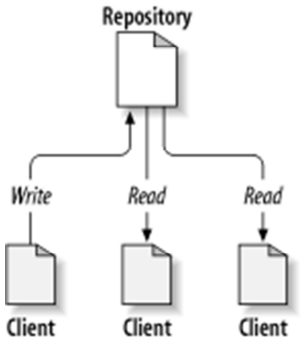
http://svnbook.red-bean.com/en/1.7/index.html

Why use version control?

- Manage file sharing
  (Specifically: Prevent people from erasing each other's modifications)
- Keep past versions of files/directories

Heart of Subversion: *The Repository*

- Typical Client/Server System
- The Repository is a kind of file server.
- However, Subversion remembers every change ever written to it!



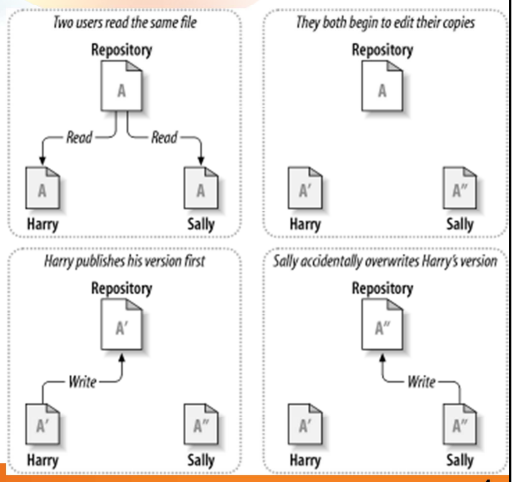
At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a *filesystem tree*—a hierarchy of files and directories. Any number of *clients* connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

Why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository *is* a kind of file server, but it's not your usual breed. What makes the repository special is that as the files in the repository are changed, the repository remembers each version of those files.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But what makes a version control client interesting is that it also has the ability to request previous states of the filesystem from the repository. A version control client can ask historical questions such as “What did this directory contain last Wednesday?” and “Who was the last person to change this file, and what changes did he make?” These are the sorts of questions that are at the heart of any version control system.

The Problem of File Sharing

- We want to avoid the following scenario:
- Overwriting each other's modifications

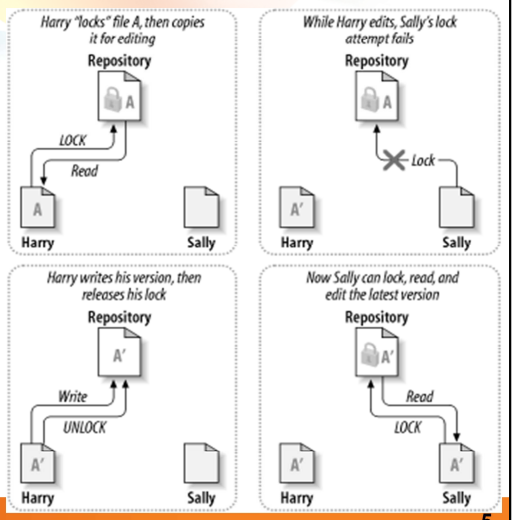


All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

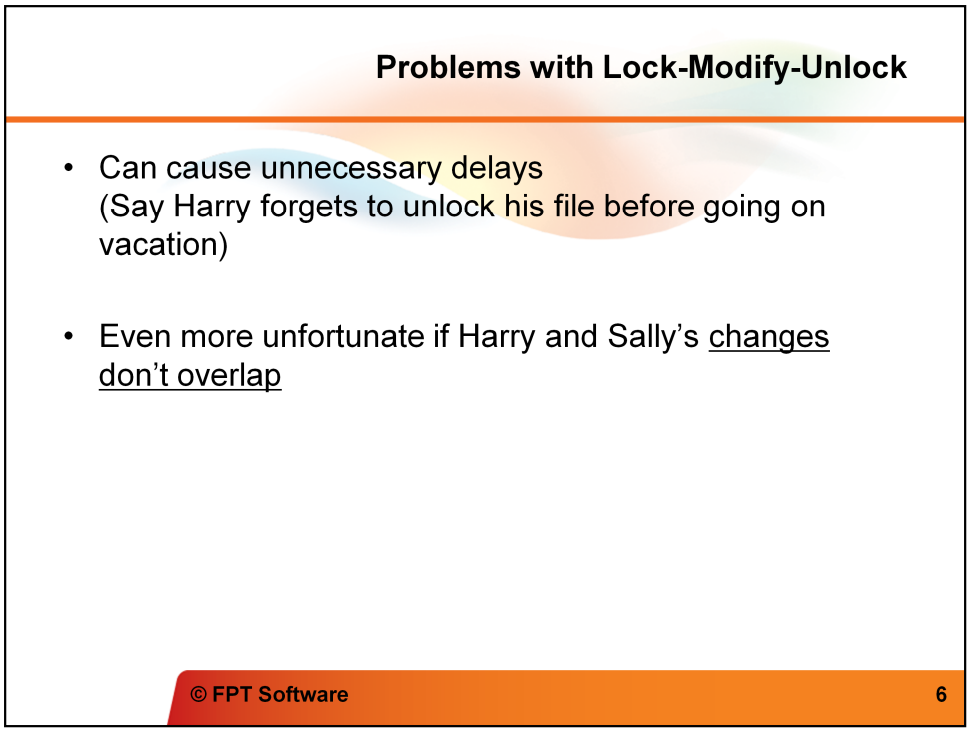
Consider the scenario shown in figure. Suppose we have two coworkers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost—or at least missing from the latest version of the file—and probably by accident. This is definitely a situation we want to avoid!

