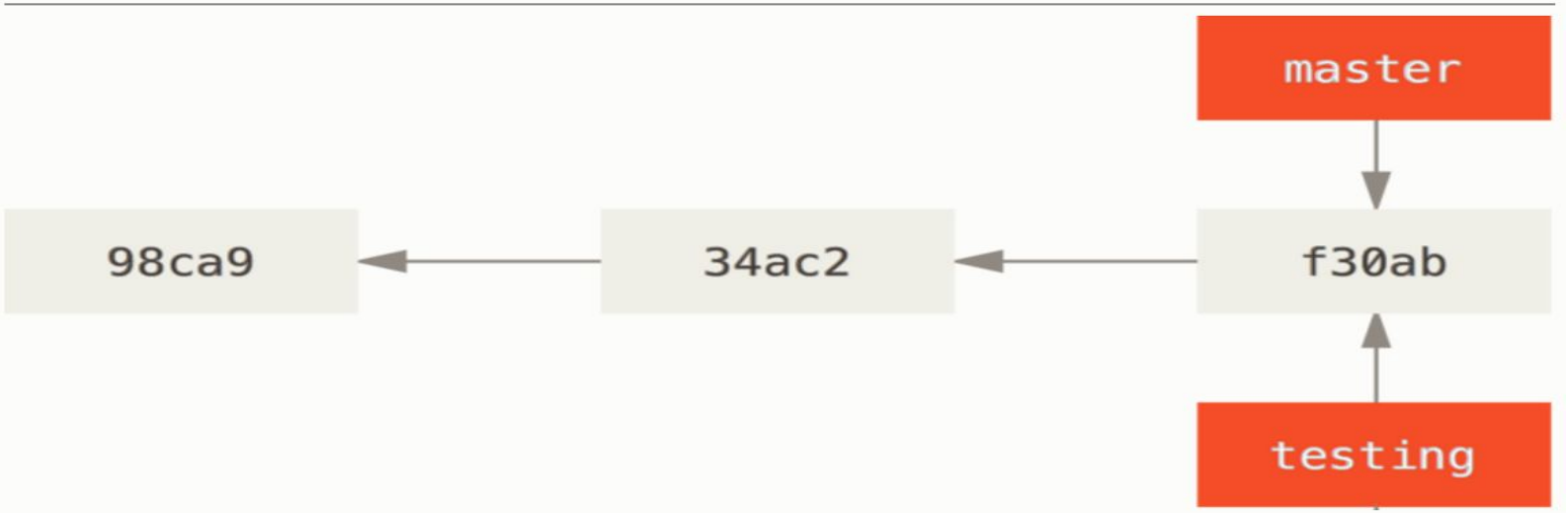
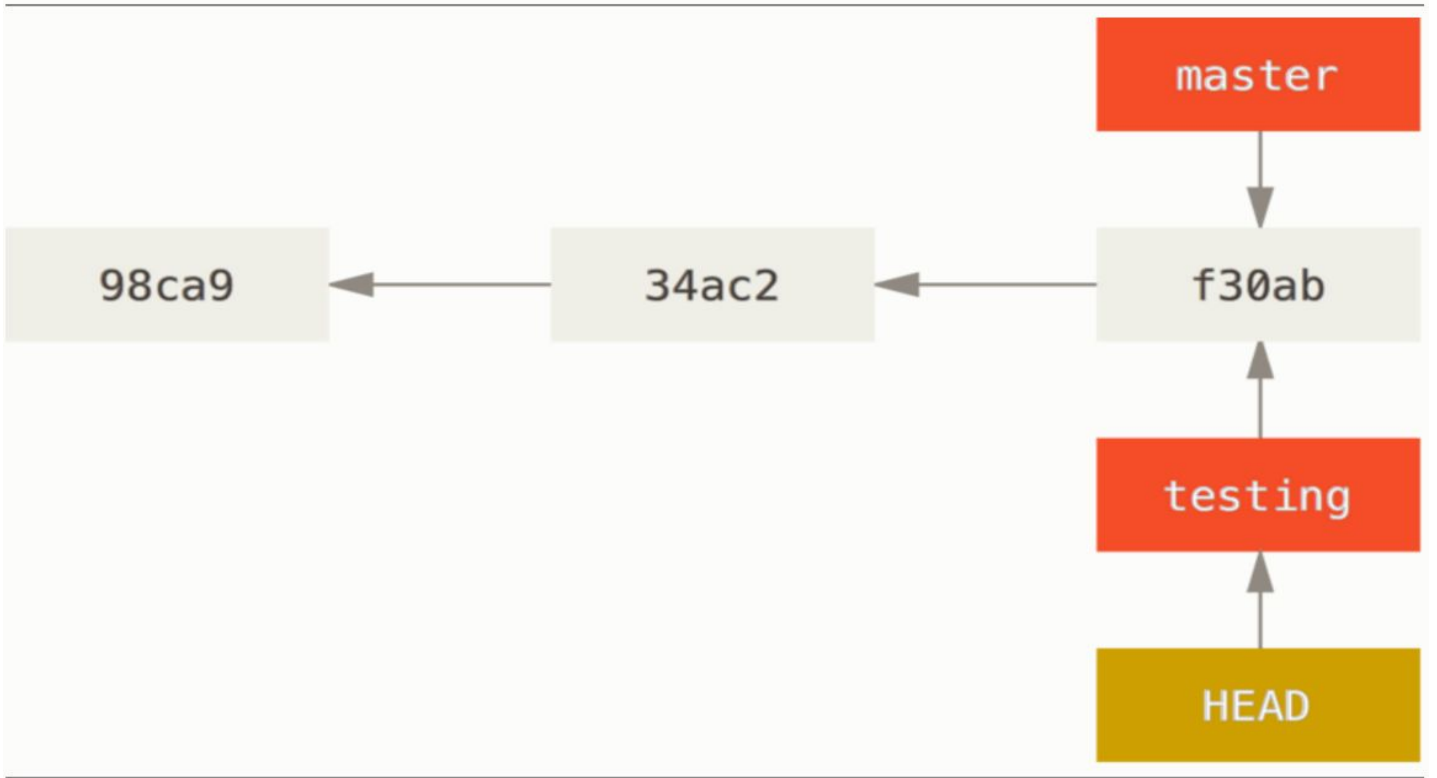


