Chapter 4

Basic Subversion Usage

In this chapter, I will walk you through the basic use of Subversion, from creating a new repository, all the way through to more complex features such as creating and merging a branch.

If you are like me, you learn best by actually sitting down at a computer and getting your feet wet. To allow you to do that, all of the examples in this chapter build on each other, one right after the other, starting with a simple Hello World project. All of the examples in this chapter assume that you are in a UNIX-like environment, such as Linux or Mac OS X. For the most part, they will all work if you are running in a Windows environment, with a few minor changes, such as turning forward slashes (/) in path names into backslashes (\).

We'll start the project with two files, which make up our example project. The first file is the source for our Hello World program, `hello.c`:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello World!!\n");

    return 0;
}
```

The second file is a makefile, which could be used with the `make` program to compile our fabulous application. The file is named, appropriately, `Makefile`:

```
all: hello.c
    gcc hello.c -o hello
```

4.1 Creating the Repository

Subversion stores files in a repository database (which is Berkeley DB by default, but version 1.1 also supports FSFS). So, the first thing to do is create a new repository where we can store Hello World. This is done using the `svnadmin` program, which is used for most server-side administrative tasks when using Subversion. The repository is created with the

`svnadmin create` command. First, though, you will want to create a directory in your home directory, where you can store the repository (you'll see later why creating it directly in your home directory isn't a good idea). If you were creating a repository to use on a server, for production use, you would probably want to place it somewhere other than your home directory, such as `/srv/` or `/var/`.

In the following example, `bill` should be replaced with your username on the machine where you are creating the repository. Similarly, in all future examples where you see my username, `bill`, you should replace it with your own username.

```
$ svnadmin create --fs-type fsfs /home/bill/my_repository
```

This creates an empty repository named `my_repository` in your home directory, using the filesystem-based FSFS repository backend. By choosing FSFS instead of the default Berkeley DB backend, you don't need to worry about repository wedging, which can happen if Berkeley DB is interrupted. Although wedging is not fatal to repositories, it will leave your repository in a temporarily inaccessible state, which requires the Berkeley DB recovery process to be run in order to clear the wedge.

In most situations, you will want to create a repository on a server, and access it through HTTP/HTTPS, or the Subversion server `svnserve`. For simplicity's sake, though, we'll take advantage of Subversion's capability to communicate directly with a repository on the local machine, using a local directory path, for all of the examples in this chapter.

After you've run the create command, you can look in your home directory, and you will see that Subversion has created a directory named `my_repository`. This contains the repository database. In general, you won't directly edit any files in this directory. Instead, you will interact with it through Subversion's `svn` command. If you look inside this directory, you can see that there are a bunch of files and directories, but there is little reason for you to worry about what they are for at this point. In Chapter 11, "The Joy of Automation," you will learn how you can edit some of the files in your repository to customize Subversion's behavior.

```
$ ls /home/bill/my_repository/
README.txt  conf/  dav/  db/  format  hooks/  locks/
```

4.2 Getting Files into the Repository

Now that you have created the empty repository, it's time to get the project files into it. To do this, you need to put the files into a basic directory structure for the repository, and then import the entire structure. It would be possible to make that directory structure as simple as a single directory named `hello_world`, with `hello.c` and `Makefile` inside. In practice, though, this isn't a very good directory structure to use.

If you recall from the previous chapter, Subversion does not have any built-in support for branches or tags, but instead just uses copies. This proves to be a flexible way to handle branches and tags, but if they're just copies, there is no set means for identifying what files are branches and what files are on the main source trunk. The recommended way to get

around this missing information is to create three directories in your repository, one named `branches`, another named `tags`, and a third named `trunk`. Then, by convention, you can put all branched versions of the project into the `branches` directory and all tags into the `tags` directory. The `trunk` directory will be used to store the main development line of the project.

With large, complex repositories, there are a number of different ways you can set up the directories for the trunk, branches, and tags, which can accommodate multiple projects in one repository, or facilitate different development processes. Because our test project is simple though, we'll keep the repository simple and place everything at the top level of the repository. So, to get everything set up, you first need to create an overall directory for the repository, called `repos`. Then, set up `trunk`, `branches`, and `tags` directories under that, and move the original source files for the project into the `trunk` directory.

```
$ mkdir repos
$ mkdir repos/trunk
$ mkdir repos/branches
$ mkdir repos/tags
$ ls repos
branches  tags  trunk
$ mv hello.c repos/trunk/
$ mv Makefile repos/trunk/
$ ls repos/trunk/
Makefile  hello.c
```

After the directories are created and filled, the only thing left to do is import the directory into our repository. This is done using the `import` command in the `svn` program.