One Solution: Lock-Modify-Unlock

- Only one person may modify a file at any time.
- While this occurs, others can read the file, but not write to it



Many version control systems use a lock-modify-unlock model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must “lock” a file before he can begin making changes to it. If Harry has locked a file, Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file. Figure 1.3, “The lock-modify-unlock solution” demonstrates this simple solution.

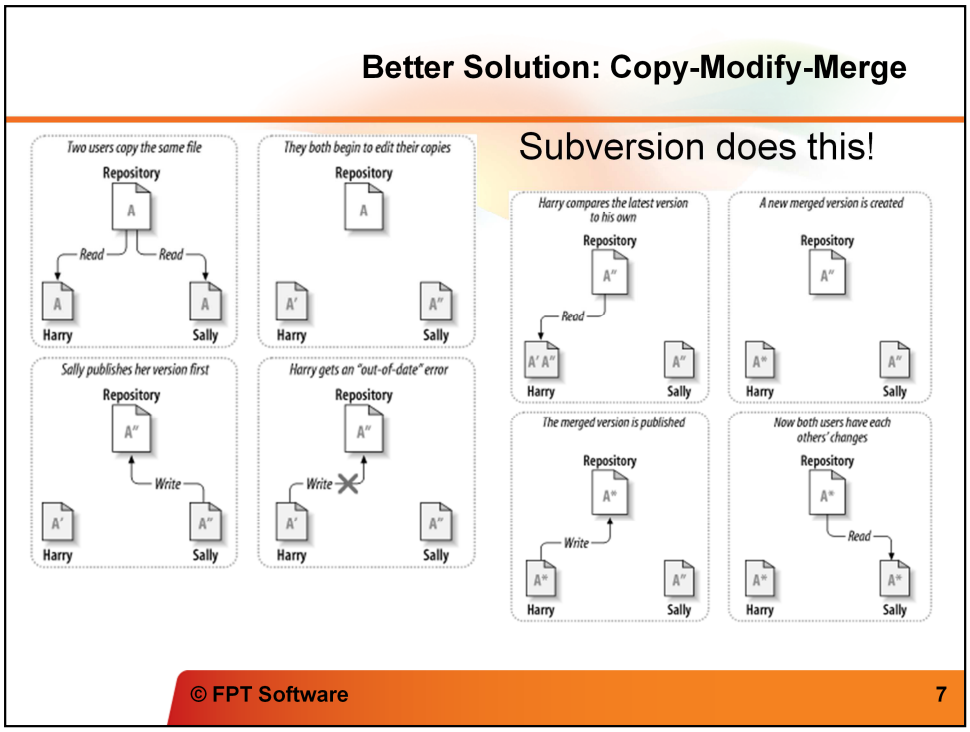


The problem with the lock-modify-unlock model is that it's a bit restrictive and often becomes a roadblock for users:

•*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.

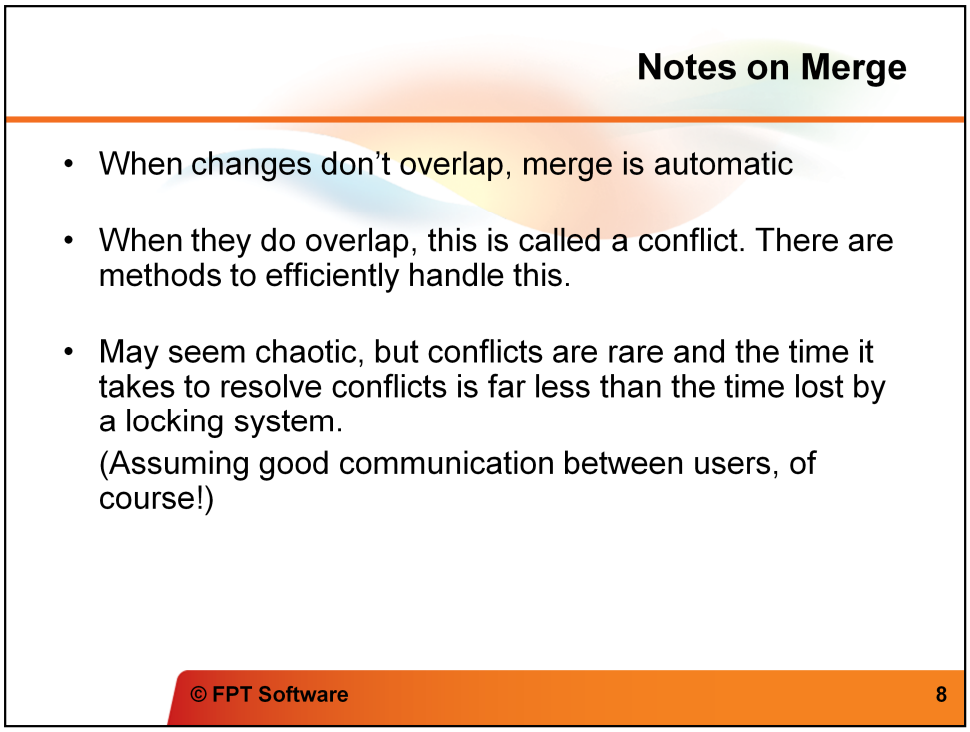
•*Locking may cause unnecessary serialization.* What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly 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.



