Decentralization allows any developer to contribute to a project without write permission on a central repository and the developers can also work with an elected central repository in a similar way to the centralized model. We list below some of the most common workflows distributed model:

* *Shared repository model*: With this flow the changes are consolidated on a central repository. This repository may be mediated by an approval system, such as Gerrit. This approval system was employed on Android project and requires that every commit goes through a code review process;
* *Pull requests model*: With this flow the changes are obtained on demand. The developer works only in their personal repository and signal to stakeholders so they can recover the changes directly from the repository. This is the model used by GitHub.

Even when a common repository is employed the use of distributed version control system implies a paradigm shift in relation to the centralized model (see Figure 1). In the distributed model, each developer has a local repository generated from the full copy (clone) of the source repository (in Figure 2, represented as a repository in the cloud).

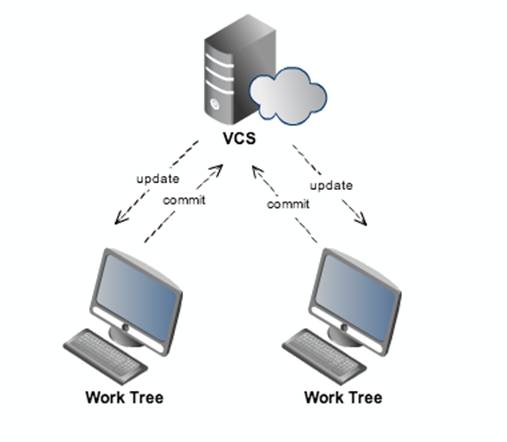


Figure : Centralised Model

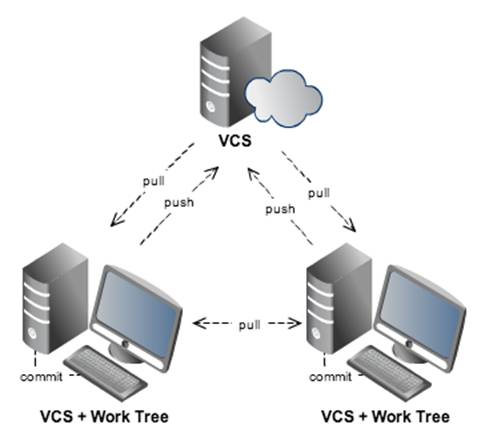
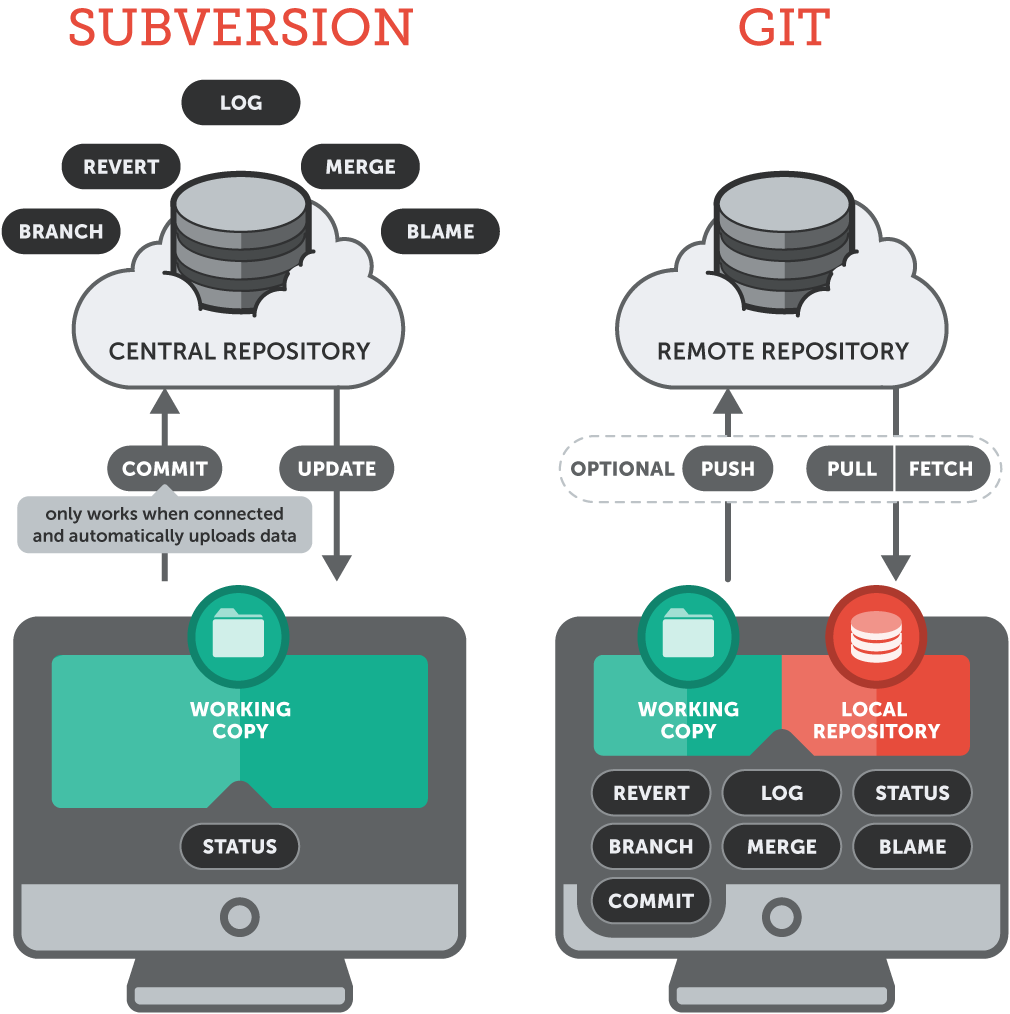
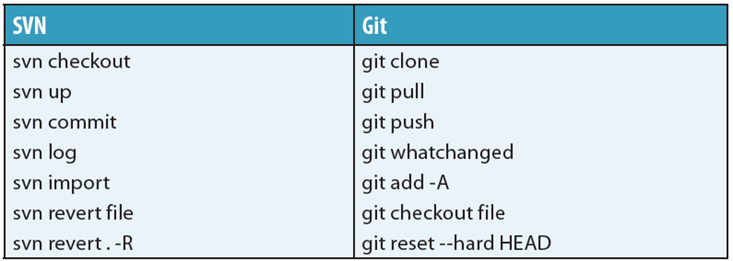


