15-410
"...Goals: Time Travel, Parallel Universes..."

Version Control
Feb. 2, 2011

Dave Eckhardt

Nathaniel Filardo

Zach Anderson (S '03)

Disclaimer

This lecture will mention one SCMS

- git

You don't need to use git

- Not even if "all the TA's do"

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**

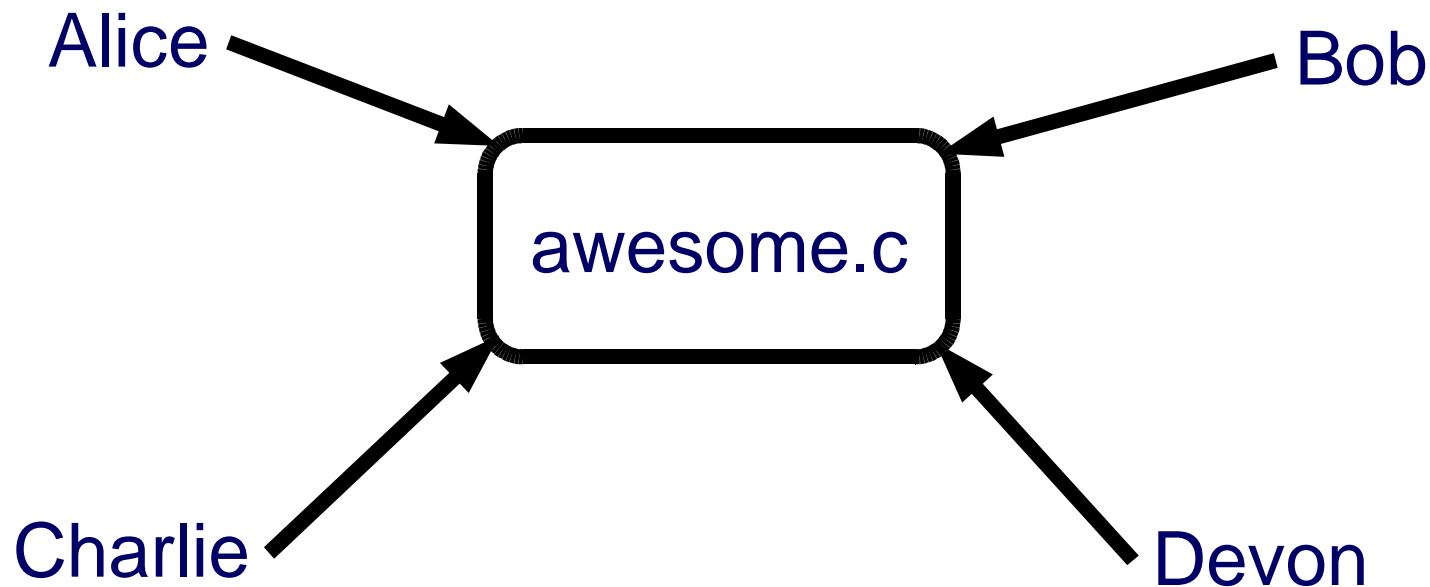– *Very* **useful for reverting from a sleepless hack session**

Parallel universes

– **Experimental universes**

– **Product-support universes**

Goal: Shared Workspace

Reduce development latency via parallelism
- **[But: Brooks, <u>Mythical Man-Month</u>]**

Alice

Bob

awesome.c

Charlie

Devon

5

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

7

How?

Keep a global repository for the project.

The Repository

Version / Revision / Configuration

- Contents of some files at a particular point in time
- aka "Snapshot"

Project

- A "sequence" of versions
  - (not really)

Repository

- Directory where projects are stored

9

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

13

Checking Out. Checking In.

Checking out

- A version is copied from the repository
  - Typically "Check out the latest"
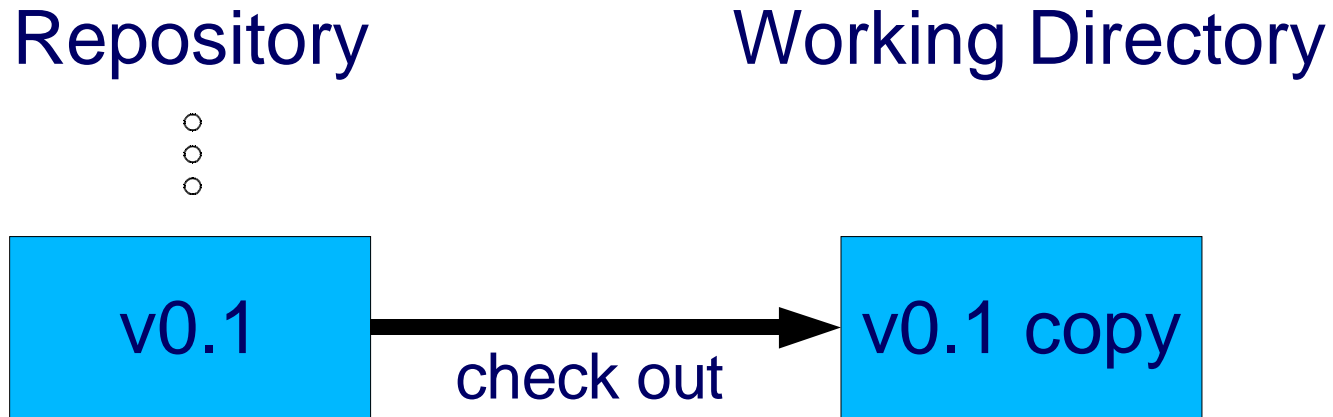 - Or: "Revision 3.1.4", "Yesterday noon"

Work

- Edit, add, remove, rename files

Checking in

- Working directory ⇒ repository *atomically*
- Result: new version

14

Checking Out. Checking In.

Repository   Working Directory

| v0.1 | → v0.1 copy |

check out

Checking Out. Checking In.

