Lunds tekniska högskola
Datavetenskap, Nov 25, 2013
EDA260 Programvaruutveckling i grupp—projekt
Labb 2 (part II: Git): Labbhandledning
Checked on Git versions: 1.8.1.2

Lab Exercise
Part II: Git: A distributed version control system

Goal

This part of the lab is intended to demonstrate basic usage of `git`, a distributed version control system.

1 Introduction

In part I of this lab, we met Alice and Bob who worked with CVS. In this part (II), you will continue to play the roles of Alice and Bob, to try out Git. As in the previous lab part, we will work from the command line. As before, keep notes of what you do during the lab.

Alice and Bob are planning to move their development to the new popular version control system Git. They want to start working in a simple way, similar to CVS, with a common central repository that they both can commit changes to, or rather, using Git parlance, they will *push* changes to it. Since Git works a bit differently from CVS, they draw two figures to get a better understanding of similarities and differences. In figure 1 it is shown how `checkout` is used in CVS to create a local workspace from the common repository. In Git, the command `clone` is used instead, which creates both a local repository and an associated workspace. The local repository is a clone of the common repository, and contains all the versions of the software. The common repository is *bare*, meaning that it does not have any associated workspace.

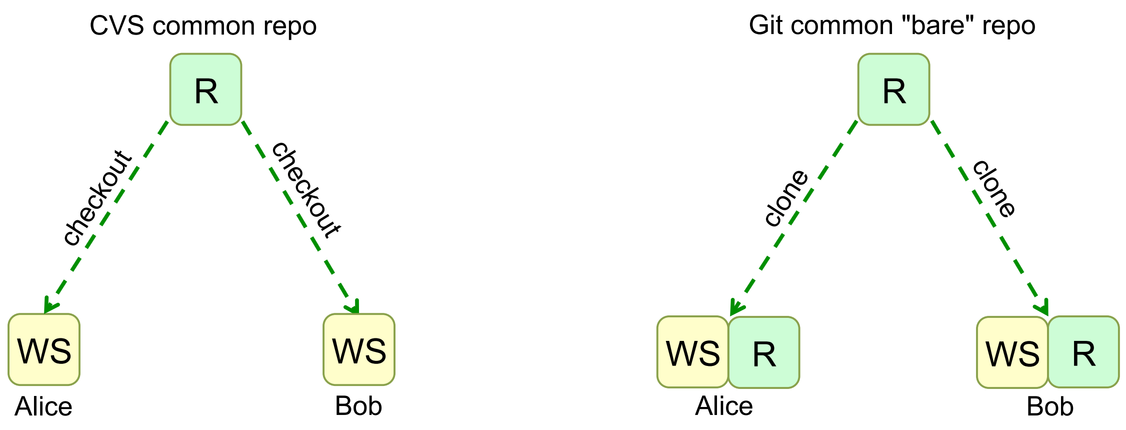

Figure 1: Creating a local workspace. In Git, a local repository is created along with the workspace.

