Change Management

Lab 4

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”

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

- Big project with multiple files
  - Bug fix required changing multiple files
  - Bug fix didn’t work
  - How to find the problem
- Figure out which parts changed (diff)
- Communicate changes with team (patch)

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

How Do We Manage Changes?

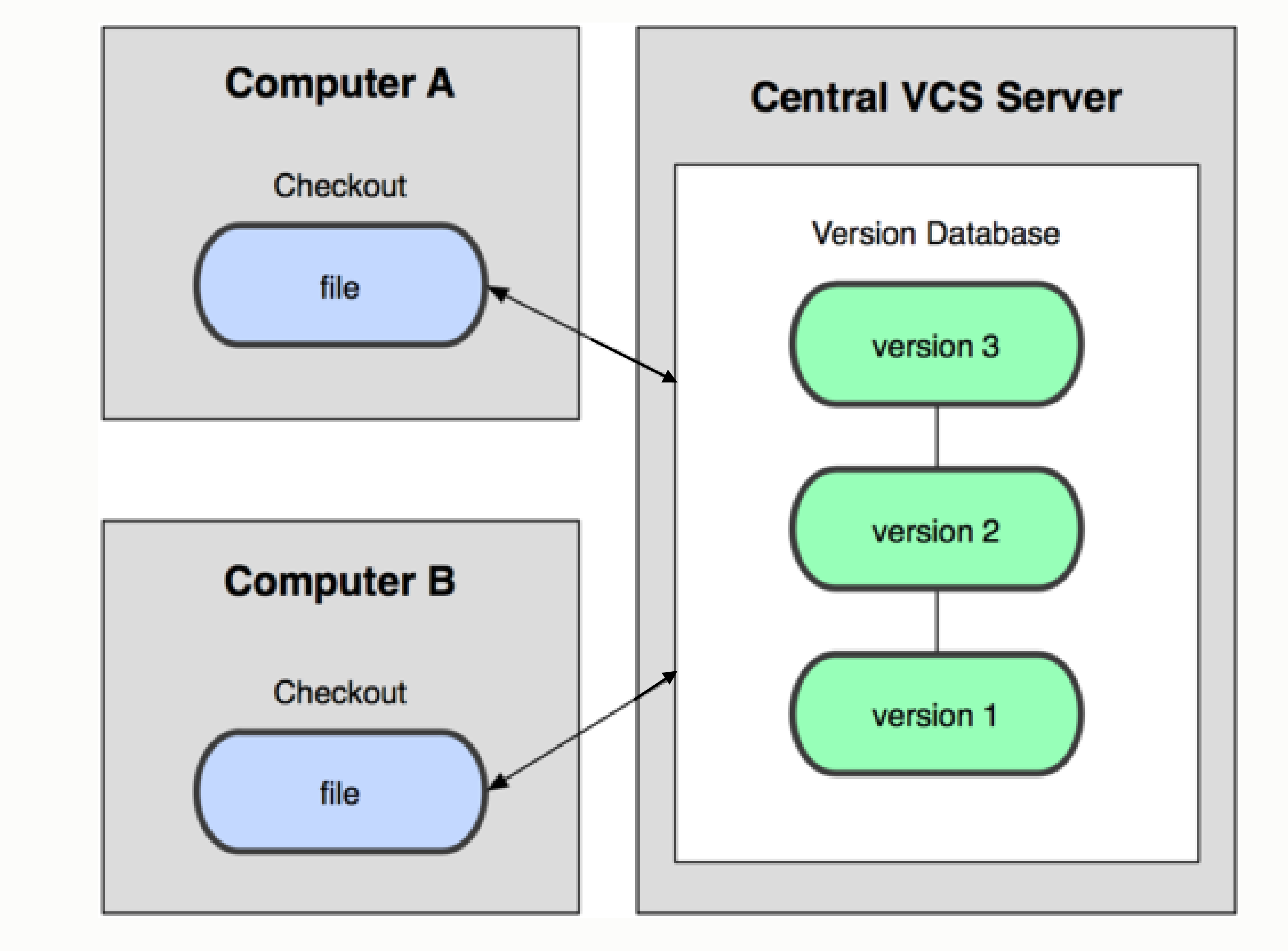
- **Version Control Tools**: track and control changes to files
  - What changes were made?
  - When were changes made?
  - Who made the changes?
  - Revert back to a previous version
- Popular Tools: Subversion, Git, Bazaar