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. 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.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently and make changes to the same file A within their copies. Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his file A is out of date. In other words, file A in the repository has somehow changed since he last copied it. So Harry asks his client to merge any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; once he has both sets of changes integrated, he saves his working copy back to the repository. Figure 1.4, “The copy-modify-merge solution” and Figure 1.5, “The copy-modify-merge solution (continued)” show this process.

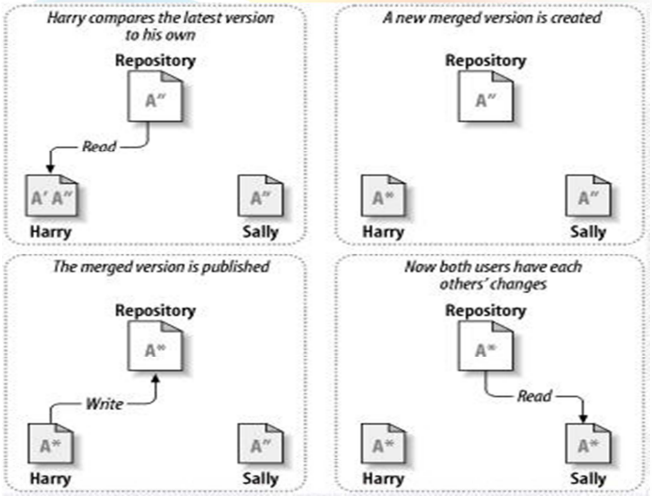


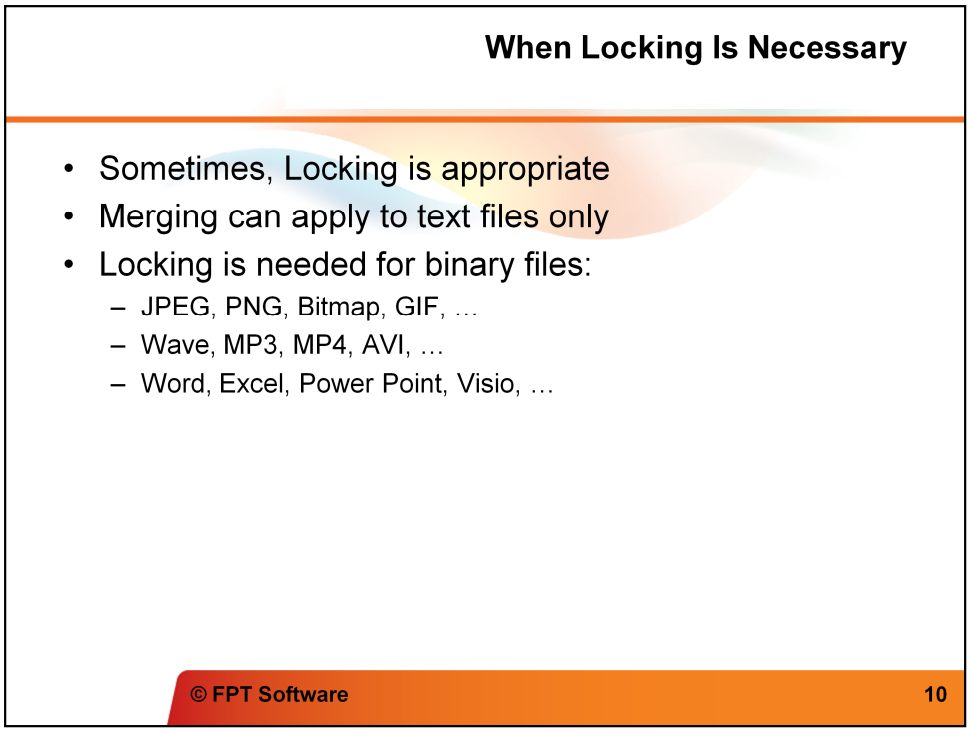
But what if Sally's changes *do* overlap with Harry's changes? What then? This situation is called a *conflict*, and it's usually not much of a problem. When Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of 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.