Repository   Working Directory

v0.1

v0.1 copy

⋮ mutate

v0.1++

Checking Out. Checking In.
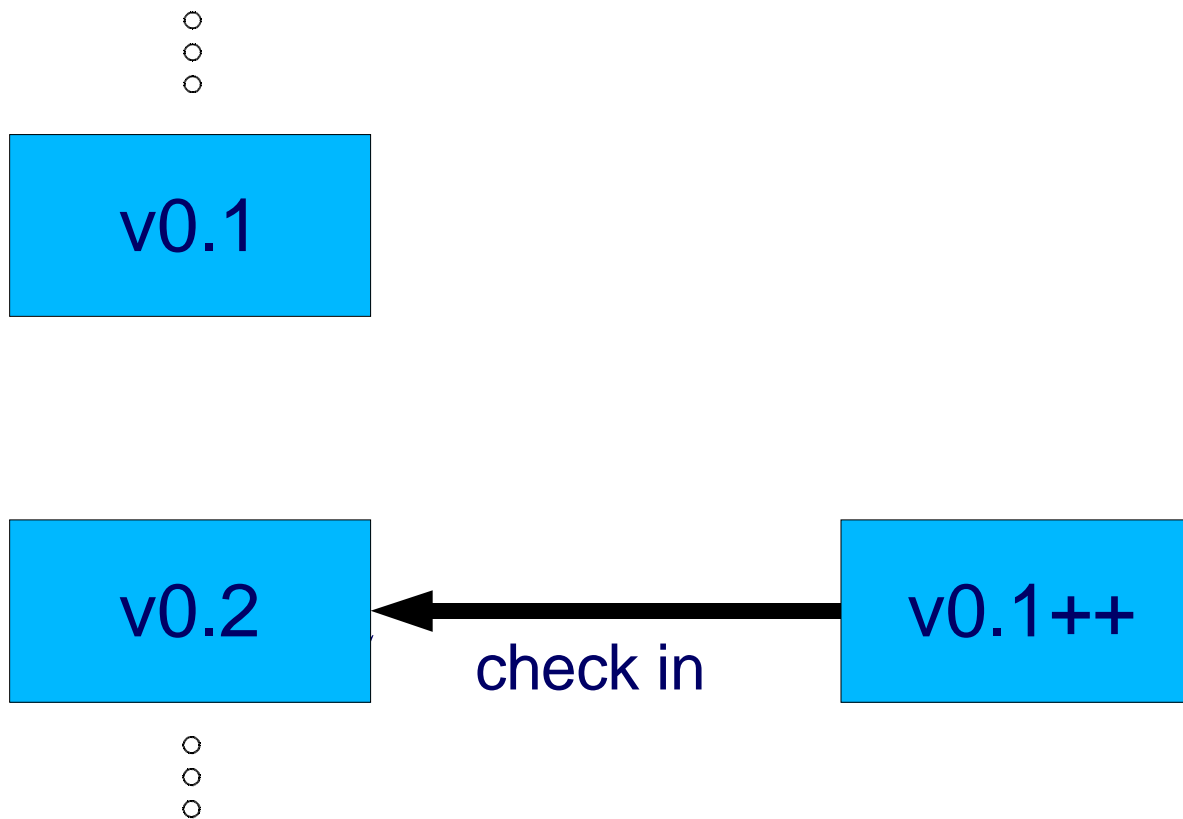
Repository   Working Directory

v0.1

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"
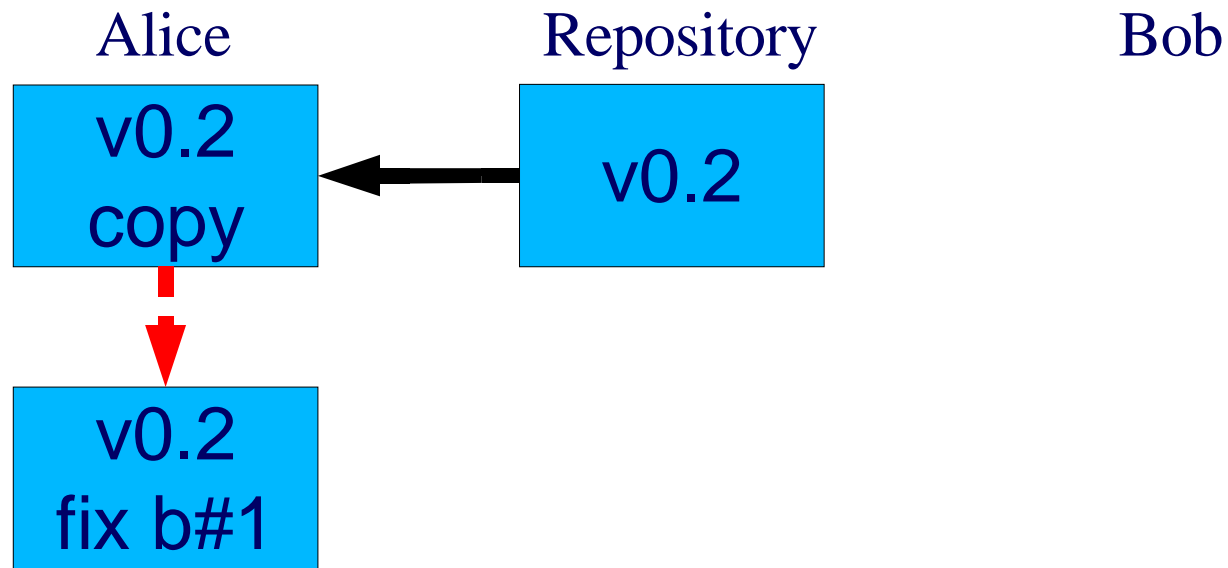- *Textual* overlap detected by revision control
- *Semantic* conflict cannot be

