15-410
“...Goals: Time Travel, Parallel Universes...”

Version Control
Sep. 19, 2018

Dave Eckhardt
Dave O'Hallaron
Nathaniel Filardo
Zach Anderson (S '03)
Outline

Motivation
Repository vs. Working Directory
Conflicts and Merging
Branching
A Brief Introduction to git
Goals

Working together should be easy

Time travel
  - Useful for challenging patents
  - Very useful for reverting from a sleepless hack session

Parallel universes
  - Experimental universes
  - Product-support universes
Goal: Shared Workspace

Reduce development latency via parallelism

Alice  awesome.c  Bob

Charlie  awesome.c  Devon
Goal: Time Travel

Retrieving old versions should be easy.

Once Upon A Time…

Alice: What happened to the code? It doesn’t work.

Charlie: Oh, I made some changes. My code is 1337!

Alice: Rawr! I want the code from last Tuesday!
Goal: Parallel Universes

Safe process for implementing new features.

- Develop bell in one universe
- Develop whistle in another
- Don't inflict B's core dumps on W
- Eventually produce bell-and-whistle release
How?

*Keep a global repository for the project.*
Definitions

**Commit / Version / Snapshot**
- Contents of some files at a particular point in time

**Project**
- A “sequence” of versions
  - (not really)

**Repository**
- Directory where projects are stored
The Repository

Stored in group-accessible location
- Old way: file system
- Modern way: “repository server”

Versions in repository visible group-wide
- Whoever has read access
- “Commit access” often separate
How?

Keep a global repository for the project.  
*Each user keeps a working directory.*
The Working Directory

Many names ("sandbox")
Where revisions happen
Typically belongs to one user
Versions are checked out to here
New versions are checked in from here
How?

Keep a global repository for the project. Each user keeps a working directory. *Concepts of checking out, and checking in*
Checking Out. Checking In.

**Checking out**
- A version is copied from the repository
  - Typically “Check out the latest”
  - Or: “Revision 3.1.4”, “Yesterday noon”

**Work**
- Edit, add, remove, rename files

**Checking in**
- Working directory ⇒ repository *atomically*
- Result: new version
Checking Out. Checking In.

Repository

- 
- 

Working Directory

v0.1

check out

v0.1 copy
Checking Out. Checking In.

Repository

○

○

v0.1

Working Directory

v0.1 copy

mutate

v0.1++
Checking Out. Checking In.

Repository

v0.1

Working Directory

v0.2

check in

v0.1++
How?

Keep a global repository for the project.
Each user keeps a working directory.
Concepts of checking out, and checking in

Mechanisms for merging
Conflicts and Merging

Two people check out.
- Both modify foo.c

Each wants to check in a new version.
- Whose is the *correct* new version?
Conflicts and Merging

Conflict

- Independent changes which “overlap”
- *Textual* overlap detected by revision control
- *Semantic* conflict cannot be

Merge displays conflicting updates per file
Pick which code goes into the new version
- A, B, NOTA

Story now, real-life example later
Alice Begins Work

Alice

v0.2

copy

v0.2

fix b#1

Repository

v0.2

Bob
Bob Arrives, Checks Out

Alice
v0.2
copy

Repository
v0.2

Bob
v0.2
copy

v0.2
fix b#1
Alice Commits, Bob Has Coffee

Alice

v0.2

copy

v0.2

fix b#1

Repository

v0.2

Bob

v0.2

copy

v0.3
Bob Fixes Something Too

Alice

v0.2

copy

v0.2

fix b#1

Repository

v0.2

Bob

v0.2

copy

v0.2

fix b#7
Wrong Outcome

Alice
- v0.2 copy
- v0.2 fix b#1

Repository
- v0.2
- v0.3

Bob
- v0.2 copy
- v0.2 fix b#7
“Arguably Less Wrong”

Alice
- v0.2 copy
- v0.2
  - v0.2 fix b#1
  - v0.3
  - v0.4

Repository
- v0.2

Bob
- v0.2 copy
- v0.2 fix b#7

Merge, Bob, Merge!

Alice

v0.2

copy

v0.2

fix b#1

Repository

v0.2

Bob

v0.2

copy

v0.2

fix b#1

v0.3

fix b#7

fix b#1

fix b#7
Committing Genuine Progress

Alice

v0.2 copy

v0.2 fix b#1

Repository

v0.2

v0.3

v0.4

Bob

v0.2 copy

v0.2 fix b#7

fix b#1

fix b#7
How?

Keep a global repository for the project.
Each user keeps a working directory.
Concepts of checking out, and checking in
Mechanisms for merging
Mechanisms for branching
Branching

A branch is a sequence of versions
   - (not really...)

Changes on one branch don't affect others

Project may contain many branches

Why branch?
   - Implement a new “major” feature
   - Begin a temporary independent sequence of development
Branching
Branch Life Cycle

“The Trunk”
- “Release 1.0”, “Release 2.0”, ...