The Copy-Modify-Merge solution





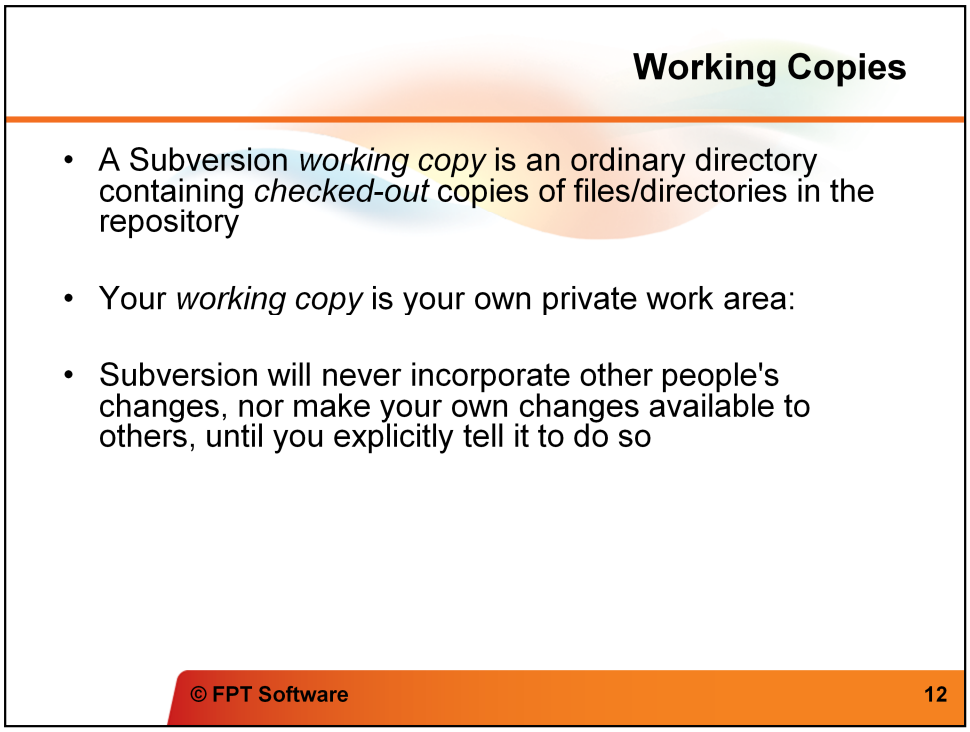
While the lock-modify-unlock model is considered generally harmful to collaboration, sometimes locking is appropriate.

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.

Access

- Repository can be accessed:
  - Via file:// server
    Easiest for us (since we're all on Madrid)
  - Via http:// server
    Requires in-depth knowledge of Apache
  - Via svn:// server
    Easiest across network access
  - Via svn+ssh://
    Same as svn://, but more secure

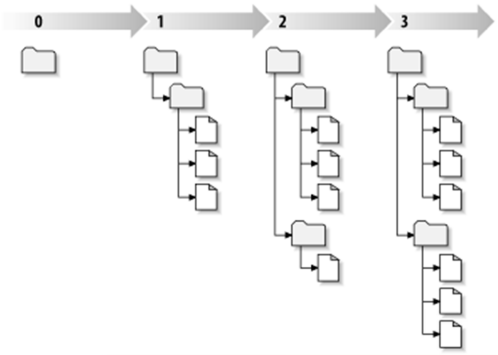


A version control system's value comes from the fact that it tracks versions of files and directories, but the rest of the software universe doesn't operate on "versions of files and directories". Most software programs understand how to operate only on a *single* version of a specific type of file. So how does a version control user interact with an abstract—and, often, remote—repository full of multiple versions of various files in a concrete fashion? How does his or her word processing software, presentation software, source code editor, web design software, or some other program—all of which trade in the currency of simple data files—get access to such files? The answer is found in the version control construct known as a *working copy*.

A working copy is, quite literally, a local copy of a particular version of a user's VCS-managed data upon which that user is free to work. Working copies[5] appear to other software just as any other local directory full of files, so those programs don't have to be "version-control-aware" in order to read from and write to that data. The task of managing the working copy and communicating changes made to its contents to and from the repository falls squarely to the version control system's client software.