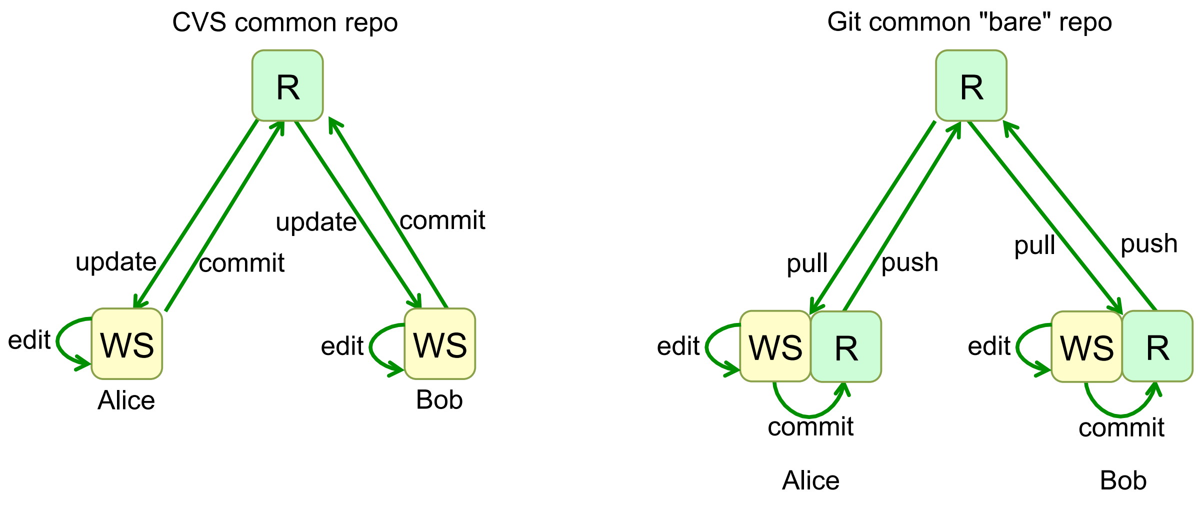

Figure 2: Working with CVS and Git. In Git, the changes are first committed to the local repository. Then the changes are pushed to the common repository.

Figure 2 shows how a developer works. In CVS, the developer *updates* the workspace to get the latest version from the repository, edits files, and then *commits* the changes to the common repository. In Git, the developer *pulls* the latest version from the common repository to his/her own local repository/workspace, then edits files, then *commits* the changes to the local repository, and finally *pushes* changes in the local repository back to the common repository.

Now, study the two figures and make sure you understand what *clone*, *pull*, *commit*, *push*, and *bare* means in Git. Write down your own descriptions here:

clone	
pull	
commit	
push	
bare	

Alice and Bob now have some doubts about Git because it would seem that it could take a lot of space to have the complete repository locally. And in CVS you can check out a single module from the repository, whereas in Git you get a workspace for the latest version of the complete repository. However, after consulting with others, they find out the following:

- Git uses compression algorithms to keep down the size of the repositories. But to keep down the size it is also important to commit only source data, like source files and text files, and avoid committing large generated binary files, like class files and jar files.
- In CVS it is common to have a large repository with many submodules, e.g., for different products or packages. But in Git, you typically have many smaller repositories instead.

Alice and Bob still think that Git looks a bit more complicated than CVS, so they wonder about what the advantages could be. Again, they consult with others and get a lot of different answers, for example the following:

- You can work locally, committing to your own local repo, without any network connection.
- You can create branches locally, without needing to make them visible to others.
- You can create your own repositories locally, so you can get the benefits of version control for your own private work, not just for collaboration.
- Instead of using a central repository, you can push and pull directly between developer repositories, creating your own work policies.
- If the central server breaks down, you will still have all your data, and can keep on working.
- It's the most popular version control system right now, so you get a lot of cool support from source code hosting providers like GitHub, BitBucket, Google Code, and others.

Encouraged by this, Alice and Bob think it is worth giving Git a try. They will build a simple HelloWorld application and version control it using Git. In the following, you should play the roles of Alice and Bob, as indicated by the box in the margin.

2 Alice starts working locally

Alice is eager to get started. She rushes ahead and creates a new directory for the project in her pvglab2 directory, and creates an empty README file for the project. *Alice*

```
$ cd pvglab2
$ mkdir HelloProject
$ cd HelloProject
$ touch README
```

Alice thinks about this wonderful news that she can have her own local git repository, and get version control on her own code. She decides to try this out before contacting Bob about any collaboration.

2.1 Configuring git

Alice has never worked with git before, so she first configures git with her name, email, and the default text editor she would like to use (`nano` is still her favorite text editor): *Alice*

```
$ git config --global user.name "Alice Wonderland"
$ git config --global user.email alicedat12@student.lth.se
$ git config --global core.editor nano
```

She then checks her current configuration, to make sure the new info is there:

```
$ git config --list
```

She wonders where this information is stored, and finds out that it is in a file called .gitconfig, in her home directory, so she looks at its content:

```
$ less ~/.gitconfig
```

Alice has also heard that the default behavior of *push* is about to change in the upcoming version 2.0 of git[1]. Alice checks what version of git she is using, and since it is not yet version 2.0, she sets the following configuration to get the new default behavior. This way, she will avoid getting a long message from git everytime she does a push.[2]:

[1]The new default behavior will be to only push the current branch, and not all local branches that are also in the common repository.

