Git Virtual File System Architecture

10/01/2017 • 15 minutes to read

In this article

By: Saeed Noursalehi

In previous articles, we have discussed the challenges of working in a Git repo with millions of files and some of our previous attempts to solve those issues. After trying out various simpler solutions, we realized that the only way to work in a repo at the scale of the Windows codebase and make it as fast as a normal sized repo is to virtualize the file system. This is why we built the Git Virtual File System (GVFS) to address the challenges of working in such a large repo.

In this article, we'll talk about the architecture of GVFS and how it allows Git to run more efficiently on the largest Git repositories. GVFS was designed to solve the following issues:

- Speed up Git commands by focusing on the set of files that a developer needs to use, not the total set of files that exist in the repo.
- Speed up network downloads by only downloading contents – current and historical – that the developer actually needs to use.

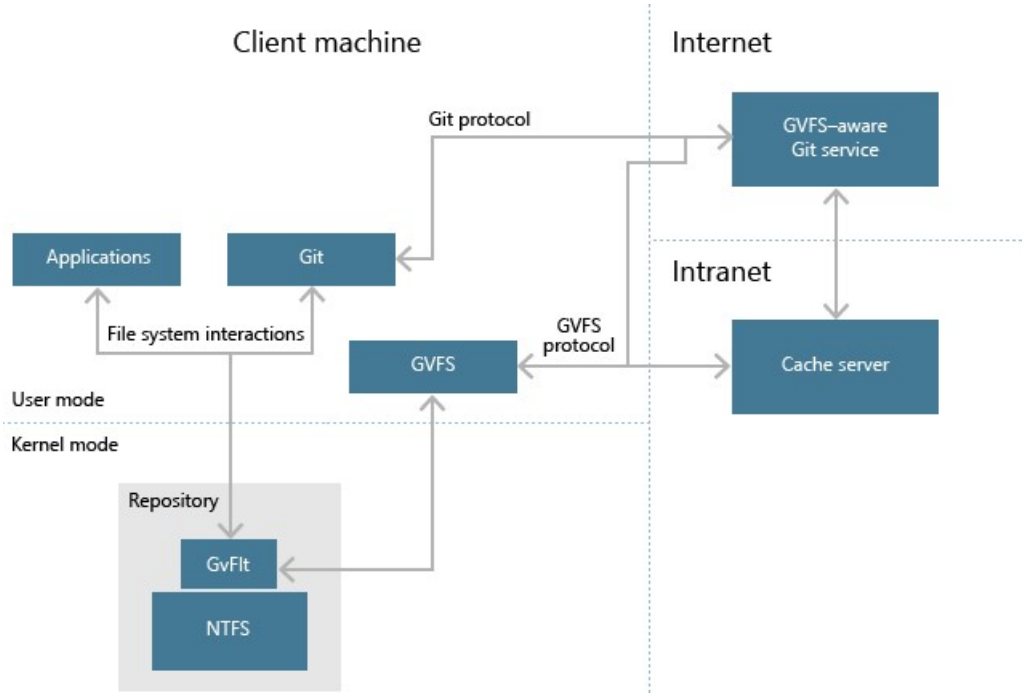
with the following constraints:

- Maintain full file system functionality, so that tools like IDEs and compilers can continue to work without being aware of GVFS.
- Maintain file system performance that matches the local file system for already-hydrated contents.

Architecture overview

GVFS is made up of the following pieces, and we'll talk about each one in detail:

- GvFlt: a file system filter driver, responsible for projecting a user mode application's view of a file system down into NTFS. (Note: GvFlt is in the process of being renamed to ProjFS, short for Windows Projected File System)
- GVFS: A user mode process that knows how to work with GvFlt to project the correct view of the file system based on the current state of the Git repo, how to respond to user edits to update Git's data structures appropriately, and how to download contents as needed to satisfy all of the above
- GVFS protocol: An HTTP protocol that allows the GVFS client to communicate with a GVFS-aware service to download Git objects on demand
- Git: Git itself needed some changes to work more efficiently on top of a virtualized file system, without affecting any behaviors that users will notice
- Cache server: A caching proxy that brings the contents of the repo closer to the end users, to reduce the latency that is inherent in fetching contents on demand
- Applications: Other applications sitting on top of GVFS need to think they are interacting with a normal Git repo and file system



