ECE2420 GIT In Class Exercise

## Overview

For ECE2420, all programming exercises will be turned in via a git push to your own personal repository. These repositories will be stored on the ECE server called git2420.ece.usu.edu. Accounts on this server have been created for each student. Your username is your fist initial followed by your last name. The default password is ‘Password123’. You should change this when you first log in.

In your home directory, you should create a directory call ‘ece2420.git’. Inside of this directory you will create a bare repository that will hold all of your programming exercises (this will be described in subsequent sections). Each exercise that you complete should be located in its own directory inside this repository.

At the end of this exercise you should have you bare repo set up on the git2420 server in a way that the instructor and assistants can clone you repository, observe your directory structure, and be ready to evaluate future programming exercises. This will leave you in a good place programming without thinking to much about turn in mechanics. If completed by 11:59 on 31 Aug, you will be awarded 25 points for the effort.

## Creating a Bare/Shared Repository (done on git2420)

The purpose of a bare shared repository is to be the location where publicly consumable (by the instructor and assitants) code is maintained. It does not have a working directory associated with it. The notion that this repository is special is by convention only – it is still just a Git repository; one of many peers.

mkdir ece2420.git

cd ece2420.git/

git init –-shared=group --bare

## Cloning a Repository – So you have a place to work (on your Linux box)

git clone ssh://git2420/home/<username>/ece2420.git clone1

1. Note the differences in the directory structure between this and the bare repo
2. Note that the objects dir is completely empty, (other than empty dirs)

## Add a File to Our Local Repository

cd clone1

vi test.c

git add test.c

git commit

Look at the local objects in clone1/.git/objects, compare SHA in the commit log message to the objects.

Take a minute to talk about Git objects (all of which are immutable):

1. Blob – file
2. Tree – like a directory; references other trees or blobs
3. Commit – points to a tree; describes entire repo at given point in time
4. Tag – like a commit, can be signed, points to a specific commit

## Staging and Un-staging File Changes

Git manages changes made to the working copy in what it refers to as the index. Things in the index are generally in one of three states:

1. Modified and will be committed
2. Modified and won’t be committed
3. Untracked

Learning how to control the index makes it very easy to control what gets committed and when. It also allows you to effectively manage the current set of changes in your working copy.

vi test.c

git diff

vi tempTest.c

git diff

git status

git add test.c

git diff

What the heck happened? Why no changes shown?

git diff -–cached

Oh… there they are. On second thought, that isn’t the file I want to commit on this commit, how do I get it out of the index? Reset is a very flexible command; we’ll get into it more later.

git reset test.c

git diff

git status

Commit the file with –a; it does the add for you automatically, but it will not add untracked files to the index. You have to do that yourself the first time.

git commit –a

git log

## The Directed A-Cyclical Graph

Git really is a tool built by a computer scientist for computer scientists. It really is useful to think of a Git tree as a DAG of commit type objects.

The current DAG looks like:

|  |
| --- |
| B  |  A  Where master -> B |

Note the ‘master -> B’ statement above; it is a reference. References are a critical concept to understand how Git works. They really are just pointers. We can make them point wherever we want to in the DAG. A branch is nothing more than a pointer to a commit.

git branch testBranch

Now we have:

|  |
| --- |
| B  |  A  Where master -> B,  testBranch -> B |

Look in .git/refs/heads; you can see it there; nothing more than a text file containing a hash. If you want to make your working copy change to what is pointed at by a different pointer, you simply have to check it out:

git checkout testBranch

git branch

git checkout master

git branch

Commits advance the pointer for the currently checked out pointer leaving other references untouched.

Checkout testBranch and add a new cool function to the source; commit your changes. Checkout master and add a different cool function; commit your changes.

Now we have:

|  |
| --- |
| C D  \ /  B  |  A  Where master -> D,  testBranch -> C |

The feature in testBranch is ready to be merged; you can see it make a node in the digraph with two parents:

git merge testBranch

git log –graph

Now we have:

|  |
| --- |
| E  / \  C D  \ /  B  |  A  Where master -> E,  testBranch -> C |

## Pushing Changes Back to the Bare Repository

We have done a lot of work up to this point, be we have really been living on an island. I use long syntax throughout this document for pedagogical purposes. However, there are **MANY** shortcuts that git has, and it will attempt to do what it ‘thinks’ you want it to do. Sometimes this is bad, so the user must exercise caution.

