1 Lab 8

Date: Oct 10, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to provide you with intermediate-level knowledge of git. After completing this lab, you should have been exposed to the following topics:

- Setting up git repositories.
- Basic git commands.
- Creating branches for independent feature development.
- Merging changes.

1.2 Background

The git homepage states: "Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency." Let's focus on the phrase "distributed version control system":

Version Control System (VCS) From atlassian: Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

One of the earliest version control systems was SCCS. CVS popularized the use of version control systems. Another popular version control system was subversion.

Distributed All of the version control systems mentioned above depend on a central server hosting a master repository. In a distributed system, there is no privileged central repository and every repository has the same status as every other repository.

In normal git usage, there are 3 areas where files can live:

Working Directory The file is in the directory which is under control by git.

Local Repository The file has been comitted to the local repository which is maintained in the `.git` directory of the working directory.

Shared Repositories The file has been comitted to a repository (usually remote) which is shared among multiple developers. Though it is possible to have multiple remote repositories associated with a working directory, usually there is just one with name `origin`.

In git, a file can be in different states:

Unknown to git The file has never been added to git.

Staged The current version of the file is set up for addition to git. This is usually done using a `git add` command. It is possible to back out from this state.

Committed All changes have been committed. This is done using the `git commit` command. Every commit has a unique ID which is a hexadecimal hash of the entire history of the repository so far. Since every commit is authenticated cryptographically, it is impossible to back out of a commit to a repository.

Current VCS allow independent branches of development. It is possible to merge changes from one branch to another. In git, a branch name is merely a way of referring to a particular commit ID.

In what follows, you can get the documentation for any git command *CMD* using a web search or by doing a `man git-`*CMD*.

1.3 Lab Tasks

Since this lab involves working with git, the directions are different from the earlier labs. Create an initial `work/lab8` directory but do **not** copy over the provided files directory. Instead, simply start up a script command in your terminal:

Assuming that you are in your `i220a` or `i220b` directory:

```
$ mkdir work/lab8
$ cd work/lab8
$ script -a lab8.log
```

1.3.1 Creating and Cloning a Bare Repository

A **bare** git repository allows you to share your work with other people. In this exercise, you will create a bare git repository which you will subsequently share with yourself.

Change over to the ~/git-repos directory you created when setting up your account for this course and create a directory to hold your bare git repository for this lab.

```
$ cd ~/git-repos   #this dir should exist as per earlier setup
$ mkdir lab8.git   #traditionally, .git extension for bare repos
```

Now create the repo:

```
$ cd lab8.git
$ git init --bare
```

Now clone this bare repository into a working directory:

```
$ mkdir -p ~/tmp #create tmp dir if it does not exist
$ cd ~/tmp

# replace LOGIN with your login-id below
$ git clone \
  ssh://LOGIN@remote.cs.binghamton.edu/home/LOGIN/git-repos/lab8.git
Cloning into 'lab8'...
warning: You appear to have cloned an empty repository.
```

where `LOGIN` is your login-id. This should work without requiring a password assuming that you have set up your account as per the earlier instructions.

This should give you a warning that you have cloned an empty repository but will create a new `lab8` working directory which you can use for subsequent development. Go into the newly created directory:

```
$ cd lab8
$ git remote -v #output the URL for your remote repo
                #referred to as "origin"
```

If you do a `ls -a` (the option `-a` means *all* and is used to show path-names starting with `.` which are normally hidden), you should see a `.git` directory which is where the local git repository is maintained. If interested, you can poke around in that directory.

Create a `work/lab8` directory in the new working directory.

```
$ mkdir -p work/lab8
$ cd work/lab8
```

You will do all your work in this newly created directory.

Copy over the README.md from the files directory:

```
$ cp ~/cs220/labs/lab8/files/README.md .
$ git status #should show README.md is untracked
```

Tell git about `README.md`:

```
$ git add README.md  #add to git staging area
$ git status --short #short status should show README.md
                     #staged for adding
$ git commit -m 'added README' #should commit to local repo
$ git branch -l #should show newly created master branch
                #the * shows the current branch
```

At this point, you have your `README.md` committed to a `master` branch only in your **local** repo. You need to `push` your changes to the remote repo, telling it to create a corresponding `master` branch set up to track your local `master` branch:

```
$ git push --set-upstream origin master #will show push to origin
$ git status -s #short status should be clean
```

