Version Control with Subversion

Introduction

- Wouldn't you like to have a time machine? Software developers already have one: it is called version control!
- Version control (aka *Revision Control System* or *Source Control System* or *Source Code Management*) is the art and science of managing information, which for software projects implies managing files and directories over time.
- A repository manages all your files and directories. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your data, or examine the history of how your data changed.

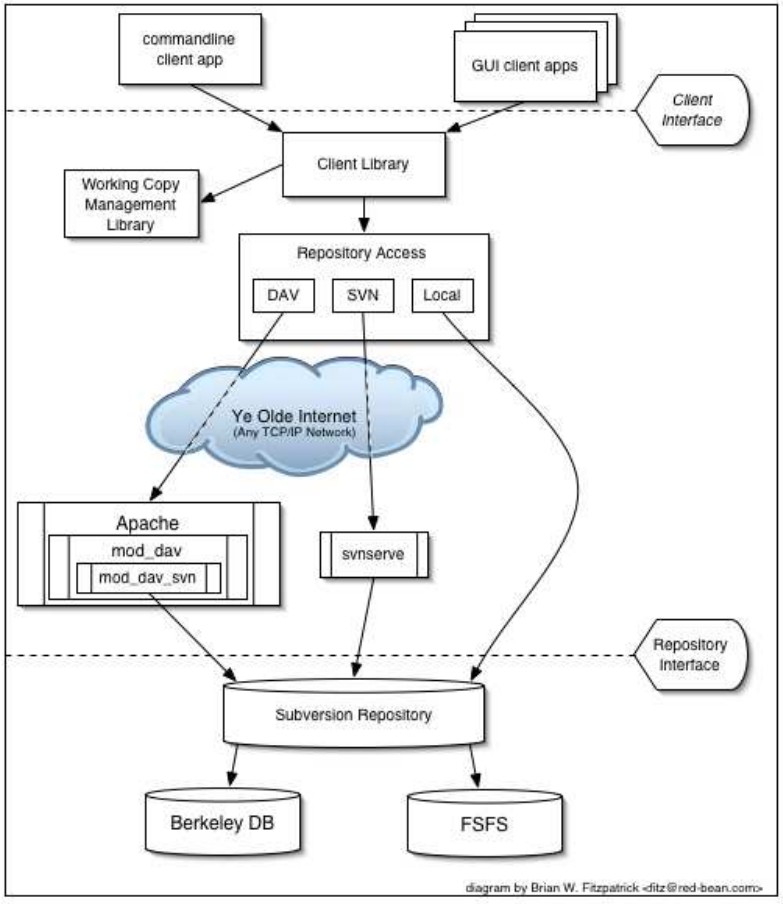
Comparison of some open-source version control systems

- RCS (Revision Control System). Simple, text based system. Included in Linux and Unix systems by default. No remote access. No directory level access.
- CVS (Concurrent Versioning System). Built on top of RCS. Adds directory level access as well as remote access.
- Subversion. A modern CVS "replacement" that isn't built on top of RCS. Allows directory access, web access (via an Apache Web server module), remote access (via ssh or svn server). Uses a centralized model with mulitple access-control possibilities.
- Git. A distributed version control system. There is no central repository like in subversion. Everyone has a copy of the repository. More complex model to learn. Useful for parallel, largely shared but permanently somewhat different lines of the same project.

Subversion Features

- Subversion allows you to attach metadata to an item. Metadata takes the form of properties. A *property* is a key/value pair. Properties are versioned as well.
- Subversion has a client/server architecture. A developer interacts with a *client* program, which communicates with a *server* program, which accesses repositories.
- A *repository* contains the versions of your items.
- Multiple clients, communication protocols, repository-access mechanisms, and repository formats are available.
- Repository formats: Berkeley Database and FSFS (preferred and default).

Subversion Architecture



Versioning Models

The core mission of a version control system is to enable collaborative editing and sharing of data. But different systems use different strategies to achieve this.

- The Lock-Modify-Unlock Solution. The repository allows only one person to change a file at a time. To change a file, one must first obtain a lock. After you store the modified file back in the repository, then we unlock the file. Example: RCS.
- The Copy-Modify-Merge Solution. Each user's client contacts the project repository and creates a personal working copy–a local reflection of the repository's files and directories. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly. Example: CVS, Subversion. However CVS and subversion still also support locking if it is needed. Typically locking is used for non-text files.

Downloading/Installing Subversion

We have subversion 1.6.x available in the lab on all workstations.

- On Fedora Linux, check the version with the command:
  `rpm -qa | grep subversion`
- If not using version 1.6 or newer, then get it using `yum` as follows:
  `yum -y update subversion* mod_dav_svn*`
- For other versions of Linux, check out the subversion website for downloads. (Subversion website: `http://subversion.tigris.org/`)
- You can also download the source code from `subversion.tigris.org` and compile it directly.

