GITHUB BASED GIT AND TFS

BY

insa gEOSPATIAL dIRECTORATE r&d tEAM

**April 5-05-2015**

**Version Control System (VCS):**

* Software that helps software developers to work together and maintain a complete history of work.

Listed below are the functions of a VCS:

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

Following are the types of VCS:

1. Centralized version control system (CVCS):- uses a central server to store all files and enables team collaboration.



Figure 1: centralized version control system [3]

1. Distributed/Decentralized version control system (DVCS).

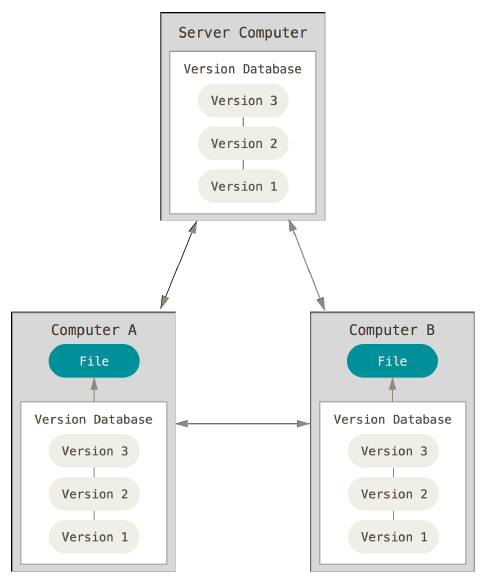


Figure 2: Decentralized version control system [3]

Concentrate only on distributed version control system and especially on Git.

If any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every checkout is really a full backup of all the data

**Git on the Server**

* The Protocols**:** - Git can use four major protocols to transfer data**: Local, HTTP, Secure Shell (SSH) and Git.**
* Local Protocol:-The most basic is the *Local protocol*, in which the remote repository is in an other directory on disk.

**THE PROS**

The pros of file-based repositories are that they’re simple and they use existing file permissions and network access. If you already have a shared file system to which your whole team has access, setting up a repository is very easy. You stick the bare repository copy somewhere everyone has shared access to and set the read/write permissions as you would for any other shared directory.

**THE CONS**

The cons of this method are that shared access is generally more difficult to set up and reach from multiple locations than basic network access. If you want to push from your laptop when you’re at home, you have to mount the remote disk, which can be difficult and slow compared to network-based access.

**The HTTP Protocols**

Git can communicate over HTTP in two different modes

1. **SMART HTTP**

The “smart” HTTP protocol operates very similarly to the SSH or Git protocols but runs over standard HTTP/S ports and can use various HTTP authentication mechanisms, meaning it’s often easier on the user than something like SSH, since you can use things like username/password basic authentication rather than having to set up SSH keys.[3 and 2]

1. **DUMB HTTP**

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler “dumb” HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server.

The beauty of the Dumb HTTP protocol is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific post-update hook, and you’re done .[3 and 2]

**Git Advantages**

* Free and open source
* Fast and small
* Implicit backup
* Security
* No need of powerful hardware(as server)
* Easier branching

**DVCS Terminologies**

* Local Repository
* Working Directory and Staging Area or Index.

**Git workflows.**

**Step 1:** You modify a file from the working directory.

**Step 2:** You add these files to the staging area.

**Step 3:** You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository as seen fig (4).

  
*Figure 4: Git work flow [3]*

**General workflow life cycle is as follows:**

1. You clone the Git repository as a working copy.
2. You modify the working copy by adding/editing files.
3. If necessary, you also update the working copy by taking other developers' changes.
4. You review the changes before commit.
5. You commit changes. If everything is fine, then you push the changes to the repository.
6. After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the workflow fig (5).



Figure 5: General workflow Git life cycle [3]

# Everyday GIT With 20 Commands and above

**Individual Developer (Standalone)**

A standalone individual developer does not exchange patches with other people, and works alone in a single repository, using the following commands.

