Version Control Systems

Motivation

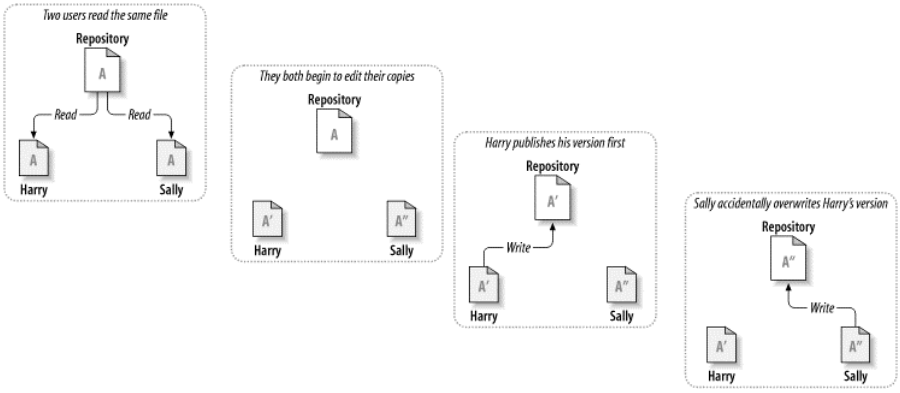
Members of a software development group need to:

- have access to the group source code (file sharing)
- work at the same time on the same files (concurrent editing)
- keep track of different versions of the same file (history)

A **Version Control System** is a special file server, designed for concurrent editing and to store history information.

Concurrent Editing

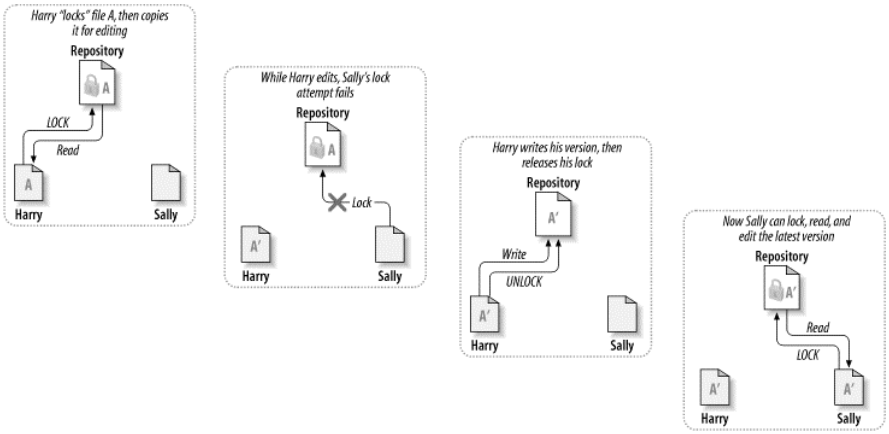
Why is concurrent editing difficult?



A normal file server (eg. NFS) can provide file sharing, but would keep only one version of each file (the most recent one).

Lock-Modify-Unlock

A simple mechanism to support concurrent editing:



The same mechanism can be used to implement multi-threaded access to shared resources.

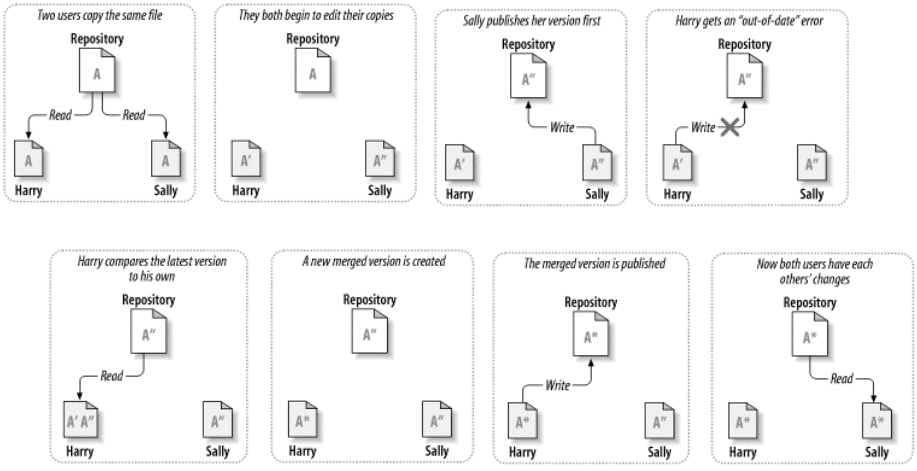
Lock-Modify-Unlock

Disadvantages of this scheme:

- *delays:* locking a file prevents concurrent editing
- *administrative overhead:* if a user forgets to release the files he has locked, an administrator has to manually remove the lock before another user can edit the files.
- *false sense of security:* locking a single file is not sufficient if there are other files depending on it

Copy-Modify-Merge

A better mechanism:



Copy-Modify-Merge

When merging, two types of changes to a file can occur

• *changes that do not overlap*: in this case merging is trivial - just take the sum of changes

• *changes that overlap*: in this case there is a **conflict** and merging can be difficult - users must communicate to decide which changes to propagate to the new version.

