Subversion

"A compelling replacement for CVS..."
Things I want to talk about

• Background
• Features
• Components
• Tools
• Some experiences
  - Surprises, limitations, and other stuff
• Useful resources
My background with Subversion

• Timeline:
  - Followed the development of Subversion since early 2002
  - Began experimenting with Subversion in late 2003
  - Began using Subversion actively for personal projects when it reached 1.0 status (Feb, 2004)
  - Began rolling Subversion out within my company (and to customers) in May, 2004

• Experience:
  - I build each new version under Solaris, OS X as it is released (though I use the "stock" binaries on Windows)
  - Primary usage is via SSH tunneling and BDB-based repositories
  - Some experimentation with WebDAV-based usage and FSFS repositories, but nothing significant
Why do version control?

• Version control is the art of managing changes to information (files).
  - For programmers, this is source code: you spend lots of time making small changes to software, and then undoing them the next day
  - For managers, this is project documentation: contracts, project plans, "engineering change orders", etc.
  - For QA, this is test plans and cases, scripts, and more project documentation
  - For technical writers, this is user's guides, manuals, FAQs, tech notes, and so on

• Version control lets you look at your information as it is now, or look at it as it was some time in the past: you can maintain *history* about your files in a "repository".

• "Anywhere you can find people using computers to manage information that changes often, there is room for version control."
• Open source version control system
• Repository structure is based on RCS format ("*,v" files)
• Centralized repository
  – can be on the local machine, or live on the network
  – work does *not* happen in the same folders as the repository (unlike RCS)
• Supported on lots of platforms
• By default, uses an optimistic approach to conflict resolution
  – "copy-merge-commit"
• A very popular tool set, but known to have some weaknesses
Where did Subversion come from?

• Early 2000:
  - CollabNet, Inc. began seeking developers to write a replacement for CVS
  - CollabNet contacted Karl Fogel to see if he was interested (he was)

• May 2000:
  - Detailed design work started
  - Team quickly attracted a community of active developers

• August 31, 2001:
  - Subversion became "self-hosting"

• February 23, 2004:
  - Subversion 1.0 released

• September 29, 2004:
  - Subversion 1.1 released
History: Subversion's early goals

- They didn't want to break new ground in version control methodology, they just wanted to fix CVS.

- Subversion would match CVS's features, and preserve the same (normal) development model
  - Centralized repository
  - "Copy-modify-merge"

- Subversion would fix CVS's most obvious flaws (e.g., management of directories, etc.).

- It did not need to be a drop-in replacement for CVS, it should be similar enough that any CVS user could make the switch with little effort.
Features (vs. CVS)

- Truly cross-platform
- Directory versioning
- Atomic commits
- "True" version history
- Consistent data handling
- More efficient network usage
- Choice of access protocols
- Versioned metadata
- Efficient branching and tagging
- Better handling of merge conflicts
- "Real" status reporting
- Built-in help ("svn help")
- "Hackability"
  - Well-defined (library-based) API for folks to use in their own programs
Access protocols

• The mechanism used to access a repository is specified as a part of a URL passed to the client programs.

• There are several protocols for accessing a Subversion repository:
  - "file:///...."
    • "local" access, via direct manipulation of the file system
  - "http://...." or "https://...."
    • access via WebDAV protocol to a Subversion-aware Apache server
  - "svn://...."
    • access via custom protocol to an "svnserve" server
    • somewhat similar to the "pserver" protocol under CVS
  - "svn+ssh://...."
    • same as "svn://....", but uses an SSH tunnel for security
    • similar to the "ext" protocol when used with "CVS_RSH" set to "ssh"
• Subversion allows users to establish "properties" for version-controlled files/folders
  - These can be user-defined properties, or standard properties that affect how Subversion handles data
  - Property content can be arbitrary, even binary data
  - Properties are versioned, just like a file’s contents

• Some examples:
  - Linking files to defect reports.
  - Details on images for web site (captions, date stamps, thumbnails, etc.)
Some standard properties

- **svn:executable**
  - No defined values; just present or not
  - Tells svn to make sure that the "execute" bit is kept enabled

- **svn:mime-type**
  - Used by Subversion to decide if a file is mergeable (type is "text/....") or not
  - Also used with Apache module to set the Content-type: header in an HTTP response