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

A Subversion client commits (that is, communicates the changes made to) any number of files and directories as a single atomic transaction. By atomic transaction, we mean simply this: either all of the changes are accepted into the repository, or none of them is. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

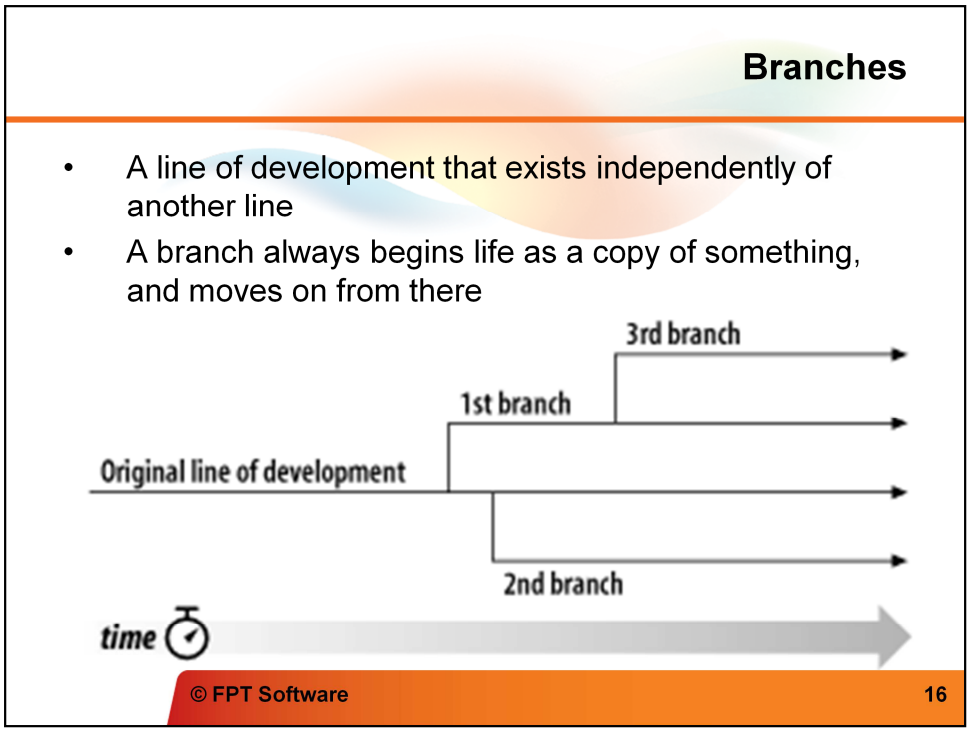
Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number assigned to the previous revision. The initial revision of a freshly created repository is numbered 0 and consists of nothing but an empty root directory.

Revisions

- We can refer to past versions of files & dir's:
  - By revision number: svn diff –r 3:4
  - By keyword: svn log –-revision HEAD
  - By dates: svn checkout –r {2002-06-22}
- Revisions are our time machine!

Revision Keywords

- HEAD: Latest revision in repository
- BASE: The “pristine” copy of the working copy (i.e. when checkout was done)
- COMMITTED: The last revision in which item actually changed (or at BASE)
- PREV: The revision just before last revision in which an item changed (COMMITTED-1)



Suppose it's your job to maintain a document for a division in your company—a handbook of some sort. One day a different division asks you for the same handbook, but with a few parts “tweaked” for them, since they do things slightly differently.

What do you do in this situation? You do the obvious: make a second copy of your document and begin maintaining the two copies separately. As each department asks you to make small changes, you incorporate them into one copy or the other.

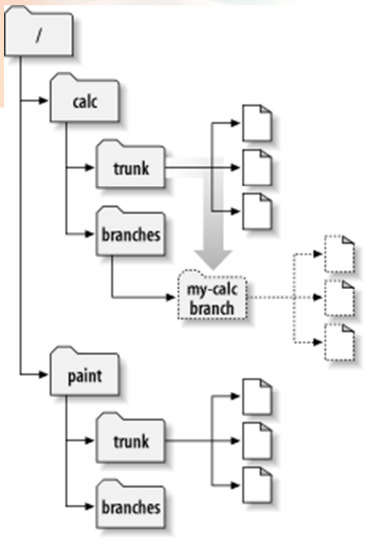
You often want to make the same change to both copies. For example, if you discover a typo in the first copy, it's very likely that the same typo exists in the second copy. The two documents are almost the same, after all; they differ only in small, specific ways.

This is the basic concept of a branch—namely, a line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of something, and moves on from there, generating its own history

Subversion has commands to help you maintain parallel branches of your files and directories. It allows you to create branches by copying your data, and remembers that the copies are related to one another. It also helps you duplicate changes from one branch to another. Finally, it can make portions of your working copy reflect different branches so that you can “mix and match” different lines of development in your daily work.

