What is going to be covered:

• Notions of the central repository, the client working copy, and the array

of repository revision trees.

• Some simple examples of how two collaborators can use Subversion to publish

and receive changes from one another, using the “copy-modify-merge” model.

• How Subversion tracks and manages information in a working copy.

1. **The Repository**

Subversion is a centralized system for sharing information. At its core is a repository, which

is a central store of data. The repository stores information in the form of a *filesystem*

*tree*—a typical hierarchy of files and directories. Any number of *clients* connect to the repository,

and then read or write to these files. By writing data, a client makes the information

available to others; by reading data, the client receives information from others. Figure

1.1, “A typical client/server system” illustrates this.

**Figure 1.1. A typical client/server system**



What makes the Subversion repository special is that *it remembers every change* ever written to it—every

change to every file, and even changes to the directory tree itself, such as the addition, deletion,

and rearrangement of files and directories.

1. **Versioning Models**

The core mission of a version control system is to enable collaborative editing and sharing

of data. But different systems use different strategies to achieve this. It's important to understand

these different strategies, for a couple of reasons.

**The Problem of File Sharing**

All version control systems have to solve the same fundamental problem: how will the system

allow users to share information, but prevent them from accidentally stepping on each

other's feet? It's all too easy for users to accidentally overwrite each other's changes in the

repository.

**Figure 1.2. The problem to avoid**



**The Lock-Modify-Unlock Solution**

**Figure 1.3. The lock-modify-unlock solution**



**The problem with the lock-modify-unlock model is that it's a bit restrictive and often becomes a roadblock for users:**

• *Locking may cause administrative problems.* Sometimes Harry will lock a file and then

forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are

tied. And then Harry goes on vacation. Now Sally has to get an administrator to release

Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.

• *Locking may cause unnecessary serialization.* What if Harry is editing the beginning of a

text file, and Sally simply wants to edit the end of the same file? These changes don't

overlap at all. They could easily edit the file simultaneously, and no great harm would

come, assuming the changes were properly merged together. There's no need for them

to take turns in this situation.

• *Locking may create a false sense of security.* Suppose Harry locks and edits file A, while

Sally simultaneously locks and edits file B. But what if A and B depend on one another,  
and the changes made to each are semantically incompatible? Suddenly A and B don't

work together anymore. The locking system was powerless to prevent the problem—yet

it somehow provided a false sense of security. It's easy for Harry and Sally to imagine

that by locking files, each is beginning a safe, insulated task, and thus they need not

bother discussing their incompatible changes early on. Locking often becomes a substitute

for real communication.

**The Copy-Modify-Merge Solution**

Subversion, CVS, and many other version control systems use a *copy-modify-merge* model

as an alternative to locking. In this model, each user's client contacts the project repository

and creates a personal *working copy*—a local reflection of the repository's files and directories.

Users then work simultaneously and independently, modifying their private copies.

Finally, the private copies are merged together into a new, final version. The version

control system often assists with the merging, but ultimately, a human being is responsible

for making it happen correctly.  
**Figure 1.4. The copy-modify-merge solution**



**Figure 1.5. The copy-modify-merge solution (continued)**



**Conflict:**

But what if Sally's changes *do* overlap with Harry's changes? What then? This situation is

called a *conflict*, and it's usually not much of a problem. When Harry asks his client to

merge the latest repository changes into his working copy, his copy of file A is somehow

flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes

and manually choose between them. Note that software can't automatically resolve conflicts;

only humans are capable of understanding and making the necessary intelligent

choices. Once Harry has manually resolved the overlapping changes—perhaps after a discussion

with Sally—he can safely save the merged file back to the repository.

The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely

smoothly. Users can work in parallel, never waiting for one another. When they work on the

same files, it turns out that most of their concurrent changes don't overlap at all; conflicts

are infrequent. And the amount of time it takes to resolve conflicts is usually far less than

the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate

poorly, both syntactic and semantic conflicts increase. No system can force users

to communicate perfectly, and no system can detect semantic conflicts. So there's no point

in being lulled into a false sense of security that a locking system will somehow prevent

conflicts; in practice, locking seems to inhibit productivity more than anything else.

