Subversion

A New Version Control System

Jim Blandy <jimb@redhat.com>
Karl Fogel <kfogel@collab.net>
Ben Collins-Sussman <sussman@collab.net>

First version: June 2000.
Last updated: $Date: 2000/12/08 01:36:45 $

Short Contents

. . . . 1
1 Goals 3
2 Model 7
3 Architecture 17
4 Deltas 19
5 Client 29
6 Protocol 33
7 Server 35
8 Future 49
9 License 53

Table of Contents

. . . . 1

1 Goals 3

1.1 Rename/removal/resurrection support 3
1.2 Text vs binary issues 4
1.3 I18N/Multilingual support 4
1.4 Branching and tagging 4
1.5 Miscellaneous new behaviors 5
1.5.1 Log messages 5
1.5.2 Client side diff plug-ins 5
1.5.3 Better merging 5
1.5.4 Conflicts resolution 5

2 Model 7

2.1 Working Directories and Repositories 7
2.2 Transactions and Revision Numbers 8
2.3 How Working Directories Track the Repository 9
2.4 Subversion Does Not Lock Files 10
2.5 Properties 11
2.6 Merging and Ancestry 12

3 Architecture 17

3.1 Client Layer 17
3.2 Network Layer 18
3.3 Filesystem Layer 18

4 Deltas 19

4.1 Text Deltas 19
4.2 Property Deltas 20
4.3 Tree Deltas 20
4.4 Postfix Text Deltas 25
4.5 Serializing Deltas via the "Editor" Interface 25

5 Client 29

5.1 Working copies and the working copy library 29
5.1.1 The layout of working copies 29
5.1.2 The working copy management library 30
5.2 The repository access library 31
5.3 The client operation library 31

6 Protocol 33

7 Server ... 35
7.1 Filesystem ... 35
7.1.1 Filesystem Overview ... 35
7.1.2 API ... 35
7.1.3 Repository Structure ... 35
7.1.3.1 Schema ... 36
7.1.3.2 Bubble-Up Method ... 37
7.1.3.3 Diffy Storage ... 45
7.1.4 Implementation ... 46
7.2 Server Library ... 46
7.2.1 Security ... 46
7.2.1.1 Username Authentication ... 46
7.2.1.2 Authorization ... 47

8 Future ... 49
8.1 High Priority Items ... 49
8.1.1 Annotation ... 49
8.1.2 Repository Conversion ... 49
8.1.3 User Interface for Branches and Tags ... 49
8.2 Blue Sky Items ... 50
8.2.1 Server Grep ... 50
8.2.2 Scripting Language ... 50
8.2.3 Smart conflict resolution ... 50
8.2.4 Mirroring Servers ... 50
8.2.5 Inter-Repository Communication ... 50
8.2.6 SQL Back-End ... 51
8.2.7 Digital Signatures ... 51
8.2.8 Import/Export format ... 51
8.2.9 SMTP access ... 51

9 License ... 53

1 Goals

The goal of the Subversion project is write a version control system that takes over CVS's current and future user base[1]. The first release has all the major features of CVS, plus certain new features that CVS users often wish they had. In general, Subversion works like CVS, except where there's a compelling reason to be different.

So what does Subversion have that CVS doesn't?

- It versions directories, file-metadata, renames, copies and removals/resurrections. In other words, Subversion records the changes users make to directory trees, not just changes to file contents.
- Tagging and branching are constant-time and constant-space.
- It is natively client-server, hence much more maintainable than CVS. (In CVS, the client-server protocol was added as an afterthought. This means that most new features have to be implemented twice, or at least more than once: code for the local case, and code for the client-server case.)
- The repository is organized efficiently and comprehensibly. (Without going into too much detail, let's just say that CVS's repository structure is showing its age.)
- Commits are atomic. Each commit results in a single revision number, which refers to the state of the entire tree. Files no longer have their own revision numbers.
- The locking scheme is only as strict as absolutely necessary. Reads are never locked, and writes lock only the files being written, for only as long as needed.
- It has internationalization support.
- It handles binary files gracefully (experience has shown that CVS's binary file handling is prone to user error).
- It has a fully-realized internal concept of users and access permissions.
- It takes advantage of the Net's experience with CVS by choosing better default behaviors for certain situations.

Some of these advantages are clear and require no further discussion. Others are not so obvious, and are explained in greater detail below.

1.1 Rename/removal/resurrection support

Full rename support means you can trace through ancestry by name *or* by entity. For example, if you say "Give me revision 12 of foo.c", do you mean revision 12 of the file whose name is *now* foo.c (but perhaps it was named bar.c back at revision 12), or the file whose name was foo.c in revision 12 (perhaps that file no longer exists, or has a different name now)? In Subversion, both interpretations are available to the user.

---

[1] If you're not familiar with CVS or its shortcomings, then skip to Chapter 2 [Model], page 7

1.2 Text vs binary issues

Historically, binary files have been problematic in CVS for two unrelated reasons: keyword expansion, and line-end conversion.

- *Keyword expansion* is when CVS expands "$Revision$" into "$Revision 1.1$", for example. There are a number of keywords in CVS: "$Author$", "$Date$", and so on.

- *Line-end conversion* is when CVS gives plaintext files the appropriate line-ending conventions for the working copy's platform. For example, Unix working copies use LF, but Windows working copies use CRLF. (Like CVS, the Subversion repository stores text files in Unix LF format).

Both keyword substitution and line-end conversion are sensible only for plain text files. CVS only recognizes two file types anyway: plaintext and binary. And CVS assumes files are plain text unless you tell it otherwise.

Subversion recognizes the same two types. The question is, how does it determine a file's type? Experience with CVS suggests that assuming text unless told otherwise is a losing strategy – people frequently forget to mark images and other opaque formats as binary, then later they wonder why CVS mangled their data. So Subversion assumes a file is binary, unless it matches a standard text pattern (.c, .h, .pl, .html, .txt, README, and so on). When necessary, the user can explicitly set the type for a file or file pattern.

Text files undergo line-end conversion by default. Users can turn line-end conversion on or off per file pattern, or per file. Text files do *not* undergo keyword substitution by default, on the theory that if someone wants substitution and isn't getting it, they'll look in the manual; but if they are getting it and didn't want it, they might just be confused and not know what to do. Users can turn substitution on or off per project, or per file pattern, or per file.

Both of these changes are done on the client side; the repository does not even know about them.

Changes to file type are versioned – the type is associated with a particular revision of the file, and new revisions inherit from previous revisions except when told otherwise.

1.3 I18N/Multilingual support

Subversion is internationalized – commands, user messages, and errors can be customized to the appropriate human language at build-time (or run time, if that's not much harder).

File names and contents may be multilingual; Subversion does not assume an ASCII-only universe. For purposes of keyword expansion and line-end conversion, Subversion also understands the UTF-* encodings (but not necessarily all of them by the first release).

1.4 Branching and tagging

Subversion supports branching and tagging with one efficient operation: 'clone'. To clone a tree is to copy it, to create another tree exactly like it (except that the new tree knows its ancestry relationship to the old one).

At the moment of creation, a clone requires only a small, constant amount of space in the repository – most of its storage is shared with the original tree. If you never commit anything on the clone, then it's just like a CVS tag. If you start committing on it, then it's a branch. Voila! This also implies CVS's "vendor branching" feature, since Subversion has real rename and directory support.

Note that from the user's point of view, there are still separate branch and tag commands, with the latter initializing the clone as read-only (i.e., if a static snapshot is going to become an active line of development, one at least wants users to be aware of the change).

1.5 Miscellaneous new behaviors

1.5.1 Log messages

Subversion has a flexible log message policy (a small matter, but one dear to our hearts).

Log messages should be a matter of project policy, not version control software policy. If a user commits with no log message, then Subversion defaults to an empty message. (CVS tries to require log messages, but fails: we've all seen empty log messages in CVS, where the user committed with deliberately empty quotes. Let's stop the madness now.)

1.5.2 Client side diff plug-ins

Subversion supports client-side plug-in diff programs.

There is no need for Subversion to have every possible diff mechanism built in. It can invoke a user-specified client-side diff program on the two revisions of the file(s) locally.

1.5.3 Better merging

Subversion remembers what has already been merged in and what hasn't, thereby avoiding the problem, familiar to CVS users, of spurious conflicts on repeated merges.

For details, See Section 2.6 [Merging and Ancestry], page 12.

1.5.4 Conflicts resolution

For text files, Subversion resolves conflicts similarly to CVS, by folding repository changes into the working files with conflict markers. But, for *both* text and binary files, Subversion also always puts the pristine repository revision in one temporary file, and the pristine working copy revision in another temporary file.

Thus, in a text conflict, the user has three files readily at hand,

1. the original working copy file
2. the repository revision from which the update was taken
3. the combined file, with conflict markers

and in a binary file conflict, the user at least has 1 and 2.

When the conflict has been resolved and the working copy is committed, Subversion can automatically remove the temporary pristine files.

A more general solution would allow plug-in merge resolution tools on the client side; but this is not scheduled for the first release (see Chapter 8 [Future], page 49). Note that users can use their own merge tools anyway, since all the original files are available.

2 Model

This chapter explains the user's view of Subversion — what "objects" you interact with, how they behave, and how they relate to each other.

2.1 Working Directories and Repositories

Suppose you are using Subversion to manage a software project. There are two things you will interact with: your working directory, and the repository.

Your *working directory* is an ordinary directory tree, on your local system, containing your project's sources. You can edit these files and compile your program from them in the usual way. Your working directory is your own private work area: Subversion never changes the files in your working directory, or publishes the changes you make there, until you explicitly tell it to do so.