GvFlt

Some optional background reading for those who are curious: GvFlt is a file system filter driver that makes heavy use of reparse points. We won't get into the technical details of how filter drivers work, but the simplified description is that filter drivers allow us to extend and modify the behavior of the NTFS file system. On Windows, every file system driver can have a stack of one or more filter drivers sitting on top of it. These filter drivers can intercept messages intended for the file system, and they can modify, reject, or ignore those messages, or their responses, as they see fit.

At a high level, what GvFlt does is intercept messages for enumerating directories and opening files. It then sends a message to a user mode application such as GVFS (known as the provider), so that the provider can specify what the contents of the directory or file should be. Let's go over some of the details of how this is accomplished.

Virtual file system objects

GvFlt introduces some new states for directories and files. A directory can be in one of these states:

- Virtual. The directory does not actually exist on disk, but it appears as if it does because GvFlt injects it into the enumeration result.
- Placeholder. A placeholder directory is an NTFS directory with a reparse point, and any of its children can exist on disk, while others can be virtual items enumerated by the provider.
- Full. A full directory is a normal NTFS directory.

A file can be in the following states:

- Virtual. The file does not actually exist on disk, but it appears as if it does because GvFlt injects it into the enumeration result.
- Placeholder. A placeholder file is an NTFS file with a reparse point, and no file contents. The file object and its metadata exist on disk, but its contents do not.
- Hydrated placeholder. A hydrated placeholder is also an NTFS file with a reparse point, but it does have its contents.
- Full. A full file is a normal NTFS file, with no reparse point. This file on disk now contains changes made

by the user.
- Tombstone. A tombstone file records that a file with the same name was deleted. This prevents a deleted file from being immediately resurrected because it is being projected by the provider. It also allows providers to project a readonly view of their file system, without having to worry about what modifications the user has made.

And here is how a file transitions through those states:

- Initial state: virtual. When a placeholder directory is enumerated, its children initially exist only in the enumeration response, with no representation on disk.
- Virtual -> placeholder. The first time a read handle is opened to a file, GvFlt creates a placeholder file on disk. This handle can be used to read the metadata of the file.
- Placeholder -> hydrated placeholder. If a read handle is used to read the contents of a placeholder file, GvFlt asks the provider for those contents and writes them into the file.
- Any sort of placeholder -> full file. If the user writes to the file contents, it gets converted into a normal NTFS file.
- Any state -> tombstone. If a file is deleted, GvFlt records a tombstone in its place.

Virtual projection

Now let's go over how all those states enable a provider application to create a virtual file system.

A provider application starts things off by creating a directory and informing GvFlt to treat the directory as a virtualization root. The first time a user enumerates the root directory, the following happens:

- GvFlt intercepts the enumeration request. Because the virtualization root is always a partial directory, GvFlt will send an enumeration request up to the provider to ask it what the contents of the directory should be.
- The provider receives the enumeration request from GvFlt. The provider decides what the contents of the directory should be, and it responds to GvFlt with a set of child files and directories.
- GvFlt combines the list of files/directories from the provider with what's physically on disk, and returns that full set in the enumeration response that goes back up the file system stack.
- When the provider supplies the contents of that root directory, it specifies if each item is a file or directory. For each child that is a file, when the user reads the file, GvFlt calls the provider to hydrate its contents. For each child that is a directory, the whole process recurses, and in this way the contents of the file system are populated on demand as they are accessed.

File system events

GvFlt also provides a set of notifications of file system events, so that the provider can respond to, and sometimes prevent, changes in the file system. For example, the provider might need to know if a file was modified, or it might need to block deletes of certain files.