Note: **When Locking Is Necessary**

While the lock-modify-unlock model is considered generally harmful to collaboration,

sometimes locking is appropriate. The copy-modify-merge model is based on the assumption that files are contextually mergeable—that is, that the majority of the files in the repository are line-based text

files (such as program source code). But for files with binary formats, such as artwork

or sound, it's often impossible to merge conflicting changes. In these situations, it

really is necessary for users to take strict turns when changing the file. Without serialized

access, somebody ends up wasting time on changes that are ultimately discarded.

While Subversion is primarily a copy-modify-merge system, it still recognizes the

need to lock an occasional file, and thus provides mechanisms for this.

1. **Subversion Repository URLs**

But there are some nuances in Subversion's handling of URLs that are notable. For example,

URLs containing the file:// access method (used for local repositories) must, in

accordance with convention, have either a server name of localhost or no server name

at all:

$ svn checkout file:///var/svn/repos

…$

svn checkout file://localhost/var/svn/repos

…

**Repository URLs**

You can access Subversion repositories through many different methods—on local

disk or through various network protocols, depending on how your administrator has

set things up for you. A repository location, however, is always a URL. Table 1.1,

“Repository access URLs” describes how different URL schemes map to the available

access methods.

**Table 1.1. Repository access URLs**

|  |  |
| --- | --- |
| **Schema** | **Access method** |
| file:/// | Direct repository access (on local disk) |
| http:// | Access via WebDAV protocol to Subversionaware  Apache server |
| https:// | Same as http://, but with SSL encryption. |
| svn:// | Access via custom protocol to an svnserve  Server |
| svn+ssh:// | Same as svn://, but through an SSH tunnel. |

1. **Working Copies**

You've already read about working copies; now we'll demonstrate how the Subversion client

creates and uses them.

A Subversion working copy is an ordinary directory tree on your local system, containing a

collection of files. You can edit these files however you wish, and if they're source code

files, you can compile your program from them in the usual way. Your working copy is your

own private work area: Subversion will never incorporate other people's changes, nor

make your own changes available to others, until you explicitly tell it to do so. You can

even have multiple working copies of the same project.

A working copy also contains some extra files, created and maintained by Subversion, to

help it carry out these commands. In particular, each directory in your working copy contains

a subdirectory named .svn, also known as the working copy's *administrative directory*.

The files in each administrative directory help Subversion recognize which files contain

unpublished changes, and which files are out of date with respect to others' work.

A typical Subversion repository often holds the files (or source code) for several projects;

usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement,

a user's working copy will usually correspond to a particular subtree of the repository.

For example, suppose you have a repository that contains two software projects, paint

and calc. Each project lives in its own top-level subdirectory, as shown in Figure 1.6, “The

repository's filesystem”.

**Figure 1.6. The repository's filesystem**



**Check out**:  
To get a working copy, you must *check out* some subtree of the repository. (The term

*check out* may sound like it has something to do with locking or reserving resources, but it

doesn't; it simply creates a private copy of the project for you.) For example, if you check

out /calc, you will get a working copy like this:

$ svn checkout http://svn.example.com/repos/calc

A calc/Makefile

A calc/integer.c

A calc/button.c

Checked out revision 56.

$ ls -A calc

Makefile button.c integer.c .svn/

The list of letter As in the left margin indicates that Subversion is adding a number of items

to your working copy. You now have a personal copy of the repository's /calc directory,

with one additional entry—.svn—which holds the extra information needed by Subversion,

as mentioned earlier.

**Committing:**  
Suppose you make changes to button.c. Since the .svn directory remembers the file's

original modification date and contents, Subversion can tell that you've changed the file.

