Practical Version Control and Issue Tracking

James Hetherington

University College London

1 Practical Version Control and Issue Tracking

1.1 What Version Control is For

- Managing Code Inventory
  - “When did I introduce this bug”?
  - Undoing Mistakes
- Working with other programmers
  - “How can I merge my work with Jim’s”

1.2 What is version control?

Do some programming

`my_vcs commit`

Program some more

Realise mistake

`my_vcs rollback`

Mistake is undone

1.3 What is version control? (Team version)

Sue	James
Create some code	
`my_vcs commit`	
	Join the team
	`my_vcs checkout`
	do some programming
	`my_vcs commit`
`my_vcs update`	more programming
Do some programming	
	`my_vcs commit`
`my_vcs commit`	
Oh Noes! Error!	
`my_vcs update`	
`my_vcs merge`	
`my_vcs commit`	
	`my_vcs commit`
	Error again…

2 Centralised Version Control

2.1 Centralised VCS concepts

- There is one, linear history of changes on the server or **repository**
- Each revision has a unique, sequential identifier (1,2,3,4…)
- You have a **working copy**
- You **update** the working copy to match the state of the repository
- If someone else has changed the repository while you were working:
- You update to get their changes

- You have to **resolve conflicts**
- Then you commit

2.2 Centralised VCS diagram

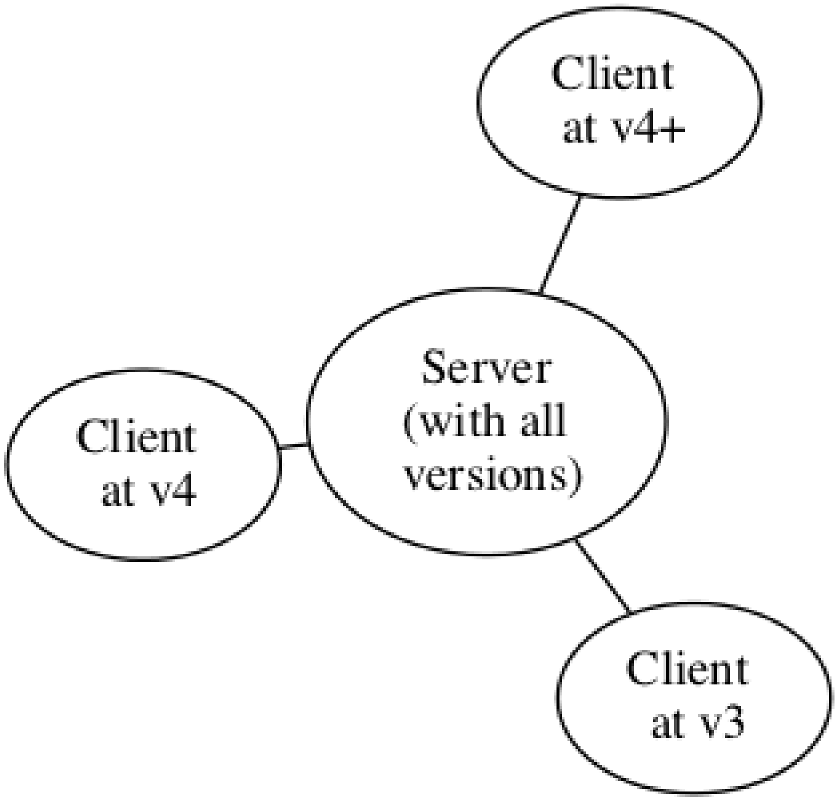


Figure 1: A centralised server with three clients

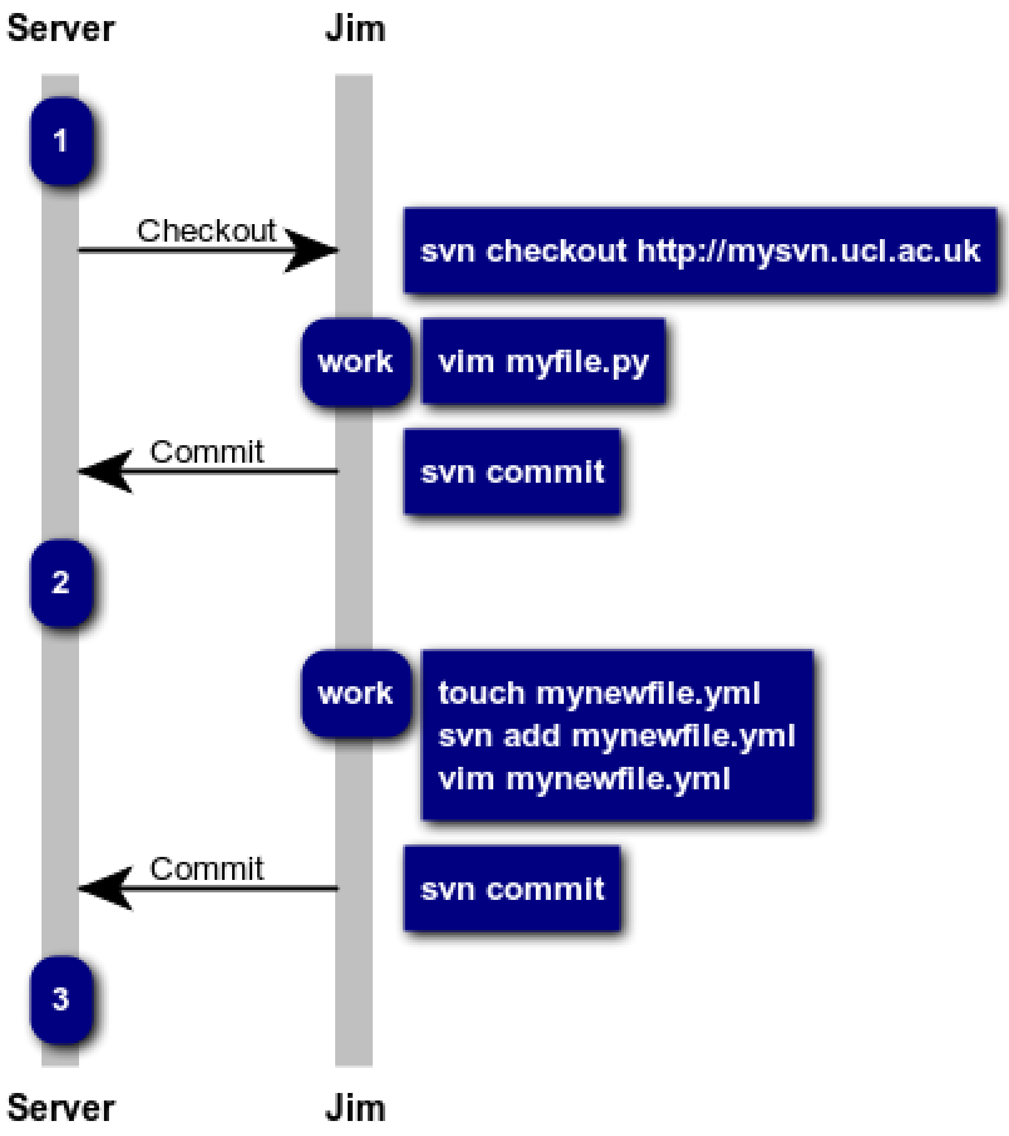
2.3 Centralised VCS solo workflow

2.4 Centralised VCS team workflow

2.5 Centralised VCS conflicted workflow

2.6 Resolving conflicts

On update, you get a prompt like:



```
svn checkout http://mysvn.ucl.ac.uk
vim myfile.py
svn commit
touch mynewfile.yml
svn add mynewfile.yml
vim mynewfile.yml
svn commit
```