Centralized vs. Distributed VCS

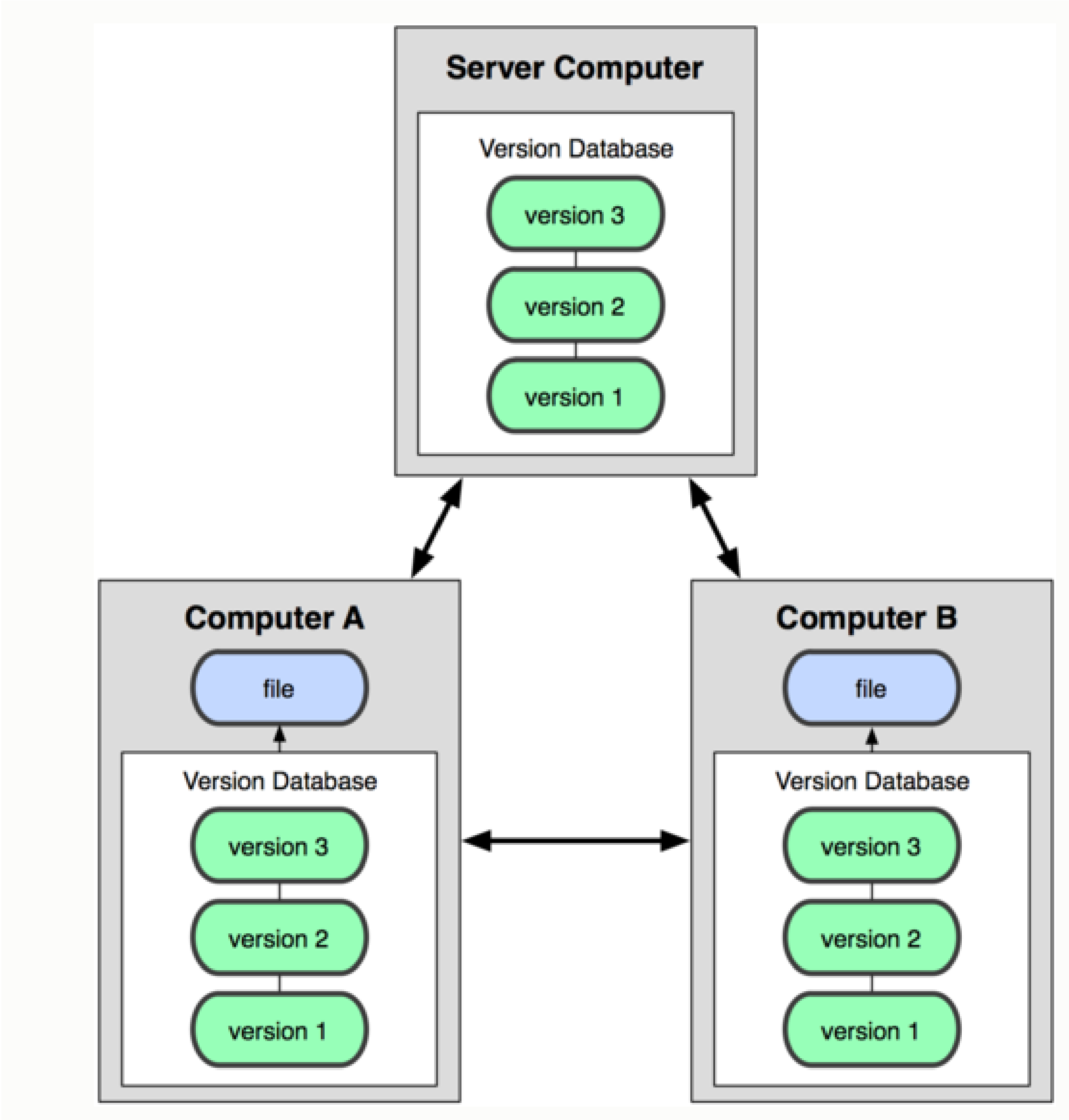
- Single central copy of the project history on a server
- Changes are uploaded to the server
- Other programmers can get changes from the server
- Examples: SVN, CVS

- 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

Centralized



Distributed



Centralized: Pros and Cons

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

<u>Pros</u>

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

<u>Cons</u>

- Single point of failure (no backups!)