However, Subversion does not make your changes public until you explicitly tell it to. The

act of publishing your changes is more commonly known as *committing* (or *checking in*)

changes to the repository.

To publish your changes to others, you can use Subversion's **svn commit** command:

$ svn commit button.c -m "Fixed a typo in button.c."

Sending button.c

Transmitting file data .

Committed revision 57.

Now your changes to button.c have been committed to the repository, with a note describing

your change (namely, that you fixed a typo). If another user checks out a working

copy of /calc, she will see your changes in the latest version of the file.

**Updating or Synchronizing:**

Suppose you have a collaborator, Sally, who checked out a working copy of /calc at the

same time you did. When you commit your change to button.c, Sally's working copy is

left unchanged; Subversion modifies working copies only at the user's request.

To bring her project up to date, Sally can ask Subversion to *update* her working copy, by

using the **svn update** command. This will incorporate your changes into her working copy,

as well as any others that have been committed since she checked it out.

$ pwd

/home/sally/calc

$ ls -A

Makefile button.c integer.c .svn/

$ svn update

U button.c

Updated to revision 57.

The output from the **svn update** command indicates that Subversion updated the contents

of button.c. Note that Sally didn't need to specify which files to update; Subversion uses

the information in the .svn directory as well as further information in the repository, to decide

which files need to be brought up to date.

**Atomic transaction:**

An **svn commit** operation publishes changes to any number of files and directories as a

single atomic transaction. In your working copy, you can change files' contents; create, delete, rename, and copy files and directories; and then commit a complete set of changes as

an atomic transaction.

By atomic transaction, we mean simply this: either all of the changes happen in the repository,

or none of them happens. Subversion tries to retain this atomicity in the face of program

crashes, system crashes, network problems, and other users' actions.

**Revisions:**

Each time the repository accepts a commit, this creates a new state of the filesystem 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 0 and consists of nothing but an empty root directory.

It's important to note that working copies 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 copy from a repository whose most recent revision is 4:

calc/Makefile:4

integer.c:4

button.c:4

At the moment, this working directory corresponds exactly to revision 4 in the repository.

However, suppose you make a change to button.c, and commit that change. Assuming

no other commits have taken place, your commit will create revision 5 of the repository,

and your working copy will now look like this:

calc/Makefile:4

integer.c:4

button.c:5

Suppose that, at this point, Sally commits a change to integer.c, creating revision 6. If

you use **svn update** to bring your working copy up to date, it will look like this:

calc/Makefile:6

integer.c:6

button.c:6

Sally's change to integer.c will appear in your working copy, and your change will still

be present in button.c. In this example, the text of Makefile is identical in revisions 4,

5, and 6, but Subversion will mark your working copy of Makefile with revision 6 to indicate

that it is still current. So, after you do a clean update at the top of your working copy, it

will generally correspond to exactly one revision in the repository.

Figure 1.7, “The repository” illustrates a nice way to visualize the repository. Imagine an array

of revision numbers, starting at 0, stretching from left to right. Each revision number

has a filesystem tree hanging below it, and each tree is a “snapshot” of the way the repository

looked after a commit.

**Figure 1.7. The repository**



**Note: Global Revision Numbers**

Unlike most version control systems, Subversion's revision numbers apply to *entire*

*trees*, not individual files. Each revision number selects an entire tree, a particular

state of the repository after some committed change. Another way to think about it is

that revision N represents the state of the repository filesystem after the Nth commit.

When Subversion users talk about “revision 5 of foo.c,” they really mean “foo.c as

it appears in revision 5.” Notice that in general, revisions N and M of a file do *not* necessarily

differ! Many other version control systems use per-file revision numbers, so

this concept may seem unusual at first.

1. **How Working Copies Track the Repository**

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

in the .svn/ administrative area:

• What revision your working file is based on (this is called the file's *working revision*)

• A timestamp recording when the local copy was last updated by the repository

Given this information, by talking to the repository, Subversion can tell which of the following

four states a working 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 working revision. An **svn commit** of the file will do

nothing, and an **svn update** of the file will do nothing.