Figure 2: Solo workflow for SVN

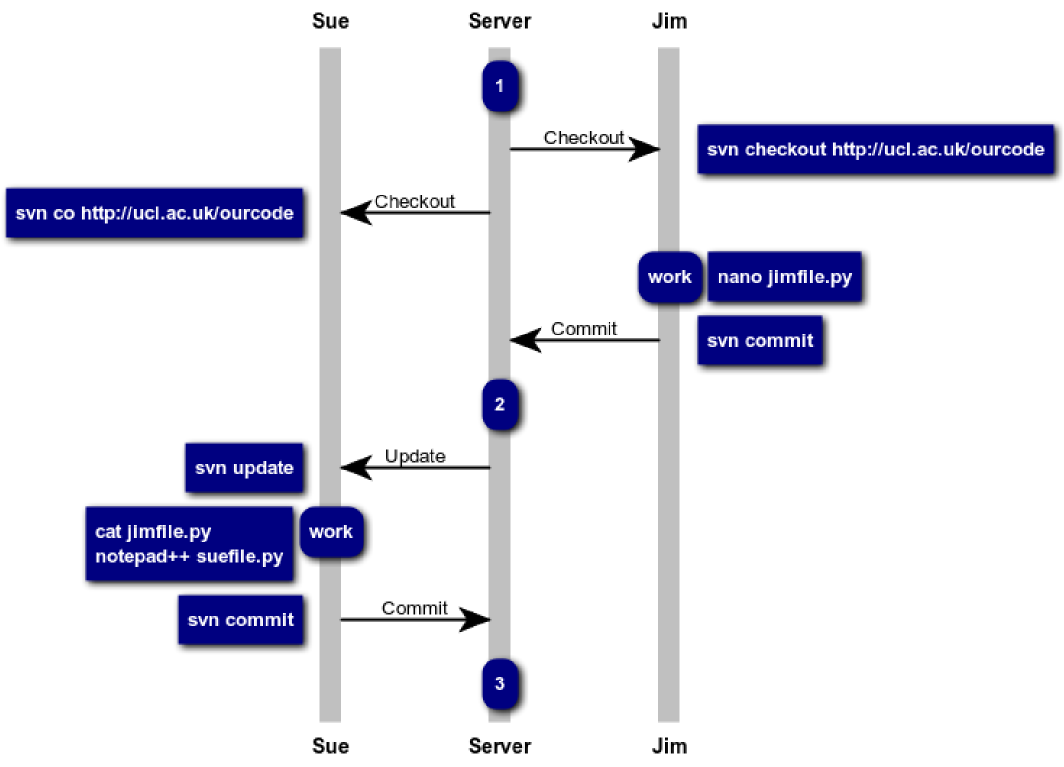


Figure 3: Team workflow for SVN

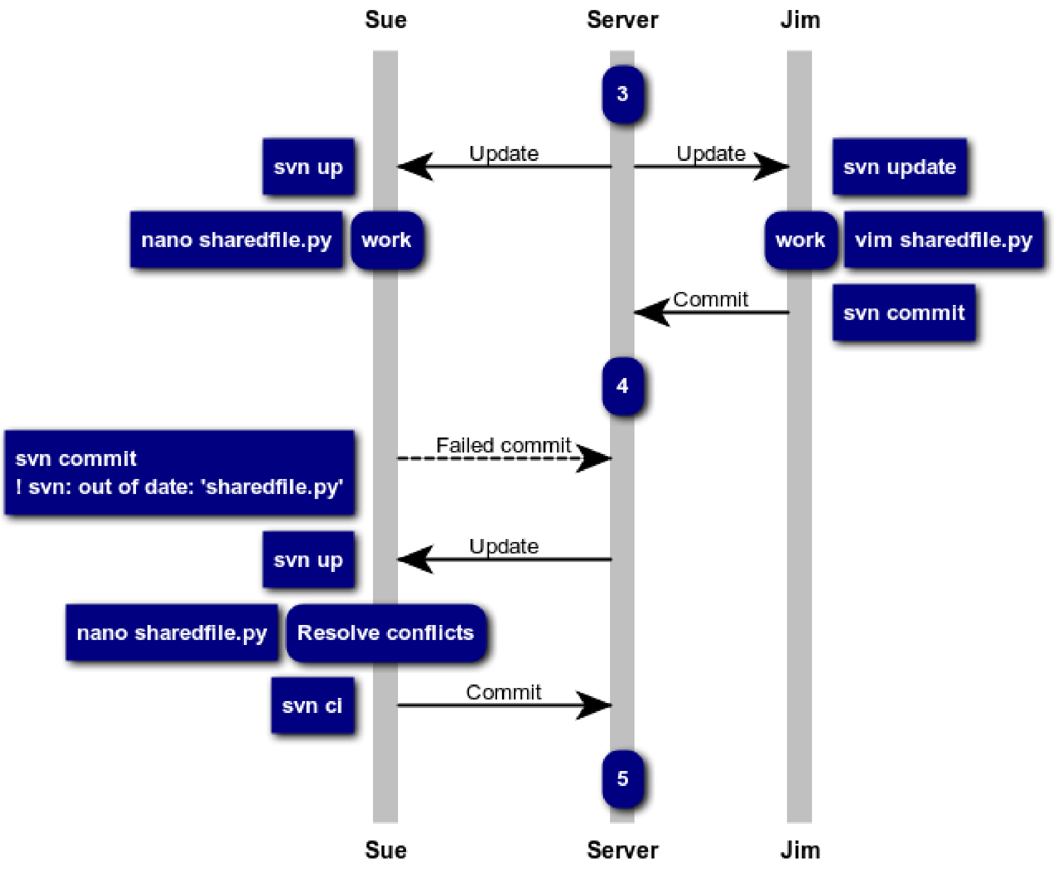


Figure 4: Conflicted workflow for SVN

```
svn update
> Conflict discovered in 'sharedfile.py'.
> Select: (p) postpone, (e) edit, (mc) mine-conflict ...
```

If you choose `(e)` the conflicted file will look something like:

```
Whatever was in the file before the conflicted bit
<<<<<<< .mine
Sue's content
=======
Jim's content
>>>>>>> .r4
Content after the conflicted bit
```

It is your duty to edit this to fix conflicts, then save.