We want to ‘push’ our changes over to the bare repository so other people can consume them. The push step is the critical point of no return, **not the original commit.** We can undo/manipulate almost anything in our clone before it is pushed. After it is pushed, life is harder.

git push origin refs/heads/master:refs/heads/master

The origin parameter above is a shortcut for ssh://optimusBuild2/home/njensen/gittest/DemoBare.git. We could have typed the whole server/path if we had so desired. See .git/config to understand how the expansion is made.

The second parameter with the colon is more complex. If you understand it, you will most likely be proficient with Git. The part before the colon is the source and the part after is the destination. We are basically replacing the contents of the destination ref with the contents of the source ref.

What if we want to share our branch with the rest of the world?

git push origin refs/heads/testBranch:refs/heads/nateTestBranch

Note that things can be called something different between repos; go inspect the actual refs in the various directories.

This push added a couple more pointers to our DAG, as well as updated the bare/public repo. You can see what it added with the command: git branch –av.

We also want to connect the remote tracking branch with the local branch. Check out the .git/config file before and after issuing the commands below:

git branch --set-upstream testBranch refs/remotes/origin/nateTestBranch

git branch --set-upstream master refs/remotes/origin/master

Now we have:

|  |  |
| --- | --- |
| Clone1  E  / \  C D  \ /  B  |  A  Where:  master -> E,  testBranch -> C  remotes/origin/master -> E  remotes/origin/nateTestBranch -> C | DemoBare.git  E  / \  C D  \ /  B  |  A  Where:  master -> E,  nateTestBranch -> C |

## Another Developer Wants Our Changes

git clone ssh://git2420/home/<username>/ece2420.git clone2

cd clone2

git branch –av

git checkout nateTestBranch

clone2 can now be used for active development; all three repositories are full repositories that can stand on their own. The new developer creates a new file and commits it on the nateTestBranch branch.

vi test2.c

git add test2.c

git commit

Clone2:

|  |
| --- |
| F E  \ / \  C D  \ /  B  |  A  Where:  master -> E,  nateTestBranch -> F  remotes/origin/master -> E  remotes/origin/nateTestBranch -> C |

git status

Questions:

1. Why do we see what we see?
2. How did this commit impact the bare public repo?

Push this change back to the master repo

git push origin refs/heads/nateTestBranch:refs/heads/nateTestBranch

git status

## Back on clone1 a Developer Has Been Busy (perhaps concurrently)

Make another file in the branch and commit it

vi test3.c

git add test3.c

git commit

git status

Questions:

1. Only one ahead? Why? Clone2 just pushed something right?
2. Can the state of ‘our’ remote pointers ever be affected by an outside entity?

Try and push this latest commit to the bare repository

git push origin refs/heads/testBranch:refs/heads/nateTestBranch

Why does this fail? What is a fast forward merge?

When current branch is descendant of another… note that we don’t get a merge commit when we do this either.

Let’s get our remote tracking pointer for this branch in sync with the bare repo. Note how the git status changes as we update the upstream remote pointer/reference.

git status

git fetch origin refs/heads/nateTestBranch:refs/remotes/origin/nateTestBranch

git status

Now merge in these changes so when we push the push will be a simple fast forward; then do the push. Note that a fetch and a merge can be combined into an operation called a pull. It can make things quicker, but it really is still a fetch and a merge under the sheets.

git merge refs/remotes/origin/nateTestBranch

git push origin refs/heads/testBranch:refs/heads/nateTestBranch

Clone1

|  |
| --- |
| H  / \  F G E  \ | / \  \ | / \  C D  \ /  \ /  B  |  A  Where:  master -> E,  nateTestBranch -> H  remotes/origin/master -> E  remotes/origin/nateTestBranch -> H |

## Keeping Lineage Simple with Rebase

There is what appears to be an unneeded branch and merge that occurred and generated the merge commit ‘H’. There is way to keep things like this a bit cleaner with a rebase command rather than a merge.