Figure 3-4. Two branches pointing into the same series of commits

- HEAD: Git records the branch you are currently working on by a pointer HEAD
- git branch will only create a new branch, but will not switch branch
- To switch to another branch

  git checkout <branch_name>

```
$ git checkout testing
```



- To create a branch and switch to it

  git checkout -b <branch_name>

Q1: What will the commit graph be like after running the following command?

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

```
$ git checkout master
```

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

You can see the whole history by

`git log --oneline --decorate --graph --all`

Q2: Why do we need branching?

Why Branching?

- Experiment with code without affecting main branch
- Separate projects that once had a common code base
- Two versions of the project

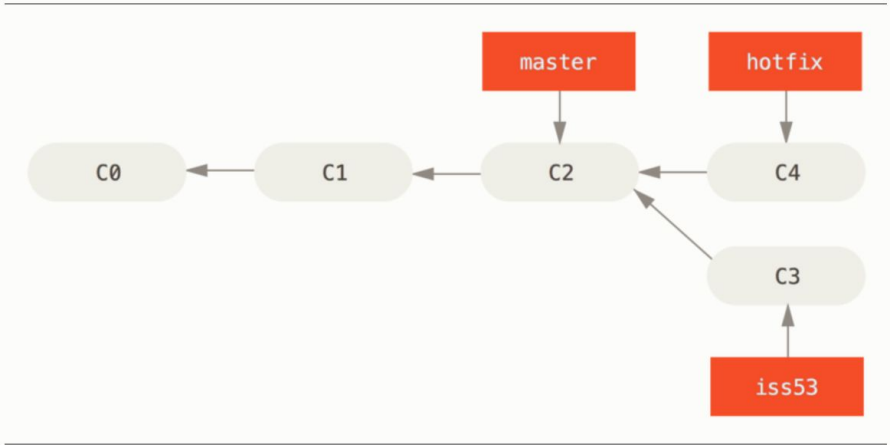
Basic Merging

- To merge another branch into current branch

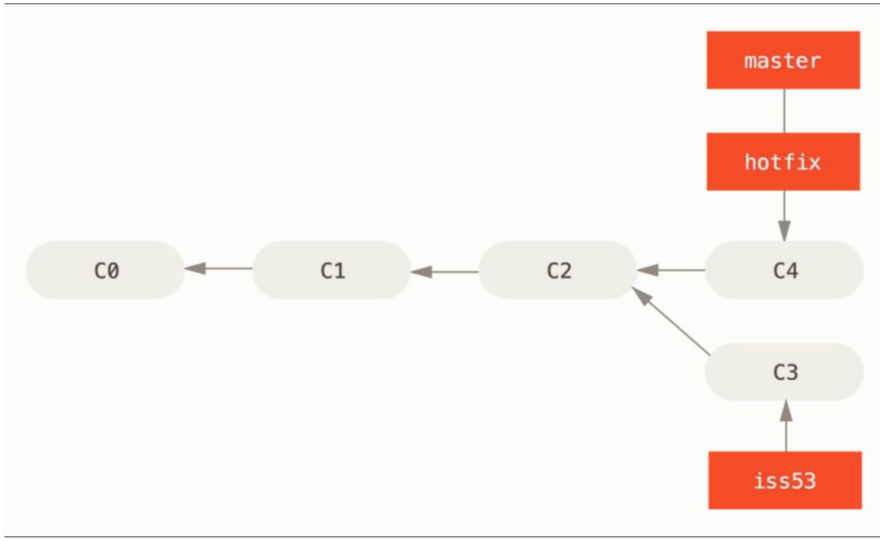