2.7 Revisiting history

Update to a particular revision:

```
svn up -r 3
```

See the differences between your working area and a revision

```
svn diff #To most recent version
svn diff -r 3
```

See what you've changed:

```
svn status
```

Get rid of changes to a file:

```
svn revert myfile.py
```

3 Distributed Version Control

3.1 Distributed versus centralised

Centralised	Distributed
Server has history	Every user has full history
Your computer has one snapshot	Many local branches
To access history, need internet	History always available
You commit to remote server	Users synchronise histories
cvs, subversion(svn)	git, mercurial (hg), bazaar (bzr)

3.2 Distributed VCS in principle

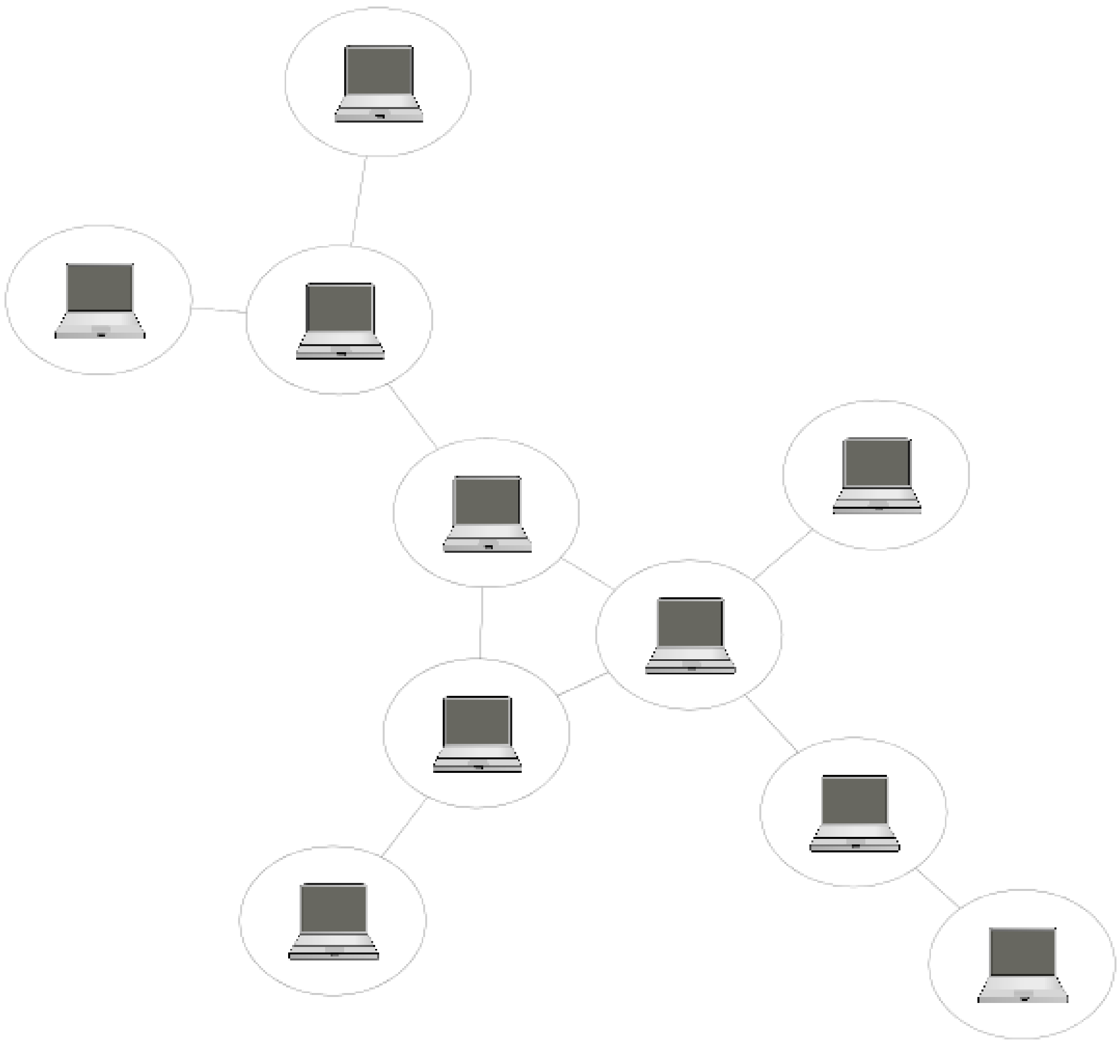


Figure 5: How distributed VCS works in principle

3.3 Distributed VCS in practice

3.4 Pragmatic Distributed VCS

Subversion	Git
`svn checkout <URL>`	`git clone <URL>`
`svn commit`	`git commit -a; git push`
`svn up`	`git pull`

`svn status`	`git status`
`svn diff`	`git diff`

3.5 Why Go Distributed?

- Easy to start a repository (no server needed)
- Easy to start a server
- Can work without internets
- Better merges
- Easy branching
- More widespread support

3.6 Why Not Go Distributed?

- More complex commands
- More confusing

3.7 Distributed VCS concepts

- Each revision has a parent that it is based on
- These revisions form a graph
- The most recent revision in each copy is the HEAD
- Each revision has a unique hash code
- In Sue's copy, revision 43 is ab3578d6
- Jim thinks that is revision 38
- When you *pull* from a *remote* the histories might conflict
- Histories are *merged* together

3.8 A revision graph

3.9 Distributed VCS concepts (2)

- You have a *working copy*
- You pick a subset of the changes in your working copy to add to the next commit
- Changes to be included in the next commit are kept in a **staging area** (a.k.a. **index**)
- When you commit you commit:
- From the staging area
- To the local repository
- You **push** to **remote** repositories to share or publish
- You **pull** (or fetch) to bring in changes from a remote