Merge displays conflicting updates per file
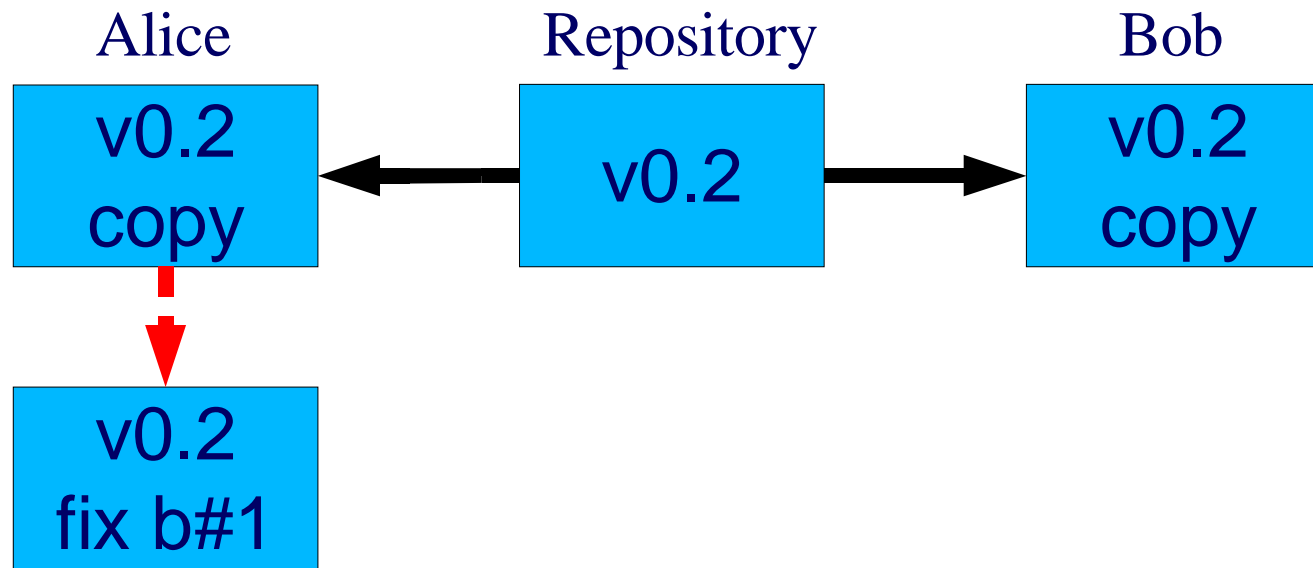
Pick which code goes into the new version
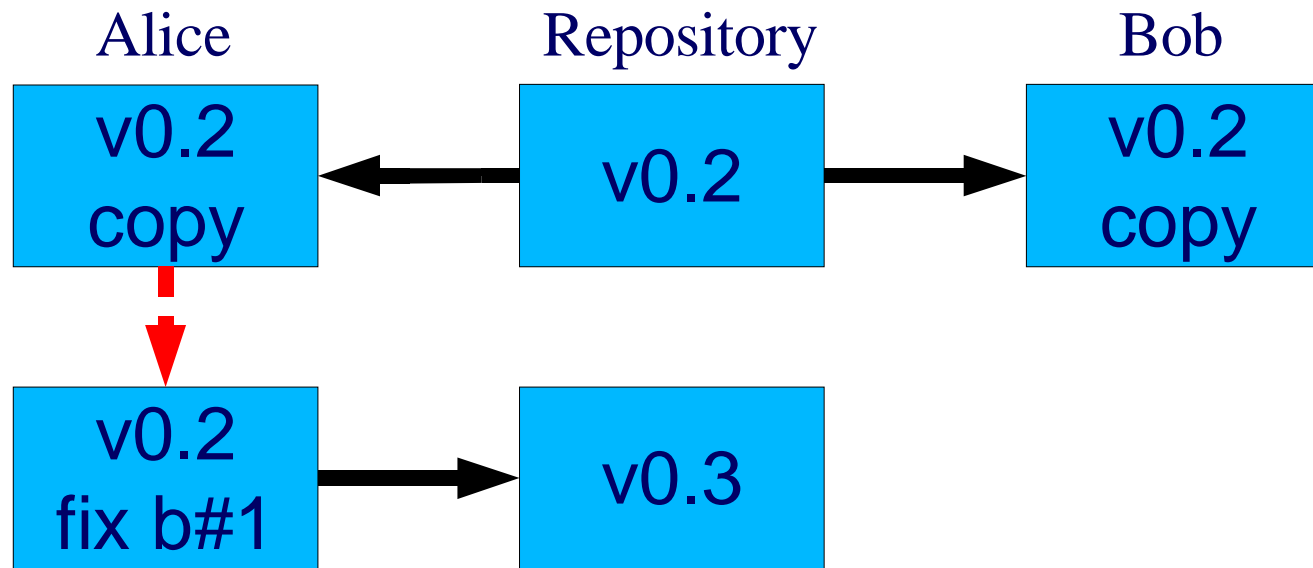
- A, B, NOTA
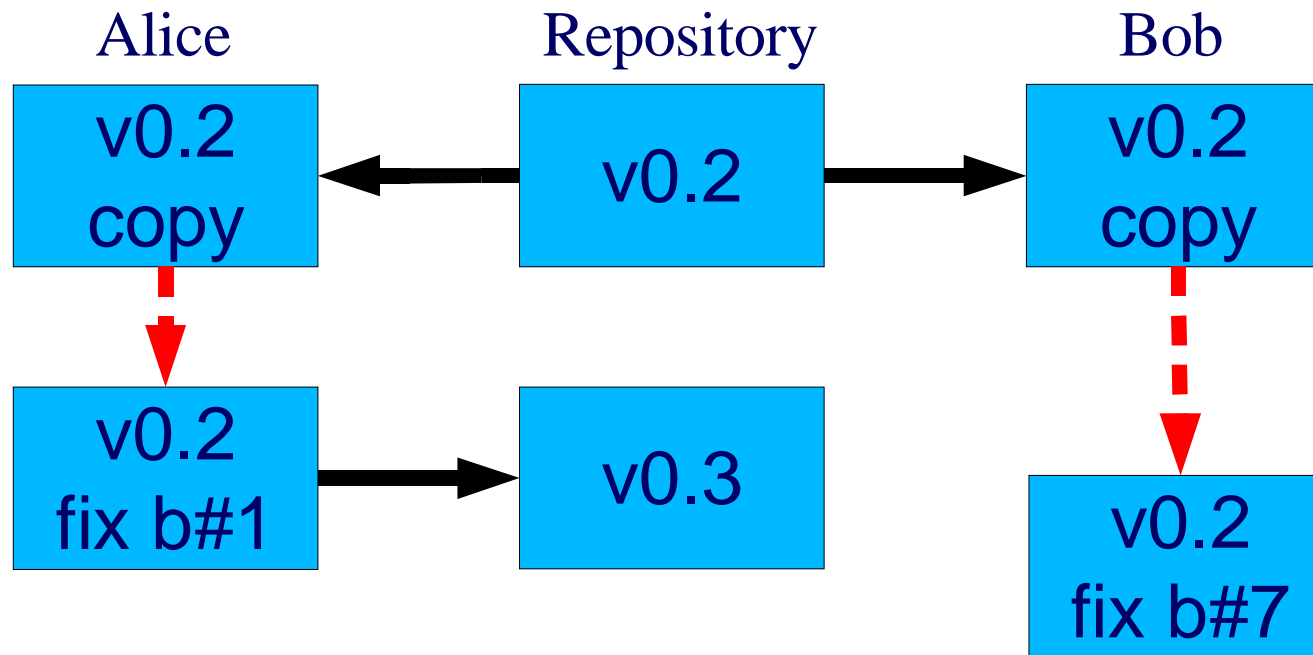
Story now, real-life example later

Alice Begins Work

Alice   Repository   Bob

v0.2
copy

v0.2

v0.2
fix b#1

Bob Arrives, Checks Out

Alice   Repository   Bob

| v0.2 copy | ←→ | v0.2 | ←→ | v0.2 copy |

v0.2
fix b#1

Alice Commits, Bob Has Coffee

Alice   Repository   Bob

| v0.2 copy | ← → | v0.2 | → | v0.2 copy |

| v0.2 fix b#1 | → | v0.3 |

Bob Fixes Something Too

Alice

Repository

Bob

Wrong Outcome

Alice   Repository   Bob

v0.2 copy	← v0.2 →	v0.2 copy
v0.2 fix b#1	→ v0.3 ←	v0.2 fix b#7
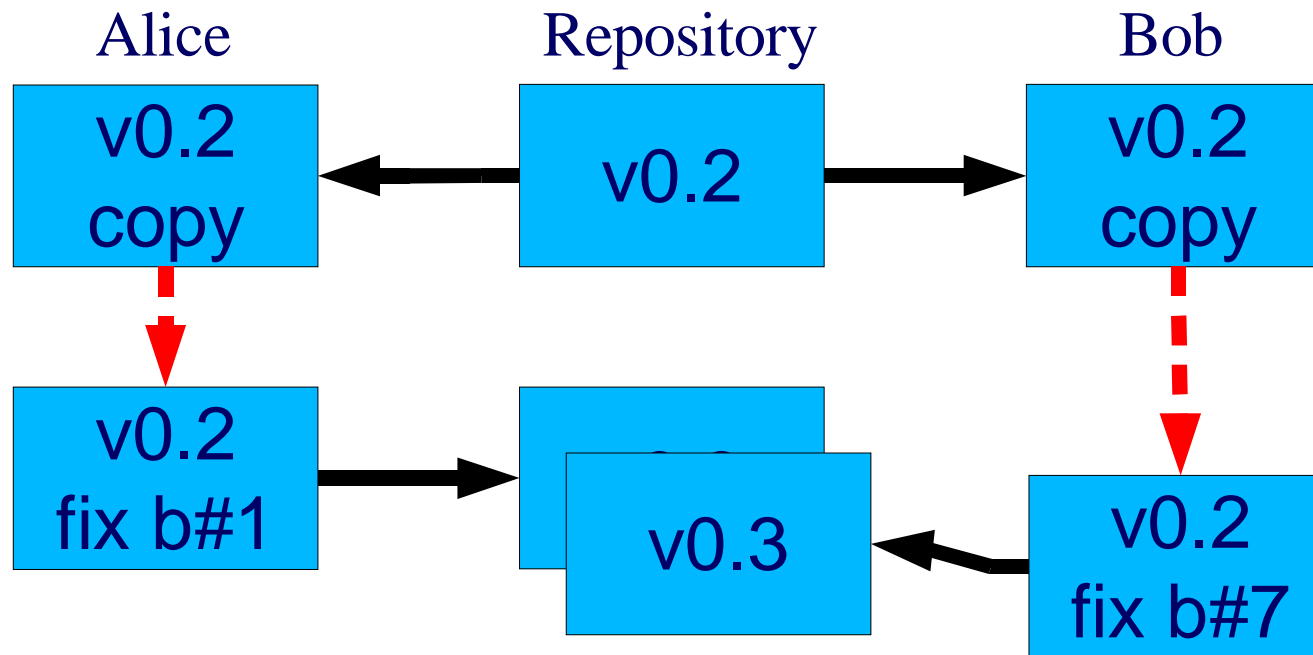
"Arguably Less Wrong"

