Advantages of DVCS

* huge in open-source
* in dvcs each system is a version control system unlike the cvcs where only the centralized server is one.
* Branching architecture is different in both, working directory changes with change in the branch checked out in case of dvcs whereas a folder exists for each new branch in case of cvcs.
* In dvcs changes can be made and committed in the clones and only pushed (or merged after review) to the remote repository.
* Context switching using branching is simple and quick.  This plays well in the case of working on bug fixes and developing new features in a software development context.  Being distributed, branching allows for individuals to create their own personal source branches which are not readily accessible by others.

Advantages of DVCS over CVCS

(Atlassian - blog)

* Everything except pushing and pulling can be done without an Internet connection. So you can work on a plane, and you won’t be forced to commit several bugfixes as one big changeset
* Performing actions other than pushing and pulling change-sets is extremely fast because DVCS only needs to access the hard drive, not a remote server.
* Committing new change-sets can be done locally without anyone else seeing them. Once you have a group of change-sets ready, they can all be pushed at once.
* Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback before showing the changes to everyone.

Disadvantages of DVCS over CVCS

(Atlassian-blog)

* If a project contains many large, binary files that cannot be easily compressed, the space needed to store all versions of these files can accumulate quickly.
* If a project has a very long history (50,000 changesets or more), downloading the entire history can take an impractical amount of time and disk space.

Problems solved by DVCS

|  |  |
| --- | --- |
| Problem in cvcs | Solved by dvcs |
| Single point of failure. | Each client connected to remote repository is a version control system and can perform very well as a server in case of failure of the remote server. |
| Remote commits are slow. | Introduced concept of local commits: no network, wicked fast. |
| Unsolicited changes that may break your build. | Developer pushes their changes continuously.  Integrators review the code changes and can bring about changes to the reference code asynchronously. |
| Merging is painful. | Merging is breeze. |

Comparative study

In due course of our evaluation, we found that subversion is versatile and a feature rich tool which user friendly tools and interfaces. This would be an automatic choice for project development teams which work in a closely knit work environment. It has a small learning curve and a new user can be brought up to speed very quickly. But on the down side, if the project involves a lot Merging and Conflict in future, it might be difficult to manage them when volume increases.

In this case, git has an upper hand. Git has a lot of handy features, very effective in exposing merge conflicts and managing multiple parallel versions. Although it has a steep learning curve, it is very versatile and works well in a dynamic project environment. Open Source projects often look towards Git as an effective VCS due to its versatility. Most projects these days are moving away from the Centralized approach to the Distributed approach for the same reasons. Based on our experience and findings so far, it is our opinion that Distributed approach outweighs the pros and outdoes the cons of Centralized approach. Hence, we feel that Git is the way to go moving forward.

Key advantage of DVCS

With distributed, the authoritative or central source is the source you want it to be, rather than being constrained by the system into having to have your source in one place.  
There have been occasions where we have had to use one of the developer’s local repositories when the central server has been down.  
This is simply making a decision that the entire team is aware of, that you are going to push / pull to / from an alternative repository.  
Hg has it’s own inbuilt web server, so this is very easy to do.

Scenario of better cure from data loss

While there is a slight danger a users hard drive could crash before he pushes his changes, that danger exists with a centralized system too.

However, the centralized system has a far worse disaster scenario. What if that “IT-guaranteed backup” of your subversion server you’ve been so confident in turns out to be corrupted? What if the backup job was modified by some guy who was late for his daughter’s recital and didn’t quite get it done right?

With a DVCS, you have no loss of data whatsoever. Just figure out who has the most recent commit – they have a COMPLETE copy of the repository. Within minutes you can have another centralized repository setup with absolutely no loss of data.

But you say – the same thing is true of subversion! We all have a copy of the project on our hard drive.

No, you don’t. You all have a copy of one VERSION of your application. It may not even be working or complete. And all those previous versions? Log entries? Metadata? Gone. Unless your backup is good, you’re hosed.

If fault-tolerance is crucial to your organization, your safest bet is DVCS with a backed-up shared repository. That way, you’re completely covered even if a backup should fail.

Pushing changes more often or not???

if you don’t push/pull changes to at least one other machine, local HDD failure results in loss of work – so you either need to push every time – and lose the benefit of not having to be online – or you have to accept that it’s more likely you’ll lose work.