Creating and Populating a Repository

- This is an administrative task. A developer would not normally need to do this.

```
svnadmin create ~/svn
cd ~/cs453
mkdir -p project1/branches project1/tags project1/trunk
gvim project1/trunk/hello.c
svn import project1 \
 svn+ssh://HOSTNAME/HOME/svn/project1 -m "import the project"
```

  Here `HOSTNAME` is the Internet name of the server and `HOME` is the full path to your home directory on the server. Also, if your local user name is different than your user name on the server, you will need to use `username@HOSTNAME`.
- The `branches`, `tags`, and `trunk` directories are a convention. They are not required (but highly recommended).
- Repository layouts: Vanilla, Strawberry or Chocolate!
  - A single repository with all projects folders in it.
  - Separate repository for each project.
  - A handful of repositories, each with multiple related projects.

Access Mechanisms

Schema	Access Method
file:///	direct repository access (on local disk)
http://	access via WebDAV protocol to Subversion-aware Apache server
https://	same as http://, but with SSL encryption.
svn://	access via custom protocol to an svnserve server
svn+ssh://	same as svn://, but encrypted via an SSH tunnel.

To be able to use `svn+ssh://` conveniently, you will need to setup automatic login with `ssh`. Here is how to set that up.

- Look in your `~/.ssh` directory on your local machine. There should be two files, `id_rsa` and `id_rsa.pub`. If not, create them using the command `ssh-keygen -t rsa`.
- Append your local `id_rsa.pub` to the remote host's `~/.ssh/authorized_keys`. If the remote host doesn't have a `~/.ssh` folder, then create one and then create the `authorized_keys` file in it.
- Change the permissions on `~/.ssh/authorized_keys` using `chmod 600 ~/.ssh/authorized_keys`

Checking-Out a Working Copy

This is a development task. A *working copy* is a private workspace in which you can make changes.

```
cd ~/cs453
svn checkout svn+ssh://HOSTNAME/HOME/svn/project1/trunk project1
```

Notes:

- The URL identifies the version to checkout.
- The last argument names the destination folder.
- Note that we are specifying `trunk` because we want to work on the main line of development. Do not use `svn+ssh://HOSTNAME/HOME/svn/project1/` or you will be checking out all the branches as well!

Working on a Working Copy

You can now change your working copy. Let's change `hello.c` and add a **Makefile**.

```
cd project1
gvim hello.c # format nicely
gvim Makefile
svn add Makefile
svn status -u
svn diff
svn commit -m "make it nice"
gvim hello.c
svn status -u
svn diff
svn commit -m "add stdio.h"
```

Notes:

- The whole tree we check-in gets a new version number.

Subversion Properties

- *Properties* are name/value pairs associated with files. The names and values of the properties can be whatever you want them to be, with the constraint that the names must be human-readable text. And the best part about these properties is that they, too, are versioned, just like the textual contents of your files.
- *Subversion Special Properties* always start with the keyword `svn:`. An useful one is `svn:keywords`. Some keywords are `Date, Revision, URL, Author, Id`. Including the keywords as $*keyword*$ in your files allows Subversion to auto-magically expand them.

```
cd ~/cs453/hw
gvim hello.c # add Id and Revision keywords
svn commit -m "Add svn id keywords"
svn propset svn:keywords "Id Revision" hello.c
svn diff
svn commit -m "commit properties"
gvim hello.c
```

Checking subversion version from your code

Add the following to the top of your file.

```
/* $Id$ */
static char *svnid = "$Id$";
```

Then, each time you commit subversion expands Id to the subversion id. You need to set the property on the file for this to work as discussed in the previous slide.
Note that the `Rev` or `Id` property only shows the last subversion revision number in which the current file was modified. *It is not the same as the global revision number for the whole repository.*

Incorporating global revision numbers

To find the global revision number of a working copy, use the svnversion command. The following example shows how to automate this to include the version number in your code.

```
##
## on every build, record the working copy revision string
##
svn_version.c: FORCE
    echo -n 'const char* svn_version(void) { const char* SVN_Version = "' \
                                          > svn_version.c
    svnversion -n .                      >> svn_version.c
    echo '"; return SVN_Version; }'      >> svn_version.c

FORCE:
##
## Then any executable that links in svn_version.o will be able
## to call the function svn_version() to get a string that
## describes exactly what revision was built.
```

Basic Work Cycle

- Get a working copy
  - `svn checkout` (`co`)
- Update your working copy
  - `svn update` (`up`)
- Make changes
  - `gvim`
  - `svn add`
  - `svn mkdir`
  - `svn delete` (`rm`)
  - `svn copy` (`cp`)
  - `svn move` (`mv`)
- Examine your changes
  - `svn status` (`st`)
  - `svn diff`
  - `svn revert`
- Merge others' changes into your working copy
  - `svn update` (`up`)
  - `svn resolved`
- Commit your changes
  - `svn commit` (`ci`)
- Getting help on `svn` options.
  - `svn help commit`

Changing repositories

- Sometimes we want to move our repository from one machine to another. First commit any changes from all working copies. Then pack up the repository as shown below and unpack it on the new server.

```
tar czvf subversion.tar.gz subversion
scp subversion.tar.gz newserver://newpath/
ssh newserver
cd newpath
tar xzvf subversion.tar.gz
```

- Now repoint your working copies to the new URL with the switch command.

  svn switch –relocate OLD-URL NEW-URL

- The `switch` (without the relocate option) command can also be used to reflect changing the name of a repository.