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

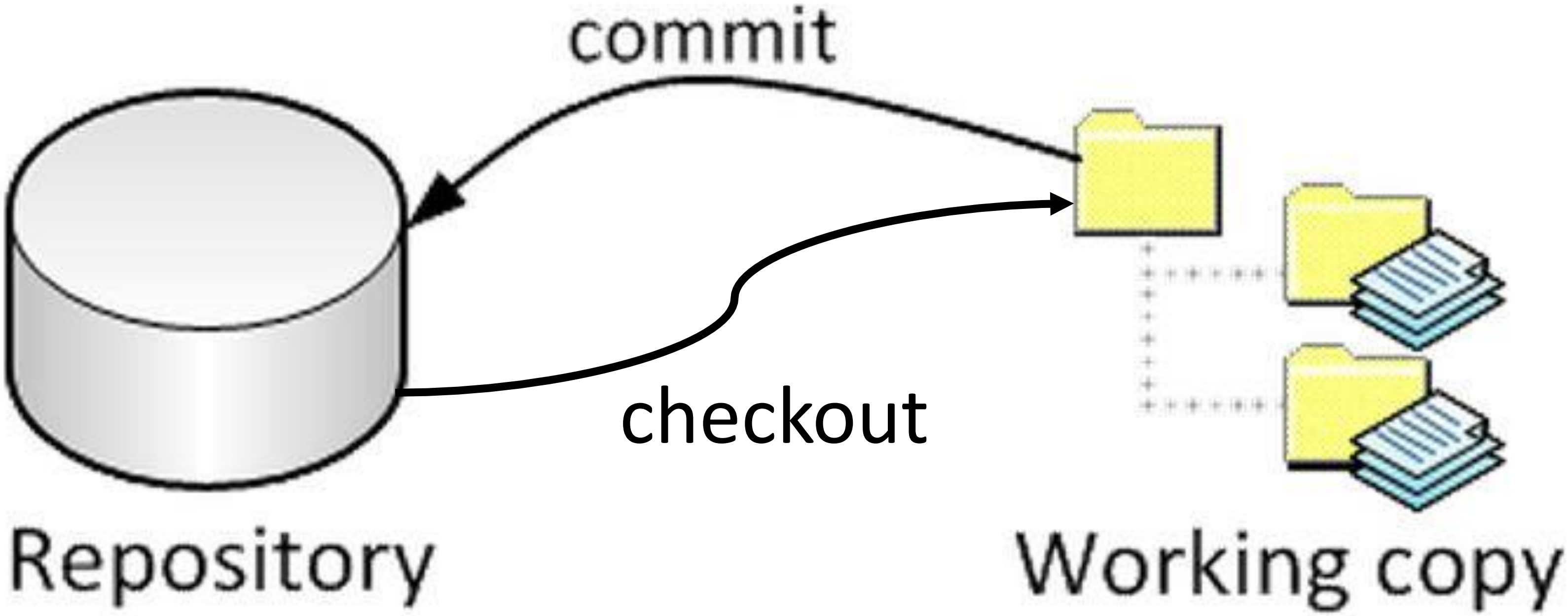
Version Control Terminology

- **Repository**
  - A database usually stored on a server that contains:
    - A set of files and directories
    - The full history and different versions of a project
- **Working Copy**
  - A local copy of files from a repository at a specific time or revision

