SVK - a visual guide

Version 0.2
Generated: 17. July 2005

Russell Brown

Changes

Version	Date	Author	Description
0.2	205/07/17	Russ Brown	Added section numbering Added this 'Changes' page Put visual descriptions into a dedicated section Completely re-vamped diagrams (thanks autrijus & clkao) Imported 'Introduction' from svk wiki Made headers and footers work better
0.1	2005/07/15	Russ Brown	First release

Table of Contents

1 Introduction

1.1 What Is SVK?

SVK is a distributed version control system built upon Subversion's underlying filesystem.

1.2 SVK's History

The idea of source control management has been around for many years. Early on the centralized repository scheme became the standard for many developer teams. In more recent times, distributed source control has become part of the normal work flow process. CVS is a widely known and used standard for Source Control Management. Because of some shortcomings, a team of developers decided to create an improved version of CVS called Subversion (svn). Subversion, however is still a centralized repository system, but because it was a new design, it was developed to be more flexible than its predecessor. Because of this fact, developers are able to take key components and connect them together in ways that the original designers never dreamt of. SVK is one of those dreams. SVK, utilizing the underlying svn API, is able to create a local repository on the users machine.

Aside from SVK's decentralized features, SVK is also a very useful too for those who want to continue to use a centralized source control system: SVK is most definitely not restricted solely to decentralized environments.

1.3 SVK's Features

The biggest feature of SVK is the ability to do disconnected distributed development while keeping the ability to perform version control. When reconnected, any changes can be merged back into a central repository, distributed via standard channels, or hosted for others to connect to and merge from.

Another great benefit is that anyone with Subversion experience will feel right at home since many of the commands are similar. After the difference in principles is understood, usage of SVK should be similar with some added benefits, such as better performance and better merge tracking.

For a new user to version control, SVK has a relatively small core command set and is perfect for a single user wishing to have the benefit of working under version control.

1.4 About this book

The purpose of this book is to help new users learn how to use SVK, by describing how the system is laid out and how certain commands work in terms of data flow.

This book is not aimed at people completely new to change control: a level of familiarity with concepts such as checking in and checking out is required.

2 Topology

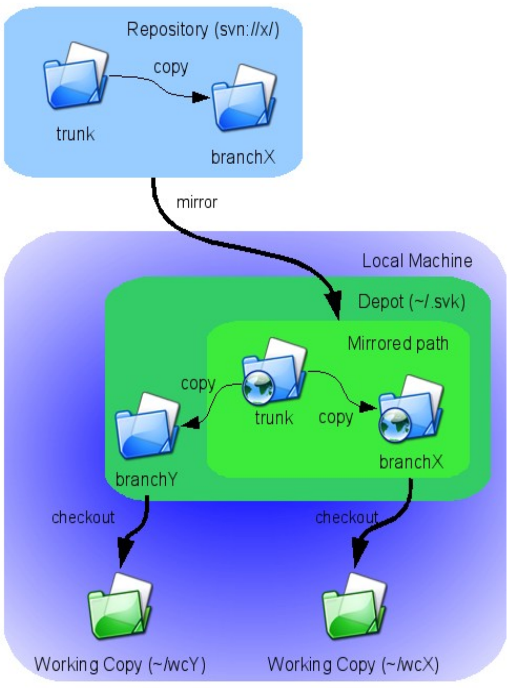
Many people find it easier to learn how to use a system if they know how the system works. They also find it easier to learn things like commands if they understand how the commands work and why you'd want to use them in the first place.

This section presents svk's topology: how the data it manages hangs together and where it is located.

SVK is a very flexible system that can be used in a great numbers of ways. It's therefore very difficult to draw a useful diagram that makes clear how it all works in one go.

This document therefore ignores svk's ability to mirror repositories other than Subversion, and also its ability to operation completely detached from a repository.

The diagram on the right shows the basic topology of svk, split into two areas:



2.1 Repository

The blue area at the top represents the Subversion repository itself. This is the place that standard Subversion users check code out from and commit code to. It could be on a server accessible via http(s) or the svn protocol, or a local or NFS-born file:// repository.

Here we show that the repository contains the trunk and one branch, called branchX. The branch was created by copying the trunk, as is the standard way of doing such things in Subversion.

2.2 Local Machine

The shaded area represents the user's local machine. This is where svk does all of its work.

The dark green area represents the depot, which contains a local mirror of the repository (shown as the lighter-green area). We see that it also contains the trunk and branchX, and the mirror knows that the branch was copied from the mirror. That information is important for merge tracking and also for keeping the mirror small, since the size of the branch is based only on the changes made on it, rather than the original files that were copied (thereby taking advantage of Subversion's cheap copies mechanism).