1. Commit a change in clone2 and push (the ‘I’ commit below)
2. Commit a change in clone1 (the ‘J’ commit below) and attempt to push; breaks due to non Fast Forward
3. Do a fetch in clone1

At this point, rather than make a merge commit that will join the divergent code lines, what if we unrolled our latest commit, pulled in changes made in clone2, and then replayed our change sets on top of the updated commit chain? Rebase does exactly this. Note that this can cause conflicts. Why?

git rebase refs/remotes/origin/nateTestBranch

git push origin refs/heads/testBranch:refs/remotes/origin/nateTestBranch

Clone1

|  |
| --- |
| J’  |  J I  \ /  H  / \  F G E  \ | / \  \ | / \  C D  \ /  \ /  B  |  A  Where:  master -> E,  nateTestBranch -> J’  remotes/origin/master -> E  remotes/origin/nateTestBranch -> J’ |

## Moving and Manipulating References

Git reset is a command that can change where a reference points; it always updates the current branch to a specified commit. It can also do various things with the index and working copy depending on options passed:

--soft – do nothing to either index or WC

--hard – change both the index and the WC

--mixed (default) – change the index but not the WC

Reset is a very helpful command when you want to manipulate the pointers in your local clone. Prime example is that you have made a commit and then you want to undo it. In general, you are free to mangle your own commits that have not been pushed in any way that suits you. If you are going to change the references in a way that will affect changes that others may have already pulled, extreme caution must be exercised. Remember, the **push** should generally be considered the point of no return.

There is also another command that only updates references and will leave a working copy unchanged, (very similar to git reset -–soft). It also doesn’t manipulate the currently checked out reference. With this command you can arbitrarily point any reference to any location -- and you bear the consequences that brings.

git update-ref <ref> <sha1>

Git also has the notion of symbolic refs. Simply think of this as a reference to another reference rather than to a SHA1. For example in clone1:

cat .git/HEAD

As you can see, this doesn’t point to an object. This is also the only symbolic ref this training will discuss. In a non-bare repository, this reference will tell you which branch you currently have checked out. Check out a different branch and the contents of this reference will change. Now move back to the bare repository. Do a git log; you will see that you only see commits from the master tree. If we want to see other branch commits, we need to update the symbolic ref to point to a different reference.

git symbolic-ref HEAD refs/heads/nateTestBranch

Also note that this will change what is checked out by default during a clone. After executing the command below, the clone3 will have the nateTestBranch checked out.

git clone ssh://optimusbuild2/home/njensen/gittest/DemoBare.git/ clone3

Deleting a reference has a bizarre syntax, that becomes less bizarre if you really understand what a refspec is and how it works. For example to delete the nateTestBranch, (a tag is no different) on the bare repo, the command would look like:

git push origin :refs/heads/nateTestBranch

The way to think of this is that we are taking ‘nothing’ and pushing it over top of an existing reference.

## Resolving Conflicts

A Git merge can generate merge conflict much like SVN. There are several way to deal with it; keep in mind that git status will show you which files are conflicted and how.

1. We know that we have the right code in our WC
   1. git checkout -–ours <file>
   2. git add file
   3. git commit
2. We know that the repo already had the right code
   1. git checkout -–theirs <file>
   2. git add file
   3. git commit
3. When we need to massage the source file by hand
   1. vi file (shows you ‘ours’ and ‘theirs’ in the file)
   2. git add file
   3. git commit

You can merge many branches all at the same time, but be careful you can also get into trouble when you have multiple merges. Reading a 5 way diff really sucks. Git has no method to lock a binary file at all. For our small development team, we have (mis?)managed this simply by communication on the team.

One more gotcha: If you are doing a rebase *ours* and *theirs* have exactly the reverse meaning of what you might initially think. This is because we have accepted the base tree as ‘ours’ and we are replaying ‘theirs’ commits onto it.

## Setting Aside Working Copy Changes Temporarily

Oftentimes when you are doing a major development effort, you find a bug in an unrelated area. You say to yourself: “It sure would be nice if I could set aside these invasive changes for a second and fix this bug.” Well, you can.

cd clone1

vi test.c

git stash save “My major changes”

git diff

git stash show stash@{0}

<make and commit the bug fix>

git stash pop stash@{0}

git diff

<go back to work on major changes>