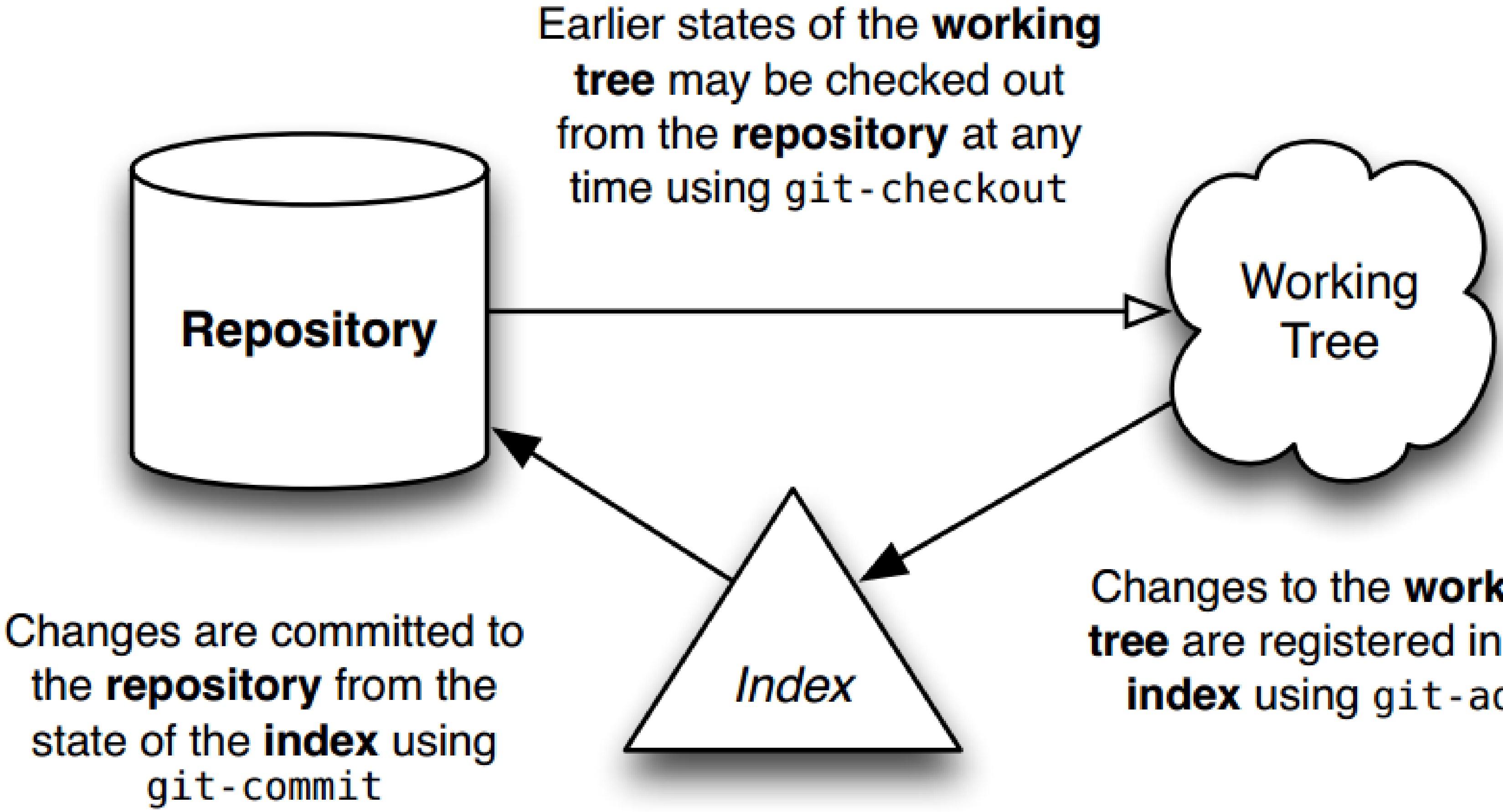
More Terminology

- **Check-Out**
  - The act of creating a local working copy from the repository
- **Commit**
  - The action of writing the changes made in the working copy back to the repository

Big Picture



Git Workflow



Earlier states of the **working tree** may be checked out from the **repository** at any time using `git-checkout`

Changes to the **working tree** are registered in the **index** using `git-add`

Changes are committed to the **repository** from the state of the **index** using `git-commit`

Git Architecture

- git has 4 object types to implement source control:
  - **Blobs**
    - Like filesystem files (sequence of bytes)
    - Stored in .git/objects

I am a **git blob**

  - **Trees**
    - Like filesystem directories
    - Can include other git trees or blobs

I am a **git tree**

  - **Commits**
    - Created when “git commit” is executed
    - Points to the top-level tree of the project at the point of commit
    - Contains name of committer, time of commit and hash of current tree