[2]If Alice used an old version of git (older than 1.7.11), she should instead do
`git config --global push.default current` to get a similar behavior.

```
$ git --version
$ git config --global push.default simple
```

2.2 Creating a local repository

Alice creates a local repository simply by doing the git command *init* in her new project directory, adding all files (currently just the README file), and committing them to the new repository: Alice

```
$ cd HelloProject
$ git init
$ git add .
$ git commit -m "New project"
```

She wonders where git placed the repository, and finds it in a subdirectory `.git`. She takes a look at the contents of the repo, just out of curiosity:

```
$ ls -l .git
```

She notes that there are files and directories called `HEAD`, `branches`, `config`, etc., and she thinks this looks just like the kind of stuff you would expect in a version control repository.

2.3 Working with the local repository

Alice now starts working with her local repository. She edits the README file, and commits again, checking the git status in between the commands, and checking the commit log: Alice

```
$ git status
$ nano README
$ git status
$ git commit -a -m "Updated README"
$ git status
$ git log
```

She notes some things:

- She is working on a branch called *master*
- Before committing she needs to add new and changed files to the so called *staging area*
- Changed files can be staged by using the -a option for *commit*, instead of explictly doing an *add*
- When there is nothing to commit, the working directory is said to be *clean*

Alice also finds out that each commit is identified by a long hexadecimal string, like

`6b8736fe84cb3e842352af055e0b3948e602fc80`

She finds out that this is a hash value of the contents of the committed version and its complete history of earlier versions. Alice also finds out that these hash values (produced by the SHA-1 secure hash algorithm) are sufficiently unique, so that it is extremely unlikely that two different commits would get the same hash code. In fact, she read somewhere that a hash collision might occur if you had 2**80 commits, which incidentally is the same order of magnitude as the number of atoms in the universe. So Alice feels pretty confident it will not happen in her project. She also finds out that Git uses these hash values internally to identify files, directories, etc., to be able to decide extremely quickly if two files/trees/commits, etc. are equal or not, i.e., without having to actually compare the contents.

Alice notes that commands like *commit* and *push* use seven-digit hexadecimal numbers, like `6b8736f`, and she finds out that this is simply a way of abbreviating the long hash values by only showing the first seven digits.

3 Setting up a common repository

Alice would now like to create a common repository, so that she can invite Bob to start collaborating with her on the new project. She will simply place the common repository somewhere where both she and Bob can access it. Unfortunately it doesn't work to use the ordinary student machines, because Alice and Bob are on different disk partitions, so Bob will not be able to write to files in her account. But both Alice and Bob have accounts on a lab machine `pvggit.cs.lth.se`. She will create the repository on her user account there, and use the git option *shared*, meaning that the unix access privileges are set to *group read and write*. This way, Bob can push to the repository, even if it is located on Alice's account.

Alice is a little worried about Charlie, since he also has an account on the lab machine, and could access the repo as well, even if he is not on their team. But she will handle this by simply not telling him the name of the repository. He will not be able to find it since her home directory on the lab server has no read access for group or other users. And perhaps Alice will move the repository to GitHub or some other commercial provider later, where there is better support for access control.

3.1 Creating the common repository

Alice

Alice now creates the common repository on the lab server. She makes it *shared*, so that Bob will be able to push into it, and *bare*, because this is the common repository that does not need any associated working directory. She calls the new repository `HelloProject.git`, because she has heard that bare repositories are usually given a name with the extension `.git`. Instead of logging in on the lab server, she simply performs a remote command using `ssh` (secure shell). By the way, she has the same username and password on this server as she has on the CVS server.

```
$ ssh alice@pvggit.cs.lth.se "git init --bare --shared HelloProject.git"
```

This creates the common repository as a directory HelloProject.git in her home directory on the lab server.

Out of curiosity, Alice takes a look into the new repository, again using an `ssh` command:

```
$ ssh alice@pvggit.cs.lth.se "ls -l HelloProject.git"
```

She notes that there are files and directories called `HEAD`, `branches`, `config`, etc., just like in her own local repository. She also notes that all the files and directories are marked with the group `students`, and have group read and write access, so Bob's git commands will be able to manipulate the files.

3.2 Connecting the developer repository to the common repository

Alice

Alice now needs to connect her developer repository to the new common repository and push its contents to it. She goes to the developer repository and performs the following magic for doing this:

```
$ cd pvglab2/HelloProject
$ git remote add origin alice@pvggit.cs.lth.se:/home/alice/HelloProject
$ git push -u origin master
```

These commands will make it look like the developer repository was cloned from the common repository (although we in fact created the repositories in the opposite order).