Release 1.0 *maintenance* branch
- After 1.0: 1.0.1, 1.0.2, ...
- Bug-fix updates as long as 1.0 has users

Internal *development* branches
- 1.1.1, 1.1.2, ...
- Probably 1.1.1.client, 1.1.1.server
Branch Life Cycle

“Development excursion” branch model
  - Create branch to fix bug #99 in v1.1
  - One or more people make 7 changes
  - Branch “collapses” back to trunk
    - Merge 1.1.bug99.7 against 1.1.12
    - Result: 1.1.13
    - There will be no 1.1.bug99.8
      - In some systems, there can't be
Branch Life Cycle

“Controlled isolation” branch model

- Server people work on 1.3.server
  - Fix server code
  - Run stable client test suite vs. new server
- Client people work on 1.3.client
  - Fix client code
  - Run new client test suite vs. stable server
- Note
  - Branches do *not* collapse after one merge!
Recommendation for 15-410

You can use Mercurial (hg) or SVN if you like

Current TA favorite: git
  - It can do what you need
    - (plus a vast array of things you don't need)
  - It's unlikely to suddenly vanish
  - It's “very likely” (25%?) to be chosen by your next boss

Be careful about online git/hg providers
  - Your work will probably be public and searchable – see syllabus!
git Repositories

Local repo and project files

Global repo, typically "bare"

Local repo and project files
Getting started

Create global repo (one partner, exactly once):

$ cd ~/410/$USER/mygroup/REPOSITORY

$ git init --bare p2

Creates a “.git” subdirectory
- Contains a hash-tree of all entities ever seen by the version control system.
- Also contains things like config, heads, remotes, and other goodies.

Clone global repo in working directory (each partner)

$ cd ~/410/$USER/scratch

$ git clone file://$HOME/410/$USER/mygroup/REPOSITORy/p2
Getting started

Make first commit (one partner)

$ cd ~/410/$USER/scratch/p2
$ cp ~/410/pub/gitignore .gitignore
$ git add .gitignore
$ git commit -m "Initial commit"
$ git push origin master –u

Get updated copy of repo (other partner)

$ git pull
Populating the world

Adding new files for git to track

$ git add file1 file2 ...
  - To add *every* file in a directory
    $ git add dir/
    • Rarely what you want!!!
Making Local Commits

$ git commit –a –m “commit message”
or
$ git commit –a

- Fires up your $EDITOR and asks you for commentary.

- Adds a new snapshot to LOCAL repository's history
  - Your partner has no idea that this has happened.
Sharing Your Work

How do local changes become non-local?

**Pull**

$ git pull

Pulls changes from a remote repository.

- Git has a notion of “default remote”

**Push**

$ git push

- Pushes changes from the local repo into the remote.
Conflicts and Merging

Suppose this hello.c is in the repository:

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```
Conflicts and Merging

Suppose Alice and Charlie each check out this version, and make changes:

Alice's Version
```
#include <stdlib.h>
#include <stdio.h>
#define SUPER 0

int main(void)
{
    /* prints "Hello World"
       to stdout */
    printf("Hello World!\n");
    return SUPER;
}
```

Charlie's Version
```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    /* this, like, says hello, and stuff */
    printf("Hello Hercules!\n");
    return 42;
}
Conflicts and Merging

Suppose Alice “checks in” first

$ git commit -a ⇒ ok
$ git push ⇒ ok

Now Charlie...

$ git commit -a ⇒ ok, but invisible to Alice
$ git push ⇒ fail!
$ git pull ⇒ Alice's changes “appear”
$ ...edit...
$ git commit -a && git push
Merge Mutilation

There wasn't a conflict “here”
Conflicts are entirely textual!

#include <stdlib.h>
#include <stdio.h>
#endif
#define SUPER 0

int main(void) {

<<<<<<<<<<< HEAD:hello.c
  /* this, like, says hello, and stuff */
  printf("Hello Hercules!\n");
  return 42;

========
  /* prints "Hello World" to stdout */
  printf("Hello World!\n");
  return SUPER;

>>>>>>> 12341234abcd5656efef787890900123456789ab:hello.c
}

Division between conflicting commits
Branching and Merging

To create a branch and switch to it:

$ git branch hotfix
$ git checkout hotfix

or

$ git checkout -b hotfix

To merge hotfix back in to the master (default initial) branch:

$ git checkout master
$ git merge hotfix

To view your branches and last commit on each:

$ git branch -v
Information

To get a summary of changes to files and which files are staged to be committed:

$ git status

To ask about changes in the past:

$ git log
Suggestions

“Commit early and often”
- So you can locally track history, roll back...

“Push good news”
- Build, test, push to shared space

“Pull often”
- Big merges are painful merges

Develop a convention for commit entries
- Type of revision (bug-fix, commenting, etc.)
- Meaningful, short descriptions
Suggestions

“Backups”
- “push” and “pull” do a lot
- Snapshotting your central repository every now and then may be smart

When to branch?
- Bug fixing?
  - Check out, fix, check in to same branch
- Trying ZFOD fork since regular fork works?
  - Branching probably a good idea.
- For “backed up but not released to partner”
Summary

We can now:
- Create projects
- Check source in/out
- Merge and branch
- Share our work

See GIT documentation
- 15-410 “git intro” web page – specific help
- Lots of documentation online (many features)
- Search for “git tutorial”
Further Reading

Pro Git
- This is a book-length object by Scott Chacon
- It is available free online (but...)
- It is a good way to actually understand git

“Git for Computer Scientists”
“Git from the Bottom Up”
“Git Magic”
“How to use git to lose data”
  - This is a locally-produced heresy