This svk user has also created a local branch: branchY. This branch is also copied from trunk, but note that it exists outside the mirrored path: the repository never gets to know about it. It's a sandbox which the user can use to develop an idea of some project even while detached from the repository, and throw it away when he's bored with it.

At the bottom of the 'Local Machine' section we see two working copies. These are created using the 'svk checkout' command. This user has chosen to check out the two branches that he has in his depot: branchX and branchY. He hasn't checked out the trunk at all at this time.

The Working Copy is the area in which the user actually does his work. It's a directory that contains the files stored in the branch that was checked out from the repository.

3 Visual Command Descriptions

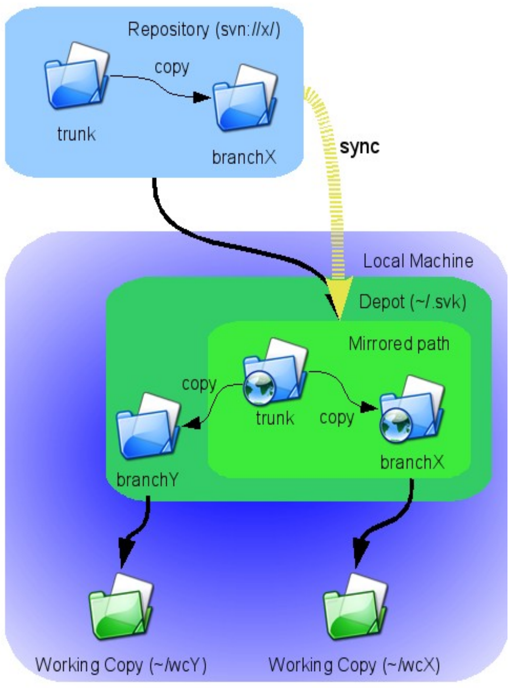
This section attempts to explain how the data-altering commands work in svk. It builds on the same diagram as in the topology section, so you can easily see the relationship between the commands and the topology, as well as each other.

Many of svk's commands are read-only. These all work on the working copy and/or depot only. They don't need access to any remote repository, which is where svk's 'detached working' ability comes from.

When viewing the diagrams for a given command, bear in mind that the arrow represent the flow of information for a given operation. Also note that come commands actually perform a number of distinct commands to make up their effect. These are shown on the diagrams in the form of numbered arrows.

3.1 Sync (sync, sy)

3.1.1 The diagram



The diagram on the right shows the flow of the sync command. As you can see, the information flows from the repository to the depot. Specifically, the information goes tot he mirrored path in the depot. Note that in this case the sync isn't tied to any particular branch on the repository: it works on the full mirrored path (which can and often does include the entire repository).

3.1.2 What does it do?

The sync command retries new revisions from the repository and adds them to the mirror in the depot. It is simply a stream of changesets that are applied in chronological order.

If you have mirrored the entire repository, sync will retrieve changesets for all branches in that repository. So for example a given sync could contain two commits to the trunk and two commits to some other branch.

3.1.3 Why do I want to do it?

Without syncing, you will never see new revisions committed by other users of the repository. Syncing often is a good habit to get into. Note that syncing is often done for you by certain commands (assuming that they're being used on a mirrored path).

3.1.4 How does it work?

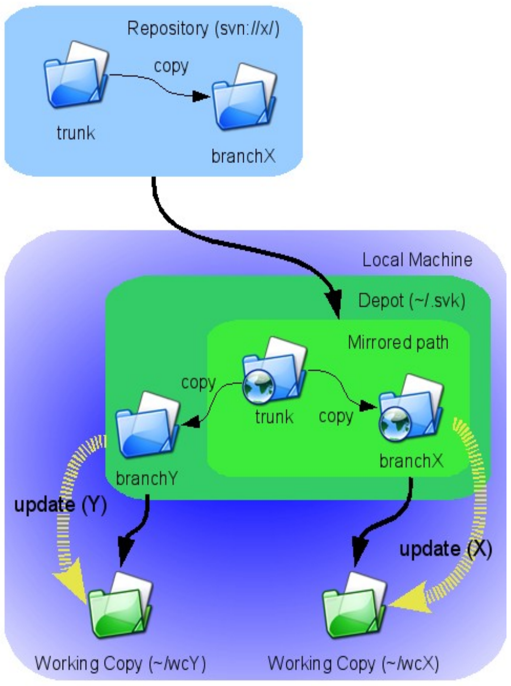
When svk mirrors a remote repository, it makes a note of the highest revision number that is has mirrored. Initially, this will be revision 0: the mirror command does not actually pull down any revisions at all. The sync command obtains the current latest revision number from the repository and if it is greater than the revision number it has recorded, it retrieves the new revisions and commits them into the depot. svk will tell you about the revisions it is committing: it will output lines like this:

```
Committed revision 1274 from revision 1273.
```

This means that is has just committed revision 1273 from the repository into your depot. The commit to your depot happened at revision 1274. It is entirely expected that these two numbers drift increasingly apart over time.