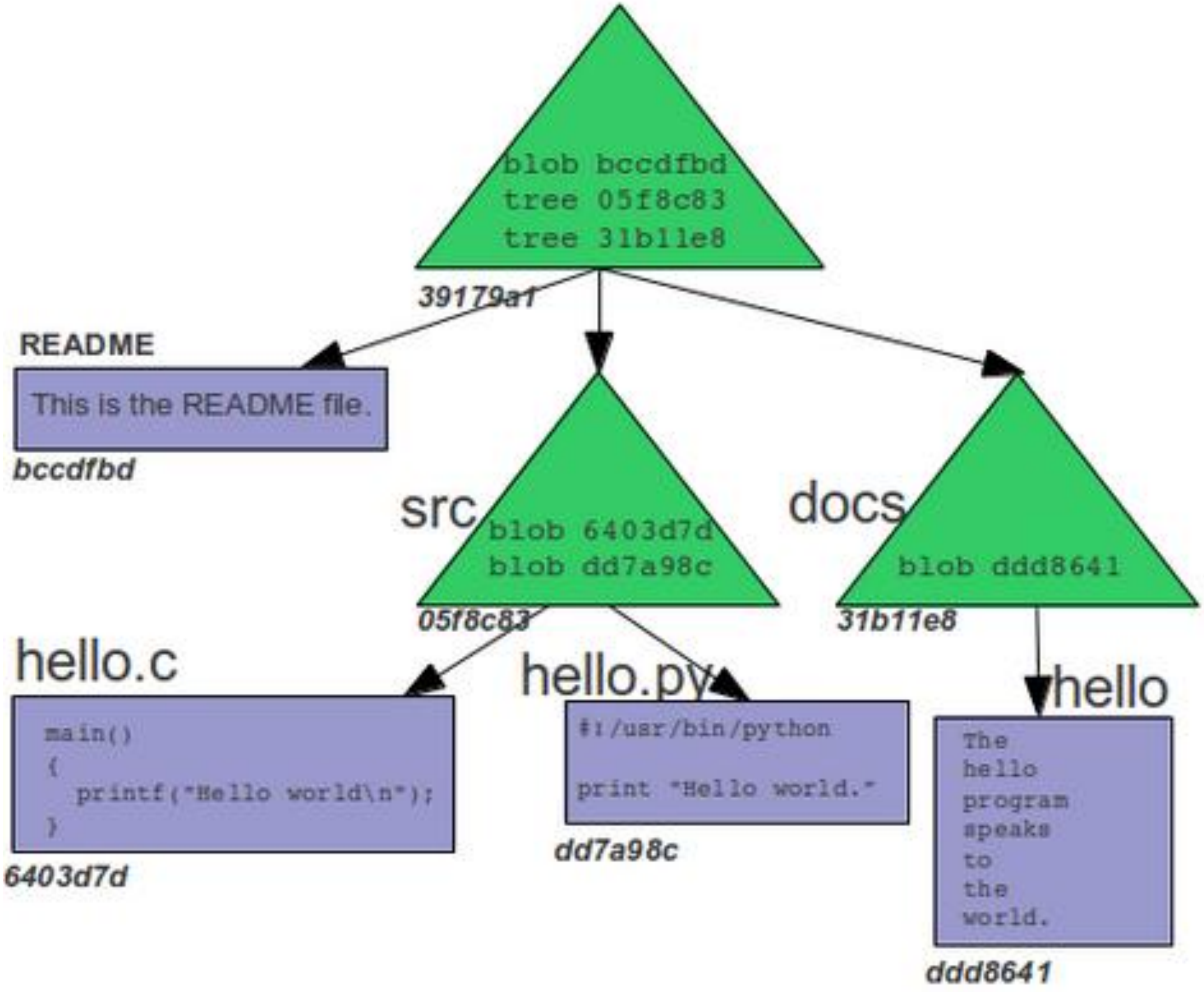
I am a **git commit**

  - **Tags**
    - Give names to commit objects for convenience
    - Include tag name, the commit referred to, tag message, tagger info

I am a **git tag**

Objects uniquely identified with hashes

Git Object Store w/o Commits



```
blob bccdfbd
tree 05f8c83
tree 31b11e8
```

This is the README file.

```
blob 6403d7d
blob dd7a98c
```

```
blob ddd8641
```