After you've made some changes to the files in your working directory, and verified that they work properly, Subversion provides commands to publish your changes to the other people working with you on your project. If they publish their own changes, Subversion provides commands to incorporate those changes into your working directory.

A working directory contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, these files help Subversion recognize which files contain unpublished changes, and which files are out-of-date with respect to others' work.

While your working directory is for your use alone, the *repository* is the common public record you share with everyone else working on the project. To publish your changes, you use Subversion to put them in the repository. (What this means, exactly, we explain below.) Once your changes are in the repository, others can tell Subversion to incorporate your changes into their working directories. In a collaborative environment like this, each user will typically have their own working directory (or perhaps more than one), and all the working directories will be backed by a single repository, shared amongst all the users.

A Subversion repository holds a single directory tree, and records the history of changes to that tree. The repository retains enough information to recreate any prior state of the tree, compute the differences between any two prior trees, and report the relations between files in the tree — which files are derived from which other files.

A Subversion repository can hold the source code for several projects; usually, each project is a subdirectory in the tree. In this arrangement, a working directory will usually correspond to a particular subtree of the repository.

For example, suppose you have a repository laid out like this:

```
/trunk/paint/Makefile
             canvas.c
             brush.c
       write/Makefile
             document.c
             search.c
```

In other words, the repository's root directory has a single subdirectory named '`trunk`', which itself contains two subdirectories: '`paint`' and '`write`'.

To get a working directory, you must *check out* some subtree of the repository. If you check out '/trunk/write', you will get a working directory like this:

```
write/Makefile
      document.c
      search.c
      SVN/
```

This working directory is a copy of the repository's '/trunk/write' directory, with one additional entry — 'SVN' — which holds the extra information needed by Subversion, as mentioned above.

Suppose you make changes to 'search.c'. Since the 'SVN' directory remembers the file's modification date and original contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to.

To publish your changes, you can use Subversion's 'commit' command:

```
$ pwd
/home/jimb/write
$ ls
Makefile   SVN/    document.c    search.c
$ svn commit search.c
$
```

Now your changes to 'search.c' have been committed to the repository; if another user checks out a working copy of '/trunk/write', they will see your text.

Suppose you have a collaborator, Felix, who checked out a working directory of '/trunk/write' at the same time you did. When you commit your change to 'search.c', Felix's working copy is left unchanged; Subversion only modifies working directories at the user's request.

To bring his working directory up to date, Felix can use the Subversion 'update' command. This will incorporate your changes into his working directory, as well as any others that have been committed since he checked it out.

```
$ pwd
/home/felix/write
$ ls
Makefile    SVN/   document.c    search.c
$ svn update
U search.c
$
```

The output from the 'svn update' command indicates that Subversion updated the contents of 'search.c'. Note that Felix didn't need to specify which files to update; Subversion uses the information in the 'SVN' directory, and further information in the repository, to decide which files need to be brought up to date.

We explain below what happens when both you and Felix make changes to the same file.

2.2 Transactions and Revision Numbers

A Subversion 'commit' operation can publish changes to any number of files and directories as a single atomic transaction. In your working directory, you can change files' contents, create, delete, rename and copy files and directories, and then commit the completed set of changes as a unit.

In the repository, each commit is treated as an atomic transaction: either all the commit's changes take place, or none of them take place. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions. We may call a commit a *transaction* when we want to emphasize its indivisible nature.

Each time the repository accepts a transaction, this creates a new state of the tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number of the previous revision. The initial revision of a freshly created repository is numbered zero, and consists of an empty root directory.

Since each transaction creates a new revision, with its own number, we can also use these numbers to refer to transactions; transaction n is the transaction which created revision n. There is no transaction numbered zero.

Unlike those of many other systems, Subversion's revision numbers apply to an entire tree, not individual files. Each revision number selects an entire tree.

It's important to note that working directories do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working directory from a repository whose most recent revision is 4:

```
write/Makefile:4
      document.c:4
      search.c:4
```

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to '`search.c`', and commit that change. Assuming no other commits have taken place, your commit will create revision 5 of the repository, and your working directory will look like this:

```
write/Makefile:4
      document.c:4
      search.c:5
```

Suppose that, at this point, Felix commits a change to '`document.c`', creating revision 6. If you use '`svn update`' to bring your working directory up to date, then it will look like this:

```
write/Makefile:6
      document.c:6
      search.c:6
```

Felix's changes to '`document.c`' will appear in your working copy of that file, and your change will still be present in '`search.c`'. In this example, the text of '`Makefile`' is identical in revisions 4, 5, and 6, but Subversion will mark your working copy with revision 6 to indicate that it is still current. So, after you do a clean update at the root of your working directory, your working directory will generally correspond exactly to some revision in the repository.

2.3 How Working Directories Track the Repository

For each file in a working directory, Subversion records two essential pieces of information:

- what revision of what repository file your working copy is based on (this is called the file's *base revision*), and
- a timestamp recording when the local copy was last updated.

Given this information, by talking to the repository, Subversion can tell which of the following four states a file is in:

- **Unchanged, and current.** The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its base revision.
- **Locally changed, and current**. The file has been changed in the working directory, and no changes to that file have been committed to the repository since its base revision. There are local changes that have not been committed to the repository.
- **Unchanged, and out-of-date**. The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the public revision.
- **Locally changed, and out-of-date**. The file has been changed both in the working directory, and in the repository. The file should be updated; Subversion will attempt to merge the public changes with the local changes. If it can't complete the merge in a plausible way automatically, Subversion leaves it to the user to resolve the conflict.

2.4 Subversion Does Not Lock Files

Subversion does not prevent two users from making changes to the same file at the same time. For example, if both you and Felix have checked out working directories of '`/trunk/write`', Subversion will allow both of you to change '`write/search.c`' in your working directories. Then, the following sequence of events will occur:

- Suppose Felix tries to commit his changes to '`search.c`' first. His commit will succeed, and his text will appear in the latest revision in the repository.
- When you attempt to commit your changes to '`search.c`', Subversion will reject your commit, and tell you that you must update '`search.c`' before you can commit it.
- When you update '`search.c`', Subversion will try to merge Felix's changes from the repository with your local changes. By default, Subversion merges as if it were applying a patch: if your local changes do not overlap textually with Felix's, then all is well; otherwise, Subversion leaves it to you to resolve the overlapping changes. In either case, Subversion carefully preserves a copy of the original pre-merge text.
- Once you have verified that Felix's changes and your changes have been merged correctly, you can commit the new revision of '`search.c`', which now contains everyone's changes.

Some version control systems provide "locks", which prevent others from changing a file once one person has begun working on it. In our experience, merging is preferable to locks, because:

- changes usually do not conflict, so Subversion's behavior does the right thing by default, while locking can interfere with legitimate work;
- locking can prevent conflicts within a file, but not conflicts between files (say, between a C header file and another file that includes it), so it doesn't really solve the problem; and finally,

- people often forget that they are holding locks, resulting in unnecessary delays and friction.

Of course, the merge process needs to be under the users' control. Patch is not appropriate for files with rigid formats, like images or executables. Subversion allows users to customize its merging behavior on a per-file basis. You can direct Subversion to refuse to merge changes to certain files, and simply present you with the two original texts to choose from. Or, you can direct Subversion to merge using a tool which respects the semantics of the file format.

2.5 Properties

Files generally have interesting attributes beyond their contents: owners and groups, access permissions, creation and modification times, and so on. Subversion attempts to preserve these attributes, or at least record them, when doing so would be meaningful. However, different operating systems support very different sets of file attributes: Windows NT supports access control lists, while Linux provides only the simpler traditional Unix permission bits.

In order to interoperate well with clients on many different operating systems, Subversion supports *property lists*, a simple, general-purpose mechanism which clients can use to store arbitrary out-of-band information about files.

A property list is a set of name / value pairs. A property name is an arbitrary text string, expressed as a Unicode UTF-8 string, canonically decomposed and ordered. A property value is an arbitrary string of bytes. Property values may be of any size, but Subversion may not handle very large property values efficiently. No two properties in a given a property list may have the same name. Although the word 'list' usually denotes an ordered sequence, there is no fixed order to the properties in a property list; the term 'property list' is historical.

Each revision number, file, directory, and directory entry in the Subversion repository, has its own property list. Subversion puts these property lists to several uses:

- Clients use properties to store file attributes, as described above. For example, the Unix Subversion client records the files' permission bits as the value of a property called '`svn-posix-access-permission`'. Operating systems which allow files to have more than one name, like Windows 95, can use directory entry property lists to record files' alternative names.
- The Subversion server uses properties to hold attributes of its own, and allow clients to read and modify them. For example, the '`svn-acl`' property holds an access control list which the Subversion server uses to regulate access to repository files.
- Users can invent properties of their own, to store arbitrary information for use by scripts, build environments, and so on. Names of user properties should be URI's, to avoid conflicts between organizations.

Property lists are versioned, just like file contents. You can change properties in your working directory, but those changes are not visible in the repository until you commit your local changes. If you do commit a change to a property value, other users will see your change when they update their working directories.

2.6 Merging and Ancestry

Subversion defines merges the same way CVS does: to merge means to take a set of previously committed changes and apply them, as a patch, to a working copy. This change can then be committed, like any other change. (In Subversion's case, the patch may include changes to directory trees, not just file contents.)

As defined thus far, merging is equivalent to hand-editing the working copy into the same state as would result from the patch application. In fact, in CVS there *is* no difference – it is equivalent to just editing the files, and there is no record of which ancestors these particular changes came from. Unfortunately, this leads to conflicts when users unintentionally merge the same changes again. (Experienced CVS users avoid this problem by using branch- and merge-point tags, but that involves a lot of unwieldy bookkeeping.)