  git merge <branch_name>
- Two types
  - fast-forward
  - Three-way merge

Fast-forward

- merge one commit with a commit that can be reached by following the first commit's history
- Git merge will simply move the pointer forward




```
$ git checkout master
$ git merge hotfix
```



Three-way merge

- Three parties: two snapshots pointed to by the branch tips and the common ancestor of the two
- Git create a new snapshot from the merge and automatically create a new commit pointing to it
- Git will find the appropriate common ancestor automatically

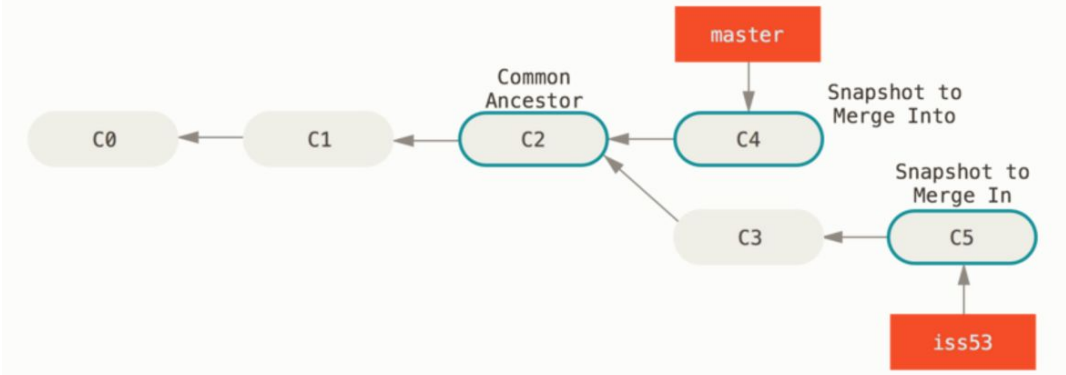


Figure 3-16. Three snapshots used in a typical merge

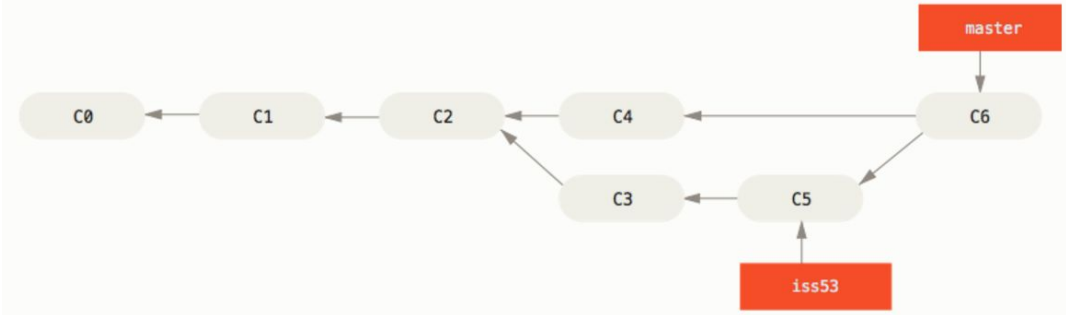


Figure 3-17. A merge commit

Basic merge conflict

- Usually git will do merge automatically
- Conflict arises when you changed the same part of the same file differently in the two branches you're merging together
- The new commit object will not be created
- You need to resolve conflicts manually