Figure : Decentralised Model

When the developer is working on his/her local copy, the code changes are made without network latency (offline). The synchronization between the local repository and remote repository can be done in batches, rather than the individual transfer of commits over the network as in the centralized model. In the distributed version control system, sending commits to another repository is the operation called *push* and the receipt of commits is called *pull*.

A distributed version control system is able to automatically detect any movement and/or files renaming, thus allowing the merge code between repositories even when files have different names and/or locations. This operation is known as a *merge-through-rename*. In Git this is possible because it creates a versioning of the content (snapshots) of each file, as opposed to the versioning of the file itself and its deltas, as in SVN.







|  |  |
| --- | --- |
| SVN | Git |
| Checkout trunk  svn checkout [http://svn.cern.ch/guest/AliRoot/trunk](http://svn.cern.ch/guest/AliRoot/trunk/) | git clonehttp://git.cern.ch/pub/AliRoot  (Note that "trunk" becomes "master")  Clone is done only once, checkout of the branches/tags is done inside the same working directory, one after the other.  If you want to checkout more than one branch/tag in parallel please follow the instructions from [Section 5](http://aliceinfo.cern.ch/Offline/node/2912/#ParallelGit). |
| Checkout a tag  svn checkout <http://svn.cern.ch/guest/AliRoot/tags/v5-04-72-AN> | Clone is done only once, checkout of the branches/tags is done inside the same working directory, one after the other  git clonehttp://git.cern.ch/pub/AliRoot  cd AliRoot  1. Checkout in "HEAD detached mode". When HEAD is detached, it points directly to a commit—instead of indirectly pointing to one through a branch. You can think of a detached HEAD as being on an unnamed branch.  git checkout v5-04-72-AN  git checkout v5-04-73-AN  A "git branch" will show  $ git branch \* (detached from v5-04-72-AN)   master 2. Checkout into a local branch            git checkout -b <new\_branch\_name> <tag>  git checkout v5-04-72-AN-local v5-04-72-AN  or to just keep it simple, use the same name for the branch as for the tag  git checkout -b v5-04-72-AN v5-04-72-AN    If you want to checkout more than one branch/tag in parallel please follow the instructions from [Section 5](http://aliceinfo.cern.ch/Offline/node/2912/#ParallelGit). |
| Checkout a branch  svn checkout [http://svn.cern.ch/guest/AliRoot/branches/TPCdev](http://svn.cern.ch/guest/AliRoot/%28trunk/branches/tags%29) | Clone is done only once, checkout of the branches/tags is done inside the same working directory, one after the other  git clonehttp://git.cern.ch/pub/AliRoot  cd AliRoot  git checkout TPCdev  git checkout AN-201306-15  If you want to checkout more than one branch/tag in parallel please follow the instructions from [Section 5](http://aliceinfo.cern.ch/Offline/node/2912/#ParallelGit). |
| List tags and branches  svn list [http://svn.cern.ch/guest/AliRoot](http://svn.cern.ch/guest/AliRoot/%28trunk/branches/tags%29) | List tags and branches on local repository  git tag/branch  List tags and branches on remote repository  git remote show *<remote>*  <remote> is usually *origin* |
| svn blame *file* | git blame *file* |
| svn info | There is no complete git equivalent of svn info command.  1. git log - will show on the first line the current branch in the working directory  2. [git-info](https://github.com/gitbits/git-info/) contribution script - download all 3 scrips: git-info, git-pager and git-editor, make them executable and add them to $PATH |