3.2 *Update (update, up)*

3.2.1 The Diagram



The diagram on the right shows the flow of the update command (assuming you don't pass any parameters).

This user has run update in both of his working copies. The two updates are separate commands. Note that information moves from the depot to the working copy, not involving the repository at all.

What does it do?

The update command works on a working copy. It retrieves new changes from the depot that the working copy is checked out from and applies them to the working copy. The command takes as an argument the path to update, which defaults to the current directory. By default update works recursively.

Why do I want to do it?

Use update regularly if you are working on a branch that other people could be committing changes to. Delaying this too long increases the potential for conflicts, since there is more chance that files you are working on may be affected by the update.

How does it work?

svk knows the current revisions of all files in your working copy. When you run update, it compares these revisions against the latest revision of the files in the depot, and if it is greater it retrieves the differences and applies them to the files in your working copy.

If svk updates a file that you have changed locally, it attempts to merge the changes in with yours. If there are conflicts, you will be asked to resolve them.

Note that update does not involve the repository at all. You can use this command with no network connection. The concept of a 'mirrored path' is irrelevant to update under normal circumstances. Because of this, update will not update your working copy with new changes that have been committed to the repository that you have mirrored. To get these, you must first do a sync, and then an update.

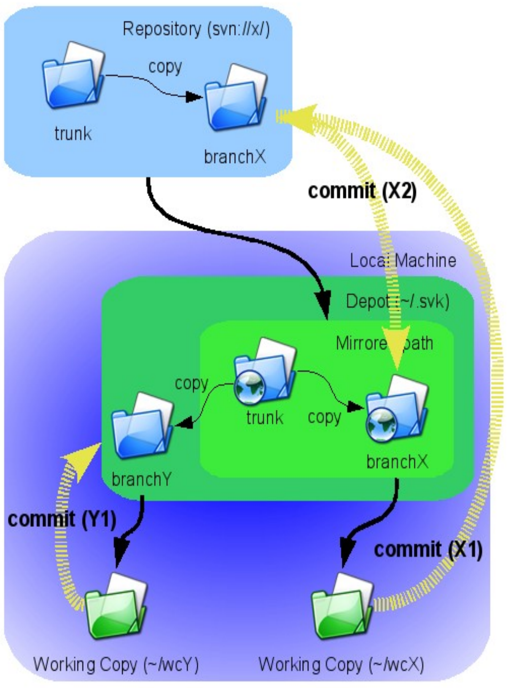
svk provides a convenient shortcut to performing both of these operations in sequence:

```
svk update -s
```

This will do a sync first, and then an update. Note that while the update stage is specific to the relevant working copy, the sync is just s standard sync which downloads all revisions from the mirrored server regardless of what specific paths they changed.

3.3 Commit (commit, ci)

3.3.1 The Diagram



The diagram on the right shows two separate commit actions: one on wcX and the other on wcY. Note that the commit from wcX involves the repository, since it involves a mirrored path. The commit to wcY only involves the depot, since it's a local branch.

What does it do?

Commit sends changes made in a working copy to the depot, via the repository if the working copy is checked out from a mirrored path.

Why do I want to do it?

Committing sets your changes in stone. Whatever happens, you will always be able to get back to the version of the files you commit. So it's good to commit whenever you reach a logical point at which you may want to get back to (for example before you do some refactoring or try an alternative algorithm). It's also the way that other users of the branch you are committing to can get at your changes (using update).

How does it work?