In Subversion, merges are remembered by recording *ancestry sets*. A revision's ancestry set is the set of all changes "accounted for" in that revision. By maintaining ancestry sets, and consulting them when doing merges, Subversion can detect when it would apply the same patch twice, and spare users much bookkeeping. Ancestry sets are stored as properties.

In the examples below, bear in mind that revision numbers usually refer to changes, rather than the full contents of that revision. For example, "the change A:4" means "the delta that resulted in A:4", not "the full contents of A:4".

The simplest ancestor sets are associated with linear histories. For example, here's the history of a file A:

```
 _____        _____        _____        _____        _____
|     |      |     |      |     |      |     |      |     |
| A:1 |----->| A:2 |----->| A:3 |----->| A:4 |----->| A:5 |
|_____|      |_____|      |_____|      |_____|      |_____|
```

The ancestor set of A:5 is:

```
{ A:1, A:2, A:3, A:4, A:5 }
```

That is, it includes the change that brought A from nothing to A:1, the change from A:1 to A:2, and so on to A:5. From now on, ranges like this will be represented with a more compact notation:

```
{ A:1-5 }
```

Now assume there's a branch B based, or "rooted", at A:2. (This postulates an entirely different revision history, of course, and the global revision numbers in the diagrams will change to reflect it.) Here's what the project looks like with the branch:

```
 _____        _____        _____        _____        _____        _____
|     |      |     |      |     |      |     |      |     |      |     |
| A:1 |----->| A:2 |----->| A:4 |----->| A:6 |----->| A:8 |----->| A:9 |
|_____|      |_____|      |_____|      |_____|      |_____|      |_____|
                \
                 \
                  \  _____        _____        _____
                   \|     |      |     |      |     |
                    | B:3 |----->| B:5 |----->| B:7 |
                    |_____|      |_____|      |_____|
```

If we produce A:9 by merging the B branch back into the trunk

```
 _____        _____        _____        _____        _____        _____
|     |      |     |      |     |      |     |      |     |      |     |
| A:1 |----->| A:2 |----->| A:4 |----->| A:6 |----->| A:8 |---.->| A:9 |
|_____|      |_____|      |_____|      |_____|      |_____|  /   |_____|
                \                                            |
                 \                                           |
                  \  _____        _____        _____        /
                   \|     |      |     |      |     |      /
                    | B:3 |----->| B:5 |----->| B:7 |--->-'
                    |_____|      |_____|      |_____|
```

then what will A:9's ancestor set be?

```
{ A:1, A:2, A:4, A:6, A:8, A:9, B:3, B:5, B:7}
```

or more compactly:

```
{ A:1-9, B:3-7 }
```

(It's all right that each file's ranges seem to include non-changes; this is just a notational convenience, and you can think of the non-changes as either not being included, or being included but being null deltas as far as that file is concerned).

All changes along the B line are accounted for (changes B:3-7), and so are all changes along the A line, including both the merge and any non-merge-related edits made before the commit.

Although this merge happened to include all the branch changes, that needn't be the case. For example, the next time we merge the B line

```
 _____     _____     _____     _____     _____      _____      _____
|     |   |     |   |     |   |     |   |     |    |     |    |     |
| A:1 |-->| A:2 |-->| A:4 |-->| A:6 |-->| A:8 |-.->| A:9 |-.->|A:11 |
|_____|   |_____|   |_____|   |_____|   |_____| |  |_____| |  |_____|
             \                                 /           |
              \                               /            |
               \  _____     _____     _____  / _____       |
                \|     |   |     |   |     |/ |     |     /
                 | B:3 |-->| B:5 |-->| B:7 |-->|B:10 |->-'
                 |_____|   |_____|   |_____|   |_____|
```

Subversion will know that A's ancestry set already contains B:3-7, so only the difference between B:7 and B:10 will be applied. A's new ancestry will be

```
{ A:1-11, B:3-10 }
```

But why limit ourselves to contiguous ranges? An ancestry set is truly a set – it can be any subset of the changes available:

```
 _____        _____        _____        _____        _____        _____
|     |      |     |      |     |      |     |      |     |      |     |
| A:1 |----->| A:2 |----->| A:4 |----->| A:6 |----->| A:8 |--.-->|A:10 |
|_____|      |_____|      |_____|      |_____|      |_____| /    |_____|
                |                                          /
                |                ______________________.__/
                |               /                      |
                |              /                       |
                \           __/_                      _|__
                 \         {    }                    {    }
                  \  _____        _____        _____        _____
                   \|     |      |     |      |     |      |     |
                    | B:3 |----->| B:5 |----->| B:7 |----->| B:9 |----->
                    |_____|      |_____|      |_____|      |_____|
```

In this diagram, the change from B:3-5 and the change from B:7-9 are merged into a working copy whose ancestry set (so far) is { A:1-8 } plus any local changes. After committing, A:10's ancestry set is

```
{ A:1-10, B:5, B:9 }
```

Clearly, saying "Let's merge branch B into A" is a little ambiguous. It usually means "Merge all the changes accounted for in B's tip into A", but it *might* mean "Merge the single change that resulted in B's tip into A".

Any merge, when viewed in detail, is an application of a particular set of changes – not necessarily adjacent ones – to a working copy. The user-level interface may allow some of these changes to be specified implicitly. For example, many merges involve a single, contiguous range of changes, with one or both ends of the range easily deducible from context (i.e., branch root to branch tip). These inference rules are not specified here, but it should be clear in most contexts how they work.

Because each node knows its ancestors, Subversion never merges the same change twice (unless you force it to). For example, if after the above merge, you tell Subversion to merge all B changes into A, Subversion will notice that two of them have already been merged, and so merge only the other two changes, resulting in a final ancestry set of:

```
{ A:1-10, B:3-9 }
```

This description of merging and ancestry applies to both intra- and inter-repository merges. However, inter-repository merging will probably not be implemented until a future release of Subversion (see Chapter 8 [Future], page 49).

3 Architecture

Subversion is conceptually divided into a number of separable layers.

Assuming that the programmatic interface of each layer is well-defined, it is easy to customize the different parts of the system. Contributors can write new client apps, new network protocols, new server processes, new server features, and new storage back-ends.

The following diagram illustrates the "layered" architecture, and where each particular interface lies.

```
                   +--------------------+
                   | commandline or GUI |
                   |   client app       |
        +----------+--------------------+----------+ <=== Client interface
        |              Client Library              |
        |                                          |
        |        +----+                            |
        |        |    |                            |
+-------+--------+    +--------------+--+----------+ <=== Network interface
| Working Copy   |    |    Remote    |  | Local    |
| Management lib |    | Repos Access |  | Repos    |
+----------------+    +--------------+  | Access   |
                      |     neon     |  |          |
                      +--------------+  |          |
                         ^              |          |
                        /               |          |
                  DAV  /                |          |
                      /                 |          |
                     v                  |          |
             +---------+                |          |
             |         |                |          |
             | Apache  |                |          |
             |         |                |          |
             +---------+                |          |
             | mod_DAV |                |          |
           +-------------+              |          |
           | mod_DAV_SVN |              |          |
+----------+-------------+--------------+----------+ <=== Filesystem interface
|                                                  |
|               Subversion Filesystem              |
|                                                  |
+--------------------------------------------------+
```

3.1 Client Layer

The Subversion client, which may be either command-line or GUI, draws on three libraries.

The first library provides an API for managing the client's working copy of a project. This includes operations like renaming or removal of files, patching files, extracting local diffs, and routines for maintaining administrative files in the '`SVN/`' directory.

The second library provides an API for exchanging information with a Subversion repository. This includes the ability to read files, write new revisions of files, and ask the repository to compare a working copy against its latest revision. Note that there are two implementations of this interface: one designed to talk to a repository over a network, and one designed to work with a repository on local disk.

The third library provides general client functions such as `update()` and `commit()`, which may involve one or both of the other two client libraries.

For details, See Chapter 5 [Client], page 29.

3.2 Network Layer

The network layer's job is to move the repository API requests over a wire.

On the client side, a network library translates these requests into a set of WebDAV DeltaV requests. The information is sent over TCP/IP to an Apache server. Apache is used for the following reasons:

- it is time-tested and extremely stable;
- it has built-in load-balancing;
- it has built-in proxy and firewall support;
- it has authentication and encryption features;
- it allows client-side caching;
- it has an extensible module system

Our suspicion is that any attempt to write a dedicated "Subversion server" (with a "Subversion protocol") would inevitably end up evolving towards Apache's already-existing feature set. (However, Subversion's layered architecture certainly doesn't *prevent* anyone from writing a totally new network

An appropriate Apache module will translate the DAV requests into API calls against a particular repository.

For details, See Chapter 6 [Protocol], page 33.

3.3 Filesystem Layer

When the requests reach a particular repository, they are interpreted by the *Subversion Filesystem library*. The Subversion Filesystem is a custom Unix-like filesystem, with a twist: writes are revisioned and atomic, and no data is ever deleted! This filesystem is currently implemented on top of a normal filesystem, using Berkeley DBM files.

For a more detailed explanation: See Chapter 7 [Server], page 35.

4 Deltas

Subversion uses three kinds of deltas:

- A *tree delta* describes the difference between two arbitrary directory trees, the way a traditional patch describes the difference between two files. For example, the delta between directories A and B could be applied to A, to produce B.

  Tree deltas can also carry ancestry information, indicating how the files in one tree are related to files in the other tree. And deltas can describe changes to file meta-information, like permission bits, creation dates, and so on. The repository and working copy use deltas to communicate changes.
- A *text delta* describes changes to a string of bytes, such as the contents of a file. It is analogous to traditional patch format, except that it works equally well on binary and text files, and is not invertible (because context and deleted data are not recorded).
- A *property delta* describes changes to a list of named properties (see Section 2.5 [Properties], page 11).