```
$ svn import --message "Initial import" repos file:///home/bill/ ¬
  repositories/my_repository
Adding         repos/trunk
Adding         repos/trunk/hello.c
Adding         repos/branches
Adding         repos/tags

Committed revision 1.
```

The `--message "Initial import"` option in the preceding example is used to tell Subversion what to use as a log message for the import. If you omit the `--message` option when you are importing or committing files to the repository, Subversion will automatically open an editor for you,[1] which will allow you to type a log message as long and complex as you need it to be.

Now that the repository structure has been imported, you can delete the original files. Everything should now be stored in the database, and ready for you to check out a working directory and begin hacking.

1. See Section 7.2, "Editing the Configuration Files."

4.3 Creating a Working Copy

The working copy is where you make all of your changes to the files in the repository. You check out the working copy directory by running the `svn checkout` command, and it contacts the repository to retrieve a copy of the most recent revision of all the data in your repository. A local directory tree that matches the tree inside the repository will be created, and the downloaded working directory files will be placed in there.

```
$ svn checkout file:///home/bill/my_repository/trunk my_repos_trunk
A  my_repos_trunk/hello.c
A  my_repos_trunk/Makefile
Checked out revision 1.
```

As you can see, Subversion has checked out the `trunk` directory from your repository, creating a local working copy directory with the name `my_repos_trunk`, along with the files `hello.c` and `Makefile` that were stored in `trunk`. You'll notice, however, that the `branches` and `tags` directories were not checked out. Subversion will let you check out the entire repository at the top level, but doing so is generally not good practice. If you do, you may end up with multiple local copies of the source tree, because branches and tags are made by copying files. Instead, if you only check out the main trunk, you will ensure that you only have one version of the files at a time in your working copy. If you need to access specific branches or tags, you can either check them out on an individual basis, into their own working copies, or switch files in your trunk working copy to point to other locations in the repository (e.g. branches or tags), which I'll show you how to do in a later section.

Now, if you look closely at your new working copy, you can see that Subversion also has placed one additional directory in the directory that you checked out.

```
$ ls my_repos_trunk
Makefile  hello.c
$ ls -A my_repos_trunk
.svn  Makefile  hello.c
```

When you check out a repository, Subversion places a `.svn` directory in every directory of the repository. Inside these directories, Subversion places a wide variety of metadata about the working directory, including what repository the working directory comes from and what revisions of each file have been checked out. It also stores complete pristine versions of the last checked-out revision of each file in the working directory. This allows Subversion to provide you with diffs showing what changes you have made locally to a file, without needing to contact the server.

4.4 Editing Files

Now that you have checked out a working copy of the repository, it's time to edit some files. Let's say, for example, that you decide that your Hello World program needs to tell everyone about a glorious new system that you've just discovered. So, you bring up your favorite text editor and modify `hello.c`, so that it now looks like this:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Subversion Rocks!!\n);

    return 0;
}
```

Whew! After a big change like that, it can be hard to remember everything that you've done since the repository was checked out. Sounds like it's time to learn about Subversion's query commands.[2]

Subversion provides you with a couple of different commands for querying the current state of the working directory. The first, `svn status`, shows the current status of local files. You can see whether files have been added, modified, deleted, and a number of other things. Running it on your current working directory shows that one file has changed:

```
$ svn status my_repos_trunk
M      my_repos_trunk/hello.c
```

Each output line from SVN (in this case, only one) shows the state of a file in the working directory tree, with files that haven't changed since the last update omitted. As you can see, the `hello.c` file is listed, with an `M` that informs you that the local file has been modified.

Just knowing that the file has been modified, though, doesn't tell you a whole lot. It would be significantly more useful if you could see exactly what has been modified. This is where the `svn diff` command comes in. With the diff command, you can see the difference between the local copy of the file and the last version to be updated from the repository (you can also use the diff command to compare with revisions other than the most recent, as you'll see in Chapter 5, "Working with a Working Copy").

```
$ svn diff my_repos_trunk/hello.c
Index: my_repos_trunk/hello.c
===================================================================
--- my_repos_trunk/hello.c (revision 1)
+++ my_repos_trunk/hello.c (working copy)
@@ -2,7 +2,7 @@

 int main(int argc, char** argv)
 {
-    printf("Hello World!!\n");
+       printf("Subversion Rocks!!\n");

        return 0;
 }
```