**4.2 Update commands**

|  |  |
| --- | --- |
| SVN | Git |
| Status of the working directory  svn status | In Git case we find 2 use cases  1. Status of the local repository  git status  2. Status of the local repository compared to a remote  git fetch  git log  git merge (if you want to merge to local repository) |
| Updating working directory  svn update | There are 2 ways to integrate changes: [merge](http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging) or [rebase](http://git-scm.com/book/en/Git-Branching-Rebasing)  1. Merge - the changes are merged directly creating a new commit with a new history.  git pull or  git fetch && git merge  (the last one will allow to see all files that were touched on the central repository)  2. Rebase - A rebase rolls back your history to the point where you forked from the remote, applies the remote changes, and then reapplies your local changes commit by commit. This rewrites history and should therefore only be used on unpublished branches.  git fetch && git rebase or  git pull --rebase (preferred method)  To better understand the difference between the usage of one or the other:  <http://gitolite.com/tips/git-pull--rebase.html>  <http://stackoverflow.com/questions/804115/git-rebase-vs-git-merge>  <http://stackoverflow.com/questions/3357122/git-pull-vs-git-fetch-git-rebase> |
| Switching between branches  svn checkout [http://svn.cern.ch/guest/AliRoot/branches/AN-2013-05-16/](http://svn.cern.ch/guest/AliRoot/%28trunk/branches/tags%29)  svn switch AN-2013-05-24 | git clonehttp://git.cern.ch/pub/AliRoot  cd AliRoot  git checkout AN-2013-05-16  git checkout AN-2013-05-24 |
| Committing your changes  svn commit | !Before commit always *pull/fetch* the changes from central repository!  1. Commit to local repository  Git allows to "add" changes to a prestage area. Therefore you always need to add the changes to the index before committing them.  git add *ModifiedFile*  git commit  or  git commit -a  2. Push the changes to remote repository  git push  The command above will push all local repository to central repository.  If only the current branch has to be pushed then use  git push --set-upstream origin current\_branch\_name  Another option is through  git config  push.default *upstream*  *push\_value =* [*nothing/matching/upstream/simple/current*](http://nothing/matching/upstream/simple/current) |
|  | git add *file* |
| svn rm *file* | git rm *file* |
| svn mv *file* | git mv *file* |
| svn revert *rev* | git revert *rev*  If a commit has been made somewhere in the project's history, and you later decide that the commit is wrong and should not have been done, then *git revert* is the tool for the job. It will undo the changes introduced by the bad commit, recording the "undo" in the history.  git checkout *rev*  If you have modified a file in your working tree, but haven't committed the change, then you can use *git checkout* to checkout a fresh-from-repository copy of the file.  git reset --hard *rev*  If you have made a commit, but haven't shared it with anyone else and you decide you don't want it, then you can use git resetto rewrite the history so that it looks as though you never made that commit. |