Merge, Bob, Merge!

Alice
Repository
Bob

v0.2 copy ← v0.2 → v0.2 copy

v0.2 fix b#1 → v0.3

v0.2 fix b#7

fix b#1 fix b#7
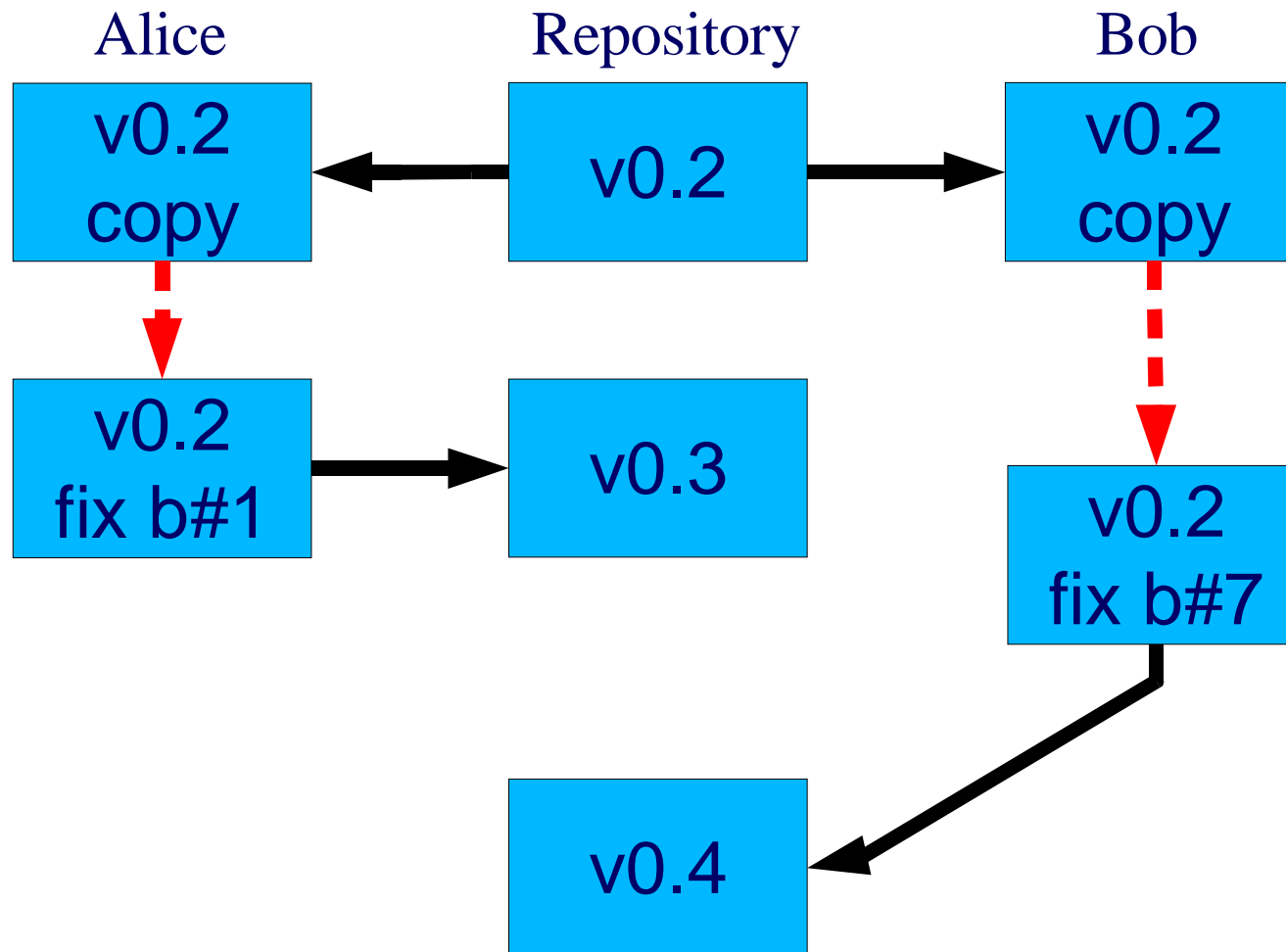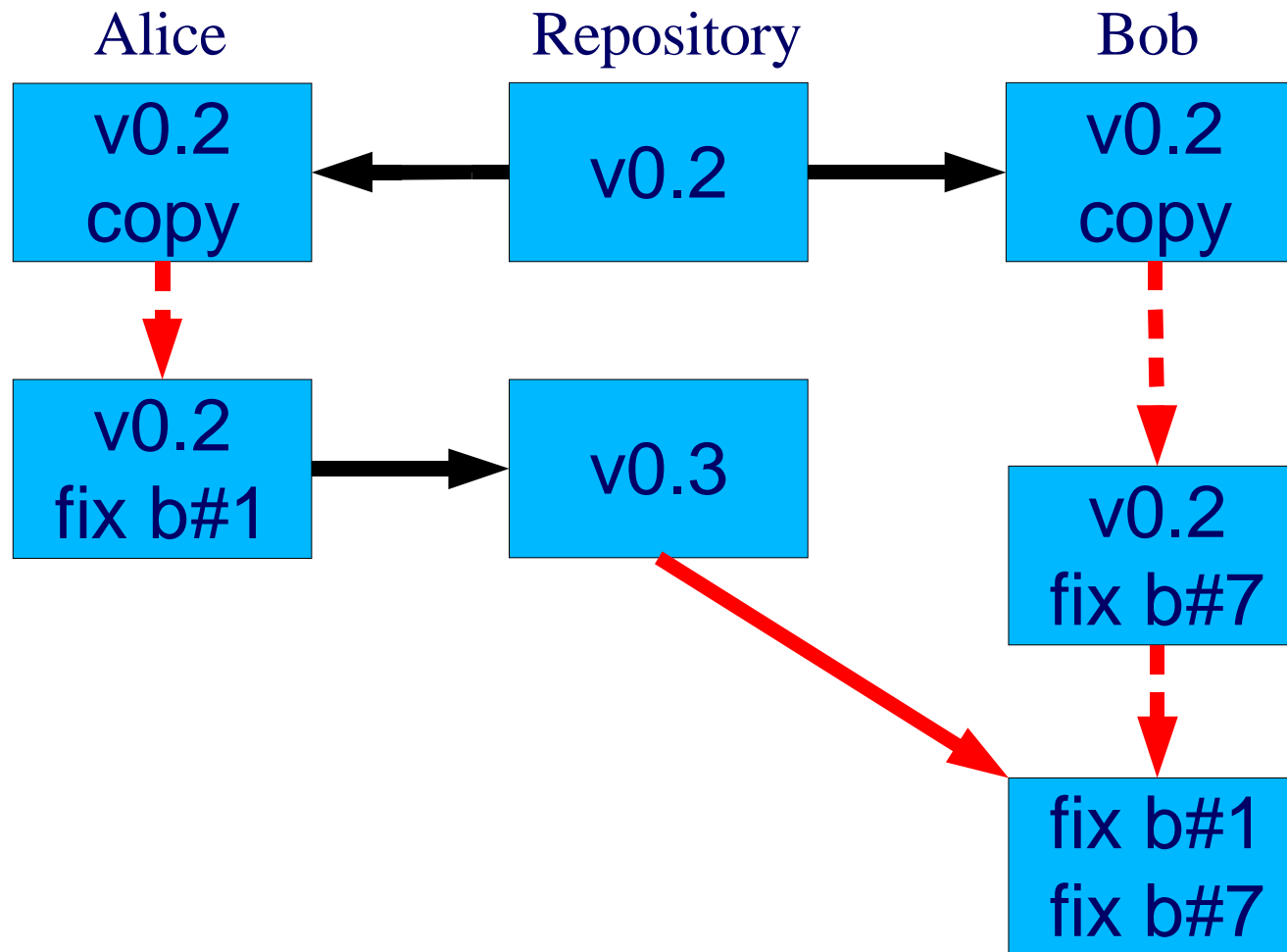
Committing Genuine Progress
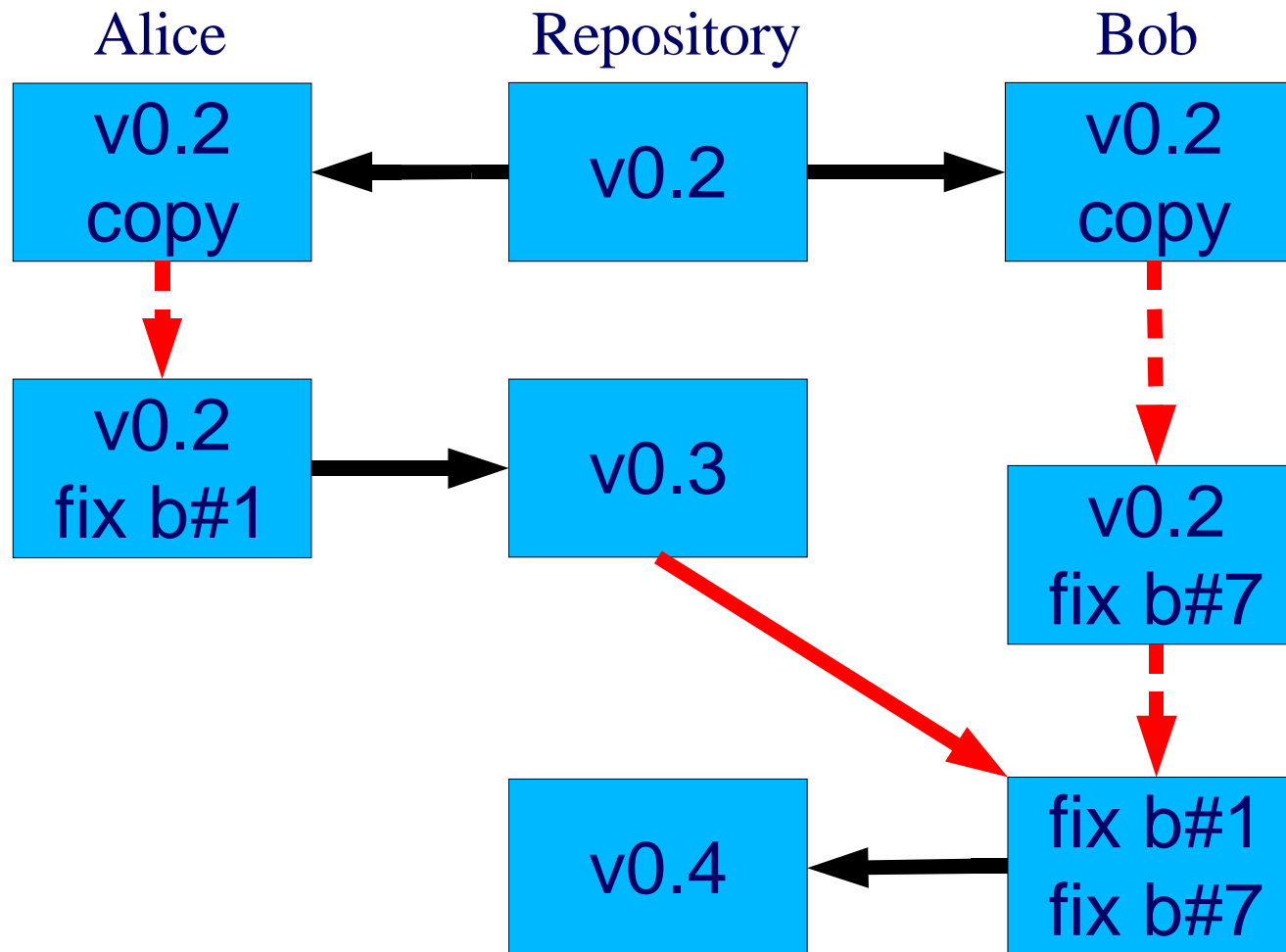
How?