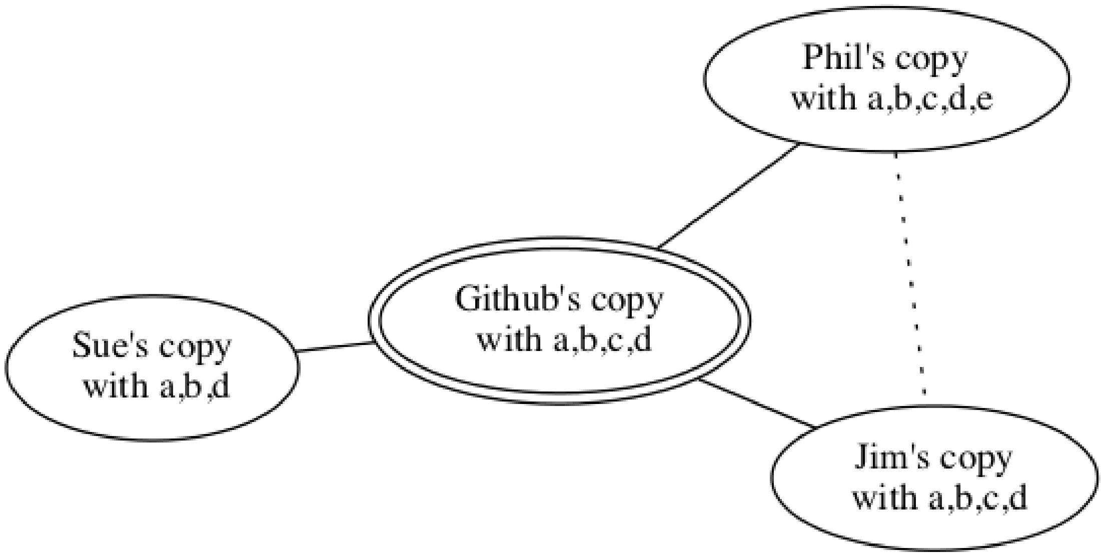


Figure 6: How distributed VCS works in practice

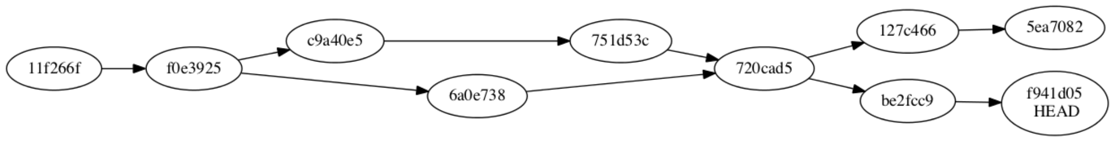


Figure 7: Revisions form a graph

3.10 The Levels of Git

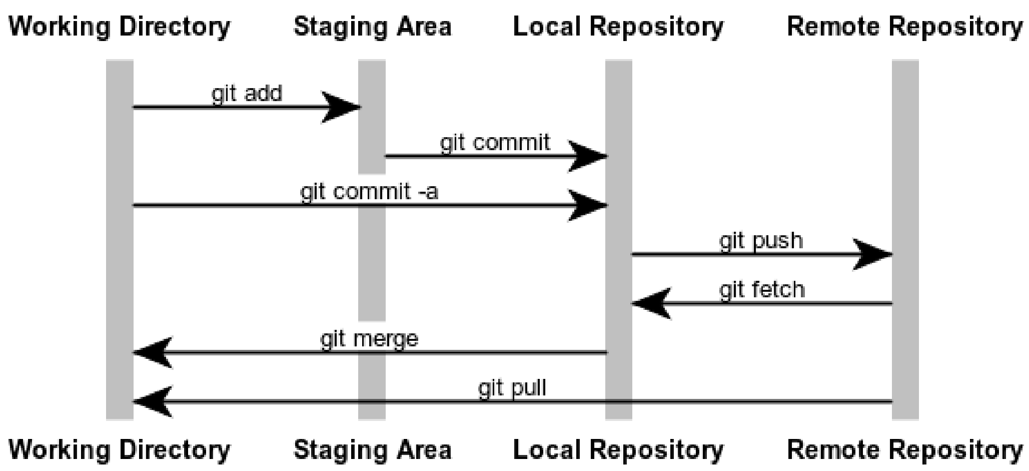


Figure 8: The relationship between the staging area, working directory, and repositories in git.