Try to merge iss53 to current branch

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Check conflicts using git status

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

- Git adds standard conflict-resolution markers to the files that have conflicts
- you can open them manually and resolve those 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
```

Manually process the conflicting lines

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

- Check whether conflicts have been resolved

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

- Run git commit to submit the changes

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   index.html
#
```

Hints for Assignment 4

Cite Jin Wang

Homework 4

- Publish the patch made in lab 4
- Create a new branch “quote” of version 3.0
  - $ git checkout v3.0 –b quote
- Use patch from lab 4 to modify this branch
  - $ patch –pnum < quote-3.0-patch.txt
- Modify the change log in *diffutils* directory
  - Add entry for your changes into those in the change log

Homework 4

- Commit changes to the new branch

  $ git add . *$ git commit –F [change log file]*

- Generate a patch that other people can use to get your changes

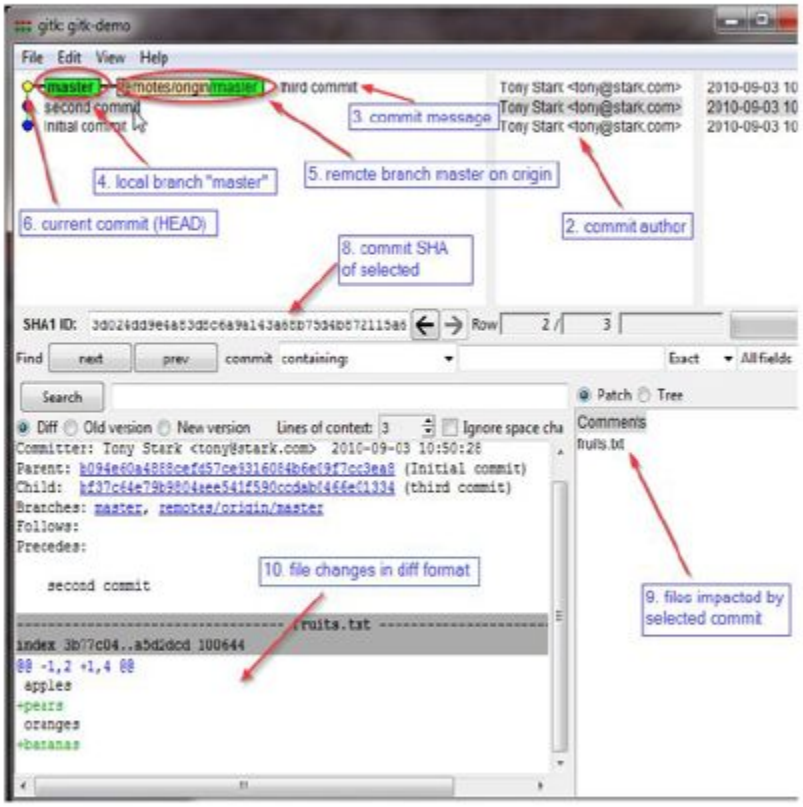
  $ git format-patch -[num] –stdout > [patch file]

- Test your partners patch
  - Choose a partner from this class
  - Apply patch with command *git am*
  - Build and test with command *make check*

Homework 4 -- Gitk

- A repository browser
  - Visualize commit graphs
  - Understand the structure of repo
  - Tutorial: [Use gitk to understand git]

  See supplement materials



Homework 4 -- Gitk

- Usage
  - ssh –X for linux and MacOS
  - Select “X11” option if using putty (Windows)
  - See supplement materials [Putty X11 forwarding]
- Run gitk in the ~/eggert/src/gnu/emacs directory
  - Need first update your path
    export PATH=/usr/local/cs/bin:$PATH
  - Run X locally before running gitk
    Xming on Windows