2. See how I set you up for that one with a smooth, effortless transition?

As you can see, the diff command gives you an overview of the changes made to the file, including both removed information and added information. The header portion of the output identifies which files have been diffed. In this case, it shows that the original file was revision 1 of `hello.c`, and all lines from that which have been removed in the latest version (which it notes, is the working copy) are marked with a - sign. Additionally, all lines added to the working copy, but not in revision 1, are marked with a +. The `@@ -2,7 +2,7 @@` tells you that the diff to follow shows lines two through seven from both versions of `hello.c`. For each section of a file that has changed, the diff will show the changes, as well as a few of the unchanged lines before and after the change. These can help you get your bearings as to which section of the file it is that you are seeing changed.

4.5 Committing Changes

Now that you've made some changes to the project, it's time to commit those changes back to the repository. This is done with the `svn commit` command, as follows.

```
$ cd my_repos_trunk/
$ svn commit --message "Changed program output"
Sending        hello.c
Transmitting file data .
Committed revision 2.
```

When you run the commit command, Subversion sends the changes you have made to the repository, where a new revision is created with the changes applied to the files in the repository. As soon as the commit is complete, other users are able to update their own working copies of the repository and retrieve the updates that you have just committed.

As you can see, the output from the commit command says that Subversion committed revision two. This is the global revision number of the repository. Whenever any user commits a change to the repository, Subversion increments the revision number of the entire repository by one. This way, you are always able to refer to a snapshot of the repository at a given point in time, using the revision number. Unlike CVS, there is no need to remember that revision 10 of file A was matched with revision 15 of file B.

4.6 Viewing the Logs

After you have multiple revisions committed to the repository, you will likely find a time when you want to review the history of changes you have made. This can be done using the `svn log` command, which displays the commit logs for a file. If multiple files are given, Subversion aggregates the logs for all of the files into a single log output, showing the log entries for each revision where at least one of the listed files changed. If a directory is given, SVN will output the log information for not only the given directory, but also all files and subdirectories contained within the directory given.

You can view the log for the `hello.c` file by running the following.

```
$ svn log hello.c
------------------------------------------------------------------
r2 | bill | 2004-07-11 04:45:12 -0500 (Sun, 11 Jul 2004) | 1 line

Changed program output
------------------------------------------------------------------
r1 | bill | 2004-07-08 16:28:57 -0500 (Thu, 08 Jul 2004) | 1 line

Initial import
------------------------------------------------------------------
```

Looking at the output, you can see that it shows both of the revisions that you have committed, along with the name of the user who made the commit, the time of the commit, the total number of lines that were changed in files that were part of the commit (in this case, just one), and the log message that you gave for each commit.

4.7 Creating a Tag

It's hard to improve upon perfection, so you decide that it's time to release your Hello World application so that others can bask in its glory. You would, however, like to be able to continue work on version 2.0 of Hello World (with many new features) after you release version 1.0. To ensure that you always have access to exactly what you released as version 1.0 (in case, for example, you later find a bug that you want to fix), it would be handy to mark the revision of the repository that made up the version 1.0 release. You could do this by writing down the revision number somewhere, but the easier (and more reliable) way to keep track of the version 1.0 release is to create a tag.

Subversion has no explicit concept of a "tag." Instead, it simply uses lightweight copies of the files being tagged. So, to create a tag, you just have to use the `svn copy` command to create a copy of the files included in the release in the `tags` directory that you created when you made the initial repository. In order to avoid the expense of actually making a copy of the files in the directory (as opposed to just marking them as copied), and because you never checked out the `tags` directory into your working copy, it is best to perform the copy entirely in the repository, by running the following command.

```
$ svn copy --message "Tagged version 1.0 release" file:///home/bill/ ¬
  my_repository/trunk/ file:///home/bill/my_repository/tags/version_1_0 ¬
  /

Committed revision 3.
```

This performs the copy inside the repository immediately, and creates a new revision. You can see that the copy occurred by using the `svn list` command to see the contents of the repository.

```
$ svn list file:///home/bill/my_repository/tags
version_1_0/
```

```
$ svn list file:///home/bill/my_repository/tags/version_1_0
Makefile
hello.c
```

4.8 Creating a Branch

You should have tags pretty well down at this point, so let's take a look at branches. Say, for example, that your boss isn't yet quite as enlightened as you are, and decides you need to release a version of Hello World that touts that *other* version control system. Because you know he's heading down a dead-end path, though, you don't want to stop development on your already excellent version of Hello World. The solution is to create a branch of the project, which will allow you to take the project in a different direction, while maintaining the current development path in parallel.