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

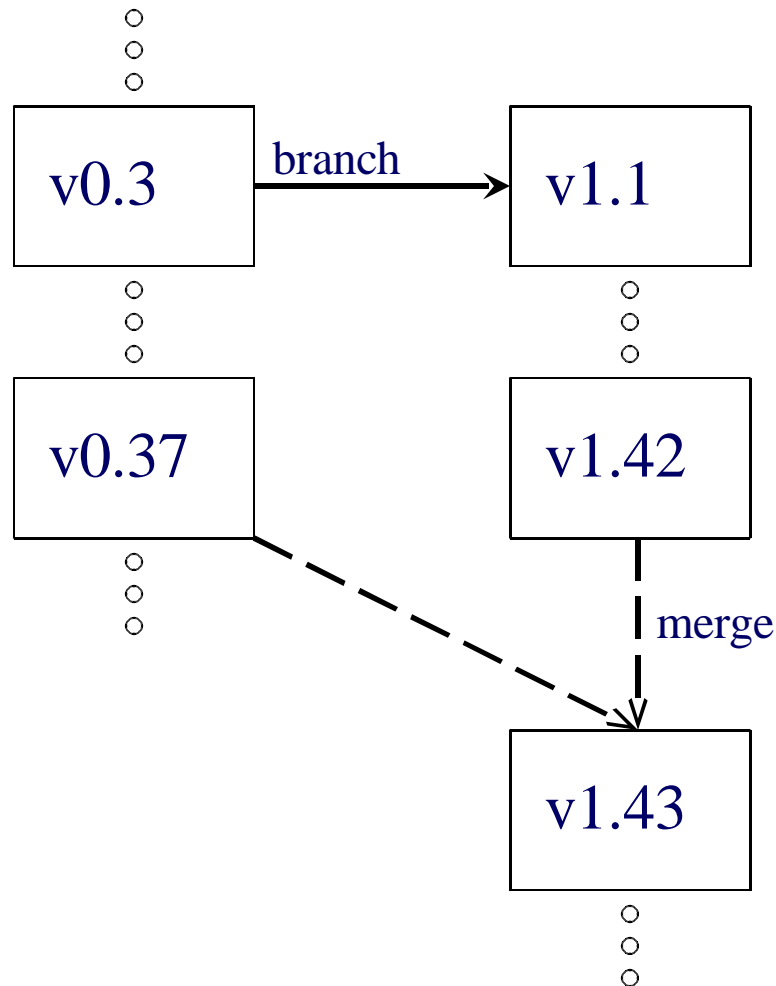**A branch is a *sequence of versions***

- (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

v0.3 — branch → v1.1

v0.37

v1.42

v1.43

The actual branching and merging take place in a particular user's working directory, but this is what such a sequence would look like to the repository.

merge

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**

32

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

33

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!

34

Branch Life Cycle

"Controlled isolation" branch model

- Periodic merges - example
    - 1.3.server.45, 1.3.12 ⇒ 1.3.13
    - 1.3.client.112, 1.3.13 ⇒ 1.3.14
    - Each group can keep working while one person "pushes up" a version to the parent
- When should server team "pull down" 1.3.14 changes?
    - 1.3.server.47, 1.3.14 ⇒ 1.3.server.48?
    - 1.3.server.99, 1.3.14 ⇒ 1.3.server.100?

35

Branch Life Cycle

Successful development branch
- Merged back to parent
- No further versions

Unsuccessful development branch
- Some changes pulled out?
- No further versions

Maintenance branch
- "End of Life": No further versions

Are Branches *Deleted?*

Consider the repository "data structure"
- Revisions of each file (coded as deltas)
- Revisions of the directory tree

Branch delete
- *Complicated* data structure update
  - [Not a well-tested code path]
- Generally a bad idea
  - History could *always* be useful later...

37

Source Control Opinions

CVS

- still widely used
- mature, lots of features
- default behavior often wrong

SubVersion (svn)

- SVN > CVS (design)
- SVN > CVS (size)
- Doesn't work in AFS
- Yes, it does
- No, it doesn't?

Perforce

- commercial
- reasonable design
- works well (big server)

BitKeeper

- ~~Favored by Linus Torvalds~~
- "Special" license restrictions

git

- Favored by Linus Torvalds

Source Control Opinions

Others

- Mercurial ("hg")
  - Merge-once branches
- Bazaar ("bzr")
- Monotone
- arch/tla
- darcs ("patch algebra")

Generally

- Promising plans
- Some rough edges
- *Many* use cases covered
- Ready yet?

Eckhardt's Raves

CVS

- Commit: atomic if you are careful
- Named snapshots: if you are careful
- Branching: works if you are careful
- *Core operations* require care & expertise!!!

Many commercial products

- Require full-time person, huge machine
- Punitive click-click-click GUI
- Poor understanding of data structure requirements

40

Recommendation for 15-410

You can use CVS if you're used to it
- Better: SVN, hg, darcs, ...

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" (20%) to be chosen by your next boss

41

Getting Started

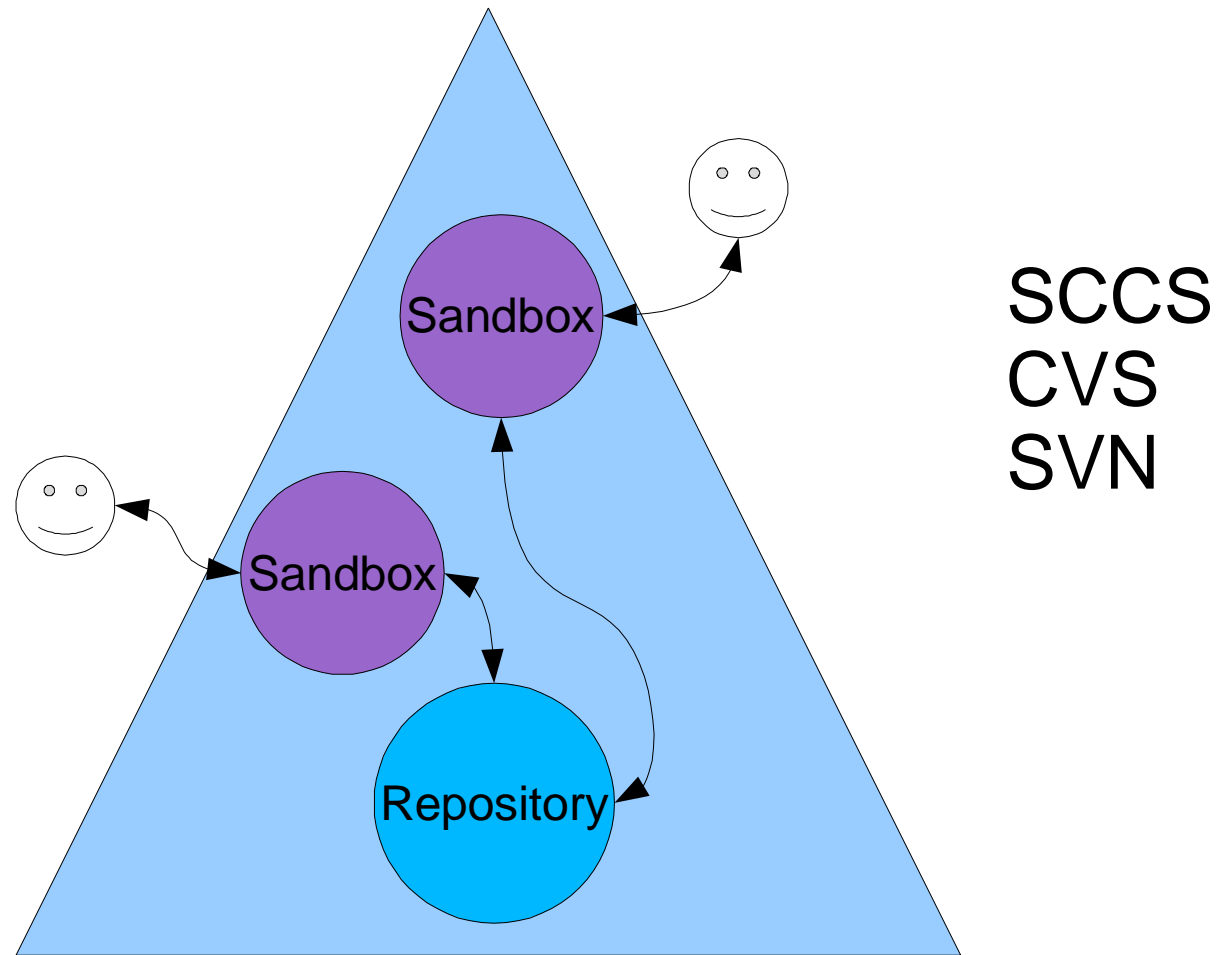Already installed on Andrew Linux systems!