4.3 Advanced commands

|  |  |
| --- | --- |
| SVN | Git |
| Branching  svn copy [http://svn.cern.ch/guest/AliRoot/branches/TPCdev http://svn.cern.ch/guest/AliRoot/branches/TPCdev\_new](http://svn.cern.ch/guest/AliRoot/%28trunk/branches/tags%29)  svn switch [http://svn.cern.ch/guest/AliRoot/branches/TPCdev\_new](http://svn.cern.ch/guest/AliRoot/%28trunk/branches/tags%29) | git branch TPCdev\_new TPCdev  git checkout TPCdev\_new  Make changes and if needed push the branch to central repository  git push |
| Merging  svn merge [http://svn.cern.ch/guest/AliRoot/branches/AN-2013-05-16/](http://svn.cern.ch/guest/AliRoot/%28trunk/branches/tags%29) | git merge AN-2013-05-16 |

* SVN is the third implementation of a [**revision** control](http://stackoverflow.com/questions/1056912/source-control-vs-revision-control/1056947#1056947): [RCS, then CVS and finally SVN](http://www.ibr.cs.tu-bs.de/kb/revision-control.html) manage directories of versioned data. SVN offers VCS features (labeling and merging), but its tag is just a directory copy (like a branch, except you are not "supposed" to touch anything in a tag directory), and its merge is still complicated, currently based on meta-data added to remember what has already been merged.
* Git is a **file content management** (a tool made to merge files), **evolved into a true Version Control System**, based on a DAG ([Directed Acyclic Graph](http://en.wikipedia.org/wiki/Directed_acyclic_graph)) of commits, where branches are part of the history of datas (and not a data itself), and where tags are a true meta-data.

To say they are not "fundamentally" different because you can achieve the same thing, resolve the same problem, is... plain false on so many levels.

* if you have many complex merges, doing them with SVN will be longer and more error prone. if you have to create many branches, you will need to manage them and merge them, again much more easily with Git than with SVN, especially if a high number of files are involved (the speed then becomes important)
* if you have partial merges for a work in progress, you will take advantage of the Git staging area (index) to commit only what you need, stash the rest, and move on on another branch.
* if you need offline development... well with Git you are always "online", with your own local repository, whatever the workflow you want to follow with other repositories.
* One is an extended revision tool, the other a true version control system.
* One is suited small to medium monolithic project with simple merge workflow and (not too much) parallel versions. SVN is enough for that purpose, and you may not need all the Git features.
* The other allows for medium to large projects based on multiple components ([one repo per component](http://stackoverflow.com/questions/984707/what-are-the-git-limits/984973#984973)), with large number of files to merges between multiple branches in a complex merge workflow, parallel versions in branches, retrofit merges, and so on. You could do it with SVN, but you are much better off with Git.  
  SVN simply can not manage any project of any size with any merge workflow. Git can.

Again, **their nature is fundamentally different** (which then leads to different implementation but that is not the point).  
One see revision control as directories and files, the other only see the content of the file (so much so that empty directories won't even register in Git!).

An SVN workflow looks like this:

The trunk directory represents the latest stable release of a project.  
Active feature work is developed within subdirectories under branches.  
When a feature is finished, the feature directory is merged into trunk and removed.  
Git projects are also stored within a single directory. However, Git obscures the details of its references by storing them in a special .git directory.

A Git repository stores the full history of all of its branches and tags within the .git directory.  
The latest stable release is contained within the master branch.  
Active feature work is developed in separate branches.  
When a feature is finished, the feature branch is merged into master and deleted.

Points:

SVN stores in server where as git stores in local machine and have a reference in server

when server is down..we cannot access svn and we are stucked. but git can work offline and when it comes online it updates the server

The key difference is that it is decentralized. Imagine you are a developer on the road, you develop on your laptop and you want to have source control so that you can go back 3 hours.

With Subversion, you have a Problem: The SVN Repository may be in a location you can’t reach (in your company, and you don’t have internet at the moment), you cannot commit. If you want to make a copy of your code, you have to literally copy/paste it.

With Git, you do not have this problem. Your local copy is a repository, and you can commit to it and get all benefits of source control. When you regain connectivity to the main repository, you can commit against it.