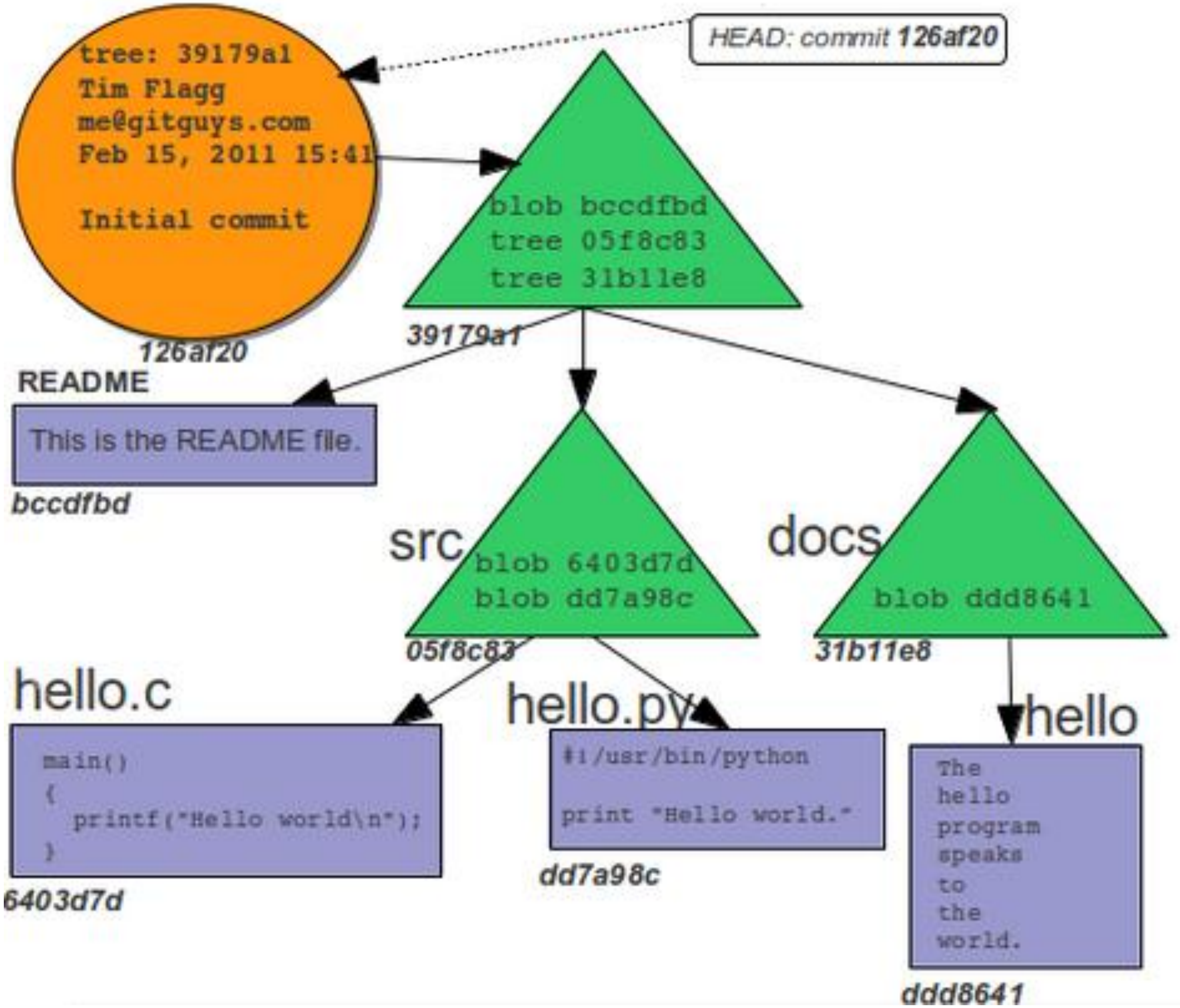
```
main()
{
  printf("Hello world\n");
}
```

```
#!/usr/bin/python

print "Hello world."
```

```
The
hello
program
speaks
to
the
world.
```