1.3.2 Saying Hello

Assuming that you are still in your `~/tmp/lab8/work/lab8` directory, copy over the provided Makefile and hello-one.c source file:

```
$ cp ~/cs220/labs/lab8/files/Makefile .
$ cp ~/cs220/labs/lab8/files/hello-one.c hello.c
```

Try building the program:

```
$ make
```

You will get several warnings about implicit declarations, but the compilation will succeed and you should be able to run the `hello` executable (the program will simply say hello to its single command-line argument).

```
$ ./hello
usage: ./hello NAME
$ ./hello john
hello john
$
```

We need to fix the warnings, but since the code seems to be working, let's save it to git:

```
$ git status -s
?? Makefile
?? hello
?? hello.c
$
```

The `?` above shows that git does not know anything about the 3 files. Normally files like executables and object files which are produced by other programs like compilers should not be committed to git. We can tell git to ignore specific such files by adding a `.gitignore` file:

```
cp ~/cs220/labs/lab8/files/.gitignore .
$ git status -s
?? .gitignore
?? Makefile
?? hello.c
$
```

Notice that the `hello` executable is no longer listed, but the `.gitignore` we just copied is listed.

Let's add all three files to git:

```
$ git add . #stage all untracked files in current dir
$ git status -s  #staged but not committed
A  .gitignore
A  Makefile
A  hello.c
#commit with message (option `-m`).
$ git commit -m 'working with warnings'
$ git status -s #should be clean
$ git push #push additions to remote
$ git log #show history of repo so far
```

Let's fix the warnings. The problem is that the necessary header files have not been `#included`. Add `#include`'s for `<stdio.h>` (for `printf()`) and `<stdlib.h>` (for `exit()`) to the start of of `hello.c` and rebuild:

```
$ make  #clean compile without warnings
# test
./hello sue
hello sue
$
```

Since we seem to have our program working, let's commit our changes:

```
$ git status -s
 M hello.c   #hell.c modified in working dir
# -a will add all modified files which git knows about
$ git commit -a -m 'working: clean compile'
$ git push #push to remote
```

You can show the history of commits for `hello.c`:

```
$ git log hello.c
```

You can restrict yourself to seeing the messages for only the latest `N` commits by specifying a `-N` option:

```
#show log message of only last commit
$ git log -1 hello.c
```

The program seems to be working, all tests seem to pass. However, it is always a good idea to review the program before declaring it complete. When you do so, you may notice that the usage error message is output using `printf()` which means it is output to standard output. However error messages should be output to standard error. Hence change the `printf(...)` for the usage message to `fprintf(stderr, ...)`.

```
make #clean compile
$ ./hello
usage: ./hello NAME
# we see usage message even when stdout is redirected
$ ./hello >/dev/null
usage: ./hello NAME
```

You can see the `diff`'s between what is in your working directory and what is committed:

```
$ git diff
diff --git a/hello.c b/hello.c
...
-    printf("usage: %s NAME\n", argv[0]);
+    fprintf(stderr, "usage: %s NAME\n", argv[0]);
...
$
```

The `-` line shows a line from the repository changed to the `+` line from your working directory. The lines printed before and after those lines provide context for the diff.

Let's commit and push:

```
git commit -a -m 'send usage message to stderr'
$ git push
$ git shortlog #simply show commit messages
```

1.3.3 A Feature Branch

The program is working but is limited in that it greets only a single name. It would be useful to change it to greet multiple names. However, we want to make sure that we do not loose the working program while making this change. You could make sure of that by making a backup copy of your working program, but with git it is only necessary to create a new branch to work on this new feature.

You can create a new git branch `hello-multi` to work on this feature:

```
$ git checkout -b hello-multi
```

This switches you to a new `hello-multi` branch. You can do development in this new branch independently of development in `master`. If you do an `ls` in

this new branch, it does not look any different from what you had originally, but if you list the branches you see that you now have 2 branches:

```
$ git branch -l
```

Let's develop our new feature. Fortunately, since this lab is only for git, the development has already been done for you:

```
$ cp ~/cs220/labs/lab8/files/hello-multi.c hello.c
```

Compile and test the new feature:

```
$ make
$ ./hello
usage: ./hello NAME...
$ ./hello john sue
hello john
hello sue
$
```

It seems to be working. Let's commit and push our changes:

```
$ git commit -a -m 'allow greeting multiple names'
# push changes onto tracking origin/hello-multi branch
$ git push -u origin hello-multi
```

1.3.4 Continuing Development on master

You can return to the `master` branch:

```
$ git checkout master
```

Recompile and re-run your program:

```
$ make
$ ./hello john sue
usage: ./hello NAME
$
```

Note that you have reverted to the original behavior where only a single name is accepted; the multi-name feature is gone.