1. **GIT is distributed, SVN is not:**

This is by far the **\*core\*** difference between GIT and other non-distributed version control systems like SVN, CVS etc. If you can catch this concept well, then you have crossed half the bridge. To add a disclaimer, GIT is not the first or only distributed VCS(version control system) currently available. There are other tools like [Bitkeeper](http://www.bitkeeper.com/), [Mercurial](http://mercurial.selenic.com/) etc. which also work on distributed mode. But, GIT does it better and comes with much more powerful features.

GIT like SVN do have centralized repository or server. But, GIT is more intended to be used in distributed mode which means, every developers checking out code from central repository/server will have their own cloned repository installed on their machine. Let’s say if you are stuck somewhere where you don’t have network connectivity, like inside the flight, basement, elevator etc. :), you will still be able to commit files, look at revision history, create branches etc. This may sound trivial for lot of people but, it is a big deal when you often bump into no-network scenario.  
And also, the distributed mode of operation is a biggest blessing for open-source software development community. Instead of creating patches & sending it thro emails, you can create a branch & send a pull request to the project team. It will help the code stay streamlined without getting lost in transport. [GitHub.com](http://www.github.com) is an awesome working example of that.

There were some rumors that the future version of subversion will be working on distributed mode. But, it’s still an unknown at this point.

1. **GIT stores content as metadata, SVN stores just files:**

Every source control systems stores the metadata of files in hidden folders like .svn, .cvs etc., whereas GIT stores entire content inside the .git folder. If you compare the size of .git folder with .svn, you will notice a big difference. So, the .git folder is the cloned repository in your machine, it has everything that the central repository has like tags, branches, version histories etc.

1. **GIT branches are not the same as SVN branches:**

Branches in SVN are nothing but just another folder in the repository. If you need to know if you had merged a branch, you need to explicitly run commands like [*svn propget svn:mergeinfo*](http://jan.baresovi.cz/dr/en/subversion-mergeinfo) to verify if it was merged or not. [Thanks Ben](http://boxysystems.com/index.php/5-fundamental-differences-between-git-svn/#comment-791) for pointing this feature :).  
So, the chance of adding up orphan branches is pretty big.

Whereas, working with GIT branches is much more easier & fun. You can quickly switch between branches from the same working directory. It helps finding un-merged branches and also help merging files fairly easily & quickly.

1. **GIT does not have a global revision no. like SVN do:**

This is one of the biggest feature I miss in GIT from SVN so far. As you may know already SVN’s revision no. is a snapshot of source code at any given time. I consider that as a biggest breakthrough moving from CVS to SVN.  
Since, GIT & SVN are conceptually different, I don’t know how you can mirror that feature in GIT. If anyone know of any tricks that can do this, please feel free to share it in the comments.  
*Update: As some of the* [*readers*](http://boxysystems.com/index.php/5-fundamental-differences-between-git-svn/#comment-718) *pointed out, you can use GIT’s SHA-1 hash key to uniquely identify the code snapshot. It may not exactly replace SVN’s easily readable numeric revision no. but, it kind of serves the same purpose.*

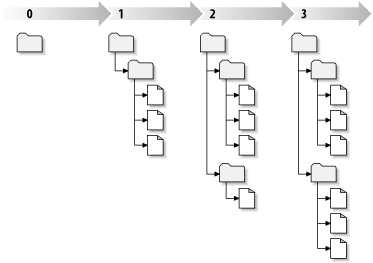
1. **GIT’s content integrity is better than SVN’s:**

GIT contents are cryptographically hashed using [SHA-1](http://en.wikipedia.org/wiki/SHA-1) hash algorithm. This will ensure the robustness of code contents by making it less prone to repository corruption due to disk failures, network issues etc. Here is an interesting discussion on GIT’s content integrity – <http://stackoverflow.com/questions/964331/git-file-integrity>

The SHA-1 hash of a commit \*is\* the revision. You can check out such a commit using e.g.  
git checkout 324dfeb and you can be sure that \*everything\* is at the state of that commit.

Please be aware also that you don’t need to use the full SHA-1 hash, only enough so that it’s unique. Normally, 5-7 chars are absolutely enough.

We don’t need to version all the ‘commits’. Just the ‘push’s. It should be technically possible to add a revision on each \*push\*, which is what most people will care about in a centrally managed repo. In fact, something like this can be done even today with a ‘post-push’ hook that adds an incremental tag on each push.