The term *delta* without qualification generally means a tree delta, unless some other meaning is clear from context.

In the examples below, deltas will be described in XML, which happens to be Subversion's import/export format. However, note that deltas are an abstract data structure, of which the XML format is merely one representation. Later, we will describe other representations: for example, there is a serialized representation (useful for streaming protocols, among other things), and a db-style representation, used for repository storage. The various representations of a given delta are (in theory, anyway) perfectly isomorphic to one another, since they describe the same underlying structure.

4.1 Text Deltas

A text delta describes the difference between two strings of bytes, the *source* string and the *target* string. Given a source string and a target string, we can compute a text delta; given a source string and a delta, we can reconstruct the target string. However, note that deltas are not invertible: you cannot always reconstruct the source string given the target string and delta.

The standard Unix "diff" format is one possible representation for text deltas; however, diffs are not ideal for internal use by a revision control system, for several reasons:

- Diffs are line-oriented, which makes them human-readable, but sometimes makes them perform poorly on binary files.
- Diffs represent a series of replacements, exchanging selected ranges of the old text with new text; again, this is easy for humans to read, but it is more expensive to compute and less compact than some alternatives.

Instead, Subversion uses the VDelta binary-diffing algorithm, as described in *Hunt, J. J., Vo, K.-P., and Tichy, W. F. An empirical study of delta algorithms. Lecture Notes in Computer Science 1167 (July 1996), 49-66.* Currently, the output of this algorithm is stored in a custom data format called *svndiff*, invented by Greg Hudson `<ghudson@mit.edu>`, a Subversion developer.

The concrete form of a text delta is a well-formed XML element, having the following form:

`<text-delta>`*data*`</text-delta>`

Here, *data* is the raw svndiff data, encoded in the MIME Base64 format.

4.2 Property Deltas

A property delta describes changes to a property list, of the sort associated with files, directories, and directory entries, and revision numbers (see Section 2.5 [Properties], page 11). A property delta can record creating, deleting, and changing the text of any number of properties.

A property delta is an unordered set of name/change pairs. No two pairs within a given property delta have the same name. A pair's name indicates the property affected, and the change indicates what happens to its value. There are two kinds of changes:

`set` *value* Change the value of the named property to the byte string *value*. If there is no property with the given name, one is added to the property list.

`delete` Remove the named property from the property list.

At the moment, the `set` command can either create or change a property value. However, this simplification means that the server cannot distinguish between a client which believes it is creating a value afresh, and a client which believes it is changing the value of an existing property. It may simplify conflict detection to divide `set` into two separate `add` and `change` operations.

In the future, we may add a `text-delta` change, which specifies a change to an existing property's value as a text delta. This would give us a compact way to describe small changes to large property values.

The concrete form of a property delta is a well-formed XML element, having the following form:

`<property-delta>`*change*`...</property-delta>`

Each *change* in a property delta has one of the following forms:

`<set name='`*name*`'>`*value*`</set>`
`<delete name='`*name*`'/>`

The *name* attribute of a `set` or `delete` element gives the name of the property to change. The *value* of a `set` element gives the new value of the property.

If either the property name or the property value contains the characters '`&`', '`<`', or '`'`', they should be replaced with the sequences '`&#38`', '`&#60`', or '`&#39`', respectively.

4.3 Tree Deltas

A tree delta describes changes between two directory trees, the *source tree* and the *target tree*. Tree deltas can describe copies, renames, and deletions of files and directories, changes to file contents, and changes to property lists. A tree delta can also carry information about how the files in the target tree are derived from the files in the source tree, if this information is available.

The format for tree deltas described here is easy to compute from a Subversion working directory, and easy to apply to a Subversion repository. Furthermore, the size of a tree delta in this format is independent of the commands used to produce the target tree — it depends only on the degree of difference between the source and target trees.

A tree delta is interpreted in the context of three parameters:

- *source-root*, the name of the directory to which this complete tree delta applies,
- *revision*, indicating a particular revision of . . .
- *source-dir*, which is a directory in the source tree that we are currently modifying to yield . . .
- . . . *target-dir* — the directory we're constructing.

When we start interpreting a tree delta, *source-root*, *source-dir*, and *target-dir* are all equal. As we walk the tree delta, *target-dir* walks the tree we are constructing, and *source-dir* walks the corresponding portion of the source tree, which we use as the original. *Source-root* remains constant as we walk the delta; we may use it to choose new source trees.

A tree delta is a list of changes of the form

`<tree-delta>`*change*`...</tree-delta>`

which describe how to edit the contents of *source-dir* to yield *target-dir*. There are three kinds of changes:

`<delete name='`*name*`'/>`
: *Source-dir* has an entry named *name*, which is not present in *target-dir*.

`<add name='`*name*`'>`*content*`</add>`
: *target-dir* has an entry named *name*, which is not present in *source-dir*; *content* describes the file or directory to which the new directory entry refers.

`<replace name='`*name*`'>`*content*`</replace>`
: Both *source-dir* and *target-dir* have an entry named *name*, which has changed; *content* describes the new file or directory.

Any entries in *source-dir* whose names aren't mentioned are assumed to appear unchanged in *target-dir*. Thus, an empty `tree-delta` element indicates that *target-dir* is identical to *source-dir*.

In the change descriptions above, each *content* takes one of the following forms:

`<file` *ancestor*`>`*prop-delta text-delta*`</file>`
: The given *target-dir* entry refers to a file, *f*. *Ancestor* indicates which file in the source tree *f* is derived from, if any.

  Prop-delta is a property delta describing how *f*'s properties differ from that ancestor; it may be omitted, indicating that the properties are unchanged.

  Text-delta is a text delta describing how to construct *f* from that ancestor; it may also be omitted, indicating that *f*'s text is identical to its ancestor's.

`<file` *ancestor*`/>`
: An abbreviation for `<file` *ancestor*`></file>` — a file element with no property or text delta, thus describing a file identical to its ancestor.

<directory *ancestor*>*prop-delta tree-delta*</directory>
: The given *target-dir* entry refers to a subdirectory, *sub*. *Ancestor* indicates which directory in the source tree *sub* is derived from, if any.

  Prop-delta is a property delta describing how *sub*'s properties differ from that ancestor; it may be omitted, indicating that the properties are unchanged.

  Tree-delta describes how to construct *sub* from that ancestor; it may be omitted, indicating that the directory is identical to its ancestor. *Tree-delta* should be interpreted with a new *target-dir* of '*target-dir*/*name*'.

  Since *tree-delta* is itself a complete tree delta structure, tree deltas are themselves trees, whose structure is a subgraph of the target tree.

<directory *ancestor*/>
: An abbreviation for <directory *ancestor*></directory> — a directory element with no property or tree delta, thus describing a directory identical to its ancestor.

The *content* of a add or replace tag may also contain a property delta, describing changes to the properties of that *directory entry*.

In the file and directory elements described above, each *ancestor* has one of the following forms:

ancestor='*path*'
: The ancestor of the new or changed file or directory is '*source-root*/*path*', in *revision*. When this appears as an attribute of a file element, the element's text delta should be applied to '*source-root*/*path*'. When this appears as an attribute of a directory element, '*source-root*/*path*' should be the new *source-dir* for interpreting that element's tree delta.

new='true'
: This indicates that the file or directory has no ancestor in the source tree. When followed by a *text-delta*, that delta should be applied to the empty file to yield the new text; when followed by a *tree-delta*, that delta should be evaluated as if *source-dir* were an imaginary empty directory.

nothing
: If neither an ancestor nor a new attribute is given, this is an abbreviation for ancestor='*source-dir*/*name*', with the same revision number. This makes the common case — files or directories modified in place — more compact.

If the *ancestor* spec is not new='true', it may also contain the text revision='*rev*', indicating a new value for *revision*, in which we should find the ancestor.

If a filename or path appearing as a *name* or *path* in the description above contains the characters '&', '<', or ''', they should be replaced with the sequences '&#38', '&#60', or '&#39', respectively.

Suppose we have the following source tree:

```
/dir1/file1
      file2
      dir2/file3
           file4
      dir3/file5
```

```
                file6
```

If we edit the contents of '`/dir1/file1`', we can describe the effect on the tree with the following tree delta, to be applied to the root:

```
<tree-delta>
  <replace name='dir1'>
    <directory>
      <tree-delta>
        <replace name='file1'>
          <file>text-delta</file>
        </replace>
      </tree-delta>
    </directory>
  </replace>
</tree-delta>
```

The outer `tree-delta` element describes the changes made to the root directory. Within the root directory, there are changes in '`dir1`', described by the nested `tree-delta`. Within '`/dir1`', there are changes in '`file1`', described by the *text-delta*.

If we had edited both '`/dir1/file1`' and '`/dir1/file2`', then there would simply be two `replace` elements in the inner `tree-delta`.

As another example, starting from the same source tree, suppose we rename '`/dir1/file1`' to '`/dir1/file8`':

```
<tree-delta>
  <replace name='dir1'>
    <directory>
      <tree-delta>
        <delete name='file1'/>
        <add name='file8'>
          <file ancestor='/dir1/file1'/>
        </add>
      </tree-delta>
    </directory>
  </replace>
</tree-delta>
```

As above, the inner `tdelta` describes how '`/dir1`' has changed: the entry for '`/dir1/file1`' has disappeared, but there is a new entry, '`/dir1/file8`', which is derived from and textually identical to '`/dir1/file1`' in the source directory. This is just an indirect way of describing the rename.