Scenario

Version Control Systems are design to help you and your team to collaborate actively on the same file. Because, as you grow, the chances of seeing different people hitting the same files get higher and higher  
Unwanted modifications can be avoided with a simple review process. If you make sure that someone looks at the new changesets you are probably going to catch most of the mistakes early in your development workflow.  
Regarding the local copy of the repository that exists in DVCS you just have to synch it regularly with your main repository to stay up-to-date. It’s actually as easy as pulling the latest version of the code.  
Finally, on the internet connection part we all need one when it comes to working together as a team 

A point to help making the Choice

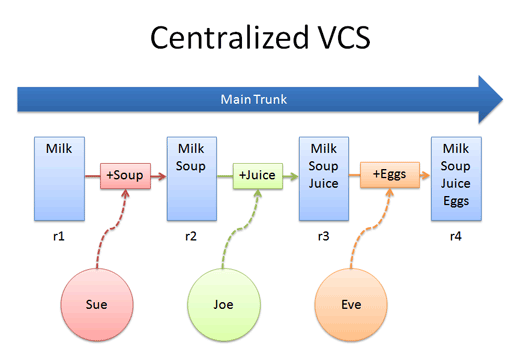
The real choice between using one type or the other is organizational -- if your project or organization wants centralized control, then a DVCS is a non-starter. If your developers are expected to work all over the country/world, without secure broadband connections to a central repository, then DVCS is probably your salvation.

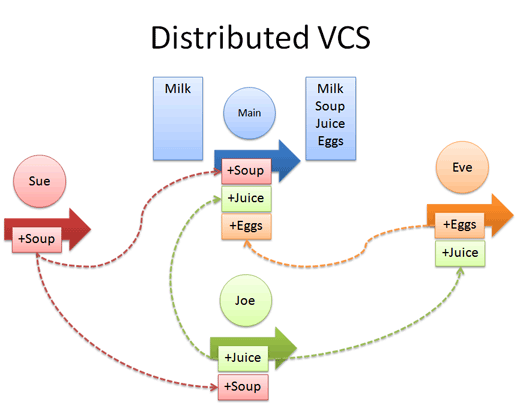
DVCS Core Advantages

* ***Everyone has their own local sandbox.***
  + You can make changes and roll back, all on your local machine.
  + No more giant checkins; your incremental history is in your repo.
* ***DVCS git works offline.***
  + You only need to be online to share changes.
  + Otherwise, you can happily stay on your local machine, checking in and undoing, no matter if the “server” is down or you’re on an airplane.
* ***DVCS git is fast.***
  + Since diffs, commits and reverts are all done locally.
  + There’s no sketchy network or server to ask for old revisions from a year ago.
* ***DVCS handles changes very well.***
  + Distributed version control systems were built around sharing changes.
  + Every change has a guid which makes it easy to track.
* ***Branching and merging is easy.***
  + Because every developer “has their own branch”, every shared change is like reverse integration.
  + The guids make it easy to automatically combine changes and avoid duplicates.
* ***With DVCS, there is less management.***
  + DVCS systems are easy to get running since there is no “always-running” server software to install.
* DVCS systems may not require you to “add” new users since you can just pick what URLs to pull from.

DVCS Core Disadvantages

* ***You still need a backup.***
  + Some claim your “backup” is the other machines that have your changes, but what if they didn’t accept them all? \*\* What if they’re offline and you have new changes?
* ***You still want a machine to push changes*** to “just in case”.
  + In Subversion, you usually dedicate a machine to store the main repo; do the same for a DVCS.
* ***There’s not really a “latest version”.***
  + If there’s no central location, you don’t immediately know whether to see others for the latest version.
  + A central location helps clarify what the latest “stable” release is.
* ***There aren’t really revision numbers.***
  + Every repo has its own revision numbers depending on the changes.
  + Instead, people refer to change numbers which are not intuitive. But, you can tag releases with meaningful names.





Left to explore

<https://www.youtube.com/watch?v=4fsSyLkBdB4#action=share>

<http://programmers.stackexchange.com/questions/134514/moving-from-a-cvcs-to-a-dvcs-in-a-big-company-what-is-the-right-way-to-do-it> **//imp 1**

<http://stackoverflow.com/questions/111031/comparison-between-centralized-and-distributed-version-control-systems> **//imp 2**

<http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/> **//done**

<https://www.appfusions.com/display/StashSCMImporter/CVCS+vs.+DVCS+In+a+Nutshell> **//done**

<http://chriswongdevblog.blogspot.in/2010/11/dvcs-vs-subversion-smackdown-round-3.html> **//done, not very useful**

Various scenarios

**HOW DO YOU CONVINCE MANAGEMENT AROUND ALL THE OFFICES OVER THE WORLD THAT ITS A SANE THING TO DO AND WILL ACTUALLY BRING HAPPINESS AND PRODUCTIVITY TO DEVELOPERS?**

* People accidentally breaking other's builds or stepping on each others' toes while submitting code revisions.

**This is not often as the responsible must wear a pink hat (hilarious stuff) until the build is fixed, but the hat is still there, and when builds get broken it affects the other offices around the world.**

**DVCS to the rescue**: Builds are now broken in a single office, people lose the fear to use version control and can actually second-guess themselves after or check with a peer by sharing their changes easier and not necessarily with the rest of the group.

* Unnecessary bureaucracy when it comes to granting access (e.g. looks like everyone is granted access to most of the code).