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 you last updated. There are local changes that

have not been committed to the repository; thus an **svn commit** of the file will succeed

in publishing your changes, and an **svn update** of the file will do nothing.

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 in order to make it current with the

latest public revision. An **svn commit** of the file will do nothing, and an **svn update** of

the file will fold the latest changes into your working copy.

Locally changed, and out of date

The file has been changed both in the working directory and in the repository. An **svn**

**commit** of the file will fail with an “out-of-date” error. The file should be updated first;

an **svn update** command will attempt to merge the public changes with the local

changes. If Subversion can't complete the merge in a plausible way automatically, it

leaves it to the user to resolve the conflict.

This may sound like a lot to keep track of, but the **svn status** command will show you the

state of any item in your working copy.

1. **Updates and commits are separate**

One of the fundamental rules of Subversion is that a “push” action does not cause a “pull,”

nor vice versa. Just because you're ready to submit new changes to the repository doesn't

mean you're ready to receive changes from other people. And if you have new changes

still in progress, **svn update** should gracefully merge repository changes into your own,

rather than forcing you to publish them.

The main side effect of this rule is that it means a working copy has to do extra bookkeeping

to track mixed revisions as well as be tolerant of the mixture. It's made more complicated

by the fact that directories themselves are versioned.

For example, suppose you have a working copy entirely at revision 10. You edit the file

foo.html and then perform an **svn commit**, which creates revision 15 in the repository.

After the commit succeeds, many new users would expect the working copy to be entirely

at revision 15, but that's not the case! Any number of changes might have happened in the

repository between revisions 10 and 15. The client knows nothing of those changes in the

repository, since you haven't yet run **svn update**, and **svn commit** doesn't pull down new

changes. If, on the other hand, **svn commit** were to automatically download the newest

changes, it would be possible to set the entire working copy to revision 15—but then we'd

be breaking the fundamental rule of “push” and “pull” remaining separate actions. Therefore,

the only safe thing the Subversion client can do is mark the one file—foo.html—as

being at revision 15. The rest of the working copy remains at revision 10. Only by running

**svn update** can the latest changes be downloaded and the whole working copy be marked

as revision 15.

1. **Mixed revisions are normal**

The fact is, *every time* you run **svn commit** your working copy ends up with some mixture

of revisions. The things you just committed are marked as having larger working revisions

than everything else. After several commits (with no updates in between), your working

copy will contain a whole mixture of revisions. Even if you're the only person using the repository,

you will still see this phenomenon. To examine your mixture of working revisions,

use the **svn status** command with the --verbose option.

Often, new users are completely unaware that their working copy contains mixed revisions.

This can be confusing, because many client commands are sensitive to the working revision

of the item they're examining. For example, the **svn log** command is used to display

the history of changes to a file or directory When the user invokes this command on a working copy object, he expects to see the entire history of the object. But if the object's working revision is quite old

(often because **svn update** hasn't been run in a long time), the history of the *older* version

of the object is shown.

1. **Mixed revisions are useful**

If your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly

*backdate* (or update to a revision older than the one you already have) portions of your

working copy to an earlier revision. Perhaps you'd like to test an earlier version of a submodule contained in a subdirectory, or perhaps you'd like to figure out when a bug first came into existence in a specific file. This is the “time machine” aspect of a version control system—the feature that allows you to

move any portion of your working copy forward and backward in history.

1. **Mixed revisions have limitations**

However you make use of mixed revisions in your working copy, there are limitations to

this flexibility.

First, you cannot commit the deletion of a file or directory that isn't fully up to date. If a newer

version of the item exists in the repository, your attempt to delete will be rejected to prevent

you from accidentally destroying changes you've not yet seen.

Second, you cannot commit a metadata change to a directory unless it's fully up to date.

A directory's working revision defines a specific set of entries and properties, and thus committing

a property change to an out-of-date directory may destroy properties you've not yet seen.