SVN Revision numbers

* It's incredibly fast. No other SCM that I have used has been able to keep up with it, and I've used a lot, including Subversion, Perforce, darcs, BitKeeper, ClearCase and CVS.
* It's fully distributed. The repository owner can't dictate how I work. I can create branches and commit changes while disconnected on my laptop, then later synchronize that with any number of other repositories.
* Synchronization can occur over many media. An SSH channel, over HTTP via WebDAV, by FTP, or by sending emails holding patches to be applied by the recipient of the message. A central repository isn't necessary, but can be used.
* Branches are even cheaper than they are in Subversion. Creating a branch is as simple as writing a 41 byte file to disk. Deleting a branch is as simple as deleting that file.
* Unlike Subversion branches carry along their complete history. without having to perform a strange copy and walk through the copy. When using Subversion I always found it awkward to look at the history of a file on branch that occurred before the branch was created. from #git: spearce: I don't understand one thing about SVN in the page. I made a branch i SVN and browsing the history showed the whole history a file in the branch
* Branch merging is simpler and more automatic in Git. In Subversion you need to remember what was the last revision you merged from so you can generate the correct merge command. Git does this automatically, and always does it right. Which means there's less chance of making a mistake when merging two branches together.
* Branch merges are recorded as part of the proper history of the repository. If I merge two branches together, or if I merge a branch back into the trunk it came from, that merge operation is recorded as part of the repostory history as having been performed by me, and when. It's hard to dispute who performed the merge when it's right there in the log.
* Creating a repository is a trivial operation: mkdir foo; cd foo; git init That's it. Which means I create a Git repository for everything these days. I tend to use one repository per class. Most of those repositories are under 1 MB in disk as they only store lecture notes, homework assignments, and my LaTeX answers.
* The repository's internal file formats are incredible simple. This means repair is very easy to do, but even better because it's so simple its very hard to get corrupted. I don't think anyone has ever had a Git repository get corrupted. I've seen Subversion with fsfs corrupt itself. And I've seen Berkley DB corrupt itself too many times to trust my code to the bdb backend of Subversion.
* Git's file format is very good at compressing data, despite it's a very simple format. The Mozilla project's CVS repository is about 3 GB; it's about 12 GB in Subversion's fsfs format. In Git it's around 300 MB.

There is no need for fancy metadata, rename tracking and so forth.  
**The only thing you need to store is the state of the tree before and after each change.**

What files were renamed? Which ones were copied? Which ones were deleted? What lines were added? Which ones were removed? Which lines had changes made inside them? Which slabs of text were copied from one file to another?  
You shouldn't have to care about any of these questions and you certainly shouldn't have to keep special tracking data in order to help you answer them: **all the changes to the tree (additions, deletes, renames, edits etc) are implicitly encoded in the delta between the two states of the tree**; you just ***track*** what is the ***content***.

**Absolutely everything can (and should) be inferred**.

Git breaks the mould because it thinks about content, not files.  
It doesn't track renames, it tracks content. And it does so at a whole-tree level.  
This is a radical departure from most version control systems.  
It doesn't bother trying to store per-file histories; it instead stores the history at the tree level.  
When you perform a diff you are comparing two trees, not two files.

The other fundamentally smart design decision is how Git does merges.  
**The merging algorithms are smart but they don't try to be too smart. Unambiguous decisions are made automatically, but when there's doubt it's up to the user to decide.**  
This is the way it should be. You don't want a machine making those decisions for you. You never will want it.  
That's the fundamental insight in the Git approach to merging: while every other version control system is trying to get smarter, Git is happily self-described as the "stupid content manager", and it's better for it.

Answers above are all correct, but I think they miss the centerpoint of git's easy merges for me. An SVN merge requires you to keep track and remember what's been merged and that's a huge PITA. From their docs:

svn merge -r 23:30 file:///tmp/repos/trunk/vendors

Now that's not killer, but if you forget whether it's 23-30 inclusive or 23-30 exclusive, or whether you've already merged some of those commits, you're hosed and you've got to go figure out the answers to avoid repeating or missing commits. God help you if you branch a branch.

With git it's just git merge and all this happens seamlessly, even if you've cherry-picked a couple commits or done any number of fantastical git-land things.

Git just makes it more difficult to screw up everyone else's repository with a bad merge.

The only real benefit is that Git is much, much faster at merging because everything is done locally and it's written in C.