* [*git-init()*](https://www.kernel.org/pub/software/scm/git/docs/git-init.html) to create a new repository.
* [*git-show-branch()*](https://www.kernel.org/pub/software/scm/git/docs/git-show-branch.html) to see where you are.
* [*git-log()*](https://www.kernel.org/pub/software/scm/git/docs/git-log.html)to see what happened.
* [*git-checkout()*](https://www.kernel.org/pub/software/scm/git/docs/git-checkout.html) and [git-branch()](https://www.kernel.org/pub/software/scm/git/docs/git-branch.html) to switch branches.
* [*git-add()*](https://www.kernel.org/pub/software/scm/git/docs/git-add.html) to manage the index file.
* [*git-diff()*](https://www.kernel.org/pub/software/scm/git/docs/git-diff.html) and [*git-status()*](https://www.kernel.org/pub/software/scm/git/docs/git-status.html) to see what you are in the middle of doing.
* [*git-commit()*](https://www.kernel.org/pub/software/scm/git/docs/git-commit.html)to advance the current branch.
* [*git-reset()*](https://www.kernel.org/pub/software/scm/git/docs/git-reset.html) and [*git-checkout()*](https://www.kernel.org/pub/software/scm/git/docs/git-checkout.html) (with pathname parameters) to undo changes.
* [*git-merge()*](https://www.kernel.org/pub/software/scm/git/docs/git-merge.html) to merge between local branches.
* [*git-rebase()*](https://www.kernel.org/pub/software/scm/git/docs/git-rebase.html)to maintain topic branches.
* [*git-tag()*](https://www.kernel.org/pub/software/scm/git/docs/git-tag.html)to mark known point.

## Individual Developer (Participant)

A developer working as a participant in a group project needs to learn how to communicate with others, and uses these commands in addition to the ones needed by a standalone developer.

* [*git-clone()*](https://www.kernel.org/pub/software/scm/git/docs/git-clone.html)from the upstream to prime your local repository.
* [*git-pull()*](https://www.kernel.org/pub/software/scm/git/docs/git-pull.html)and [git-fetch()](https://www.kernel.org/pub/software/scm/git/docs/git-fetch.html)from "origin" to keep up-to-date with the upstream.
* [git-push()](https://www.kernel.org/pub/software/scm/git/docs/git-push.html) to shared repository, if you adopt CVS style shared repository workflow.
* [git-format-patch()](https://www.kernel.org/pub/software/scm/git/docs/git-format-patch.html) to prepare e-mail submission, if you adopt Linux kernel-style public forum workflow.

## 

## Integrator

A fairly central person acting as the integrator in a group project receives changes made by others, reviews and integrates them and publishes the result for others to use, using these commands in addition to the ones needed by participants.

* [***git-am()***](https://www.kernel.org/pub/software/scm/git/docs/git-am.html) to apply patches e-mailed in from your contributors.
* [*git-pull()*](https://www.kernel.org/pub/software/scm/git/docs/git-pull.html) to merge from your trusted lieutenants.
* [*git-format-patch(*)](https://www.kernel.org/pub/software/scm/git/docs/git-format-patch.html) to prepare and send suggested alternative to contributors.
* [*git-revert()*](https://www.kernel.org/pub/software/scm/git/docs/git-revert.html) *to* undo botched commits.
* [*git-push(*)](https://www.kernel.org/pub/software/scm/git/docs/git-push.html) to publish the bleeding edge.

## Repository Administration

A repository administrator uses the following tools to set up and maintain access to the repository by developers.

* [*git-daemon()*](https://www.kernel.org/pub/software/scm/git/docs/git-daemon.html) to allow anonymous download from repository.
* [*git-shell()*](https://www.kernel.org/pub/software/scm/git/docs/git-shell.html) can be used as a restricted login shell for shared central repository users.

Why use git?