Why is it necessary to be so indirect? Consider the delta representing the result of:

1. renaming '`/dir1/file1`' to '`/dir1/tmp`',
2. renaming '`/dir1/file2`' to '`/dir1/file1`', and
3. renaming '`/dir1/tmp`' to '`/dir1/file2`'

(in other words, exchanging '`file1`' and '`file2`'):

```
<tree-delta>
  <replace name='dir1'>
    <directory>
```

```
          <tree-delta>
            <replace name='file1'>
              <file ancestor='/dir1/file2'/>
            </replace>
            <replace name='file2'>
              <file ancestor='/dir1/file1'/>
            </replace>
          </tree-delta>
        </directory>
      </replace>
    </tree-delta>
```

The indirectness allows the tree delta to capture an arbitrary rearrangement without resorting to temporary filenames.

Another example, starting from the same source tree:

1. rename '`/dir1/dir2`' to '`/dir1/dir4`',
2. rename '`/dir1/dir3`' to '`/dir1/dir2`', and
3. move '`file3`' from */dir1/dir4* to */dir1/dir2*.

Note that '`file3`''s path has remained the same, even though the directories around it have changed. Here is the tree delta:

```
    <tree-delta>
      <replace name='dir1'>
        <directory>
          <tree-delta>
            <replace name='dir2'>
              <directory ancestor='/dir1/dir3'>
                <tree-delta>
                  <add name='file3'>
                    <file ancestor='/dir1/dir2/file3'/>
                  </add>
                </tree-delta>
              </directory>
            </replace>
            <delete name='dir3'/>
            <add name='dir4'>
              <directory ancestor='/dir1/dir2'>
                <tree-delta>
                  <delete name='file3'/>
                </tree-delta>
              </directory>
            </add>
          </tree-delta>
        </directory>
      </replace>
    </tree-delta>
```

In other words:

- '`/dir1`' has changed;

- the new directory '`/dir1/dir2`' is derived from the old '`/dir1/dir3`', and contains a new entry '`file3`', derived from the old '`/dir1/dir2/file3`';
- there is no longer any '`/dir1/dir3`'; and
- the new directory '`/dir1/dir4`' is derived from the old '`/dir1/dir2`', except that its entry for '`file3`' is now gone.

Some more possible maneuvers, left as exercises for the reader:

- Delete '`dir2`', and then create a file named '`dir2`'.
- Rename '`/dir1/dir2`' to '`/dir1/dir4`'; move '`file2`' into '`/dir1/dir4`'; and move '`file3`' into */dir1/dir3*.
- Move '`dir2`' into '`dir3`', and move '`dir3`' into '`/`'.

4.4 Postfix Text Deltas

It is sometimes useful to represent a set of changes to a tree without providing text deltas in the middle of the stream. Text deltas are often large and expensive to compute, and tree deltas can be useful without them. For example, one can detect whether two changes might conflict — whether they change the same file, for example — without knowing exactly how the conflicting files changed.

For this reason, our XML representation of a tree delta allows the text deltas to come *after* the </tree-delta> closure. This allows the client to receive early notice of conflicts: during a `svn commit` command, the client sends a tree-delta to the server, which can check for skeletal conflicts and reject the commit, before the client takes the time to transmit the (possibly large) textual changes. This potentially saves quite a bit of network traffic.

In terms of XML, postfix text deltas are split into two parts. The first part appears "in-line" and contains a reference ID. The second part appears after the tree delta is complete. Here's an example:

```
<tree-delta>
  <replace name="foo.c">
     <file>
       <text-delta-ref id="123">
     </file>
  </replace>
  <add name="bar.c">
     <file>
       <text-delta-ref id="456">
     </file>
   </add>
</tree-delta>
<text-delta id="123">data</text-delta>
<text-delta id="456">data</text-delta>
```

4.5 Serializing Deltas via the "Editor" Interface

The static XML forms above are useful as an import/export format, and as a visualization aid, but we also need a way to express a delta as a *series of operations*, to implement directory tree diffing and patching. Subversion defines a standard set of such operations in the vtable `svn_delta_edit_fns_t`, a set of function prototypes which anyone may implement (see '`svn_delta.h`').

Each function in an instance of `svn_delta_edit_fns_t` (colloquially known as an *editor*) implements some distinct subtask of editing a directory tree. In fact, if you compare the editor function prototypes to the XML elements described previously, you'll notice a fairly strict correspondence: there's one function for replacing a directory, another function for replacing a file, one for adding a directory, another for adding a file, a function for deleting, and so on.

Although the editor interface was designed around the general idea of making changes to a directory tree, a specific implementation's behavior depends on its role. For example, the versioning filesystem library offers an editor that creates new revisions, while the working copy library offers an editor that updates working copies. And the network layer offers an editor that turns editing calls into wire protocol, which is then converted back into editing calls on the other side! All of these different tasks can share a single interface, because they are all fundamentally about the same thing: expressing and applying differences between directory trees.

Like the XML forms, a series of editor calls must follow certain nesting conventions; these conventions are implicit in the interface, in that some of the functions take arguments that can only be obtained from previous calls to other editor functions.

Editors can best be understood by watching one work on a real directory tree. For example:

Suppose that the user has made a number of local changes to her working copy and wants to commit them to the repository. Let's represent her changes with the same tree-delta from a previous example. Notice that she has also made textual modifications to '`file3`'; hence the in-line `<text-delta>`:

```
<tree-delta>
  <replace name='dir1'>
    <directory>
      <tree-delta>
        <replace name='dir2'>
          <directory ancestor='/dir1/dir3'>
            <tree-delta>
              <add name='file3'>
                <file ancestor='/dir1/dir2/file3'>
                  <text-delta>data</text-delta>
                </file>
              </add>
            </tree-delta>
          </directory>
        </replace>
        <delete name='dir3'/>
        <add name='dir4'>
          <directory ancestor='/dir1/dir2'>
            <tree-delta>
              <delete name='file3'/>
            </tree-delta>
```

```
          </directory>
        </add>
      </tree-delta>
    </directory>
  </replace>
</tree-delta>
```

So how does the client send this information to the server?

In a nutshell: the tree-delta is *streamed* over the network, as a series of individual commands given in depth-first order.

Let's be more specific. The server presents the client with an object of type `struct svn_delta_edit_fns_t`, colloquially known as an *editor*. An editor is really just table of functions; each function makes a change to a filesystem. Agent A (who has a private filesystem) presents an editor to agent B. Agent B then calls the editor's functions to change A's filesystem. B is said to be *driving* the editor.

As Karl Fogel likes to describe the process, if one thinks of the tree-delta as a lion, the editor is a "hoop" that the lion jumps through – each portion of the lion being decomposed through time.

B cannot call the functions in any willy-nilly order; there are some logical restrictions. In particular, as B drives the editor, it receives opaque data structures which represent directories and files. It must use and pass these structures, known as *batons*, to make further function calls.

As an example, let's watch how the client would transmit the above tree-delta to the repository. (The description below is slightly simplified. For exact interface details, see '`subversion/include/svn_delta.h`'.)

1. The repository hands an "editor" to the client.
2. The client begins by calling

   `root_baton = editor->replace_root();`

   The client now has an opaque object, *root_baton*, which represents the root of the repository's filesystem.
3. `dir1_baton = editor->replace_dir("dir1", root_baton);`

   Notice that *root_baton* gives the client free license to make any changes it wants in the repository's root directory – until, of course, it calls `editor->close_dir(root_baton)`.

   The first change made was a replacement of '`dir1`'. In return, the client now has a new opaque data structure that can be used to change '`dir1`'.
4. `dir2_baton = editor->replace_dir("dir2", "/dir1/dir3", dir1_baton);`

   The *dir1_baton* is now used to replace '`dir2`' with a directory whose ancestor is '`/dir1/dir3`'.
5. `file_baton = editor->add_file("file3", "/dir1/dir2/file3", dir2_baton);`

   Edits are now made to '`dir2`' (using *dir2_baton*). In particular, a new file is added to this directory whose ancestor is '`/dir1/dir2/file3`'.
6. Now the text-delta associated with *file_baton* needs to be transmitted:

   `window_handler = editor->apply_textdelta(file_baton);`

Text-deltas themselves, for network efficiency, are streamed in "chunks". So instead of receiving a baton object, we now have a routine that is able to receive any number of small "windows" of text-delta data.

We won't go into the details of the `svn_txdelta_*` functions right here; but suffice it to say that these routines are used for sending svndiff data to the *window_handler* routine.

7. `editor->close_file(file_baton);`

   The client is done sending the file's text-delta, so it releases the file baton.

8. `editor->close_dir(dir2_baton));`

   The client is done making changes to '`dir2`', so it releases its baton as well.

9. The client isn't yet finished with '`dir1`', however; it makes two more edits:

   ```
   editor->delete_item("dir3", dir1_baton);
   dir4_baton = editor->add_dir("dir4", "/dir1/dir2", dir1_baton);
   ```

   (The function's name is `delete_item` *rather than* `delete` *to avoid gratuitous incompatibility with C++, where* `delete` *is a reserved keyword.)*

10. Within the directory '`dir4`' (whose ancestry is '`/dir1/dir2`'), the client removes a file:

    ```
    editor->delete_item("file3", dir4_baton);
    ```

11. The client is now finished with both '`dir4`', as well as its parent '`dir1`':

    ```
    editor->close_dir(dir4_baton);
    editor->close_dir(dir1_baton);
    ```

12. The entire tree-delta is complete. The repository knows this when the root directory is closed:

    ```
    editor->close_dir(root_baton);
    ```

Of course, at any point above, the repository may reject an edit. If this is the case, the client aborts the transmission and the repository hasn't changed a bit. (Thank goodness for transactions!)

Note, however, that this "editor interface" works in the other direction as well. When the repository wishes to update a client's working copy, it is the *client's* reponisibility to give a custom editor-object to the server, and the *server* is the editor-driver.