- **svn:ignore**
  - Contains a multi-line list of file patterns (one per line) that certain operations will ignore
  - Works in conjunction with the "global-ignores" run-time configuration option
  - Similar to ".ignore" in CVS, and can be directly set from a .cvsignore file's contents:
    
    ```
    $ svn propset svn:ignore -F .cvsignore .
    property 'svn:ignore' set on '.'
    ```

Some standard properties (2)

• **svn:keywords**
  - Defines a list of keywords for which substitution will take place within a file
  - Supported set of keywords are:
    • LastChangedDate (abbreviated as Date)
    • LastChangedRevision (Revision, or Rev)
    • LastChangedBy (Author)
    • HeadURL (URL)
    • Id
  - Usage is same as in CVS (e.g., "$Author$")

• **svn:eol-style**
  - By default, Subversion doesn't adjust files to match the local conventions for "text files"; it assumes that "bytes are bytes"
  - This keyword provides for this adjustment
  - Possible values are: "native", "CRLF", "LF", or "CR"
Automatic property setting

• Most properties must be manually set by the user
• Subversion will automatically set some of them under specific circumstances
  - During an "add" or "import" operation, Subversion will try to guess if a file is human-readable; if not, its "svn:mime-type" property is set to "application/octet-stream"
  - There is an "auto-props" feature, which can be used to map filename patterns to property names and values (also during "add" or "import" operations)
    • This is controlled from the runtime configuration area (e.g., "~/.subversion/config" under *nix)
    • Users must be explicitly enable this feature, as well as setting up the mappings for filenames
Architectural overview
<table>
<thead>
<tr>
<th>Feature</th>
<th>Berkeley DB</th>
<th>FSFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitivity to interruptions</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Usable from read-only mounts</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Platform-independent storage</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Usable on network file systems</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Repository size</td>
<td>Slightly larger</td>
<td>Slightly smaller</td>
</tr>
<tr>
<td>Scalability: # of revision trees</td>
<td>No problems</td>
<td>Depends on file system</td>
</tr>
<tr>
<td>Scalability: directories with lots of files</td>
<td>Slower</td>
<td>Faster</td>
</tr>
<tr>
<td>Speed: checking out latest code</td>
<td>Faster</td>
<td>Slower</td>
</tr>
<tr>
<td>Speed: large commits</td>
<td>Slower, but work is spread through commit</td>
<td>Faster, but finalization delays can cause client timeouts</td>
</tr>
<tr>
<td>Group permissions handling</td>
<td>Sensitive to user umask issues; best if accessed by only 1 user</td>
<td>Works around umask issues</td>
</tr>
<tr>
<td>Code maturity</td>
<td>Used since 2001</td>
<td>Used since 2004</td>
</tr>
</tbody>
</table>
Subversion components

• Client-side
  - **svn**
    - The command-line client program.
  - **svnversion**
    - A program for reporting the state (in terms of revisions of the items present) of a working copy.
  - **svnlook**
    - A tool for inspecting a Subversion repository.
  - **svnadmin**
    - A tool for creating, tweaking or repairing a Subversion repository.
  - **svndumpfilter**
    - A program for filtering Subversion repository dump streams.

• Server-side
  - **mod_dav_svn**
    - A plug-in module for the Apache HTTP Server, used to make your repository available to others over a network.
  - **svnserv**
    - A custom standalone server program, which can be run as a daemon process or invoked by SSH; another way to make your repository available to others over a network.
Some useful commands

• Core commands:
  - `svn checkout`
  - `svn update`
  - `svn status`
  - `svn diff`
  - `svn revert`
  - `svn commit`
  - `svn add`
  - `svn delete`
  - `svn copy`
  - `svn move`
  - `svn import`
  - `svn merge`
  - `svn resolved`

• Property management:
  - `svn propset`
  - `svn propget`
  - `svn proplist`
  - `svn propedit`

• Miscellaneous:
  - `svn cleanup`
  - `svn cat`
  - `svn log`
• CollabNet owns the copyright to Subversion
  – The copyright license is fully compliant with the Debian Free Software Guidelines
  – In other words, anyone is free to download, modify, and redistribute Subversion as she pleases; no permission from CollabNet or anyone else is required