Git Object Store w/ First Commit

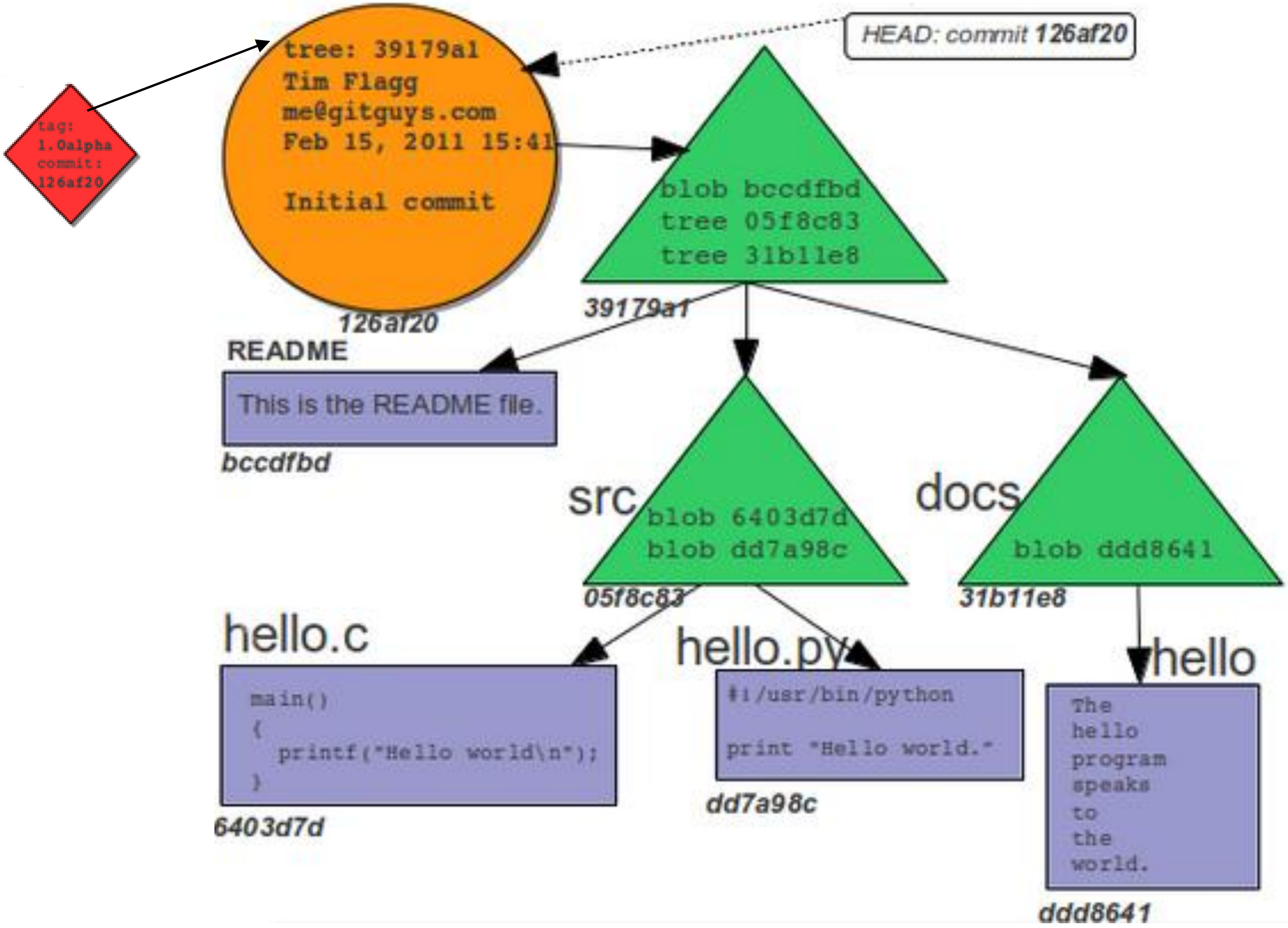


```
main()
{
  printf("Hello world\n");
}
```

```
#!/usr/bin/python

print "Hello world."
```

Git Object Store w/ First Commit



This is the README file.

```
main()
{
  printf("Hello world\n");
}
```

```
#!/usr/bin/python

print "Hello world."
```

The
hello
program
speaks
to
the
world.

Git Architecture cont'd

- **head**
  - A reference to a commit object
  - A repository can contain any number of heads
- **HEAD**
  - Refers exclusively to the currently active (checked out) head
- **Branch**
  - Refers to a head and the entire history of ancestor commits preceding that head
- **Master**
  - The default head name that git creates when the repo is first created. It refers to the default branch created in the repository.

