ECE 321L - Lab 5

Git - Distributed version control system

Git is a distributed version control system (VCS) designed to track changes in source code during software development. Developed by Linus Torvalds in 2005, Git enables multiple developers to work collaboratively on the same project without interfering with each other's work. Unlike traditional VCSs that rely on a central server to store all versions of a project's files, Git allows every developer to have a full-fledged repository on their local machine, ensuring both speed and redundancy. This local repository contains a complete history of all changes made to the project, allowing for efficient branching, merging, and collaboration. Git's decentralized nature, combined with its robustness and speed, has made it the de facto standard for version control in the software industry, especially for open-source projects. Principles of GIt:

- Distributed Version Control: Unlike centralized version control systems, Git is distributed. This means every developer has a full copy of the project history on their local machine. This approach has several advantages, including speed (since most operations are local), redundancy (every copy is a full backup), and the ability to work offline.
- Snapshots, Not Differences: Most version control systems store information as a list of file-based changes. Git, however, thinks about its data more like a series of snapshots. Every time you commit, or save the state of your project in Git, it takes a snapshot of what all your files look like at that moment and stores a reference to that snapshot.
- Nearly Every Operation is Local: Most necessary information and operations in Git are local. This means you don't need a connection to a server to view the project history or make changes. This results in faster performance and the ability to work offline.
- Integrity: Everything in Git is checksummed using a SHA-1 hash algorithm. This ensures data integrity and the assurance that, once committed, your data cannot change without Git knowing.
- Frequent Commits and Collaboration: Git encourages a workflow where changes are committed frequently, rather than in large batches. This promotes collaboration, as it's easier to merge small, incremental changes.

Importance of Git in software engineering:

- Collaboration: Git allows multiple developers to work on the same project simultaneously. With features like branching, developers can work on features independently without affecting the main codebase.
- Versioning: Git provides a detailed history of commits, making it easy to revert to a previous state, compare changes over time, or see who last modified something.
- Backup and Redundancy: Since every developer has a full copy of the project repository, it acts as a backup. If a central server fails, any of the client repositories can be copied back to restore it.
- Continuous Integration and Deployment: Modern software practices, like Continuous Integration (CI) and Continuous Deployment (CD), rely heavily on version control systems like Git.

When code is committed, it can be automatically tested and deployed, ensuring rapid and reliable release cycles.

- Accountability: Git's detailed logging means every change is tagged with the author, date, and time. This ensures accountability and helps in tracking changes and their impact.
- Code Review: With platforms like GitHub and GitLab, Git provides tools for code review. Developers can submit their changes for review through pull requests, ensuring code quality and encouraging team collaboration.

Simple set-up: Let's walk through the steps to set up a simple C project on GitHub:

1. Setting up a GitHub Account: If you don't already have a GitHub account, sign up on github.com.
2. Installing Git (No need to do this in lab)
3. Configuring Git: After installing, configure your username and email (these will be associated with your commits):

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

4. Creating a New Repository on GitHub:
   - Go to your GitHub dashboard.
   - Click the '+' icon in the upper right corner and select 'New repository'.
   - Give your repository a name, for example, "simple-c-project".
   - Choose whether you want your repository to be public (visible to everyone) or private.
   - Initialize with a README if you wish.
   - Click 'Create repository'.
5. Cloning the Repository: Once the repository is created on GitHub, clone it to your local machine:

   ```
   git clone https://github.com/YourUsername/simple-c-project.git
   ```

6. Setting Up the C Project: Navigate to your project directory

   ```
   cd simple-c-project
   ```

   and create a simple C program. Create a file named hello.c:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

7. Committing and Pushing Changes: Track your new file and commit the changes:

```
git status
git add hello.c
git commit -m "Initial commit: Added hello.c"
```

8. Push the changes to GitHub:

   ```
   git push
   ```

9. Verifying on GitHub: Refresh your GitHub repository page. You should see the `hello.c` file you just pushed.

Scenario: Imagine that you have a simple C project with a file named `hello.c`. You made some changes to this file and pushed it to GitHub. However, after reviewing, you realized that the older version of the file was better and now you want to revert to that version.

1. Making a Change: Assume the initial content of hello.c was:

```c
#include <stdio.h>

int main() {
        printf("Hello, World!\n");
        return 0;
}
```

2. Make the following change:

```c
#include <stdio.h>

int main() {
        printf("Hello, GitHub!\n");
        return 0;
}
```

3. Commit and push it:

   ```
   git add hello.c
   git commit -m "Changed World to GitHub"
   git push
   ```

4. Reverting the Change: There are multiple ways to revert changes in Git. Here are two common methods:

   - Using `git revert`: This command creates a new commit that undoes changes from a specified commit. This means the commit history remains intact, and a new commit is added to reverse the changes. First, find the commit hash of the change you want to revert:

     ```
     git log
     ```

     This will show a list of commits. Copy the hash of the commit you want to revert. Then, use git revert:

     ```
     git revert [commit_hash]
     ```

     This will open an editor with a default commit message. Save and close the editor. Now, push the changes:

`git push`

- Using `git reset`: This method involves moving the current branch pointer to the desired commit and then pushing it to the remote repository. First, find the commit hash where everything was fine:

  `git log`

  Copy the hash of the desired commit. Then, reset your main branch to that commit:

  `git reset --hard [commit_hash]`

  Now, force push to update the remote repository:

  `git push --force`

  Caution: Force pushing can be dangerous as it rewrites the commit history, which can be problematic for collaborators. It's essential to coordinate with your team before force pushing.

5. Verification: Check you `hello.c` file or your GitHub repository, you'll find that it's reverted to the older version, printing "Hello, World!".

Deliverables: Create a new Git repository to host a simple C basic calculator. You will incrementally add features to the calculator, ensuring that each addition or modification is committed separately:

1. Initialize a new repository on their local machine using `git init`. Create a basic `calculator.c` file with a main function. Commit the changes with the message "Initial commit: Created calculator.c with main function".

2. Implement a function for addition in `calculator.c`. Test the function within the main function. Commit the changes with the message "Implemented addition feature".

3. Implement a function for subtraction in `calculator.c`. Test the function within the main function. Commit the changes with the message "Implemented subtraction feature".

4. Implement a function for multiplication in `calculator.c`. Test the function within the main function. Commit the changes with the message "Implemented multiplication feature".

5. Implement a function for division in `calculator.c` (consider edge cases like division by zero). Test the function within the main function. Commit the changes with the message "Implemented division feature".

6. Introduce a user interface in the main function, allowing users to choose an operation and input numbers. Commit the changes with the message "Added user interface for calculator operations".

7. Add error handling for invalid inputs, such as entering characters instead of numbers or choosing an invalid operation. Commit the changes with the message "Implemented error handling for invalid inputs".

8. Add comments throughout `calculator.c` explaining the functionality and logic of the code. Commit the changes with the message "Added comments and documentation".

9. Create a new repository on GitHub named "simple-calculator". Link the local repository to the GitHub repository: `git remote add origin [URL_OF_GITHUB_REPO]`. Push the commits to the GitHub repository: `git push`.

Students will submit:

1. A link to their GitHub repository showcasing their commit history and the final version of the calculator program.
2. A brief reflection (around 200 words) on their experience using Git for this activity, detailing what they learned, challenges they faced, and the importance of individual commits.