• This has led to a variety of projects to build new tools to work with Subversion repositories
Some popular tools

- Some clients:
  - Core command-line tools from the Subversion team
  - TortoiseSVN (Windows Explorer integration)
  - AnkhSVN (Visual Studio integration)
  - Subclipse (Eclipse plug-in for Windows)
  - RapidSVN (wxWidgets-based GUI for Windows, OS X, *nix)
  - eSvn (Qt-based GUI for Windows, OS X, *nix)
  - Xcode (OS X)
  - SvnX (OS X 10.3+ GUI)
  - Scplugin (OS X Finder integration)
  - JSVN (Java GUI, requires command-line tools)

- Repository converters:
  - CVS to Subversion
  - Visual Source Safe to Subversion
  - Perforce to Subversion

- Web front-ends
  - Apache module (from core distribution)
  - WebSVN
Some striking things about Subversion

- File system requirements
- Using "svn status" vs. "cvs update"
- "Look ma! No network!"
- User umask issues
- Cleanup after "unexpected exits"
- Different keyword support (esp. no "log")
- Implications of the versioning strategy
  - Per-file
  - Per-project
- Tags, no; Copies, yes!
  - Parlays into recommended layout (trunk, branches, releases)
- No option for locking
- Timestamps as properties
- Conversion from CVS repositories (vs. import)
- Experience/openness issue
Useful resources

- "The Book" (Version Control with Subversion)
  - Free on the web (http://svnbook.red-bean.com/)
  - Published by O'Reilly
- The website (http://subversion.tigris.org)
- The mailing list (users@subversion.tigris.org)
  - Archived on the web site
- www.oreillynet.com
  - Lots of good articles, blogs
- Google
  - When I've run into problems (usually when trying to do a build early on), I frequently found answers quickly
Some useful articles

- **General overviews/usage**
  - "Single-user Subversion", by Rafael Garcia-Suarez
  - "Multi-user Subversion", by Rafael Garcia-Suarez

- **Comparisons to other tools**
  - "Better SCM Initiative: Comparison", by Shlomi Fish
    [http://better-scm.berlios.de/comparison/](http://better-scm.berlios.de/comparison/)
  - "Subversion for CVS users", by Mike Mason
    [http://osdir.com/Article203.phtml](http://osdir.com/Article203.phtml)
  - "Debunking BitMover’s Subversion comparison", by Karl Fogel
    [http://subversion.tigris.org/bitmover-svn.html](http://subversion.tigris.org/bitmover-svn.html)

- **Migration experiences/support**
  - "cvs2svn Documentation" (Tigris.org)
    [http://cvs2svn.tigris.org/cvs2svn.html](http://cvs2svn.tigris.org/cvs2svn.html)
  - "Migrating from CVS to Subversion", by Jeremy Mates
So if you want to be Subversive....

• The Subversion binaries for OS X and Windows can be easily obtained off the 'net

• Binaries for Solaris can be found in ~mjh/pub/subversion
  - Currently, these are for SVN 1.1.0
  - I expect to upgrade them to 1.1.3 soon
  - I also hope to convince Linus and team to start managing these for me, so that they don't have to live in my account....
Any questions?
Sample Subversion Sessions
Sample session: Creating a repository

- Creating a local repository (using Berkeley DB):

  $ svnadmin create /path/to/repos
  
  $ ls /path/to/repos
  README.txt  conf/  dav/  db/  format  hooks/  locks/

  $ ls repos/hooks/
  post-commit.tmpl  pre-revprop-change.tmpl
  post-revprop-change.tmpl  start-commit.tmpl
  pre-commit.tmpl

- Creating a local repository (using FS-based storage):

  $ svnadmin create --fs-type fsfs /path/to/repos

- Note:
  - Repository creation must be done on the machine hosting the repository; you can't specify a URL for this operation
  - File-system based storage is new to Subversion 1.1, and has therefore been less heavily tested than the BDB-based repository approach
  - For security reasons, the hook scripts are executed with an empty environment: not even the PATH is set. The scripts should therefore set the PATH themselves, or use absolute paths to programs.
Sample session: Doing a check-out

- Checking out via svn+ssh:

  $ svn checkout svn+ssh://user@host/path/to/repos
  Checked out revision 0.