4 Using Git

4.1 Distributed VCS Solo Workflow

4.2 Distributed VCS With Publishing

4.3 Distributed VCS in teams without conflicts

4.4 Distributed VCS in teams with conflicts

4.5 Working with multiple remotes

```
git remote add sue ssh://sue.ucl.ac.uk/somerepo
   # Add a second remote
git remote
   # List available remotes
git push sue
   # Push to a specific remote
   # Default is origin
```

5 Branches

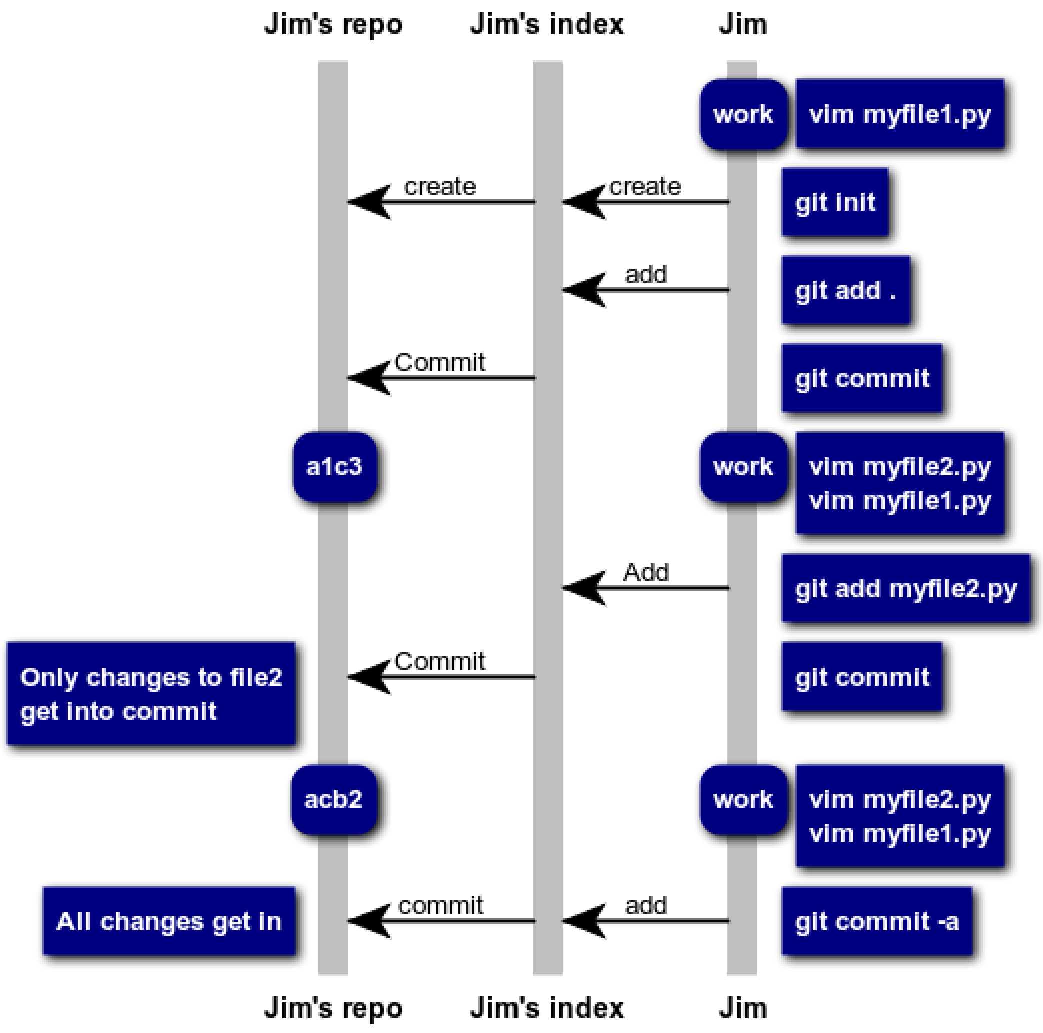


Figure 9: Working alone with git

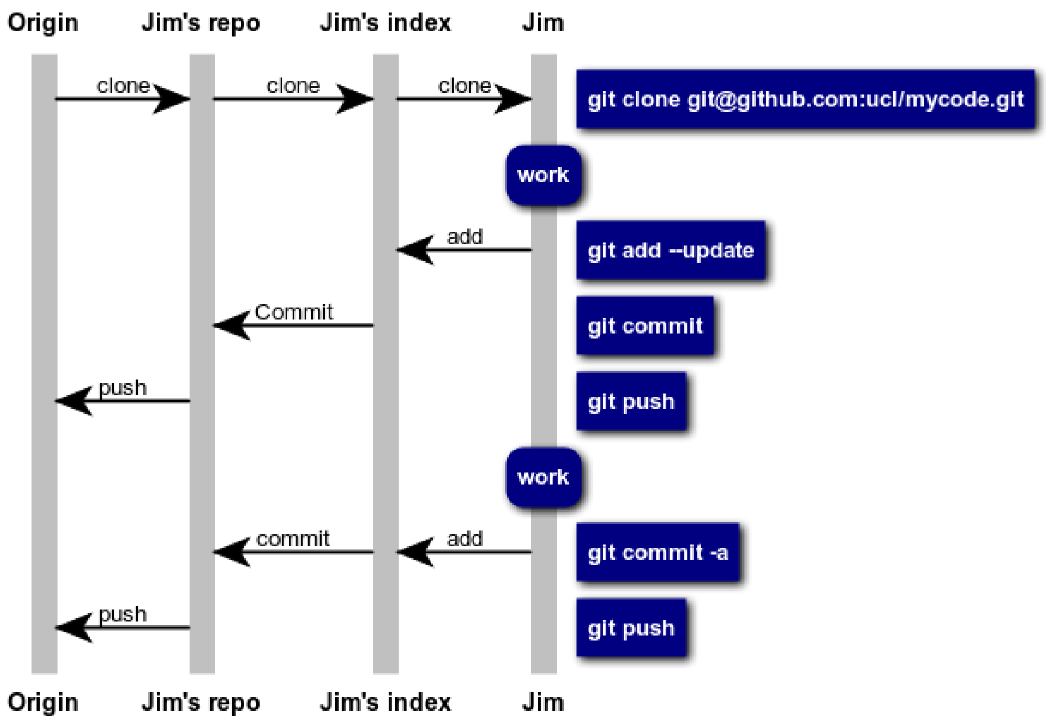


Figure 10: Publishing with git

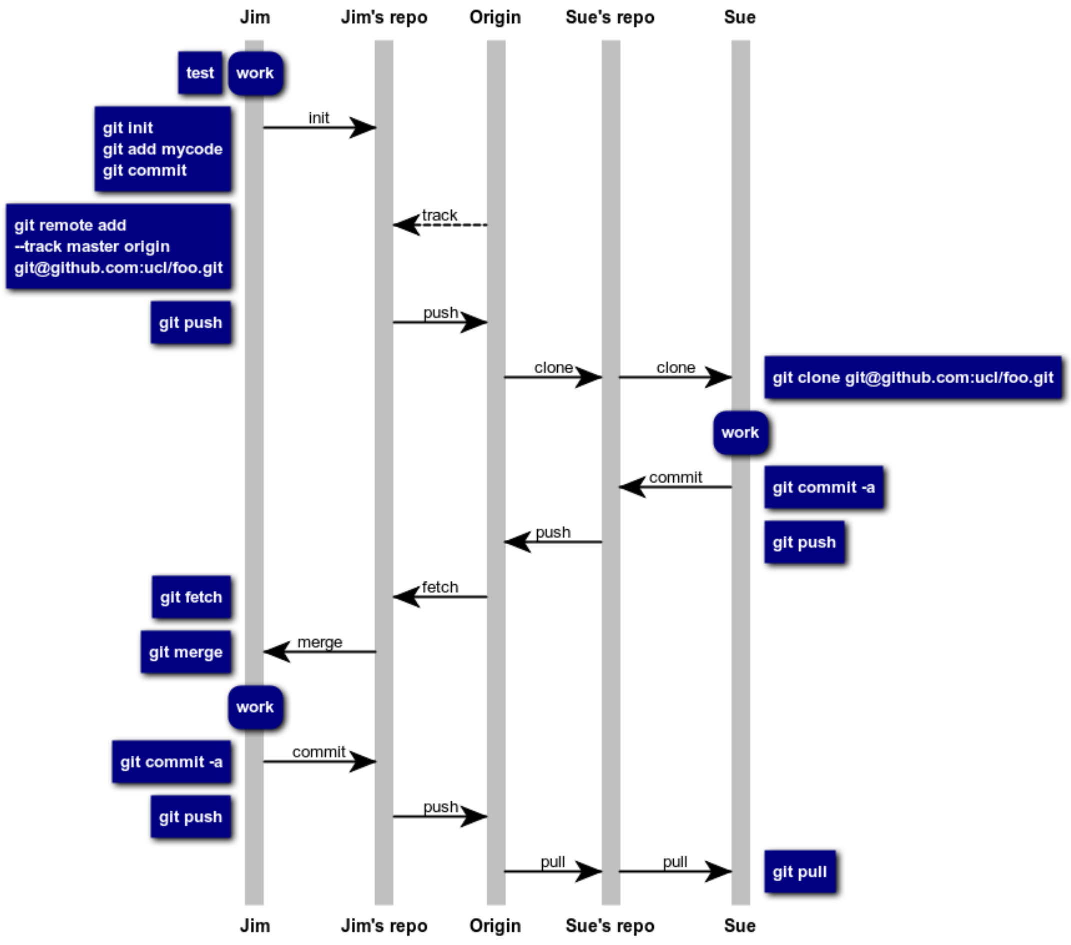


Figure 11: Teamworking in git

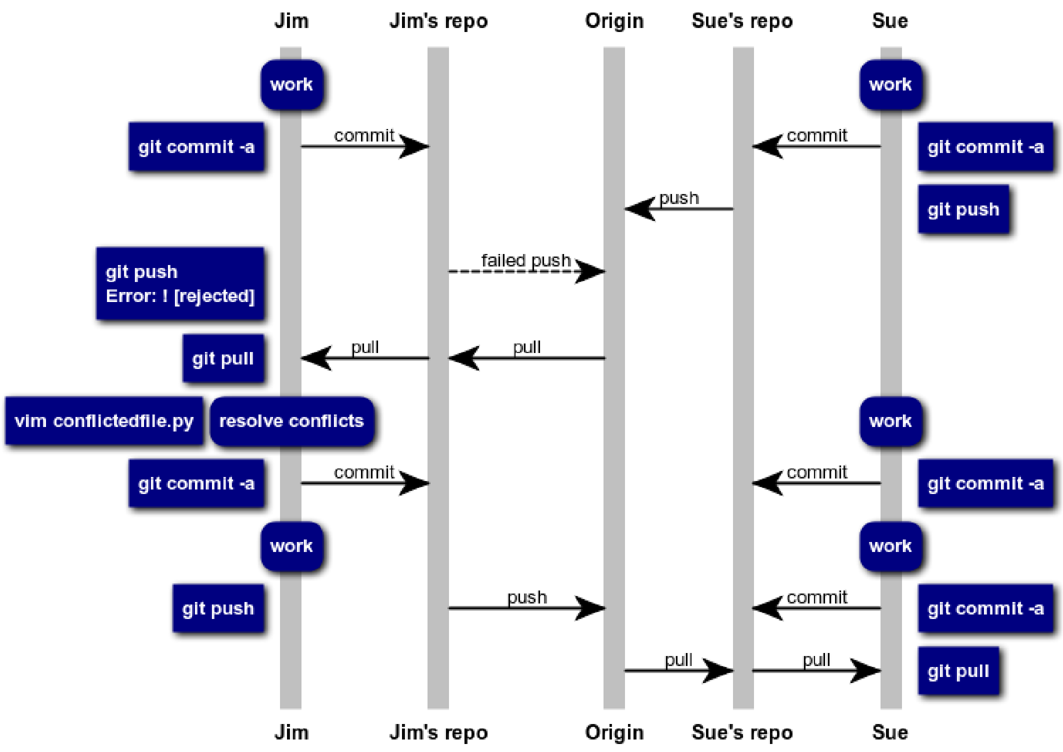


Figure 12: Teamworking in git with conflicts

5.1 Working with branches

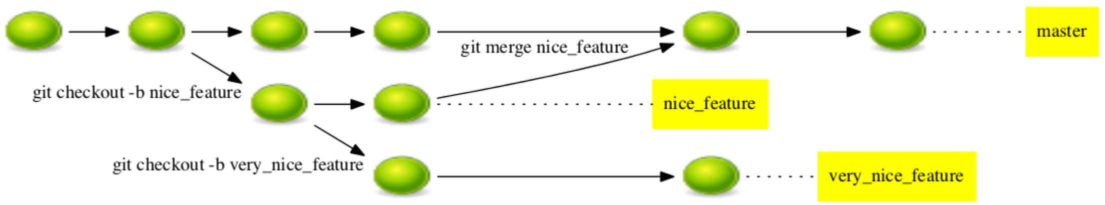

git merge nice_feature
master
git checkout -b nice_feature
nice_feature
git checkout -b very_nice_feature