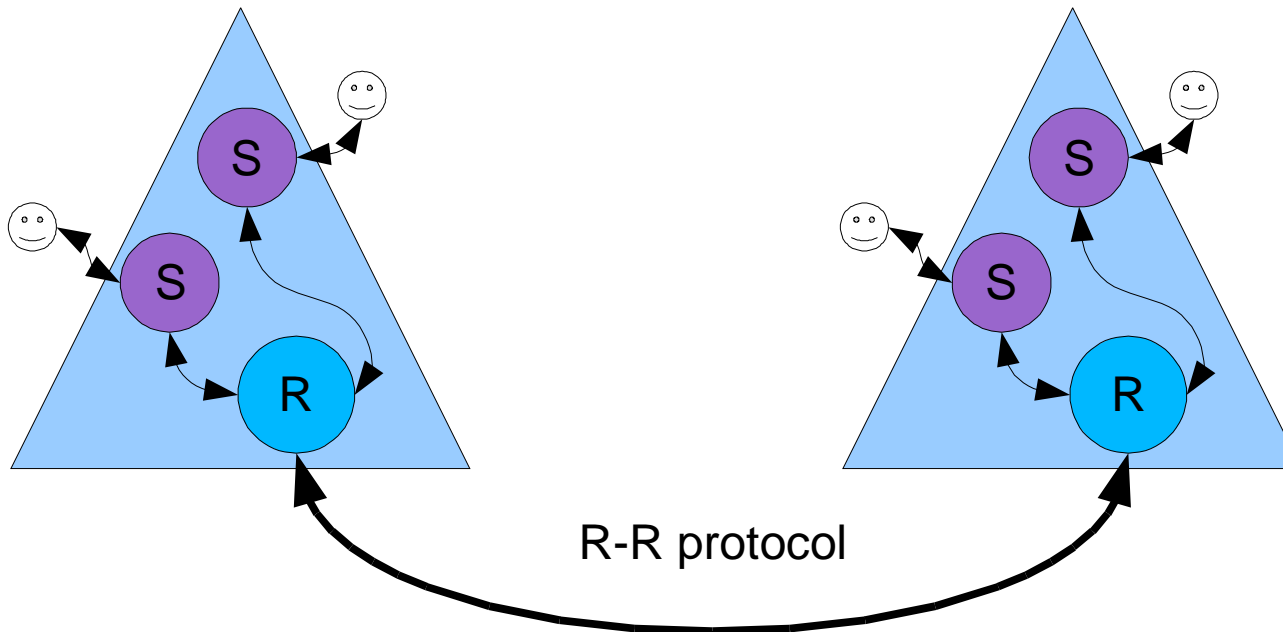Or you can install it yourself on your own.
- ("Some assembly required")

Git is a "distributed" source-control system
- ???
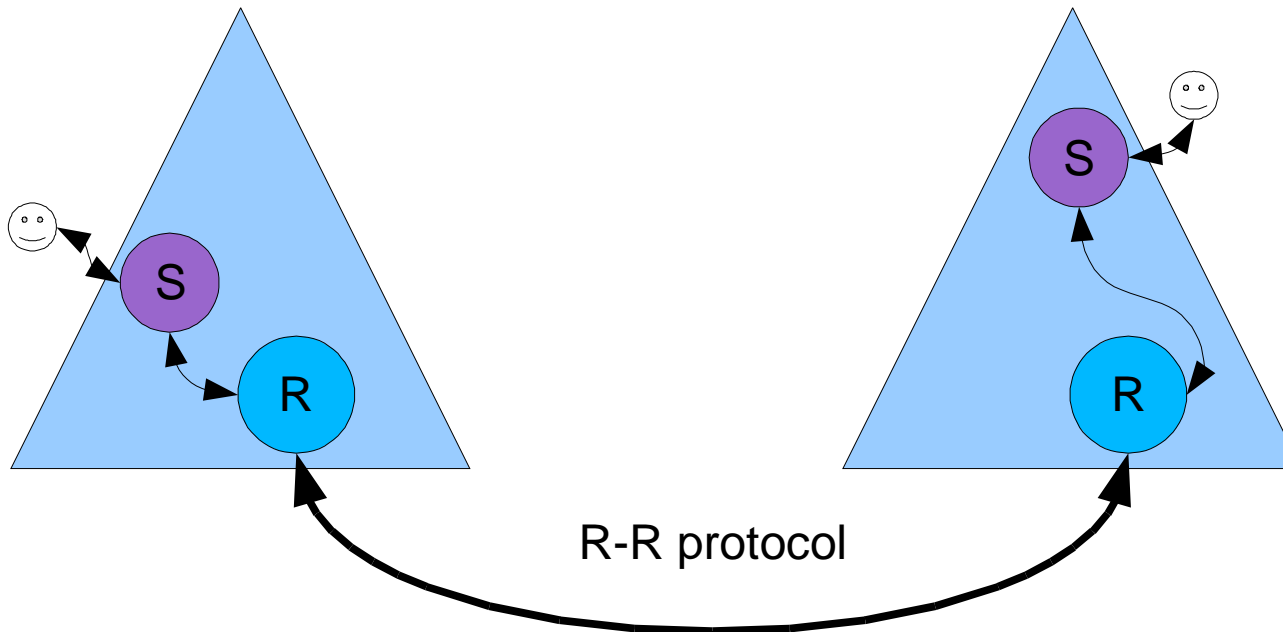
Traditional "File System" Model

SCCS
CVS
SVN

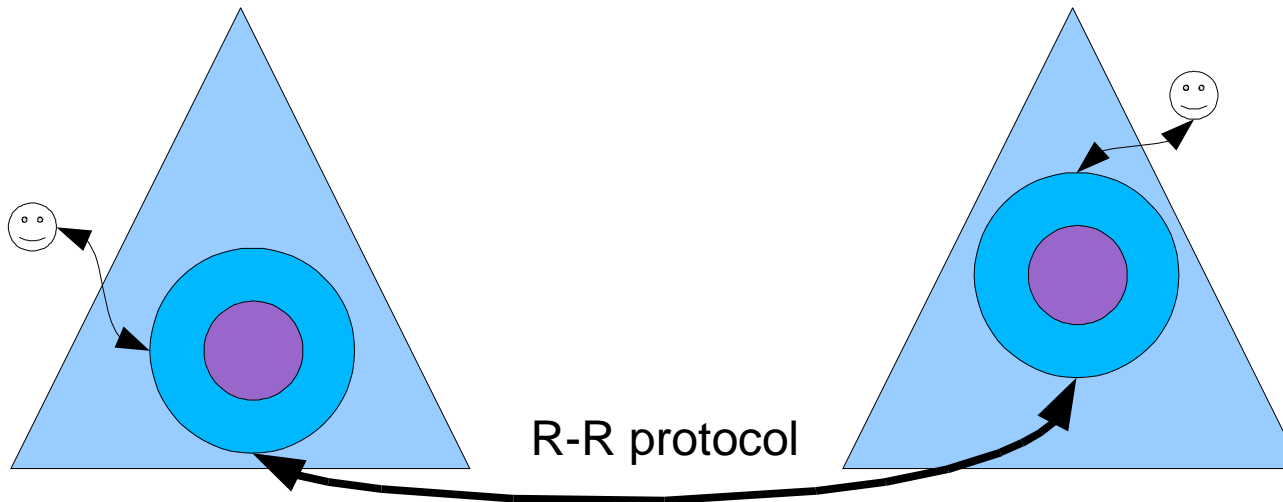# "East-Coast / West-Coast" Model

R-R protocol

Inter-repository protocol runs "sometimes".
Conflicts are tricky.
Perforce does this.