Here are the main advantages of this interface:

- *Consistency.* Tree-deltas move across the network, in both directions, using the same interface.
- *Flexibility.* Custom editor-implementations can be written to do anything one might want; the editor-driver has no idea what is happening on the other side of the interface. For example, an editor might
  - Output XML that matches the tree-delta DTD above;
  - Output human-readable descriptions of the edits taking place;
  - Modify a filesystem

Whatever the case, it's easy to "swap" editors around, and make client and server do new and interesting things.

5 Client

The Subversion client is built on three libraries. One operates strictly on the working copy and does not talk to the repository. Another talks to the repository but never changes the working copy. The third library uses the first two to provide operations such as `commit` and `update` – operations which need to both talk to the repository and change the working copy.

The initial client is a Unix-style command-line tool (like standard CVS), but it should be easy to write a GUI client as well, based on the same libraries. The libraries capture the core Subversion functionality, segregating it from user interface concerns.

This chapter describes the libraries, and the physical layout of working copies.

5.1 Working copies and the working copy library

Working copies are client-side directory trees containing both versioned data and Subversion administrative files. The functions in the working copy management library are the only functions in Subversion which operate on these trees.

5.1.1 The layout of working copies

This section gives an overview of how working copies are arranged physically, but is not a full specification of working copy layout.

As with CVS, Subversion working copies are simply directory trees with special administrative subdirectories, in this case named "SVN" instead of "CVS":

```
                       myproj
                       / | \
         _____________/  |  \______________
        /                |                 \
      SVN               src                doc
  ___/ | \___           /|\             ___/ \___
 |     |     |         / | \           |         |
base  ...   ...       /  |  \     myproj.texi   SVN
                     /   |   \              ___/ | \___
                ____/    |    \____        |     |     |
               |         |         |      base  ...   ...
              SVN      foo.c     bar.c     |
          ___/ | \___                      |
         |     |     |                     |
       base   ...   ...               myproj.texi
    ___/ \___
   |         |
 foo.c     bar.c
```

Each '`dir/SVN/`' directory records the files in '`dir`', their revision numbers and property lists, pristine revisions of all the files (for client-side delta generation), the repository from which '`dir`' came, and any local changes (such as uncommitted adds, deletes, and renames) that affect '`dir`'.

Although often it would often be possible to deduce certain information (such as the origin repository) by examining parent directories, this is avoided in favor of making each directory be as much a self-contained unit as possible.

For example, immediately after a checkout the administrative information for the entire working tree *could* be stored in one top-level file. But subdirectories instead keep track of their own revision information. This would be necessary anyway once the user starts committing new revisions for particular files, and it also makes it easier for the user to prune a big, complete tree into a small subtree and still have a valid working copy.

The SVN subdir contains:

- A '`format`' file, which indicates which version of the working copy adm format this is (so future clients can be backwards compatible easily).
- A '`repository`' file, stating where the directory came from (syntax TBD).
- A '`text-base`' directory, containing the pristine repository revisions of the files in the corresponding working directory
- An '`entries`' file, which holds revision numbers and other information for this directory and its files, and records the presence of subdirs.

  It may help to think of this file as the functional equivalent of the CVS/Entries file.
- A '`props`' directory, containing property names and values for each file in the working directory.
- A '`prop-base`' directory, containing pristine property names and values for each file in the working directory.
- A '`dir-props`' file, recording properties for this directory.
- A '`dir-prop-base`' file, recording pristine properties for this directory.
- A '`lock`' file, whose presence implies that some client is currently operating on the adminstrative area.
- A '`tmp`' directory, for holding scratch-work and helping make working copy operations more crash-proof.
- A '`log`' file. If present, indicates a list of actions that need to be taken to complete a working-copy-operation that is still "in progress".

You can read much more about these files in the file '`subversion/libsvn_wc/README`'.

5.1.2 The working copy management library

- **Requires:**
  - a working copy
- **Provides:**
  - ability to manipulate the working copy's versioned data
  - ability to manipulate the working copy's administrative files

This library performs "offline" operations on the working copy, and lives in '`subversion/libsvn_wc/`'.

The API for *libsvn_wc* is always evolving; please read the header file for a detailed description: '`subversion/include/svn_wc.h`'.

5.2 The repository access library

- **Requires:**
  - network access to a Subversion server
- **Provides:**
  - the ability to interact with a repository

This library performs operations involving communication with the repository.

The interface defined in '`subversion/include/svn_ra.h`' provides a uniform interface to both local and remote repository access.

Specifically, *libsvn_ra_dav* will provide this interface and speak to repositories using DAV requests. At some future point, another library *libsvn_ra_local* will provide the same interface – but will link directly to the filesystem library for accessing local disk repositories.

5.3 The client operation library

- **Requires:**
  - the working copy management library
  - a repository access library
- **Provides:**
  - all client-side Subversion commands

These functions correspond to user-level client commands. In theory, any client interface (command-line, GUI, emacs, Python, etc.) should be able to link to *libsvn_client* and have the ability to act as a full-featured Subversion client.

Again, the detailed API can be found in '`subversion/include/svn_client.h`'.

6 Protocol

The Subversion client library `libsvn_ra_dav` uses the *Neon* library to generate WebDAV DeltaV requests and sends them to a "Subversion-aware" Apache server.

This Apache server is running *mod_DAV* and *mod_DAV_SVN*, which translates the requests into Subversion filesystem calls.

For more info, See Section 3.2 [Network Layer], page 18.

For a detailed description of exactly how Greg Stein `<gstein@lyra.org>` is mapping the WebDAV DeltaV spec to Subversion, see his paper: '`notes/webdav-usage.html`'

For more information on WebDAV and the DeltaV extensions, see `http://www.webdav.org` and `http://www.webdav.org/deltav`.

For more information on *Neon*, see `http://www.webdav.org/neon`.

7 Server

The term “server” is ambiguous, because it has at least two different meanings: it can refer to a powerful computer which offers services to users on a network, or it can refer to a CPU process designed to receive network requests.

In Subversion, however, the *server* is just a set of libraries that implements *repositories* and makes them available to other programs. No networking is required.

There are two main libraries: the *Subversion Filesystem* library, and the *Subversion Server* library.

7.1 Filesystem

7.1.1 Filesystem Overview

- **Requires:**
  - some writable disk space
  - (for now) Berkeley DB library
- **Provides:**
  - a repository for storing files
  - concurrent client transactions
  - enforcement of user & group permissions

This library implements a hierarchical filesystem which supports atomic changes to directory trees, and records a complete history of the changes. In addition to recording changes to file and directory contents, the Subversion Filesystem records changes to file meta-data (see discussion of *properties* in Chapter 2 [Model], page 7).

7.1.2 API

There are two main files that describe the Subversion filesystem.

First, read the section below (See Section 7.1.3 [Repository Structure], page 35.) for a general overview of how the fileystem works.

Once you’ve done this, read Jim Blandy’s own structural overview, which explains how nodes and revisions are organized (among other things) in the filesystem implementation: ‘`subversion/libsvn_fs/structure`’.

Finally, read the well-documented API in ‘`subversion/include/svn_fs.h`’.

7.1.3 Repository Structure

7.1.3.1 Schema

To begin, please be sure that you're already casually familiar with Subversion's ideas of files, directories, and revision histories. If not, See Chapter 2 [Model], page 7. We can now offer precise, technical descriptions of the terms introduced there.

> A *text string* is a string of Unicode characters which is canonically decomposed and ordered, according to the rules described in the Unicode standard.
>
> A *string of bytes* is what you'd expect.
>
> A *property list* is an unordered list of properties. A *property* is a pair (*name*, *value*), where *name* is a text string, and *value* is a string of bytes. No two properties in a property list have the same name.
>
> A *file* is a property list and a string of bytes.
>
> A *node* is either a file or a directory. (We define a directory below.) Nodes are distinguished unions — you can always tell whether a node is a file or a directory.
>
> A *node table* is an array mapping some set of positive integers, called *node numbers*, onto *nodes*. If a node table maps some number *i* to some node *n*, then *i* is a *valid node number* in that table, and *node i* is *n*. Otherwise, *i* is an *invalid node number* in that table.
>
> A *directory entry* is a triple (*name*, *props*, *node*), where *name* is a text string, *props* is a property list, and *node* is a node number.
>
> A *directory* is an unordered list of directory entries, and a property list.
>
> A *revision* is a node number and a property list.
>
> A *history* is an array of revisions, indexed by a contiguous range of non-negative integers containing 0.
>
> A *repository* consists of node table and a history.

Now that we've explained the form of the data, we make some restrictions on that form.

Every revision has a root directory. Every revision's node number is a valid node number, and the node it refers to is always a directory. We call this the revision's *root directory*.

Revision 0 always contains an empty root directory. This baseline makes it easy to check out whole projects from the repository.

Directories contain only valid links. Every directory entry's *node* is a valid node number.

Directory entries can be identified by name. For any directory *d*, every directory entry in *d* has a distinct name.

There are no cycles of directories. No node is its own child.

Directories can have more than one parent. The Unix file system does not allow more than one hard link to a directory, but Subversion does allow the analogous situation. Thus, the directories in a Subversion repository form a directed acyclic graph (*DAG*), not a tree. However, it would be distracting and unhelpful to replace the familiar term "directory tree" with the unfamiliar term "directory DAG", so we still call it a "directory tree" here.

There are no dead nodes. Every node is a child of some revision's root directory.

7.1.3.2 Bubble-Up Method

This section provides a conversational explanation of how the repository actually stores and revisions file trees. It's not critical knowledge for a programmer using the Subversion Filesystem API, but most people probably still want to know what's going on "under the hood" of the repository.