Merging is a **manual** process by the user
(No AI available yet to decide which changes to take).

The amount of time it takes to resolve conflicts is far less than the time lost by a locking system.

Available Version Control Systems

• CVS (www.cvshome.org): open source, wide-spread, problems when removing and renaming files.

• Subversion (subversion.tigris.org): open source, the successor of CVS

• BitKeeper (www.bitkeeper.com): closed source, used in linux kernel development

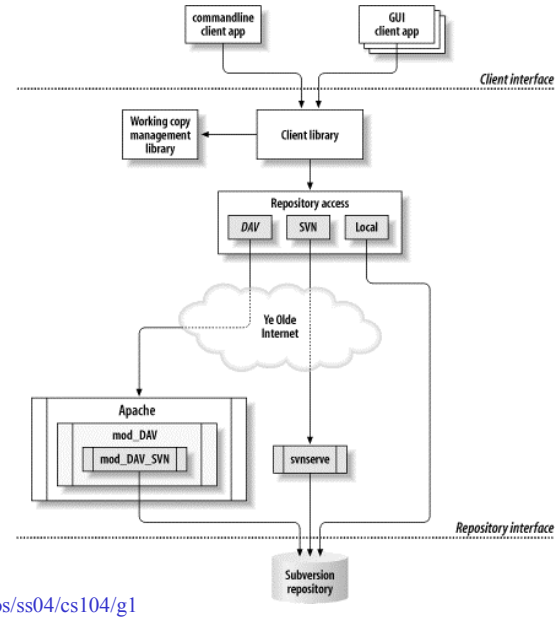
• … others?

Brief Intro to Subversion

Subversion Architecture

Clients:
- svn (Linux)
- Tortoise (Windows)

Repository Access URLs:
- file:///
- http:// + https://
- svn://



Programing II (CS104):
- https://subversion.cs.unibas.ch/repos/ss04/cs104/g1

Repository

What is it?

The *repository* is a centralized store for data:

- Stores data in form of a *filesystem tree*
- Provides read/write access to the stored data
- Remembers any modification made to it

Working Copy

What is it?

The working copy is made up of two parts:
- a *local* copy of the directory tree of a project
- an *administrative directory* named .svn in each directory, storing version control information

Notes:
- Users edit their working copy locally.
- Changes are then *committed* to the repository.
- After a commit, all other users can access the changes by *updating* their working copies to the latest revision.

Administrative Information

Information stored in `.svn` directories:

For each file, Subversion stores:
- the working revision of the file
- a timestamp of the last update of the file

Based on this information, Subversion can determine the state of the file (shown by the **status** command):
- if changes have been committed (an update will change the working copy)
- if changes have been made in the working copy (a commit will change the repository)

Basic Commands on Linux (svn)

checkout - Get an initial working copy of a project. Eg.:

```
svn checkout file:///path/to/repos/aproject
```

add - Add a new file or directory to the repository. Eg.:

```
svn add newfile.java
```

commit - Transfer local changes to the repository. Eg.:

```
svn commit -m "changes description" changed.java
```

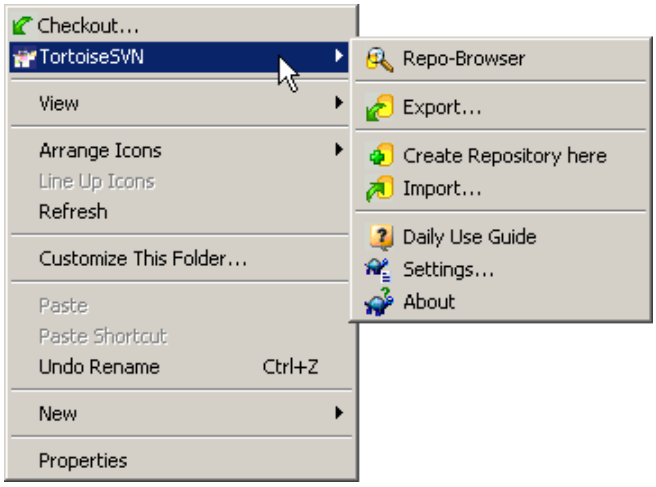
update - Apply changes committed by others to own working copy. Eg:

```
svn update outofdate.java
```