Some git Commands

- **Getting a Repository**
  - git init //creates new repository
  - git clone //gets a copy of an existing repository
- **Commits**
  - git add //adds files to the index
  - git commit // changes are added to the repo
- **Getting information**
  - git help, git status, git log, git show, git diff

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’`

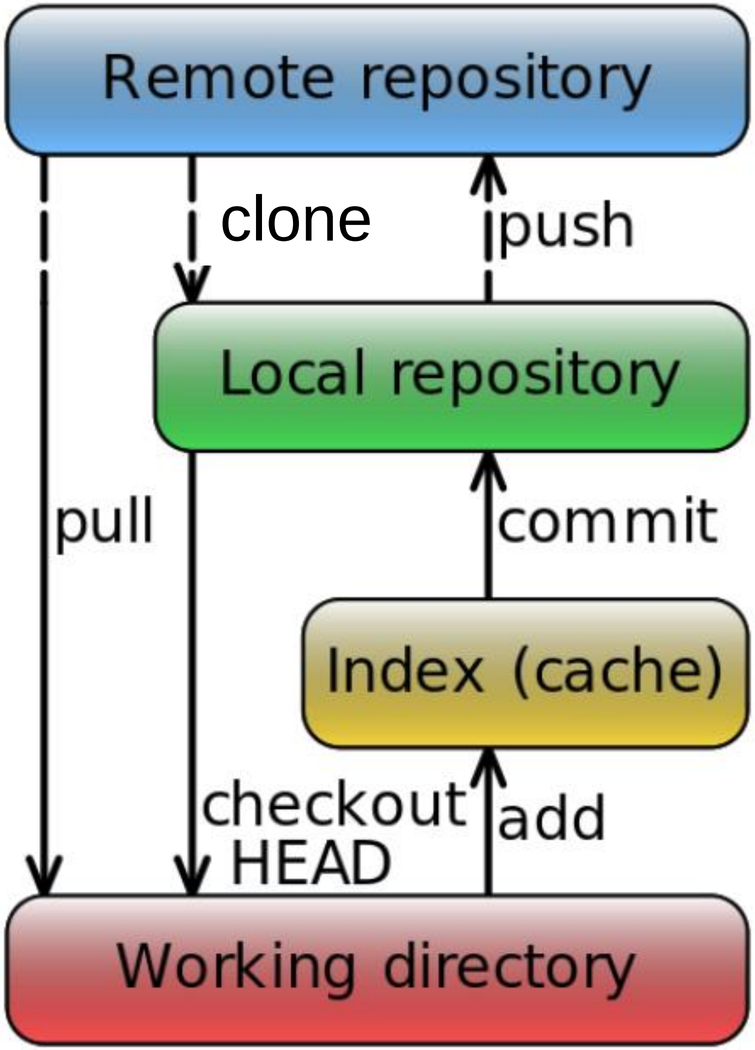
Working With Git

- `$ 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’`

Undoing What Is Done

- **git checkout**
  - Used to checkout a specific version/branch of the tree
- **git revert**
  - Reverts a commit
  - Does not delete the commit object, just applies a patch
  - Reverts can themselves be reverted!
- **Git never deletes a commit object**
  - It is very hard to lose data

Overview


Remote repository
clone
push
Local repository
pull
commit
Index (cache)
checkout
HEAD
add
Working directory

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 called “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

Backporting

Taking a certain software modification (patch) and **applying it to an older version** of the software than it was initially created for.

Steps

1) Installing Git
   - Ubuntu: $ sudo apt-get install git
   - SEASnet
     - Git is installed in /usr/local/cs/bin
     - Add it to PATH variable or use whole path
2) Make a directory ‘gitroot’ and get a copy of the Diffutils Git repository
     - $ mkdir gitroot
     - $ cd gitroot
     - $ git clone git://git.savannah.gnu.org/diffutils.git
3) Follow steps in lab and use man git to find commands

Useful Links

- Git Tutorial
  - By topic
- Git Beginner’s Tutorial
  - Step by step tutorial + testing terminal
- Git Visual Guide
  - For visualizing what each command does
- Git From The Bottom Up
  - For understanding how Git is structured and the details of how it tracks changes