Branches in Subversion are just like tags, copies of the original repository part they refer to. Therefore, you make them exactly the same way; only in this case, you will want to copy the files into the `branches` directory, instead of the `tags` directory.

```
$ svn copy --message "Created a branch of the project to make the boss ¬
  happy" file:///home/bill/my_repository/trunk/ file:///home/bill/ ¬
  my_repository/branches/cvs_version
Committed revision 4.
```

After you have created the branch, you'll need to put it into a working copy so that you can make changes to it. You could check out the branch (using `svn checkout`) into a new working copy. In fact, that will work just fine. There's a better solution, though. Instead of checking out a new working copy, you can switch your current working copy to point to the branch, instead of the `/trunk` directory that it points to now. To do this, you need to use the `svn switch` command. To switch your working copy to the branch, run the following command line.

```
$ svn switch file:///home/bill/my_repository/branches/cvs_version
Updated to revision 4.
```

The files in your working copy now point to the `branches/cvs_version/` directory, and any changes that you commit will be applied to that directory. In this particular case, running `svn switch` didn't make any changes to the files in your working copy, because the branch and your trunk are identical. Had they been different, Subversion would have updated all of your working copy files to reflect the `cvs_version/` directory that you switched to.

You can look at what directory you are currently switched to by running `svn info`. For instance, the following command line will show you that your current working copy is switched to the `cvs_version` branch (look at the `URL` line).

```
$ svn info
Path: .
```

```
URL: file:///home/bill/my_repository/branches/cvs_version
Repository UUID: 5380c965-27ea-0310-9e69-9d7dd738c2c1
Revision: 4
Node Kind: directory
Schedule: normal
Last Changed Author: bill
Last Changed Rev: 4
Last Changed Date: 2004-12-01 00:46:13 -0500 (Wed, 01 Dec 2004)
```

Now that you have switched your working copy to point to the branch, you'd probably like to make some changes to the branch. For instance, to make your boss happy, you might change `my_repository/branches/cvs_version/hello.c` to look like this:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("CVS is the best!!\n"); // Ugh! The boss made me do it

    return 0;
}
```

Then, when you commit those changes, they will be applied to the copied version of the file, but the original file will remain unaffected, as you can see in the log outputs here.

```
$ svn commit --message "Changed program output to praise CVS"
Sending        hello.c
Transmitting file data .
Committed revision 5.
```

After the commit, the branch shows the committed change.

```
$ svn log file:///home/bill/my_repository/branches/cvs_version/hello.c
------------------------------------------------------------------------
r5 | bill | 2004-07-12 23:32:11 -0500 (Mon, 12 Jul 2004) | 1 line

Changed program output to praise CVS
------------------------------------------------------------------------
r4 | bill | 2004-07-12 22:47:11 -0500 (Mon, 12 Jul 2004) | 1 line

Created a branch of the project to make the boss happy
------------------------------------------------------------------------
r2 | bill | 2004-07-11 04:45:12 -0500 (Sun, 11 Jul 2004) | 1 line

Changed program output
```

```
------------------------------------------------------------------
r1 | bill | 2004-07-08 16:28:57 -0500 (Thu, 08 Jul 2004) | 1 line

Initial import
------------------------------------------------------------------
```

But the the original `hello.c` file, still only shows the first two revisions.

```
$ svn log file:///home/bill/my_repository/trunk/hello.c
------------------------------------------------------------------
r2 | bill | 2004-07-11 04:45:12 -0500 (Sun, 11 Jul 2004) | 1 line

Changed program output
------------------------------------------------------------------
r1 | bill | 2004-07-08 16:28:57 -0500 (Thu, 08 Jul 2004) | 1 line

Initial import
------------------------------------------------------------------
```

Of course, now that you're done modifying the branch, it's a good idea to switch your working copy back to the trunk. If you don't make the switch as soon as you're done with the branch, it can be all too easy to forget and accidentally apply modifications to the wrong place.