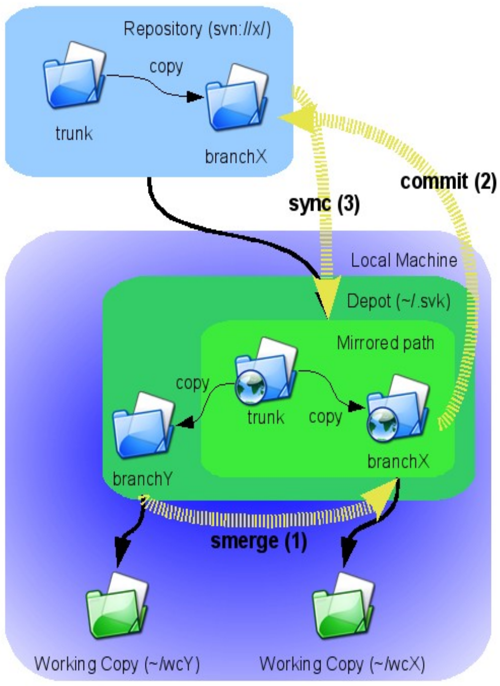
In the example on the diagram, we see that the commit in wcX first goes to the repository, as it has been checked out from a mirrored path. svk then syncs the committed revision back to the depot. You can't commit to a mirrored path unless you have a connection to the repository.

The commit from wcY goes straight to the depot, since it's a checkout of a local branch and is therefore not on a mirrored path.

3.4 Star Merge (smerge, sm)

The Diagram

The diagram on the right shows the users doing a Star Merge from branchY to branchX. Note that it is quite a complex three-stage process.



What does it do?

The Star Merge is probably one of the most clever and important things that svk adds to Subversion. It allows the merging of changes between branches without the need to explicitly track versions numbers that have been merged previously.

In the diagram, the user is doing a smerge from branchY to branchX. Since the two branches share a common ancestor (trunk), svk can figure out which changesets it needs to merge automatically. Once the merge has been calculated, it is committed as a changeset to the repository, which is then synced back like any other commit. Along with the changeset of the files themselves, svk also commits a Subversion property known as a 'merge ticket' which allows svk to track what it has merged in the past.

Why do I want to do it?

Merging between branches is core to a branch-based source control system. While svk does allow you to merge using the traditional explicit-revision method, there are very few reasons why you would sanely want to when smerge will do most of the work for you.

How does it work?

I have no idea: it's really clever. :)

Seriously, svk can track ancestry either using the path that the branch was originally copied from or from a previous merge ticket. It takes that information along with the information on what has been merged in the past to calculate what it needs to merge this time. Once the merge has been done, it updates the merge tracking information so that future merges work correctly too.

Note that smerge can also be asked to do the merge incrementally. That will cause it to commit the changesets being merged individually instead of as one single changeset.

3.5 Pull (pull, pu)

The Diagram

There is no diagram for pull, because it's a shortcut to commands that have already been described!

What does it do?

Pull merges changes from a branch's parent onto the branch, and optionally updates the branch working copy.

Why do I want to do it?

This depends on the way in which you are using branches. In many cases, pull allows you to retrieve changes from the trunk to your branch. This gets you the latest code and also allows you to ensure that the code you are writing on your branch works with the trunk code.

How does it work?

Pull takes as its argument one path: the path that you want to pull changes to. This path must be a path that has a parent: it is therefore meaningless to try a pull on the trunk (assuming that the trunk was not copied from anywhere).

If svk pull is performed in a working copy directory, you don't have to supply a path: svk will derive the depot path from the working copy.

pull works out the ancestor of the directory being pulled to. This gives it two paths to use.

Once it has the required information, pull performs the following commands for you:

1. sync (if relevant)
2. smerge from parent to specified path
3. update current working copy (if relevant)

svk does a sync first and foremost to ensure that it is working with the most recent changes (assuming that one of both of the two paths involved are on a mirrored path).

svk pull then does a smerge from you from the parent to the specified path.

If you are in a working copy checked out from the pulled path, svk will do an update for you, so your working copy contains the changes that have been pulled in.

3.6 Push (push, pu)

The Diagram

As with pull, there is no diagram for push, because it's a shortcut to commands that have already been described!

What does it do?

Push merges changes from a branch to its parent..

Why do I want to do it?

Again, this depends on the way in which you are using branches. In many cases, push allows you to send changes from our branch to the trunk, when you are ready for your changes to be used by others, or to have them rolled to live.

How does it work?

Push takes as its argument one path: the path that you want to push changes from. This path must be a path that has a parent: it is therefore meaningless to try a push on the trunk (assuming that the trunk was not copied from anywhere).

If svk push is performed in a working copy directory, you don't have to supply a path: svk will derive the depot path from the working copy.

push works out the ancestor of the directory being pushed. This gives it two paths to use.

Once it has the required information, push performs the following commands for you:

1. sync (if relevant)
2. smerge from specified path to parent

svk does a sync first and foremost to ensure that it is working with the most recent changes (assuming that one of both of the two paths involved are on a mirrored path).

svk push then does a smerge from you from the specified path to the parent.