Suppose we have a new project, at revision 1, looking like this (using CVS syntax):

```
prompt$ svn checkout myproj
U myproj/
U myproj/B
U myproj/A
U myproj/A/fish
U myproj/A/fish/tuna
prompt$
```

Only the file '`tuna`' is a regular file, everything else in myproj is a directory.

Let's see what this looks like as an abstract data structure in the repository, and how that structure works in various operations (such as update, commit, and branch).

In the diagrams that follow, lines represent parent-to-child connections in a directory hierarchy. Boxes are "nodes". A node is either a file or a directory – a letter in the upper left indicates which kind. A file node has a byte-string for its content, whereas directory nodes have a list of dir_entries, each pointing to another node.

Parent-child links go both ways (i.e., a child knows who all its parents are), but a node's name is stored only in its parent, because a node with multiple parents may have different names in different parents.

At the top of the repository is an array of revision numbers, stretching off to infinity. Since the project is at revision 1, only index 1 points to anything; it points to the root node of revision 1 of the project:

```
                    ( myproj's revision array )
       ______________________________________________________
      |___1_______2________3________4________5_________6_____...
          |
          |
       ___|_____
      |D        |
      |         |
      |   A     |      /* Two dir_entries, `A' and `B'. */
      |    \    |
      |   B \   |
      |__/___\__|
        /     \
       |       \
       |        \
    ___|___   ___\____
   |D      | |D       |
   |       | |        |
   |       | | fish   |   /* One dir_entry, `fish'. */
   |_______| |___\____|
                  \
                   \
                 ___\____
                |D       |
                |        |
                | tuna   |  /* One dir_entry, `tuna'. */
                |___\____|
                     \
                      \
                    ___\____
                   |F       |
                   |        |
                   |        |   /* (Contents of tuna not shown.) */
                   |________|
```

What happens when we modify '`tuna`' and commit? First, we make a new '`tuna`' node, containing the latest text. The new node is not connected to anything yet, it's just hanging out there in space:

```
                         ________
                        |F       |
                        |        |
                        |        |
                        |________|
```

Next, we create a *new* revision of its parent directory:

```
 --------
|D       |
|        |
| tuna   |
|___\____|
     \
      \
    ___\____
   |F       |
   |        |
   |        |
   |________|
```

We continue up the line, creating a new revision of the next parent directory:

```
 --------
|D       |
|        |
| fish   |
|___\____|
     \
      \
    ___\____
   |D       |
   |        |
   | tuna   |
   |___\____|
        \
         \
       ___\____
      |F       |
      |        |
      |        |
      |________|
```

Now it gets more tricky: we need to create a new revision of the root directory. This new root directory needs an entry to point to the “new” directory A, but directory B hasn't changed at all. Therefore, our new root directory also has an entry that still points to the *old* directory B node!

```
    ______________________________________________________
   |___1_______2________3________4________5_________6_____...
       |
       |
    ___|_____             ________
   |D        |           |D       |
   |         |           |        |
   |   A     |           |   A    |
   |    \    |           |    \   |
   |   B \   |           |   B \  |
   |__/___\__|           |__/___\_|
     /     \               /     \
    |    ___\_____________/       \
    |   /    \                     \
 ___|__/   ___\____              ___\____
|D      | |D       |            |D       |
|       | |        |            |        |
|       | | fish   |            | fish   |
|_______| |___\____|            |___\____|
               \                     \
                \                     \
              ___\____              ___\____
             |D       |            |D       |
             |        |            |        |
             | tuna   |            | tuna   |
             |___\____|            |___\____|
                  \                     \
                   \                     \
                 ___\____              ___\____
                |F       |            |F       |
                |        |            |        |
                |        |            |        |
                |________|            |________|
```

Finally, after all our new nodes are written, we finish the "bubble up" process by linking this new tree to the next available revision in the history array. In this case, the new tree becomes revision 2 in the repository.

```
    ------------------------------------------------------
   |___1_______2________3________4________5_________6______...
       |        \
       |         \__________
    ___|_____             __\_____
   |D        |           |D       |
   |         |           |        |
   |   A     |           |   A    |
   |    \    |           |    \   |
   |   B \   |           |   B \  |
   |__/___\__|           |__/___\_|
     /     \               /     \
    |    ___\_____________/       \
    |   /    \                     \
 ___|__/   ___\____              ___\____
|D      | |D       |            |D       |
|       | |        |            |        |
|       | | fish   |            | fish   |
|_______| |___\____|            |___\____|
               \                     \
                \                     \
              ___\____              ___\____
             |D       |            |D       |
             |        |            |        |
             | tuna   |            | tuna   |
             |___\____|            |___\____|
                  \                     \
                   \                     \
                 ___\____              ___\____
                |F       |            |F       |
                |        |            |        |
                |        |            |        |
                |________|            |________|
```

Generalizing on this example, you can now see that each "revision" in the repository history represents a root node of a unique tree (and an atomic commit to the whole filesystem.) There are many trees in the repository, and many of them share nodes.

Many nice behaviors come from this model:

1. **Easy reads.** If a filesystem reader wants to locate revision X of file '`foo.c`', it need only traverse the repository's history, locate revision X's root node, then walk down the tree to '`foo.c`'.
2. **Writers don't interfere with readers.** Writers can continue to create new nodes, bubbling their way up to the top, and concurrent readers cannot see the work in progress. The new tree only becomes visible to readers after the writer makes its final "link" to the repository's history.
3. **File structure is versioned.** Unlike CVS, the very structure of each tree is being saved from revision to revision. File and directory renames, additions, and deletions are part of the repository's history.

Let's demonstrate the last point by renaming the '`tuna`' to '`book`'.

We start by creating a new parent "fish" directory, except that this parent directory has a different dir_entry, one which points the *same* old file node, but has a different name:

```
    _____________________________________________________
   |___1_______2________3________4________5_________6_____...
       |        \
       |         \__________
    ___|_____             __\_____
   |D        |           |D       |
   |         |           |        |
   |   A     |           |   A    |
   |    \    |           |    \   |
   |   B \   |           |   B \  |
   |__/___\__|           |__/___\_|
     /     \               /     \
    |    ___\_____________/       \
    |   /    \                     \
 ___|__/   ___\____              ___\____
|D      | |D       |            |D       |
|       | |        |            |        |
|       | | fish   |            | fish   |
|_______| |___\____|            |___\____|
               \                     \
                \                     \
              ___\____              ___\____      ________
             |D       |            |D       |    |D       |
             |        |            |        |    |        |
             | tuna   |            | tuna   |    | book   |
             |___\____|            |___\____|    |_/______|
                  \                     \         /
                   \                     \       /
                 ___\____              ___\____ /
                |F       |            |F       |
                |        |            |        |
                |        |            |        |
                |________|            |________|
```

From here, we finish with the bubble-up process. We make new parent directories up to the top, culminating in a new root directory with two dir_entries (one points to the old "B" directory node we've had all along, the other to the new revision of "A"), and finally link the new tree to the history as revision 3:

```
    ______________________________________________________
   |___1_______2________3________4________5_________6_____...
       |        \        \_________________
       |         \__________               \
    ___|_____             __\_____        __\_____
   |D        |           |D       |      |D       |
   |         |           |        |      |        |
   |   A     |           |   A    |      |   A    |
   |    \    |           |    \   |      |    \   |
   |   B \   |           |   B \  |      |   B \  |
   |__/___\__|           |__/___\_|      |__/___\_|
     /  ___________________/_____\_________/     \
    |  / ___\_____________/       \               \
    | / /    \                     \               \
 ___|/_/   ___\____              ___\____      _____\__
|D      | |D       |            |D       |    |D       |
|       | |        |            |        |    |        |
|       | | fish   |            | fish   |    | fish   |
|_______| |___\____|            |___\____|    |___\____|
               \                     \             \
                \                     \             \
              ___\____              ___\____      ___\____
             |D       |            |D       |    |D       |
             |        |            |        |    |        |
             | tuna   |            | tuna   |    | book   |
             |___\____|            |___\____|    |_/______|
                  \                     \         /
                   \                     \       /
                 ___\____              ___\____ /
                |F       |            |F       |
                |        |            |        |
                |        |            |        |
                |________|            |________|
```

For our last example, we'll demonstrate the way "tags" and "branches" are implemented in the repository.

In a nutshell, they're one and the same thing. Because nodes are so easily shared, we simply create a *new* directory entry that points to an existing directory node. It's an extremely cheap way of copying a tree; we call this new entry a *clone*.

Let's go back to our original tree, assuming that we're at revision 6 to begin with:

```
    ------------------------------------------------------
 ...___6_______7________8________9________10_________11_____...
       |
       |
    ___|_____
   |D        |
   |         |
   |   A     |
   |    \    |
   |   B \   |
   |__/___\__|
     /     \
    |       \
    |        \
 ___|___   ___\____
|D      | |D       |
|       | |        |
|       | | fish   |
|_______| |___\____|
               \
                \
              ___\____
             |D       |
             |        |
             | tuna   |
             |___\____|
                  \
                   \
                 ___\____
                |F       |
                |        |
                |        |
                |________|
```

Let's "tag" directory A. To make the clone, we create a new dir_entry **T** in our root, pointing to A's node:

```
    ------------------------------------------------------
   |___6_______7________8________9________10_________11_____...
       |        \
       |         \
    ___|_____   __\______
   |D        | |D        |
   |         | |         |
   |   A     | |    A    |
   |    \    | |    |    |
   |   B \   | |  B |  T |
   |__/___\__| |_/__|__|_|
     /     \    /   |  |
    |    ___\__/   /  /
    |   /    \    /  /
 ___|__/   ___\__/_ /
|D      | |D       |
|       | |        |
|       | | fish   |
|_______| |___\____|
               \
                \
              ___\____
             |D       |
             |        |
             | tuna   |
             |___\____|
                  \
                   \
                 ___\____
                |F       |
                |        |
                |        |
                |________|
```

Now we're all set. In the future, the contents of directories A and B may change quite a lot. However, assuming we never make any changes to directory T, it will *always* point to a particular pristine revision of directory A. (We can use permissions to prevent anyone from writing to directory T.) Thus, T is a tag.