  $ ls -aF repos
  ./  ../  .svn/

  $ ls -aF repos/.svn
  ./    empty-file*  prop-base/  tmp/
  ../   entries*     props/    wcprops/
  README.txt* format*  text-base/
Sample session: Adding content

• Example:

$ cd repos

$ mkdir src doc

$ echo "This is a test" > doc/Test.txt

$ ls -Fa doc

./     ../    Test.txt

$ svn add doc src

A       doc
A       doc\Test.txt
A       src
A       src\HelloWorld.java

• Notes:
  - By default, Subversion will recursively add the contents of a directory to the repository. To disable this, use the "--non-recursive" flag.
  - Subversion allows you to add files, directories, and symbolic links.
Sample session: Committing changes

• Example:

  $ svn commit -m "Initial check-in"
  Adding trunk
  Adding trunk\doc
  Adding trunk\doc\Test.txt
  Adding trunk\src
  Adding trunk\src\HelloWorld.java
  Transmitting file data ..
  Committed revision 1.

• Notes:

  - As with CVS, you can either specify the check-in comment on the command line (via "-m"), or you can let Subversion open an editor in which you'll enter the description.
Sample session: A commit with differences

• Example:

```bash
$ svn commit -m "Broke the message out into two lines"
Sending src\HelloWorld.java
Transmitting file data .svn: Commit failed (details follow):
svn: Out of date: '/trunk/src/HelloWorld.java' in transaction '3'

$ svn update
G HelloWorld.java
Updated to revision 2.

$ svn commit -m "Broke the message out into two lines"
Sending src\HelloWorld.java
Transmitting file data .
Committed revision 3.
```

• Notes:
  - As with CVS, you're responsible for making sure that your local copy of the file is based on the current "HEAD" version before you can commit.
Sample session: Conflict identification

• Example:

```sh
$ svn commit -m "Checking in a conflicting change"
Sending src\HelloWorld.java
Transmitting file data .svn: Commit failed (details follow):
svn: Out of date: '/trunk/src/HelloWorld.java' in transaction '6'

$ svn update
C HelloWorld.java
Updated to revision 4.

$ ls
HelloWorld.java.mine   HelloWorld.java.r4
HelloWorld.java       HelloWorld.java.r3
```
Sample session: Committing with known conflicts

• Example:

```
$ cat HelloWorld.java
/**
<<<<<<<<< .mine
  * This is a demo class, used as sample content for
<<<<<<<<<
  * This is a class, used as an example in
>>>>>>>>> .r4
  * a demonstration of Subversion.
...

$ svn commit -m "Trying to recommit without resolution"
svn: Commit failed (details follow):
svn: Aborting commit:
'\r:/Temp/subversionDemo/prepopulated/trunk/src/HelloWorld.java' remains in conflict
```

• Notes:
  - As with CVS, you're responsible for making sure that conflicts are resolved before you may commit.
Sample session: Resolving conflict

- Example:

  ```
  $ vi HelloWorld.java  # Fixing the problems....
  $ ls                  # "Conflict files" still there
  HelloWorld.java.mine  HelloWorld.java.r4
  HelloWorld.java       HelloWorld.java.r3
  
  $ svn resolved HelloWorld.java
  Resolved conflicted state of 'HelloWorld.java'
  
  $ ls                  # "Conflict files" are now gone
  HelloWorld.java
  
  $ svn commit -m "Updated code to handle....."
  Sending    HelloWorld.java
  Transmitting file data .
  Committed revision 5.
  ```
Sample session: Creating a branch

• Example:

```bash
$ svn copy svn+ssh://user@host/.../repos/calc/trunk \
svn+ssh://user@host/.../repos/calc/branches/my-calc-branch \
-m "Creating a private branch of /calc/trunk."
Committed revision 341.

$ svn checkout \
svn+ssh://user@host/.../repos/calc/branches/my-calc-branch
```

• Notes:

  – This is the core reason for the recommendation for a "trunk" folder within the repository
  – Many entity-level operations can be performed directly on the repository (and are thus immediately committed); no working copy is actually needed at all in these cases
  – Copying directories is *cheap* (O(1)); a new directory entry is created that points at the existing tree. As modifications are made to the files within the copied directory, only that file changes.
• Unlike many other version control systems, Subversion's branches exist as *normal file system directories* in the repository, not in an extra dimension. These directories just happen to carry some extra historical information.
Merging between branches