Repository with new copy

- Creating a branch is very simple with command
  - svn copy
- Branches are cheap copies

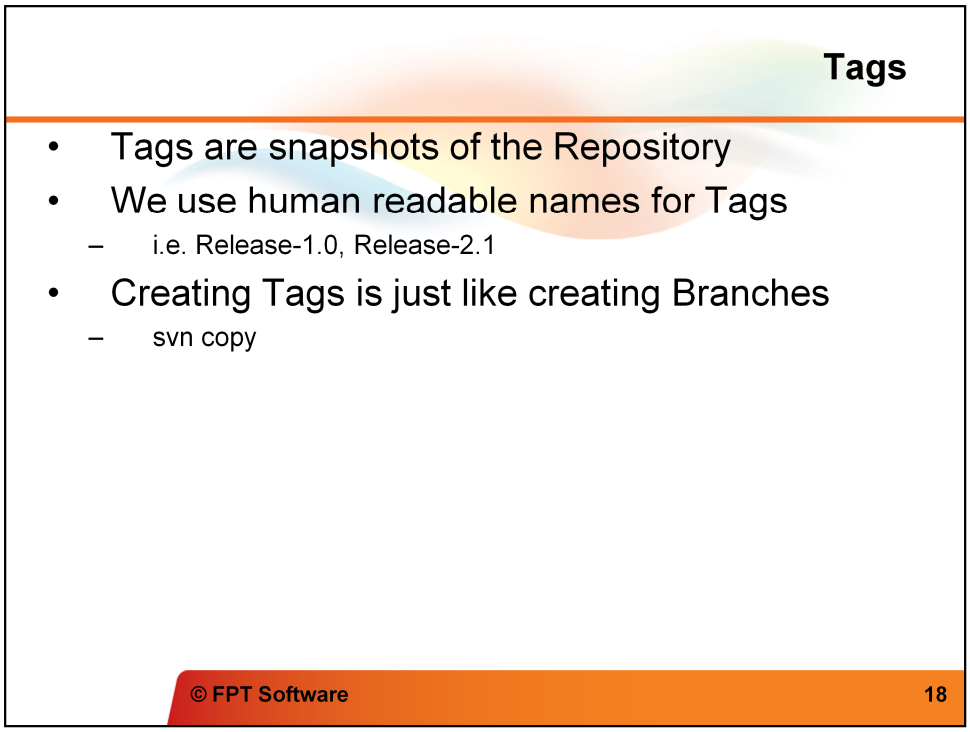


Creating a branch is very simple—you make a copy of your project tree in the repository using the **svn copy** command. Since your project's source code is rooted in the /calc/trunk directory, it's that directory that you'll copy. Where should the new copy live? Wherever you wish. The repository location in which branches are stashed is left by Subversion as a matter of project policy. Finally, your branch will need a name to distinguish it from other branches. Once again, the name you choose is unimportant to Subversion—you can use whatever name works best for you and your team.

Subversion's repository has a special design. When you copy a directory, you don't need to worry about the repository growing huge—Subversion doesn't actually duplicate any data. Instead, it creates a new directory entry that points to an *existing* tree. If you're an experienced Unix user, you'll recognize this as the same concept behind a hard link. As further changes are made to files and directories beneath the copied directory, Subversion continues to employ this hard link concept where it can. It duplicates data only when it is necessary to disambiguate different versions of objects.

This is why you'll often hear Subversion users talk about “cheap copies.” It doesn't matter how large the directory is—it takes a very tiny, constant amount of time and space to make a copy of it. In fact, this feature is the basis of how commits work in Subversion: each revision is a “cheap copy” of the previous revision, with a few items lazily changed within. (To read more about this, visit Subversion's web site and read about the “bubble up” method in Subversion's design documents.)

Of course, these internal mechanics of copying and sharing data are hidden from the user, who simply sees copies of trees. The main point here is that copies are cheap, both in time and in space. If you create a branch entirely within the repository (by running **svn copy *URL1 URL2***), it's a quick, constant-time operation. Make branches as often as you want.



Another common version control concept is a tag. A tag is just a “snapshot” of a project in time. In Subversion, this idea already seems to be everywhere. Each repository revision is exactly that—a snapshot of the filesystem after each commit.

However, people often want to give more human-friendly names to tags, such as release-1.0. And they want to make snapshots of smaller subdirectories of the filesystem. After all, it's not so easy to remember that release 1.0 of a piece of software is a particular subdirectory of revision 4822.

In Subversion, there's no difference between a tag and a branch. Both are just ordinary directories that are created by copying. Just as with branches, the only reason a copied directory is a “tag” is because *humans* have decided to treat it that way: as long as nobody ever commits to the directory, it forever remains a snapshot. If people start committing to it, it becomes a branch.