```
$ svn switch file:///home/bill/my_repository/trunk/
U  hello.c
Updated to revision 5.
```

As you can see, Subversion updates your `hello.c` file so that it represents the trunk version, rather than your modified branch version of the file.

4.9 Merging a Branch

As various branches of a repository's main trunk progress and diverge, it's sometimes necessary to use changes made on one branch in a different branch. Subversion allows you to apply these changes using the merge command. Let's say you add a line of output to the Hello World program to make it output some copyright information so that it now looks like this:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Subversion Rocks!!\n");
    printf("Copyright 2004, Bill Nagel\n");
```

```
    return 0;
}
```

This change is of course committed as usual, using `svn commit`.

```
$ svn commit --message "Added copyright information"
Sending        hello.c
Transmitting file data .
Committed revision 6.
```

Because outputting the copyright information is something that would be useful in both your Subversion-praising version and in the CVS-praising branch, it would be nice to merge these changes over to the branch. To do this, you'll use the `svn merge` command.

The merge command works by taking the difference between two revisions of a file or directory in the repository and applying those differences to a location in your working directory. In this case, you want to apply the change made to the repository trunk in revision 6 to the cvs_version branch.

First, you should run `svn log` to check which revision(s) of the repository the change you want to merge was committed on.

```
$ svn log hello.c
------------------------------------------------------------------
r6 | bill | 2004-07-13 00:23:38 -0500 (Tue, 13 Jul 2004) | 1 line

Added copyright information
------------------------------------------------------------------
r2 | bill | 2004-07-11 04:45:12 -0500 (Sun, 11 Jul 2004) | 1 line

Changed program output
------------------------------------------------------------------
r1 | bill | 2004-07-08 16:28:57 -0500 (Thu, 08 Jul 2004) | 1 line

Initial import
------------------------------------------------------------------
```

In this case, you can see that the copyright information change was applied to the main trunk in revision 6. That means that to apply those changes to the branch version, you need to apply the difference between revision 6 of `hello.c` and revision 5.

The Subversion merge command takes as parameters two different revisions of a source directory, and a working copy path to apply the changes to. Although merges are relatively easy to undo, after you have run them, it is usually a good idea to execute the merge command first with the `--dry-run` option, which will show you the files that will be changed before it applies the change. This lets you see any potential merge conflicts before they happen, which can often make them easier to deal with, and may even allow you to

eliminate the conflict before it occurs. After the merge, it is a good idea to test the merged files and make sure everything was applied correctly, before committing the merge to the repository.

To run the merge in your repository, you can do the following.

```
$ svn switch file:///home/bill/my_repository/branches/cvs_version/
U  hello.c
Updated to revision 7.
$ svn merge --dry-run -r 5:6 file:///home/bill/my_repository/trunk
U  hello.c
$ svn merge --revision 5:6 file:///home/bill/my_repository/trunk
U  hello.c
$ cat hello.c
#include <stdio.h>

int main(int argc, char** argv)
{
        printf("CVS is the best!!\n"); // Ugh! The boss made me do it
        printf("Copyright 2004, Bill Nagel\n");

        return 0;
}
$ svn commit --message "Merged with revision 6 - revision 5 in trunk/ ¬
  hello.c"
Sending        hello.c
Transmitting file data .
Committed revision 7.
$ svn switch file:///home/bill/my_repository/trunk
U  hello.c
Updated to revision 7.
```

You may notice that I explicitly stated in my log file for the merge commit which revisions were merged with the branch. It is actually very important to keep that information in the logs whenever a commit is made, because Subversion doesn't yet do any sort of tracking of merges and branches. By keeping track of the merged revisions in the logs, you can help ensure that you don't accidentally apply a merge more than once, which can have the unintended side effects of triggering spurious conflicts or even putting back changes that were taken out previously.

4.10 Handling Conflicts

Let's finish this chapter by taking a look at conflicts and how you can resolve them when they occur. Conflicts occur when Subversion is unable to merge two files together automatically. Generally, this happens when two users have independently made a change to the

same area of a file. Because Subversion doesn't actually understand the files that it merges, it has no way of figuring out which of the two versions to use. Its only recourse, in this case, is to let the user solve the conflict.

