Faculty of Computers, Informatics and Microelectronics

Technical University of Moldova

PAD

Laboratory work # 1

GIT: source code distributed systems.

Author:
Petru Negrei

Supervisor:
D. Ciorba

September 2014

1 Introduction

1.1 Topic

Git: Distributed revision control systems.

1.2 Objective

To work with Git tools and study its possibilities.

1.3 Generic requirements

1.3.1 Task

- Initialise a local git repository;
- Create and link with repository from Git Lab account;
- Create 3 branches. On each branch commit 3 modifications. Push the local modifications to the remote repository.

1.3.2 Report

Report will containt a short description of work done, and will present necesary information about tools, algorithms used or studied.

2 Implementation

2.1 Installation

It is easiest to install Git on Linux using the preferred package manager of your Linux distribution. On Debian/Ubuntu.

```
    $ apt-get install git
```

2.2 Configuration

The first thing you should do after you install Git is to set your user name and e-mail address. This is important because every Git commit uses this information, and its immutably baked into the commits you pass around.

- **Username:** First you need to tell git your name, so that it can properly label the commits you make.
- **Email:** Git saves your email address into the commits you make. We use the email address to associate your commits with your GitHub or other account.

```
git config --global user.name "Your Name Here"
git config --global user.email "your_email@youremail.com"
```

2.3 Initialize an empty repository

Go to the projects directory and type the below commands this will initialize the repository to git :

```
git init
git add .
touch README.md
git commit -m "initial commit"
```

2.4 Link local and remote repositories

Before you can send your changes of the project to the remote repository you need to link your local with the remote one. There are 2 ways to connect with the GitLab repository: HTTPS or SSH. I used the SSH method to establish a connection.

The keys are situated in the ' /.ssh' directory, first you need to verify if there are already generated keys.

```
# check id_rsa.pub file
$ ls ~/.ssh | grep id_rsa.pub
```

If the file doesn't exist then we need to generate a new SSH key, by typing the command below. After the new key is generated you need to link to your GitLab account, located under 'Profile settings', 'SSH Keys'.

```
# generate rsa key
$ ssh-keygen -t rsa -C "you_email@example.com"
# verify if the account was linked
$ ssh -T @git@gitlab.ati.utm.md
```

2.5 Remote repository

The next step is to create a git repository on you GitLab account, named *lab*. Then I linked it with my local repository by typing the following command:

```
$ git remote add origin git@gitlab.ati.utm.md:petru.negrei/lab.git
```

When you need to share your code you need to push your branch to a your remote repository. Your local branches arent automatically synchronized to the remotes you write to you have to explicitly push the branches you want to share.

And the last step is to push to the remote repository. Now your code is on remote repository, and can be seen by everyone colaborating with you.

```
    $ git push origin master
```

2.6 Working with branches

Here I will explain the workflow of laboratory most important part. I tried to emulate the work made during a real project.

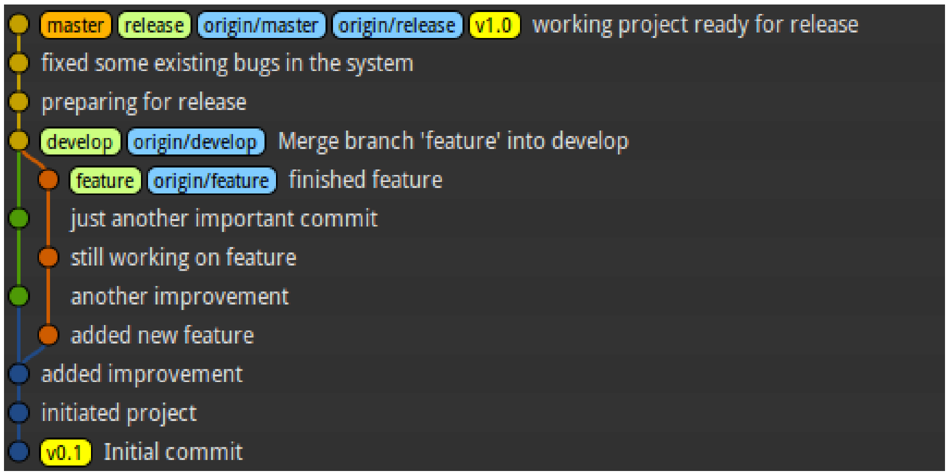
First thing I did was to use *tags*, in order to tag specific points in history as being important. Generally, people use this functionality to mark release points (v1.0, and so on), so I started by tagging *master 0.1*.

Next step I added some commit on **master** branch and then I created a new branch named **develop**. Ussually **develop** branch is were all work is done. Here I added some commits refering to *improvements* added to the project.

Next I created another branch named **feature**, this branch purpose is to contain work done relating to new feature that will be added to the project, here also I added some commits, and then merged *feature* and *develop* branches.

Next I created another branch named **release**, this branch purpose is to prepare the project for release, and to fix existing bugs and add last touches to the current version of the project. After couple commit I merged the *feature* branch and *master* branch, adding a new tag *v1.0*.

You can see the current state of the project in the figure below.



```
master release origin/master origin/release v1.0 working project ready for release
fixed some existing bugs in the system
preparing for release
develop origin/develop Merge branch 'feature' into develop
feature origin/feature finished feature
just another important commit
still working on feature
another improvement
added new feature
added improvement
initiated project
v0.1 Initial commit
```

Fig. 1 History

Below I will list some of the commands used during the implementation of the given task:

- *git status* - check the current state of the repository, showing the untracked (created files), modified of staged files.
- *git checkout -b "branch name"* - command used to create branches.
- *git checkout "branch name"* - command used in order to switch between branches.
- *git merge "branch name"* - merges the current branch with the "branch name".
- *git push and pull* - commands used to check for changes or to update your changes to the remote repository.

3 Conclussion

I previously worked with Git, which is a great Distributed Version Control System that fully mirror the repository. That allow the posibility to copy the back up from any client (that has this repo), to the server, in case the server is down, and need to be restored.

Distributed revision control systems takes a peer-to-peer approach to version control, as opposed to centralized system. Rather than a single, central repository (usually server), each peer has a working copy of the codebase as a coplete repository. Distributed revision control synchronizes repositories by exchanging data (sets of changes) between peers. Some of the important benefits of distributed systems are:

- ○ *Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.*
- ○ *Communication is only necessary when sharing changes among other peers.*
- ○ *Each working copy effectively functions as a remote backup of the codebase and of its change-history, protecting against data loss.*

Thus with the help of Git and Github (which is a repository web-based hosting service), we can have several remote repositories that can work on different machine with different users, that can collaborate with each other, in different ways simultaneosly within the same project.

Link to Repository: `https://gitlab.ati.utm.md/petru.negrei/lab`

4 References

- Scott Chacon, Pro Git, July 29, 2009 `http://git-scm.com/book`
- Git How To, `http://githowto.com/`
- Atlassian, Git Tutorials, `https://www.atlassian.com/git/tutorial`

- Vincent Driessen, A successful Git branching model, January 05, 2010, `http://nvie.com/posts/a-successful-git-branching-model/`
- Code School, Try Git, Free course, `https://www.codeschool.com/courses/try-git`
- Code School, Git Real, Free preview, `https://www.codeschool.com/courses/git-real`
- Code School, Git Real 2, Free preview, `https://www.codeschool.com/courses/git-real-2`
- Linux.conf.au 2013 - Git For Ages 4 And Up, Youtube, `https://www.youtube.com/watch?v=1ffBJ4sVUb4`