GVFS

At a high level, GVFS has a few primary responsibilities:

- Respond to GvFlt's requests to enumerate directories and hydrate files
- Understand the state of the Git repo so that it can respond to GvFlt with the correct set of files and directories to match the behavior of a non-virtualized Git repo
- Communicate with the server over the GVFS protocol to get file contents when needed
- Ensure that all Git commands produce correct results and operate efficiently

Disk layout

When GVFS first creates a new clone, it creates the following layout on disk:

Copy

```
|
|__ .gvfs
|   |__
|
|__ src  (this is the virtualization root)
    |
    |__ .git
    |   |__
    |
    |__
```

Virtual projection

Prerequisite reading: to really follow along here, you should be familiar with Git's commits, trees, blobs, index, and basic operation of commands like status, checkout, merge, rebase. See this description of Git's internals for more details.

Now let's go over how GVFS uses GvFlt to create a virtual Git repo. When cloning, GVFS sets up the `src` folder as the virtualization root so that it can project the contents of the working directory as needed. The question is, what should those contents be?

In our initial designs, GVFS would look at the current HEAD commit, and parse the tree objects referenced by that commit to know what directories and files ought to exist in the working directory. This makes sense conceptually. When you run a command like `git checkout`, that's exactly what Git does; it takes a commit, walks through its trees and blobs, and lays down files and directories accordingly. So what GVFS used to do is monitor the HEAD for changes, and whenever GvFlt asked for the contents of a directory or file, GVFS would use the current HEAD commit/trees to answer the question. However, this turns out to be wrong in some cases (as an exercise for the reader, think through Git's behavior during a `git merge` conflict, and the fact that Git pauses the merge after it has written some files to disk, but before it has updated the HEAD).

Instead, we realized that there is another Git data structure that does match what the file system ought to look like: the Git index. Git always updates its index to match the files that it has written to disk. So the index is an accurate representation of the files in a Git repo that are being tracked by Git, and this became the source of truth for GVFS's projection of a repo.

So now GVFS monitors the index file for changes, and whenever it is modified, GVFS parses it and stores the directory/file structure in memory so that it can respond efficiently to enumeration requests from GvFlt.

Making Git commands run fast

So how does all of this help with making Git commands fast? Nothing we've discussed so far helps with that directly. Downloading files on demand is great, and helps with download times, but if Git decides to walk every file in the repo during a status or checkout command, all of that lazy downloading is moot. So let's cover some more background and then we can get into the crux of the solution.

As we discussed in a previous article, having a lot of files in the repo slows down any Git command that needs to scan all the files in the working directory. As you might imagine, scanning through 3.5 million files takes a long time, not to mention that it would nullify any benefits of lazy hydration if Git methodically opens and

reads every file in the working directory.

Luckily, Git had a couple of existing features that we were able to enhance with GVFS, so that we can preserve the benefits of lazy hydration and also convince Git to only open files that need to be opened.

The first existing feature is sparse-checkout, which allows you to tell Git about a set of files and directories in the repo, and have Git only write out those files when you do run a `git checkout`. It also tells Git to ignore all other files when you run a command like `git status`, since otherwise all of those other files would look like they were deleted. This feature allows you to work with a subset of the files in a repo (though normally Git still has to download *all* contents when you run `git clone` or `git fetch`).

The other existing feature is Git's highly configurable set of ignore files, via `.gitignore` files in the repo and exclude files outside of it. Ignore files tell Git about files that it does not need to report as new untracked files in the `git status` command.

Using these two features, GVFS is able to set things up so that Git does not need to do any scans of the working directory, except on files that the user has actively modified. GVFS knows exactly which files have been modified, thanks to the file system events delivered by GvFlt. Any files that have not been modified are guaranteed to be clean, so there is no need to have `git status` open them to see if they changed, or have `git checkout` overwrite them with new contents since GVFS can simply re-project them the next time they are accessed by the user.