Practice

- Setup a server
  - Create a repository
  - Start server
  - Create project
- Checkout a working copy
- Make changes & Commit changes
- Update on client
- Merge/Resolve conflict
- Create a new branch
- Browse the repository

CMD – Setup server

- Open a CMD window
- Create a repository
  svnadmin create d:\repos
- Server config:
  echo.duydd = duydd >> d:\repos\conf\passwd (use the same account as you login to Windows)
  notepad d:\repos\conf\svnserve.conf (uncomment "password-db = passwd" line)
- Start Server
  cd d:\
  d:
  svnserve –d (startup the SVN server at svn://localhost/repos)

CMD – Setup server (Cont)

- Open another CMD
- Prepare project

```
mkdir   d:\myprj
mkdir   d:\myprj\trunk
mkdir   d:\myprj\branches
mkdir   d:\myprj\tags
```

- Import project into SVN

```
svn   mkdir   svn://localhost/repos/myprj   -m   “Make prj dir”
svn   import   d:\myprj   svn://localhost/repos/myprj   -m   “Import”
```

CMD – Checkout a working copy

- Checkout for Harry

  mkdir d:\harry\myprj\trunk

  svn co svn://localhost/repos/myprj/trunk d:\harry\myprj\trunk
- Checkout for Sally

  mkdir d:\sally\myprj\trunk

  svn co svn://localhost/repos/myprj/trunk d:\sally\myprj\trunk

CMD – Make changes & commit

- Add new file

  cd d:\harry\myprj\trunk (should cd into the folder you are working on)
  d: (go to D drive)
  echo Hello World! > d:\harry\myprj\trunk\foo.txt
  svn add foo.txt
  svn ci -m "Committing foo.txt"

- Modify existing file

  echo My name is Foo! >> foo.txt
  svn status (check the file status, you will see "M foo.txt")
  svn ci -m "Make change to foo.txt"

CMD – Update from repository

- Folder status before synchronize
  dir d:\sally\myprj\trunk (shows nothing in this folder)
  svn info d:\sally\myprj\trunk (still at revision 2)
- Update Sally's working copy
  svn up d:\sally\myprj\trunk
  dir d:\sally\myprj\trunk (shows foo.txt)
  svn info d:\sally\myprj\trunk (now at revision 4)

CMD – Produce conflicts

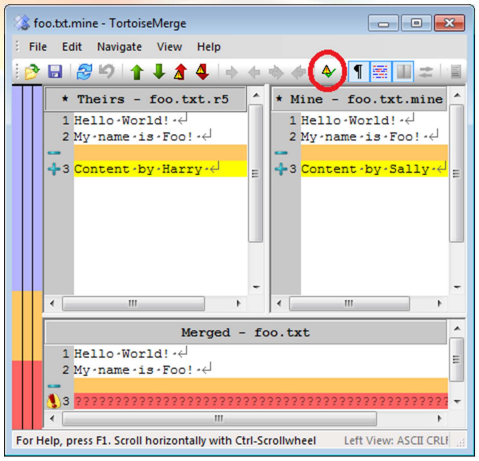
- Make change by Harry
  cd d:\harry\myprj\trunk
  d: (go to D drive)
  echo.Content by Harry >> foo.txt
  svn ci * -m "Change by Harry"
- Make change by Sally
  cd d:\sally\myprj\trunk
  d: (go to D drive)
  echo.Content by Sally >> foo.txt
  svn ci * -m "Change by Sally" (this will fails with error)

CMD – Merge & resolve conflicts

- Update the latest changes from server
  cd d:\sally\myprj\trunk
  d: (go to D drive)
  svn up (Conflict discovered in 'foo.txt')
  Choose Postpone by typing ‘p’ in the choices to resolve conflict

CMD – Resolve conflicts (Cont)

- Open Windows Explorer
- Go to d:\sally\myprj\trunk
- Right click foo.txt
- Choose menu TortoiseSVN
  ➔
  Edit Conflicts
- After merging, click Resolved button



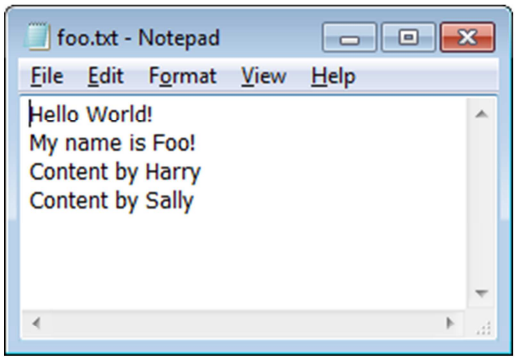
CMD – Commit after resolving

- Commit the resolved file by Sally

  cd d:\sally\myprj\trunk

  d: (go to D drive)

  svn ci * -m "Change by Sally"