Out of curiosity, Alice looks up the more precise meaning of these commands and finds out the following:

- The *remote* command sets up the common repository as the *origin* of the developer repository.
- The *push* command pushes the *master* branch of the developer repository into the origin repository.

- The *-u* option configures her master branch to *track* the master branch in the common repository. This has the effect that when she in the future does a `git push`, the changes will end up in the right place in the common repository.

Alice does not entirely grasp all this, but is happy with accepting that the magic probably does what it should. To check that everything works, Alice does a small change to the README file, commits it, and pushes it to the common repository.

```
$ nano README
$ git commit -a -m "Updated README"
$ git push
```

She also tries the pull command.

```
$ git pull
```

Of course, nothing new is pulled, since Bob has not started working yet.

Avoiding writing the password?

Alice thinks it is a bit tedious to have to write the `pvggit` password for every pull or push. She knows this can be avoided by generating an ssh key, but she thinks this is too much work for this short experiment with git.

4 Alice invites Bob to work on the project

Alice now invites Bob to work on the project. He starts by configuring git in a similar way as Alice did. *Bob*

4.1 Cloning the common repository

Bob clones the common repository into a suitable place in his own account, and checks the contents of his working directory. *Bob*

```
$ cd pvglab2
$ git clone bob@pvggit.cs.lth.se:/home/alice/HelloProject
$ cd HelloProject
$ ls -a
```

4.2 Bob starts working

Bob starts working. He edits the README file some more, commits it, and pushes back the changes to the common repository. Write down the commands he uses here: *Bob*

4.3 Alice pulls the new changes

Alice pulls down Bob's changes, and takes a look at the commit log. *Alice*

```
$ git pull
$ git log
```

4.4 Concurrent development: using stash

Alice and Bob now start experimenting with developing at the same time. Just like you should do *update* before you do *commit* in CVS, you should do *pull* before you do *commit* and *push* in Git. And just like in CVS, you should make sure that the code is clean (compiles and tests) before committing and pushing.

First, try the following scenario where Alice and Bob edit the same file:

- Alice edits the README file and saves it. *Alice*
- Bob edits the README file and saves it. *Bob*
- Alice pulls, commits, and pushes. *Alice*
- Bob pulls in order to merge with the latest version before he commits and pushes. *Bob*

What happens? It turns out that you cannot pull if your working directory contains uncommitted files that need to be merged with the pulled changes. Bob now has two options:

commit he can commit his changes and then pull

stash he can *stash* away his changes, then do pull, and later *apply* the stashed changes, merging them with the new version of the files.

The *stash* command takes all the uncommitted changes to the working directory and moves them away to a safe place, the *stash*, so that all edits are undone, and the working directory is set back to the previous *clean* state (with no uncommitted changes). The *stash apply* command takes the changes from the stash, and applies them on the working directory again, i.e., merges them with the files in the working directory.

Bob tries the stash option, checking the contents of the README file after each step. If necessary, he edits the README file before the commit, to make sure it has the desired merged content. *Bob*

```
$ git stash
$ more README
$ git pull
$ more README
$ git stash apply
$ more README
$ nano README
$ git commit -a -m "..."
$ git push
```

Bob realizes that *stash* is like a light-weight anonymous branch.

4.5 Continued concurrent development

Alice and Bob continue with their development, creating the HelloWorld application. They experiment with different ways of getting into merge situations, making sure that Alice also tries out the stash command.

Alice and Bob then reflect on what they have learnt, by writing down descriptions of git concepts and commands:

repository	
working directory	
staged area	
stash	
commit identifier	
`git config`	
`git init`	
`git clone`	
`git status`	
`git log`	
`git add`	
`git commit`	
`git push`	
`git pull`	
`git stash`	
`git stash apply`	

5 Learning more

Alice and Bob now discuss some additional things they would like to learn about Git:

- How to use Git from Eclipse.
- How to move the common repository to GitHub or BitBucket or some other project hosting provider.
- How to use branches in Git.

As a starting point, they take a look at the official Git site: `http://git-scm.com`, then they decide it is time for a break.