So when GVFS creates a new clone, it sets up two special files in the `.git` folder:

- It creates a `.git\info\sparse-checkout` file that is initially (nearly) empty. (Git requires this file to have at least one line in it, so GVFS seeds it with `.gitattributes`)
- It creates a `.git\info\always_exclude` file that contains a single line that excludes `*`. In other words, it tells Git to ignore all untracked files, since initially there are none and there's no point in enumerating the whole working directory looking for them.

Because of the combination of those two files, if you clone and then run `git status`, Git performs almost no IO in the working directory. It opens just the one `.gitattributes` file and nothing else. But because GvFlt+GVFS create a virtual projection of the working directory, if the user goes browsing around, it looks like all the files are there. So you get the best of both – all the files in the working directory look like they exist, but Git thinks they don't and so it doesn't bother looking at them.

Making Git commands run correctly

Making Git commands run fast is all well and good, but if they don't give the right results, it's all for naught. And so one of GVFS's primary responsibilities is to monitor file system activity and update the sparse-checkout and always exclude files (and if you want to dig deeper, the skip-worktree bits in the index) so that Git commands that deal with working directory files always behave *as if* this were a normal, non-virtualized repo.

The key here is that GVFS knows which files have been modified. Any file that has never been written to by the user can be safely ignored by Git, since its contents can always be projected correctly by GVFS. But once a file is modified, GVFS hands the responsibility for that file over to Git from that point on, both for tracking changes and for updating it when needed.

Whenever a file is modified, GVFS adds the path of that file to the `sparse-checkout` and `always_exclude` files (beware: the details here can get quite involved). This causes all future Git commands to consider this file and report it in `git status` if it's been modified, update it in `git checkout` if it needs to change, etc.

Similarly, when a new file is created, it is added to `always_exclude` (actually it's added as a negation, since the first line in `always_exclude` says ignore everything, and now we're saying "except for this file"), so that new files can show up as untracked files in the output of `git status`.

We've skipped over a lot of the nitty gritty details here, but hopefully you can see the overall picture. Any files that have never been modified are "owned" by GVFS, and they get projected with the correct contents when they are accessed. Any files that have been written to by the user are "owned" by Git, and must be considered by any Git command that deals with file contents.

GVFS protocol

One of the issues we faced in building GVFS is that the native Git protocol does not provide any way to download individual objects. The smallest granularity that you can ask for is a commit, along with the trees and blobs represented by that commit. In GVFS, we want to download the commits and trees, but leave all the blobs behind on the server until they are needed.

And even though we don't want to download a blob until its associated file is actually read from, we do need to know the *size* of a blob before then, because the file system requires that an enumeration result contain both file names and their sizes. The size of a blob is not stored anywhere else in the Git metadata, so we needed some way to discover the size without downloading the blob itself.

So we built the GVFS protocol to allow the GVFS client to communicate to the server and ask it for this information when needed. The basics of this protocol are that it allows the GVFS client to ask for the sizes of a set of blobs, or ask for the contents of a set of Git objects.

The GVFS protocol also allows the GVFS client to download "prefetch packs" of commits and trees, since these objects are very small and much more efficient to download in bulk rather than one at a time. In a future article, we'll also discuss how we built a caching proxy for the GVFS protocol to reduce the latency of downloading files on demand.

Final note on performance

To state the obvious, developers care deeply about performance. We have succeeded in designing GVFS such that Git commands are no longer linear on the number of files in the repo, but we're still a ways off from making a giant repo feel like a tiny one.

For example, the `git status` command now takes 2-5 seconds in the Windows repo, as opposed to several minutes without GVFS, and that's a huge win. But in any other repo, developers are used to seeing status run nearly instantaneously, and so we continue to work on ways to make status (and all other commands) faster.

Stay tuned for future articles on our caching solution, more details about speeding up particular Git commands, and what we plan to do for GVFS on other operating systems.

Saeed Noursalehi is a Principal Program Manager on the Azure DevOps team at Microsoft, and works on making Git scale for the largest teams in Microsoft