You decide that you do not want to depend on the c11 feature which allows `main()` to not return a value. So you add a `return 0;` at the bottom of `main()`, resulting in the following `tail` output:

```
$ tail -3 hello.c
  printf("hello %s\n", argv[1]);
  return 0;
}
$
```

You recompile and re-test:

```
$ make
$ ./hello
usage: ./hello NAME
$ ./hello sue
hello sue
$
```

Everything seems ok, let's commit and push this change:

```
$ git commit -a -m 'added return at end of main()'
$ git push
$
```

1.4 Merging Master Changes to Feature Branch

You would like to apply the changes from `master` to the `hello-multi` branch.

```
# switch to hello-multi branch
$ git checkout hello-multi
Switched to branch 'hello-multi'
Your branch is up to date with 'origin/hello-multi'.

# apply changes from master to current hello-multi branch
$ git merge master
Auto-merging hello.c
CONFLICT (content): Merge conflict in hello.c
Automatic merge failed; fix conflicts and then commit the result.
$
```

The problem is that the `return 0;` statement added at the end of `main()` in `master` conflicts with all the changes we've made in the `hello-multi` branch. Looking at the end of the `hello.c` file, you should see something like:

```
<<<<<<< HEAD
  for (int i = 1; i < argc; i++) {
    printf("hello %s\n", argv[i]);
  }
=======
  printf("hello %s\n", argv[1]);
  return 0;
>>>>>>> master
```

The `<<<`, `===` and `>>>` lines are conflict markers. You need to manually clean up this file so that you can proceed. Remove the conflict marker lines and well as the `printf(..., argv[1]);` line which is no longer needed. After your cleanup, `tail` should show something like the following:

```
$ tail -5 hello.c
  for (int i = 1; i < argc; i++) {
    printf("hello %s\n", argv[i]);
  }
  return 0;
}
```

You should now be able to build and re-run the feature branch:

```
$ make
$ ./hello john sue
hello john
hello sue
$
```

Commit your changes into the feature branch and push:

```
$ git commit -a -m 'merged master'
$ git push
```

1.4.1 Merge Feature Branch into Master

You can now merge the `hello-multi` feature branch into your `master` branch:

```
$ git checkout master
$ git merge hello-multi
```

Since all the changes from `master` were already in `hello-multi`, git only needs to add in the `hello-multi` changes to `master` without any merge. This is referred to as a fast-forward.

Recompile and test:

```
$ make
$ ../hello john sue
hello john
hello sue
$
```

We have the changes from hello-multi in `master`.

```
$ git branch -l #list branches
$ git shortlog hello.c #show all log messages for file
```

Note how the history encompasses work done on both `master` and `hello-multi`.

Commit and push:

```
$ git commit -a -m 'merged hello-multi'
$ git push
$
```

The commit will not commit anything to the local repository since the merge was a fast-forward and all changes are already in the local repository. However, the push will indeed push changes to the remote.

1.4.2 Getting Your Lab8 Repository into Your i220X Repository

Change over to your `i220a` or `i220b` directory:

```
$ cd ~/i220X  #X is either a or b

# specify URL for lab8 repo; replace LOGIN with your login-id
$ git remote add lab8 \
  ssh://LOGIN@remote.cs.binghamton.edu/home/LOGIN/git-repos/lab8.git
```

where `LOGIN` is your login id on `remote.cs`.

Now you can pull in your `lab8` repository:

```
$ git pull --allow-unrelated-histories lab8 master
```

Go in to the `lab8` directory:

```
$ cd work/lab8
$ ls
```

and you should see the contents of your `lab8` repository. In fact, you should see that the history has been transferred over too:

```
$ git log hello.c
```

Stop the script command which is recording your terminal session by typing a `^D`. This should create a `lab8.log` file in your current directory.

Add and commit this `lab8.log` file to your local repository. As usual, do a `git mv` of your `work/lab8` directory to `submit/lab8`, commit and push to github.

1.5 References

Official Git Site.

Scott Chacon, Ben Straub, *Pro Git*.

Reference Documentation