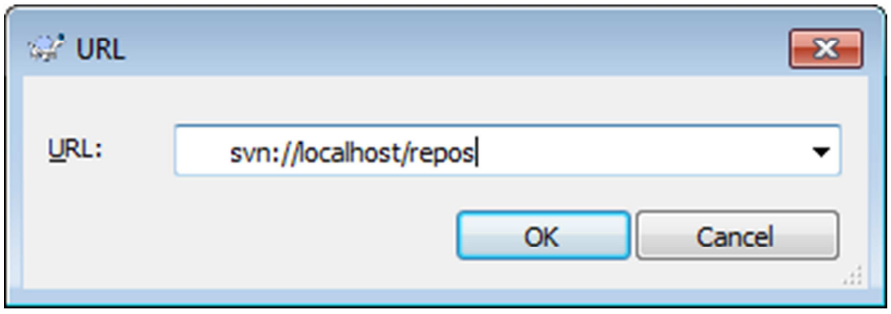
CMD – Create a new branch

- Make branch on revision #4

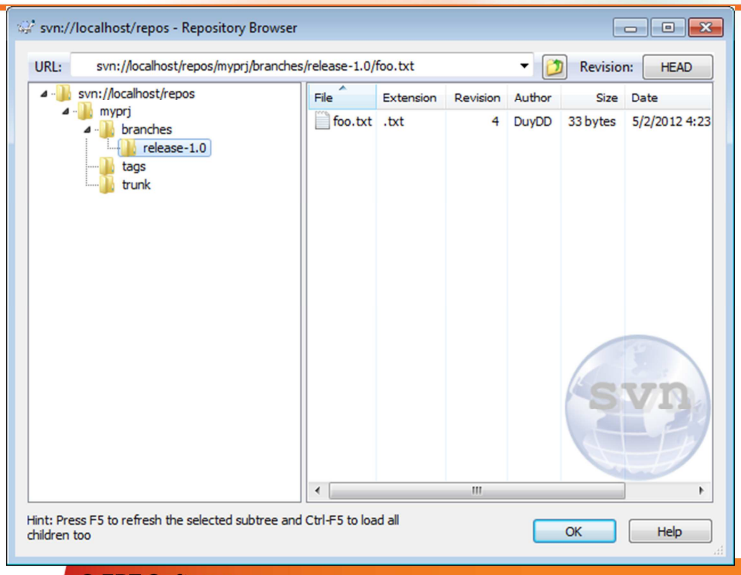
```
svn copy svn://localhost/repos/myprj/trunk svn://localhost/repos/myprj/branches/release-1.0 -r 4 -m "Make release 1.0 branch"
```

CMD – Browse the repository

- Using TortoiseSVN Repo Browser
  Right click on Windows Explorer
  Choose menu TortoiseSVN ➔ Repo Browser

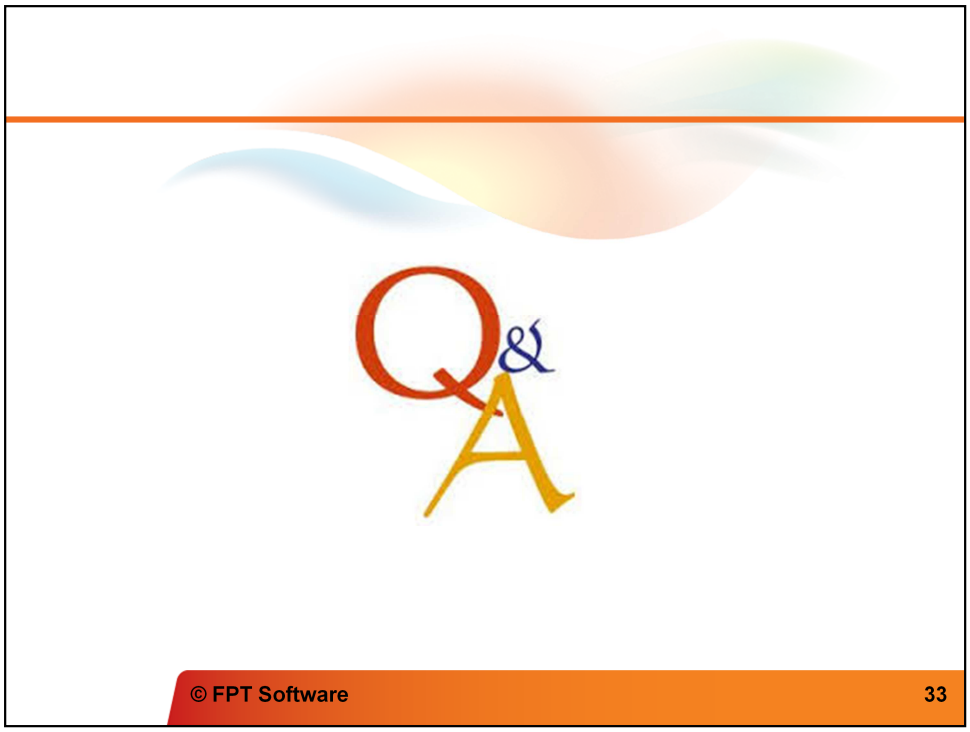


CMD – Browse the repository



More references

- Using SubversionSVN_v1.0.doc
- SVN with TortoiseSVN and GoogleCode_v1.1.doc
- MOCK_Using SVN.doc


Q&A