Before you can resolve a conflict, you have to have a conflict. So, let's create a conflict. To start, check out a new working copy, which will represent the work of a second developer.

```
$ svn checkout file:///home/bill/my_repository/trunk/ /home/bill/ ¬
  other_dev_trunk
A   other_dev_trunk/hello.c
A   other_dev_trunk/Makefile
Checked out revision 7.
```

Then, edit the file `hello.c` in your new working copy, and change the line

```
printf{"Subversion Rocks!!\n");
```

so that it reads

```
printf("Subversion is Great!!\n");
```

After the change has been made, commit it to the repository.

```
$ svn commit --message "Changed to a more conservative phrase" /home/ ¬
  bill/other_dev_trunk/hello.c
Sending        hello.c
Transmitting file data .
Committed revision 8.
```

With your changes from the new working copy committed, it's time to go back to your original working copy. Once there, edit the copy of the file `hello.c` that is stored there, *without updating the file from the repository first*. This time, change the line

```
printf("Subversion Rocks!!\n");
```

to the third, yet equally complimentary line,

```
printf("Subversion is Awesome!!\n");
```

Now, try to commit this change to `hello.c`.

```
$ svn commit --message "Decided on a more hip phrase" /home/bill/ ¬
  my_repos_trunk/hello.c
Sending        my_repos/trunk/hello.c
svn: Commit failed (details follow):
svn: Out of date: '/my_repos_trunk/hello.c' in transaction '9'
```

Well, Subversion obviously didn't like that. The reason, of course, is that Subversion won't allow you to commit changes to a file if those changes cause a conflict with previous changes, which can happen if you try to commit without first updating the working copy to the latest revision. To resolve the conflict, you first need to update your working copy to the latest revision, using `svn update`.

```
$ cd ~/my_repos_trunk/
$ svn update
C  my_repos_trunk/hello.c
Updated to revision 8.
```

Notice that there is a `C` in front of the listing for `hello.c` instead of the normal `U` for files that have been updated. The `C` is used to indicate a conflict. When a conflict such as this one occurs, Subversion does two things. First, it marks the file as being in a conflicted state. Second, it creates four versions of the conflicted file, for you to use when resolving the conflict.

```
$ ls trunk/
Makefile  hello.c  hello.c.mine  hello.c.r7  hello.c.r8
```

The first file, named `hello.c` just like the original, contains the file with each conflicted area showing both possible versions of the file. The first version shown is the version in your working copy, before the conflict occurred, and begins at a «««<. The second version shown is from the version of the file that the working copy was being merged with. It begins at the ======= that separates the two versions, and ends at a »»»>. You can see an example of this diff view here.

```
#include <stdio.h>

int main(int argc, char** argv)
{
<<<<<<< .mine
        printf("Subversion Is Great!!\n");
=======
        printf("Subversion Is Awesome!!\n");
>>>>>>> .r8
        printf("Copyright 2004, Bill Nagel\n");

        return 0;
}
```

The second version of the file, `hello.c.mine`, is a copy of the file as it existed in your working directory right before the conflict. The third version, `hello.c.r7`, is the file as it existed in your working directory the last time you checked it out, prior to any local changes. The `.r7` tells you that the file is taken from revision 6 of the repository. The

fourth and final file, `hello.c.r8`, is the file as it exists in the repository revision that is being merged into the working directory. Like the previous file, the `.r8` extension on this file tells you that it is from revision 8 of the repository.

To resolve the conflict, you need to modify the original file (`hello.c` in this case) so that it represents the resolved, final version of the file that you would like to commit to the repository as part of the next revision. In doing so, you are free to make use of any of the conflict files Subversion provides (either by copying information from them or copying the file on top of the existing version wholesale), as well as any data from other sources, as necessary. If you need to, to resolve a conflict, you could even rewrite the entire file from scratch.

After you have the conflicted file set up the way you want it, with all conflicting data merged or removed, you need to tell Subversion that you are done. This is done through the `svn resolved` command, which tells Subversion to remove the flag that marks the file as conflicted. Subversion also removes the extra versions of the file that it created when the resolved command is run. After the conflict has been marked resolved, you are free to commit the file to the repository.[3]

```
$ svn resolved hello.c
Resolved conflicted state of 'hello.c'
$ svn commit --message "Resolved conflicted state"
Sending        trunk/hello.c
Transmitting file data .
Committed revision 9.
```

4.11 Summary

In this chapter, you have walked through most everything that you will encounter in day-to-day interaction with a Subversion repository. The first few sections walked through the basics of creating a repository, including how to get the initial files into a repository and how to check out a working copy of the repository. After that, you learned the basics of editing files and committing changes, followed by some more advanced techniques such as branching and merging. Finally, you saw how to manually merge conflicts that Subversion can't handle automatically.

This ends Part I of the book. Now that you've gone through a thorough introduction to what Subversion is and how it works, Part II will delve in depth into the workings of Subversion from the point of view of a user of the Subversion client.

3. Of course, it's entirely possible that someone else may have committed yet another version of the file while you were working on resolving the conflict, which could put the file into a conflicted state again.