Technical Scale Challenges with Git

10/07/2018 • 8 minutes to read

In this article

Too many pushes
Too many branches
Too much history
Too many files
Next article in the series

By: Saeed Noursalehi

It's become "common knowledge" that Git struggles with larger repos, but what does that mean, exactly? Let's break it down. It turns out there are there a few different ways that a repo can be "too big". These factors are mostly orthogonal to each other, meaning that it's possible for a repo to grow in one dimension but not the rest. However they also often feed into one another, for example the more people there are writing code, the more files you'll tend to have too. Let's take a look at each dimension:

Too many pushes

One common workflow when working in a Git repo is that the master branch contains the latest and greatest code, and people create topic branches or feature branches off of master, make some changes, validate those changes, and then merge them back into master. Other workflows are possible as well, but in the end, those changes have to be in a shared branch in order to be useful, and therein lies the challenge. To get your changes into master, you need to be able to push to master. And the way pushing works in Git is that you have to first pull down the most recent commit from the server's master branch, apply your changes on top of that, and then push your new commit to the server. However, you can only complete that push if your commit is still based on the most recent commit on the server (to be precise, your push must result in a fast-forward merge of the server's master branch). In other words, if someone else pushed a change in between when you fetched and when you pushed, your push will fail.

For a small team, this is not a major issue. The odds of a collision in that time window are fairly small, and when the collisions do happen, it's fairly quick to fetch, redo the merge or rebase, and try to push again.

On a large team, this issue brings your productivity to a standstill. Imagine you have 400 people all trying to push to master, and imagine that it's the last day before a big milestone and everyone has lots of changes to get in. It is incredibly frustrating to get stuck in a fetch-merge-push loop, and feel like you're racing all of your teammates to get your changes in. It is also very detrimental to code quality, because frustrated developers who are just trying to finish that last change so they can go home are likely to skip steps in their validation because that's the only way to win the race.

As you add more people to a repo, the odds of a push collision increase quadratically with the number of people. So while this is a minor issue for the GVFS repo, it's a major nuisance for the Azure DevOps repo, and it would be a complete show stopper for the Windows repo.

See The Race to Push to see how we use pull requests and merge queues to solve this issue by handling that race for you.

Too many branches

Branches are cheap to create and easy to use, and people tend to create lots of them. Typically, you'll have a few branches that everyone cares about, like master and release branches, and then lots of branches that most people don't care about, like topic branches. It is very common to have 5, 10, even 15 topic branches per person – some of those are active, some of them are parked, and some are forgotten, but by default everyone has to deal with all of these branches, regardless of how useful they are (or aren't).

There are several reasons that having too many branches can be painful:

- When listing out the branches, either locally or in the web, you have to wade through all of the branches to find the few you care about.
- When fetching, the Git client has to communicate with the server to figure out what new branches and commits to send down. The protocol requires the client and server to negotiate by comparing the set of branches that each of them has, finding the latest commit that each of them knows about for each branch, and then computing what new commits to send from the server to the client. These negotiations and computations grow linearly with the number of branches, so the more branches there are, the longer it takes not just to fetch, but also to figure out *what* to fetch.
- When pushing, there is a similar negotiation, which again gets more expensive the more branches there are.

For a team as big as Azure DevOps, this problem is a nuisance, but not insurmountable. We would routinely see that when people run "git fetch", git would sit there for up to 5 minutes negotiating what to send, before the server even started sending any objects to the client. It's definitely frustrating to sit there for that long "doing nothing", but it's not necessarily the end of the world. However for a team as big as Windows, with 100K branches or more, it would mean that fetching could take hours.

And technically speaking, the problem isn't actually "too many branches", it's actually "too many refs". We'll cover that in more detail as we talk about how the Azure DevOps Limited Refs feature solves this problem by only showing you the branches that you care about.

Too much history

One interesting aspect of working with a DVCS like Git is that when you clone, you get not only all of the latest sources, but also all of the history copied to your local machine. This enables some very nice things, such as the ability to work offline, or the ability to quickly browse history or create new branches from any point in history. However, these benefits also come at a cost: you have to download the entire history when you clone, and you have to download all new changes every time you fetch. For a small repo like GVFS, this is no issue at all. For a medium repo like Azure DevOps, it ranges from a minor nuisance to a major nuisance, depending on where in the world you're cloning from (e.g. cloning takes much longer for devs in India than for devs in Redmond). But for an XL repo like Windows, it takes long enough that it goes beyond nuisance and become a blocking issue. History can grow for a few different reasons.

- The more edits you make to files, the more the history grows. And the more files and people you have, the more edits you're going to have. But for the most part, Git does pretty well here, thanks the way it stores its backend objects. All objects in history are both deltified and compressed, resulting in a fairly compact storage, as long as the files are diffable and compressible.

- So the more the significant contribution to growth of history is edits to undiffable and uncompressible files, i.e. binaries. The more binaries there are in a repo, and the more churn those binaries go through, the faster the history grows.

- And while a few important branches can be protected with server-side policies to prevent people from pushing unwanted changes, it's very simple in Git to create a topic branch, commit whatever you want in there, and push it to the server. So one careless (or inexperienced) person can inflict pain on everyone else on the team.

There are some solutions already available to help with these issues:

- Packaging systems like NuGet, npm, etc. By pulling build outputs and 3rd party dependencies out of the repo and consuming them as packages, we can eliminate some of the problems associated with fast history growth due to binary files.

- Git LFS. For cases where binary files really need to be treated as sources, a tool like LFS can help because it stores the actual contents of the binaries in a separate content store, and the Git repo itself doesn't have to contain every historical copy of the file. This tool comes with some tradeoffs, but it does a nice job of keeping the size under control for certain scenarios.

However, there are cases where none of these solutions are ideal. We'll talk in more depth later on about the core issue of sources vs binaries, and the range of tools and approaches that various teams have used to deal with these issues.

The other solution that we've created to help with this situation is the Git Virtual File System (GVFS). Handling the size of history is actually just a secondary benefit of GVFS, but for repos that are large enough to require virtualization, the size of history becomes much less of an issue.

Too many files

The biggest challenge for the Windows repo is that even after you manage to clone the whole thing to your machine, local Git operations simply take too long, to the point where the repo would be completely unusable. Some examples:

- 'git status' takes up to 10 minutes, even where there are no dirty files in the repo
- 'git checkout' can take 3 hours
- 'git commit' takes 30 minutes

Why do these operations take so long? Two fundamental reasons:

- Git maintains an index file that contains a flat list of every file in the repo. This creates overhead for every single command that needs the index, since Git has to read and parse the index every time you run those commands.
- When you run an operation like status or checkout, Git essentially has to walk through every single file in your working directory, compare those files to the related entry in the index, and then compare the entries in the index to the corresponding objects in the Git tree. With over 3M files in the repo, the sheer cost of all that file IO means there's no way for the operation to complete quickly enough to feel responsive.

This is the problem that GVFS is uniquely positioned to solve, without forcing a rewrite of all of Git.

You can read more about the design history of GVFS, and coming soon we'll discuss the inner workings of GVFS in much more detail.

Next article in the series

- Limited Refs

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