However, if we *do* decide to allow commits in directory T, and now our repository tree increments to revision 8, then T becomes a branch. Specifically, it's a branch of directory A which "broke off" from the main line at revision 8.

7.1.3.3 Diffy Storage

You may have been thinking, "Gee, this bubble up method seems nice, but it sure wastes a lot of space. Every commit to the repository creates an entire line of new directory nodes!"

Like many other revision control systems, Subversion stores changes as differences. It doesn't make complete copies of nodes; instead, it stores the *latest* revision as a full text, and previous revisions as a succession of reverse diffs (the word "diff" is used loosely here – for files, it means vdeltas, for directories, it means a format that expresses changes to directories).

7.1.4 Implementation

For the initial release of Subversion,

- The filesystem will be implemented as a library on Unix.
- The filesystem's data will probably be stored in a collection of .db files, using the Berkeley Database library.[1] (For more information, see Sleepycat Software (`http://www.sleepycat.com`).)

7.2 Server Library

This library no longer exists in the current Subversion design; all functionality that it would have provided will come out Apache and mod_DAV_SVN.

Only a small part of the spec has been left behind – mainly to provide ideas for the future.

7.2.1 Security

Subversion has two levels of security, both abstracted.

7.2.1.1 Username Authentication

First of all, the Subversion filesystem does *not* authenticate users – it never asks a client user to prove his/her identity. Instead, authentication is soley the responsibility of the network layer. Every Subversion user *must* have an account in an external database that is understood by some network layer. (See Section 3.2 [Network Layer], page 18.)

For example, for a vanilla Apache network layer, each client user could authenticate against a standard Apache access control file. More complex network layers could authenticate against '`/etc/passwd`' or a SQL database.

Once the network layer has authenticated the user, it passes three arguments to the Subversion server:

- **auth_username** – the authenticated username
- **auth_method** – the method in which this user was authenticated
- **auth_domain** – the domain the user is coming from

This information is then used to "authorize" a client's actions.

---

[1] In the future, of course, contributors are free modify the Subversion filesystem to operate with more powerful SQL database.

7.2.1.2 Authorization

By *authorization*, we mean answering the question:

"Is user X allowed to perform action Y on data Z?"

The Subversion filesystem provides a single interface for answering this question.

```
str_t svn_authorize (auth_user, auth_method, auth_domain, action, path);
```

This interface is implemented by a server-side plug-in library which is free to define any sort of security scheme it wishes. The return value of this function is either NULL (meaning that authorization was denied), or *username*, where *username* is the specific username to use when performing the action against the Subversion filesystem.

The first release of Subversion will ship with a plug-in that manages this information in a file called 'svn_security'; each repository has its own copy this file. It's worth mentioning how this file works.

The 'svn_security' file does three things:

1. **Map authorized usernames to filesystem usernames.** For example, suppose Jim Blandy is able to contact the Subversion server in many different ways, using different network layers. It's important that the *same* username be used when talking to the filesystem. We call this username the *Subversion username* or *canonical* username. Here's an example of this mapping:

```
jimb : Jim Blandy :   jim/unix/localhost,  \
                      jimb/smtp/jimb@red-bean.com, \
                      jimb/kerberos/redhat.com, \
                      jblandy/DAV/red-bean.com
```

2. **Define Roles** A *role* is a set of performable actions. Here are some examples of roles:

```
visitor : checkout update
hacker : checkout update commit
core-programmer : checkout update commit add rm mv
nitpicker : checkout update annotate
admin : checkout update commit add rm mv import
god : all
offlimits: none
```

3. **Map roles to users.** For each canonical username, specify specific roles this person has for particular filesystem paths:

```
joe : visitor /trunk/gnome : nitpicker /trunk/foo
kfogel : admin /trunk/subversion : offlimits /trunk/subversion/foo.c
zeke : hacker /trunk/comp-tools : visitor /trunk/comp-tools/gdb
```

Remember, this is only one example of a back-end implementation of `svn_authorize ()`. Programmers are free to write new server-side plug-ins that implement different security schemas (for example, authorizing against a SQL database.)

8 Future

Here are some thoughts on features to include in later releases of Subversion.

8.1 High Priority Items

These are things that *should* be released in Subversion 1.0, if we really want to achieve the goal of creating a compelling replacement for CVS.

The lists below are a bit dated; a more recent wishlist can be found in '`IDEAS`' in the source-tree.

8.1.1 Annotation

We need to provide annotation of individual files (i.e. who wrote which line in which revision), because CVS does this reasonably well.

The information needed to annotate files should be stored by the filesystem as properties.

The "annotate" method itself should be implemented as a server-side plug-in. This makes the output much more hackable.

8.1.2 Repository Conversion

We need to write a tool which will convert a CVS repository into an SVN repository, preserving all history. This is absolutely critical in persuading people to switch to SVN.

Bob Miller `<kbob@jogger-egg.com>` has already started work on this tool, which can be found in '`tools/cvs2svn/`'.

8.1.3 User Interface for Branches and Tags

This is a tricky issue.

The Subversion Filesystem implements branches and tags by using "clones"; both branches and tags appear as ordinary directories in the filesystem.

We have been told that this implementation is quite confusing for end-users. (The closed-source version-control system *Perforce* uses this method.) It's apparently confusing, when browsing the filesystem, to know which subdirectories represents the main-lines of a project, and which represent branches or tags. This problem is made worse by users accidentally (or carelessly) creating clone directories in bad places, such as inside the trees they're cloning.

There are a number of proposed solutions being discussed. They all center on two principles:

- Recommending a repository filesystem structure that cleanly divides trunks, branches, and tags;
- Implementing client behavior that "hides" this repository structure from the end user, and allows them to continue using intuitive branch and tag commands.

8.2 Blue Sky Items

These are features we'd like to see in Subversion someday, but aren't critical for a 1.0 release.

8.2.1 Server Grep

Write a server-side plug-in which can quickly search a filesystem's text and properties.

8.2.2 Scripting Language

Write a server-side plug-in which provides glue between *svn_main* and libguile.so, libperl.so, or libpython.so. The server becomes **very** extensible if it has an interpreted language built into it; it's also very nice for writing test suites!

8.2.3 Smart conflict resolution

Certain kinds of conflicts can be resolved without human intervention. For example, files like '`/etc/passwd`' just need to keep lines unique by username and user ID.

Right now, merging new repository data into a modified working copy of a passwd file can result in a textual conflict even when there's no "semantic conflict". But if the Subversion client knew something about the format of passwd files, then it could merge without flagging a conflict.

A similar rule could be used for ChangeLogs, based on the dates in the header lines. And so on.

Since all merging takes place on the client, these "smart merges" should be implemented as a client-side plug-in.

8.2.4 Mirroring Servers

This is like the ClearCase multisite feature. Essentially, it is a redundant distributed repository. The repository exists on two or more cooperatively mirroring servers (each one presumably being close, network-wise, to its intended users). A commit from any user is visible on all the servers.

The best way to implement this is by creating a "hierarchy" of Subversion servers, much like the DNS or NIS system. We can define a server *master* to contain the "authoritative" repository. We can then set up any number of *slave* servers to mirror the master. The slave servers exist primarily as local caches; it makes `reads` and `updates` faster for geographically disperse users. When a user wishes to `commit`, however, her delta is always sent to the master server. After the master accepts the change, the delta is automatically "pushed" out to the caching slave servers.

8.2.5 Inter-Repository Communication

This is one that people request a lot: the ability to commit changes first to a local "working repository" (not visible to the rest of the world), and then commit what's in the working repository to the real repository (with the several commits maybe being folded into one commit).

Why do people want this? It may be the psychological comfort of making a snapshot whenever one reaches a good stopping point, but not necessarily wanting all those "comfort points" to become publically-visible commits.

The best way to implement this is to allow "clones" to cross between repositories. (See Section 7.1.3.2 [Bubble-Up Method], page 37.)

In other words, Joe Hacker sets up a personal Subversion repository on his desktop machine; he creates a local "clone" of a subtree from a public repository. He commits to his clone (which turns it into a branch), and when he's done, he performs a branch-merge back into the public repository.

8.2.6 SQL Back-End

The initial release of the Subversion filesystem will use Berkeley DB to store data on disk. However, in the future, a SQL database may be used to add sophisticated query support across the filesystem (e.g. "what revisions of what files has property `xyz`?").

To support this feature, Someone could rewrite the filesystem back-end to speak SQL, in addition to the default Berkeley DB support.

8.2.7 Digital Signatures

A few people have mentioned cryptographic signing of deltas. It's a cool idea, and we should leave the door open for it.

8.2.8 Import/Export format

Suggested by Jason Robbins; he mentioned this as a nice thing to have someday. It makes repositories easy to transport.

Subversion already has an XML DTD that it uses for testing updates and commits; this DTD may be 80% of the "exchange" format we're looking for.

8.2.9 SMTP access

Write a totally independent Network Layer which is an SMTP server on one end, and speaks to the Subversion server library on the back end. It would neat to be able to "mail in" commits, or receive working-copy updates through e-mail.

9 License

This software is licensed as described in the file COPYING, which you should have received as part of this distribution. The terms are also available at http://subversion.tigris.org/license-1.html. If newer versions of this license are posted there, you may use a newer version instead, at your option.