* It's fast
* You don't need access to a server
* Amazingly good at merging simultaneous changes
* Everyone's using it

**GitHub**

**: -** is a web-based hosting service for software development projects that uses the Git revision control system.

**Basic use Github**

* Change some files
* See what you've changed

git status

git diff

git log

* Indicate what changes to save

*git add*

* Commit to those changes

*git commit*

* Push the changes to GitHub

*git push*

* Pull changes from your collaborator

*git pull*

*git fetch*

*git merge*

Bottom of Form

## Adding an existing project to GitHub using the command line

1. [Create a new repository](https://help.github.com/articles/creating-a-new-repository) on GitHub. To avoid errors, do not initialize the new repository with README, license, or gitignore files. You can add these files after your project has been pushed to GitHub.
2. Open Terminal (for Mac users) or the command prompt (for Windows and Linux users).
3. Change the current working directory to your local project.
4. Initialize the local directory as a Git repository. Example **git init**
5. Add the files in your new local repository. This stages them for the first commit.

Example:-

* *git add .*
* Adds the files in the local repository and stages them for commit. To upstage a file, use “*git* reset HEAD *YOUR-FILE”*

1. **Commit the files that you've staged in your local repository.**

Example

* *git commit -m 'First commit'*
* Commits the tracked changes and prepares them to be pushed to a remote repository. To remove this commit and modify the file, use *'git reset --soft HEAD~1'* and commit and add the file again.

1. At the top of your GitHub repository's Quick Setup page, click to copy the remote repository URL.
2. The Command prompt, [add the URL for the remote repository](https://help.github.com/articles/adding-a-remote) where your local repository will be pushed.

Example

* git remote add origin remote repository URL
* Sets the new remote
* *git remote -v*
* Verifies the new remote URL.

**Note:** GitHub for Windows users should use the command *git remote* set-url origin instead of git remote add origin. [1]

1. [*Push the changes*](https://help.github.com/articles/pushing-to-a-remote)in your local repository to GitHub.

* Example git *push* origin master
* Pushes the changes in your local repository up to the remote repository you specified as the origin

**Why use GitHub?**

* It takes care of the server aspects of git
* Graphical user interface for git
* Exploring code and its history
* Facilitates:
* Learning from others
* Seeing what people are up to Contributing to others' code
* Lowers the barrier to collaboration
* "There's a type in your documentation." vs."Here's a correction for your documentation”

**Set up GitHub repository**

* Get a GitHub account
* Click the "Create a new repo" button
* Give it a name and description
* Click the "Create repository" button
* Back at the command line:

*Example:-* git remote add origin

<https://github.com/username/repo>

git push -u origin master

To create a GitHub repository, I generally first set things up locally (using git init and then a bit of git add and git commit).Then go to GitHub and click the “Create a new repo” button. Give it a name and description and click “Create repository.”

The back at the command line, you use git remote add to indicate the Github address; then Git push to push everything to GitHub.

**GIT**

**Strengths**

* It has full repository support when offline from others. It also has fast local repository access.
* You can easily have multiple repositories and highly flexible topologies. You can use repositories in circumstances where branches might be used in a centralized server approach, which can, therefore, help with scalability. Since all the effort required to work with the repository is performed on the client, DVCS solutions typically have more modest hardware requirements on the server.
* It encourages frequent check-ins to a local repository, thus providing the capability to track those changes and see the evolution of the code.
* It is well-suited to many open source project workflows. It allows participation in the project without any centralized server granting permission. It works well for large projects with many partly independent developers responsible for certain areas.
* Because of the way DVCS systems typically track changes, and because the nature of having distributed repositories means that merges happen more frequently, DVCS merges are usually less likely to produce conflicts, compared with similar changes merged from separate branches in a centralized version control system. However, merges can still obviously conflict, and the more the code has changed between merges, the more likely it is to require effort in performing the merge.
* As each working copy of the repository is a copy of the entire repository, including history, backups of that repository are implicit in each client. This increases the disaster recovery options without requiring any centralized overhead.
* DVCS systems provide a greater number of workflows when managing fi le versions. While this vast degree of freedom can be overwhelming to new-comers, once a basic workflow is established in the team it is quickly understandable.