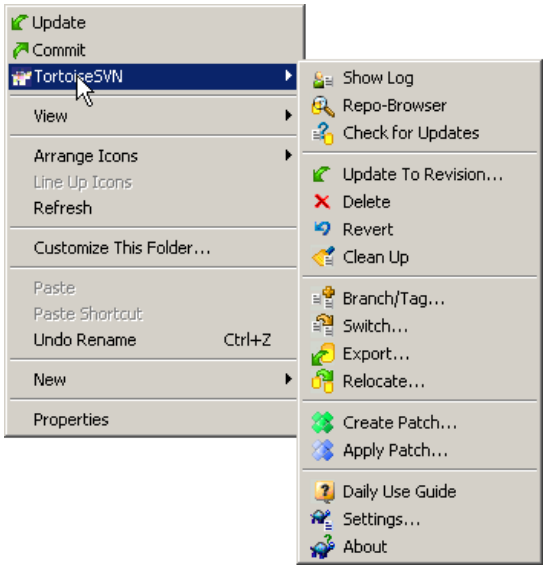
help - Explains available commands Eg:

```
svn help
svn help add
```

Basic Commands on Windows (TortoiseSVN)



Basic Commands on Windows (TortoiseSVN)



Revisions

What is it?

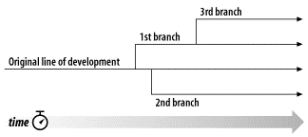
A **revision** is the state of the *filesystem tree* after a commit.

- The state of the repository after each commit is called a **revision**.
- To each revision, a natural number is assigned which identifies the revision uniquely.
- Revision numbers always increase, starting from 0.

Notes:
- revisions are assigned to the whole tree: a certain file can be left unchanged through different revisions
- files in the repository have always the same revision
- files in the working copy might have different revisions

Branches

What is it?



Branches:
- A Branch is an independent line of development that shares a common history with another line of development
- It begins life as a copy of something and moves on from there generating its own history

What are they good for?
- Branches are usually used to try out new features without disturbing the main branch of development with compiler errors and bugs
- As soon as the new feature of the branch is stable enough the branch is merged back into the main branch (the trunk) and removed

Tags

What is it?

• A tag is a snapshot of a project in time, E.g. each repository revision is a tag - a snapshot of the filesystem after each commit.
• To make it easier to remember snapshots, *Tags* get human-friendly names like „release-1.0“.
• Like Branches, Tags are created as copies of something

What are they good for?

• To save the state of a project that one later wants to refer to

Branches and Tags continued

Do they differ?

• No, not really. For Subversion Branches and Tags are the same - they are directories in the filesystem and handled in the same way.

• From a user perspective yes - Branches are lines of development. That means, you make changes to them. Tags are used to fixate a certain state of your development, so you should not commit to a Tag - although you could do that.

Programming II (CS104)

Repository Layout

Each Group has its own toplevel directory that only the team members have access to.

Example: Project directory of group **g1** is

https://subversion.cs.unibas.ch/repos/ss04/cs104/g1/

Within each of the group directories three subdirectories exist for software development:

- **Trunk** - Here the main development takes place
- **Branches** - Here you create Branches from Trunk
- **Tags** - Here you create Tags from Trunk

Example Work Cycle on Linux (Part 1)

- Checkout a working copy

```
% svn checkout https://subversion.cs.unibas.ch/repos/ss04/cs104/demo          // Note the https:// ...
A  demo/trunk
A  demo/branches
A  demo/tags
Checked out revision 7.
% cd demo/trunk
```

- Make changes

Create a directory and a new file

```
% svn mkdir doc                    // create directory doc and schedule it for addition to repository
A         doc
% cd doc
% vi index.html                    // create a new file
% svn add index.html               // schedule the new file for addition to repository
A         index.html
```

- Commit changes

```
% cd ..
% pwd
demo/trunk
% svn commit -m „Started project documentation“   // commit doc and index.html to repository
Adding        trunk/doc
Adding        trunk/doc/index.html
Transmitting file data .
Committed revision 8.
```

Example Work Cycle (Part 2)

- Update the working copy

Update all files and directories to the most current version

```
% svn update
```

Go to a particular older revision for all files and directories

```
% svn update -r 3
```

- check log messages

```
% svn log
```

Subversion Client Installation

Linux:

Download latest tarball from http://subversion.tigris.org/project_packages.html

```
# tar xzvf subversion-1.0.1.tar.gz
# cd subversion-1.0.1
# ./configure --without-apache --without-apxs --with-ssl --enable-shared --enable-static
# make install
```

Windows: TortoiseSVN

Download MSI Installer from http://tortoisesvn.tigris.org/download.html

Documentation

General: http://svnbook.red-bean.com/

Windows specific: http://tortoisesvn.tigris.org/docs.html

Commandline Help:

```
svn help                    // List available commands
svn help <command>          // Help and argument list
                            // for <command>
```