Merges and Conflicts

Suppose two people are working on the same file.

```
cd project1
#add comment on top/bottom
gvim hello.c
svn commit -m "made some changes"
```

```
svn checkout svn+ssh://HOSTNAME/HOME/svn/project1/trunk project2
cd project2
gvim hello.c # add comment on top/bottom
svn commit -m "made some changes"
---fails due to conflict---
svn update
gvim hello.c  # fix conflict
svn resolved program.c
svn commit -m "made some changes"
```

Branches

A developer typically wants to work on a task without interference from other people and without interfering with other people. A task may take days or weeks to complete. While working on task a developer should check-in intermediate versions, because:

- A repository typically resides on a high-quality storage device (e.g., RAID).
- A repository typically is backed-up carefully.
- A developer may want to work on different computers, on different network segments.

The solution is to work on a *branch*. Subversion uses directories for branches. To Subversion, there is nothing special about a "branch" directory.
When Subversion makes a copy, it is a "cheap copy" requiring a constant amount of space. When you change one of the copies, it really makes the copy. This technique is also called *copy-on-write*.
Recall that our hello-world project directory had three subdirectories: `branches`, `tags`, and `trunk`. This is just a convention. The `branches` directory is for branches.

Branch Example

```
svn copy svn+ssh://HOSTNAME/HOME/svn/project1/trunk \
svn+ssh://HOSTNAME/HOME/svn/project1/branches/mytask \
-m "a real big task"

svn checkout svn+ssh://HOSTNAME/HOME/svn/project1/branches/mytask mytask
cd mytask
svn info
```

Merging from a Branch

Now, we can work on our branch in isolation, checking-in our changes, for as long as we like.

```
cd project1
gvim hello.c # add new function
svn status -u
svn diff
svn ci -m "add new function"
gvim hello.c # call it
svn ci -m "call new function"
```

Merging from a Branch (contd.)

Eventually, we may want to merge our changes from our branch back to the trunk, so others may benefit from them.

```
svn co svn+ssh://HOSTNAME/HOME/svn/project1/trunk project1
cd project1
#what is the earliest revision on the branch?
svn log --verbose --stop-on-copy \
                  svn+ssh://HOSTNAME/HOME/svn/project1/branches/mytask
svn merge -r 8:10  svn+ssh://HOSTNAME/HOME/svn/project1/branches/mytask
gvim project1.c #check merge result
svn ci -m "merged -r 8:10 from mytask branch"
svn info
```

Notes on Merging

- We created a trunk working copy, to merge to.
- We needed to determine, and specify, the starting and ending versions on our branch, so the right changes are merged.
- We merge from our branch, in the repository.
- We mention the merge versions in the check-in message, so we won't accidentally merge those changes again, if we keep working on my branch.
- Now, the trunk has our changes.

GUIs for Subversion

There are many GUI tools available for subversion. However, I would recommend the following.

- **Subclipse plugin**. First, check if you already have the Subclipse plugin under *Help* → *Software Updates* → *Manage Configuration*. Otherwise go to `http://subclipse.tigris.org/install.html` for step-by-step instructions on how to install the Subclipse plugin from Eclipse. The installation of Subclispe plugin requires that you have write access to the folder that contains the Eclipse installation.
- **TortoiseSVN** is a nice stand-alone GUI for subversion that works for Linux and MS Windows.

Subclipse Plugin for Eclipse

The subclipse plugin gives you most of the subversion functionality in Eclipse.

- We can use the SVN perspective to browse repositories. From there we can checkout a subversion project as an Eclipse project.
- Subclipse supports the `https://` and `svn+ssh://` protocols but does not support the `file://` protocol.
- A new menu *Team* gives you access to subversion commands from your project. All of the concepts we have covered can be accessed from the subclipse plugin except for the administrative commands.
- You can share a existing project in Eclipse using the menu items *Team* $\rightarrow$ *Share project...*. To share a projects, you need to know the URL of an existing repository.

References

- http://subversion.tigris.org/ Homepage for the subversion project.
- http://svnbook.red-bean.com Excellent book for subversion.
- Thanks to Jim Buffenbarger for providing me notes on subversion from his Software Engineering class. His notes/examples were used extensively in this document.