**Weaknesses**

* Using developer repositories can reduce the frequency with which changes are synced with the rest of the team, leading to a loss of visibility as to the progress of teams overall.
* There is no centralized backup of progress for developers until changes are pushed to a central repository.
* Current DVCS solutions lack some security, auditing and reporting capabilities common to enterprise requirements such as the ability to control access by the path level. path in version control. Access permissions are controlled at the repository level, not at
* Most centralized systems (such as SVN and Team Foundation Server) allow for optional locking of fi les preventing later merge conflicts. The nature of DVCS tools makes this impossible.
* Since the entire repository is cloned to every machine, there can be an issue moving the large repositories across the network. This is often avoided by having multiple smaller repositories rather than just a single global repository.
* At the time of this writing the integrated tooling or the tooling on Windows is not at the same level of maturation as the most popular centralized version control systems such as Team Foundation Server or Subversion.

Pros:

* Distributed (fast as it is local!).
* Good branching.
* Good merging algorithms.

Cons:

* Visual Studio & Windows integration.
* Learning curve.

TFS

Team Foundation Server delivers source control, work item tracking, Team Foundation Build, a team project portal Web site, reporting, and project management capabilities. Team Foundation Server also includes a data warehouse where data from work item tracking, source control, builds, and testing tools are stored.

A logical Team Foundation Server is made up of two components: an application-tier server, made up primarily of Web services, and a data-tier server, made up primarily of several SQL Server 2005 /2008 databases. In Team Foundation Server deployments, these two components are referred to as the application-tier server and the data-tier server. The application-tier server and data-tier server can be deployed on one server or two servers. In the single-server configuration, the application-tier and data-tier services and programs are hosted on the same computer. In the dual server configuration, one server is the application-tier server, and the other is the data-tier server. This data is used by Team Foundation for its built-in reporting functionality.

All team projects are stored and managed with Team Foundation Server. To begin working on a team project, you must first connect to the appropriate Team Foundation Server. Be sure you are a member of a group on the Team Foundation Server with appropriate permissions before attempting to connect to Team Foundation Server, or that your account has been added directly to the Team Foundation Server and configured with the permissions appropriate to your role in the project.

Pros:

* Full integration with VS2010.
* Integration with work items & projects.
* Flexibility.

Cons:

* Flexibility [I know...but in order to configure it “just right” it can be time-consuming!]
* Heavyweight back-end requirements [Two server setup recommended, backup/restore more costly & time-consume.

**Recommendation**

* We will recommends, GitHub/Git is better than TFS. Because of Github is free and easily to get (free and open source), distributed (fast as it is local!), good merging algorithms, not need of powerfully hardware, simplicity backup and good branching.

**Reference**

1 it-ebooks.com pro git , 2nd edition,Scott Chacon and Ben Straub[1]

1. Version control system ,karl Broman[3]
2. Fast version control git tutorialspoint [4]
3. Www. github.com [2]

Appendex

Press enter to submit commands

> git init

$ git status

$ git status

$ git add octocat.txt

$ git status

$ git commit -m "Add cute octocat story"

$ git add '\*.txt'

$ git commit -m 'Add all the octocat txt files'\

$ git log

$ git remote add origin https://github.com/try-git/try\_git.git

$ git push -u origin master

$ git pull origin master

$ git diff HEAD

$ git add octofamily/octodog.txt

$ git diff --staged

$ git reset octofamily/octodog.txt

$ git checkout -- octocat.txt

$ git branch clean\_up

$ git checkout clean\_up

$ git rm '\*.txt'

$ git commit -m "Remove all the cats"

$ git checkout master

$ git merge clean\_up

$ git branch -d clean\_up

$ git push