**There is a dedicated group of admins who take 1 to 3 days to grant access to the code, meanwhile other devs must ask their peers for the code just to look at it... but AFAIK, the often wait instead.**

**DVCS to the rescue:** Administration does not have to be centralized on a little group of overwhealmed people, new employees get the code from the project's responsible (1st level) or a peer with repository access (2nd level) who could even review their code and submit it along with his/her changes.

* Complaints with regards to slowness due to network latency. **Not that directly, but sometimes a) The perforce server is down, b) Internet is down, c) Internet or perforce server is slow. Yes, not that often, but it is annoying enough to kick you off your train of thought sometimes.**

**DVCS to the rescue**: When using the DVCS you obviously don't need a network connection. Worst case scenario?, share your changes over LAN or sneakernet (with patch queues).

* Trouble when trying to collaborate with external providers that do not necessarily belong to the organization. **This happens often enough and security is a concern here AFAIK, and clients suffer from the access bureaucracy**.

**DVCS to the rescue**: So granting access to a subrepo here would be the ideal solution here... also, clients themselves could use version control among themselves without needing to setup their own server and they would be able to integrate better to the internal workflow as currently Perforce access access to the local network, and thus more bureaucracy which is not necessarily security here, as external collaborators still have access to thumbdrives and email. Actual security would be having externals work in situ... in a dungeon or something, and grope them à la TSA on their way out to make sure they are not getting the code out.

* Workaround to integrate modules from other repositories. Requires additional maintenance/effort from the developers. \*These guys actually wrote themselves (I think) a plugin to make perforce fetch files from other repositories in other parts \*

**DVCS to the rescue**: Their solution looks a lot like the mercurial subrepo feature, wich uses a file (.hgsub) to indicate what code goes where, but they have to maintain the other repositories' states in the same file (which is done separately and automatically by mercurial in the .hgsubstate file).

* Working with the source code requires to mark files for edit in advance. This adds an annoying extra step IMHO. **If you don't do this the file will be in read only mode and you will not be able to edit it on your IDE, also, this IMHO makes the local copy prone to low integrity. This trait of perforce makes me think there was a lazy developer that didn't want to make the client check the files against the repo before versioning the changes. It also makes me think/personify Perforce as an idiot paranoid that says** ***"DONT CHANGE THE FILES WITHOUT TELLING ME OR THE SQUIRRELS WILL COME AND EAT ME!!!"***.

**DVCS to the rescue**: Well... I guess any other SCM software knows there are no carnivore squirrels.

* Little project integrity in developers' machines (see previous point; its harder to add changes without previously indicating them to Perforce). **You don't need a DVCS for this... but if you make changes locally and for some reason forgot to add them, you are going to have a hard time detecting it**. Once you detect it there is a two step command line instruction thing I saw somewhere to detect changes you did without telling Perforce.

Advantages of svn(cvcs)

* atomic commits
  + one transaction to add/update/delete files
  + promotes grouping related changes together in one commit to easily identify and revert
* full revision history retained for files that are renamed, copied, moved or removed
* no file locking (by design)
  + file locking is extremely counter-productive (especially for large teams in multiple timezones)
  + SVN's automatic merging and conflict resolution tools make locking unnecessary
* working copy of files are always writable and all changes are local until committed
  + allows you to easily revert unwanted changes at commit time
* fast and flexible update/commits
  + all tasks can be done from any level easily (updates, commits, merges, reverts, etc)
* branching and tagging are cheap operations
  + simple to create a tag/branch and merge between them
* ease of access to the commit log
  + provides high visibility to changes, file history, integrated file comparison
  + ease of reverting changes to a file or an entire commit
* easy to setup email triggers (on checkins to keep team informed of changes, etc)
* integrates with everything
  + windows explorer integration via Tortoise
  + with leading IDEs (Eclipse, IntelliJ)
  + Agile/continuous integration tools (Atlassian, Jira, Fisheye, TeamCity, Hudson, etc.)
  + various websites to provide project/committer stats (Ohloh, etc)
* ease of setup and administration

Small projects

1. If your development is linear and simpler (without requiring branches and parallel work), you should stick with Subversion

2.SVN is less complex than git.

To some extent, the two schemes are equivalent:

* A distributed VCS can trivially emulate a centralised one if you just always push your changes to some designated upstream repository after every local commit.
* A centralised VCS won't usually be able to emulate a distributed one quite as naturally, but you can get something very similar if you use something like [quilt](http://savannah.nongnu.org/projects/quilt) on top of it. Quilt, if you're not familiar with it, is a tool for managing large sets of patches on top of some upstream project. The idea here is that the DVCS commit command is implemented by creating a new patch, and the push command is implemented by committing every outstanding patch to the centralised VCS and then discarding the patch files. This sounds a bit awkward, but in practice it actually works rather nicely.