very_nice_feature

Figure 13: Using branches

5.2 Working with branches in git

```
git branch # Tell me what branches exist

   * master # Asterisk tells me which one
     experiment # I am currently on

git checkout -b somebranch # Make a new branch
git checkout master # Switch to an existing branch
```

5.3 Sharing branches

```
git push -u origin experiment # Share a recently
                              # made branch
git push origin experiment #Republish a branch
git branch -r #Discover remote branches
git checkout origin/some_branch #Get a branch
                                #from a remote
```

5.4 Merging branches

```
git checkout master # Switch to master branch
git merge experiment # Merge the branch in
git branch -d experiment # Delete branch locally
git push --delete experiment # Delete published branch
```

5.5 A good branch strategy

- A `production` branch: code used for active work
- A `develop` branch: for general new code

- `feature` branches: for specific new ideas
- `release` branches: when you share code with others
- Useful for isolated bug fixes

5.6 Tagging

Easy to read labels for revisions Produce real results *only* with tagged revisions

```
git tag -a v1.3
git push --tags
```

5.7 Branching and tagging in subversion

- Subversion doesn't have real branches and tags
- Instead, each is a separate whole copy
- But you can still merge between copies

6 Hosting Servers

6.1 Hosting a server

- Any repository can be a remote for pulls
- Can pull/push over shared folders or ssh
- Pushing to someone's working copy is dangerous
- Use `git init --bare` to make a copy for pushing
- You don't need to create a "server"

6.2 Hosting a server in the cloud

- Many online services
- Github, bitbucket, sourceforge…
- I recommend GitHub