• Changes between branches (or the trunk) can be merged together with the "svn merge" command
  - This allows you to specify a range of versions from which the change set is to be extracted (e.g., "apply the changes that were made in the trunk between versions 1234 and HEAD, inclusive, to my branch")
  - You can specify that a "dry run" of the merge is to take place, which simply shows you the resulting status that each affected file would have at the end of the merge (e.g., updated, added, conflicting, etc.)
A caution about merging

• Given two different branches of development that need to be merged, a frequent mistake is to just merge the current copy of the source branch with a current copy of the target
  - "svn merge" operates like "svn diff": comparing the latest trees will not merely describe the set of changes you made to your branch; it will also show the removal of changes in the target branch that never happened in the source tree!

• You need to compare the initial state of your source branch to its final state, and use that as the basis for the merge
Sample session: Merging from the trunk into a branch

- Example:

  # Find out where the branch was created....
  $ svn log --verbose --stop-on-copy \
  svn+ssh://user@host/.../repos/calc/branches/my-calc-branch
  ...

---------------------------------------------------------------

r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
Branching and Merging
55
A /calc/branches/my-calc-branch (from /calc/trunk:340)

# Do the merge
$ cd calc/trunk
$ svn update
At revision 405.

$ svn merge -r 341:HEAD \
  svn+ssh://user@host/.../repos/calc/branches/my-calc-branch
U integer.c
U button.c
U Makefile

$ svn status
M integer.c
M button.c
M Makefile

# Examine the diffs, compile, test, etc., and then commit the changes when
# you think it's ready....
$ svn commit -m "Merged my-calc-branch changes r341:405 into the trunk."
More about merging

- Commit messages from a merge should always reference the range of changes that was merged in.
  - That way, you can restrict future ranges from the source branch to just the changes that are new since the last merge.
  - By including the range in the commit message, you can quickly find the end point for the range in the log messages, and thus restrict yourself to "lastEndPoint+1:HEAD".

- Merging can also be used to roll back changes, by specifying a reverse range:

  ```
  $ pwd
  /Users/jsmith/project/calc/trunk
  
  $ svn merge -r 303:302 \
  \n  $ svn+ssh://user@host/.../repos/calc/trunk
  U integer.c
  
  $ svn status
  M integer.c
  
  $ svn diff # verify the changes were removed
  . . .
  
  $ svn commit -m "Undoing change committed in version 303"
  . . .
  ```
Some Subversion "cheat sheets"

Handy snippets to have lying around
(someday, I'll put them in a Wiki....)
Cheat sheet: Status codes

- **U**: Updated from the server
- **A**: Added to your working copy
- **D**: Deleted from the working copy
- **R**: Replaced in the working copy (old one deleted, new one added)
- **G**: Updated locally changed file from server; merge was successful
- **C**: Updated locally changed file from server; conflicts detected
- **L**: Lock detected in the .svn directory for this entity
- **I**: Not managed by Subversion, and explicitly ignored
- **S**: This file/directory has been switched to a branch
- **?**: Not managed by Subversion
- **!**: Managed by Subversion, but missing/incomplete
- **~**: Versioned as file/directory/link, but type has changed
Cheat sheet: Useful standard properties

- **svn:executable**
  - No defined values; just present or not
  - Tells svn to make sure that the "execute" bit is kept enabled

- **svn:mime-type**
  - "text/...." tells Subversion the file is "human-mergeable"
  - Used by Apache module to set the Content-type: header

- **svn:ignore**
  - Multi-line list of file patterns (one per line)
  - Additive to the "global-ignores" run-time configuration option
  - Can be directly set from a .cvsignore file's contents:
    
    ```
    $ svn propset svn:ignore -F .cvsignore .
    ```

- **svn:keywords**
  - Defines a list of keywords for which substitution will take place within a file
  - Supported set of keywords are:
    - LastChangedDate (Date)
    - LastChangedRevision (Revision, or Rev)
    - LastChangedBy (Author)
    - HeadURL (URL)
    - Id

- **svn:eol-style**
  - Possible values are: "native", "CRLF", "LF", or "CR"