Laptop Model!

R-R protocol

Sandbox-repository protocol.
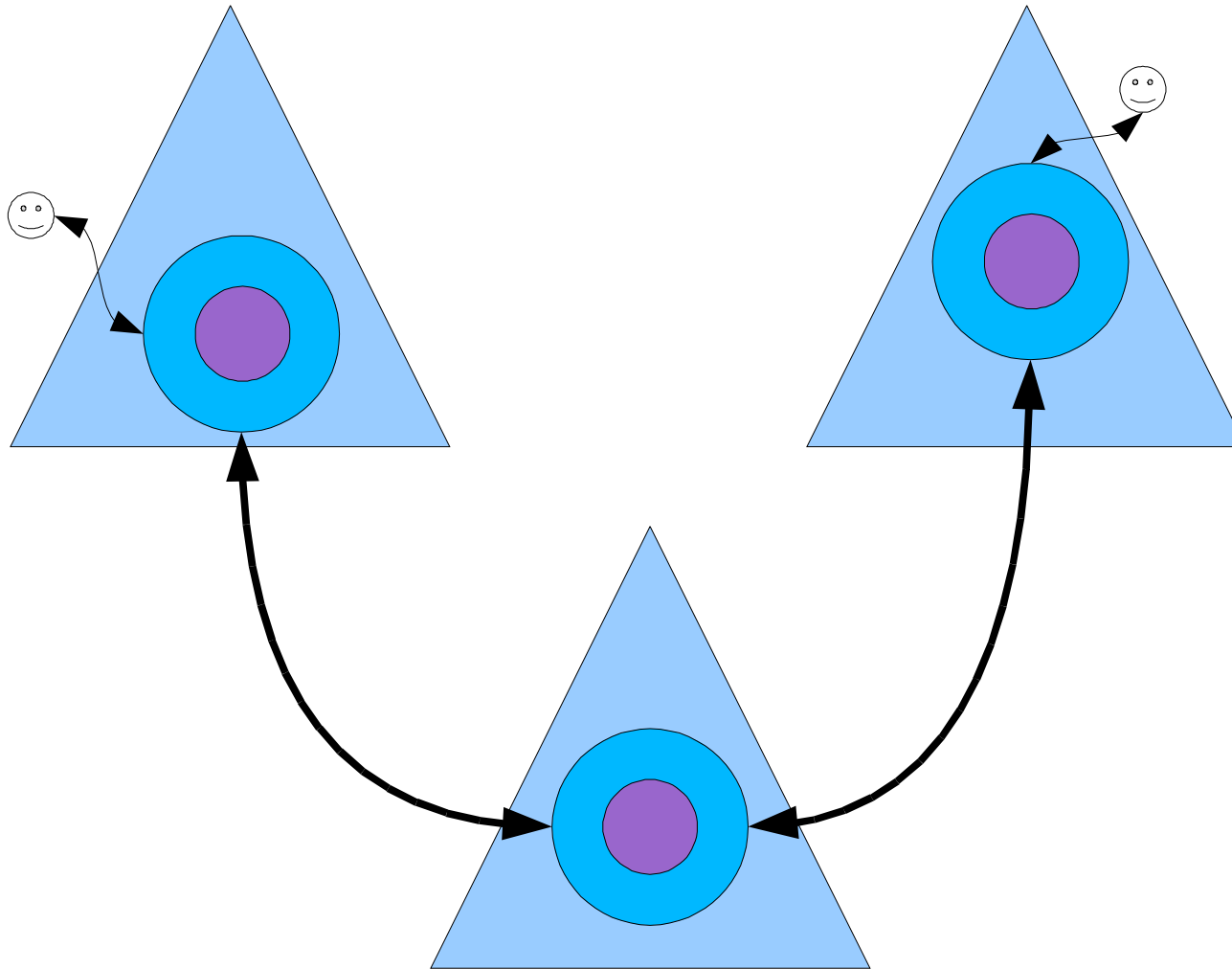Also, inter-repository protocol.
More protocols == more fun?

"Distributed Version Control"

Repository holds current files *and* metadata.
Inter-repository protocol is tricky (no "before").
Whose laptop do we release to customers from??

hg, git
darcs

"The Repository"

Creating A New Project

Anywhere, but safest in a blank directory:

```
$ git init
```

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.

Populating the world

Adding Files

    `$ git add file1 file2 ...`

    – **To add *every* file in a directory**

        `$ git add dir/`

        • **Rarely what you want!!!**

 These are "staged" operations...

    – **"Add" requires a commit just like "edit" does.**

Checking In

Commit Yourself!

`$ git commit -a`

- Fires up your $EDITOR and asks you for commentary.
- Can restrict which files on the command line, or even use –interactive.
- Adds a new snapshot to *LOCAL* repository's history
  - Your partner has *no idea that this has happened.*

50

Sharing Your Work

How do changes become non-local?

Pull

```
$ git pull [remote-path/URL]
```

- **Pulls changes from a remote repository.**
- **Git has a notion of "default remote"**

Push

```
$ git push [remote-path]
```

- **Pushes changes from the local repo into the remote.**

Checking Out A Project

Making a new checkout:

```
$ git clone remote-path/URL [local-name]
```

- Clones the remote repository
- All set for you to work in.
- The default push/pull target is the remote you copied.

You can use this mechanism to "branch".

- Git also supports named branches in a repo.
- See "man git-branch" or any of the other docs.

52

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;
}
```

53

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;
}
```

54

Conflicts and Merging

Suppose Alice "checks in" first

```
$ git commit -a  ⇒ ok
$ git push       ⇒ ok
```

Now Charlie...

```
$ git commit -a  ⇒ ok
$ 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!

commit:file name

Division between
conflicting commits

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

#define SUPER 0

int main(void)
{
<<<<<<< HEAD:hello.c
        /* this, like, says hello, and stuff */
        printf("Hello Hercules!");
        return 42;
=======
        /* prints "Hello World" to stdout */
        printf("Hello World!");
        return SUPER;
>>>>>>> 12341234abcd5656efef787890900123456789ab:hello.c
}
```

56

Information

To get a summary of changes:

```
$ git status
```

To ask about changes in the past:

```
$ git log
```

Branching

To create the first version of a new branch:

```
$ git branch Experimental_VM

$ git checkout Experimental_VM
```

To merge master with branch:

```
$ git checkout master

$ git merge Experimental_VM
```

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 COW 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

See GIT documentation

- 15-410 "git intro" web page – specific help
- Lots of documentation online (*many* features)
- Search for "gittutorial"

61

Further Reading

"Git for Computer Scientists"

"Git from the Bottom Up"

"Git Magic"