EXHIBIT 7

January 7, 2014 OPEN SOURCE · INFRA · PERFORMANCE · OPTIMIZATION

Scaling Mercurial at Facebook

 Durham Goode  Siddharth Agarwal

With thousands of commits a week across hundreds of thousands of files, Facebook's main source repository is enormous--many times larger than even the Linux kernel, which checked in at 17 million lines of code and 44,000 files in 2013. Given our size and complexity—and Facebook's practice of shipping code twice a day--improving our source control is one way we help our engineers move fast.

Choosing a source control system

Two years ago, as we saw our repository continue to grow at a staggering rate, we sat down and extrapolated our growth forward a few years. Based on those projections, it appeared likely that our then-current technology, a Subversion server with a Git mirror, would become a productivity bottleneck very soon. We looked at the available options and found none that were both fast and easy to use at scale.

Our code base has grown organically and its internal dependencies are very complex. We could have spent a lot of time making it more modular in a way that would be friendly to a source control tool, but there are a number of benefits to using a single repository. Even at our current scale, we often make large changes throughout our code base, and having a single repository is useful for continuous modernization. Splitting it up would make large, atomic refactorings more difficult. On top of that, the idea that the scaling constraints of our source control system should dictate our code structure just doesn't sit well with us.

We realized that we'd have to solve this ourselves. But instead of building a new system from scratch, we decided to take an existing one and make it scale. Our engineers were comfortable with Git and we preferred to stay with a familiar tool, so we took a long, hard look at improving it to work at scale. After much deliberation, we concluded that Git's internals would be difficult to work with for an ambitious scaling project.

Instead, we chose to improve **Mercurial**. Mercurial is a distributed source control system similar to Git, with many equivalent features. Importantly, it's written mostly in clean, modular

Python (with some native code for hot paths), making it deeply extensible. Just as importantly, the Mercurial developer community is actively helping us address our scaling problems by reviewing our patches and keeping our scale in mind when designing new features.

When we first started working on Mercurial, we found that it was slower than Git in several notable areas. To narrow this performance gap, we've contributed over 500 patches to Mercurial over the last year and a half. These range from new graph **algorithms** to **rewrites of tight loops in native code**. These helped, but we also wanted to make more fundamental changes to address the problem of scale.

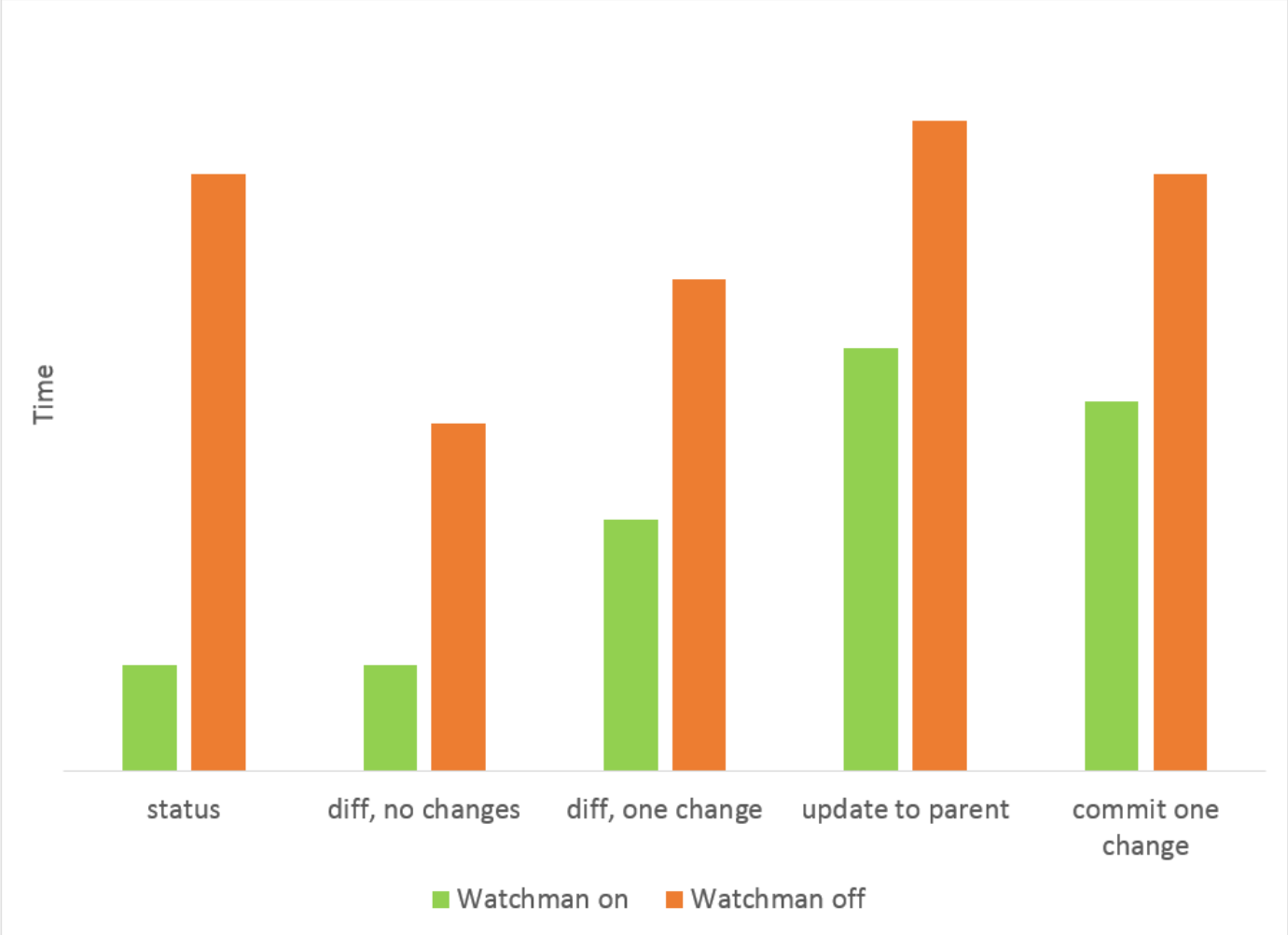
Speeding up file status operations

For a repository as large as ours, a major bottleneck is simply finding out what files have changed. Git examines every file and naturally becomes slower and slower as the number of files increases, while Perforce "cheats" by forcing users to tell it which files they are going to edit. The Git approach doesn't scale, and the Perforce approach isn't friendly.

We solved this by monitoring the file system for changes. This has been tried before, **even for Mercurial**, but making it work reliably is surprisingly challenging. We decided to query our build system's file monitor, **Watchman**, to see which files have changed. Mercurial's design made integrating with Watchman straightforward, but we expected Watchman to have bugs, so we developed a strategy to address them safely.

Through heavy stress testing and internal dogfooding, we identified and fixed many of the issues and race conditions that are common in file system monitoring. In particular, we ran a beta test on all our engineers' machines, comparing Watchman's answers for real user queries with the actual file system results and logging any differences. After a couple months of monitoring and fixing discrepancies in usage, we got the rate low enough that we were comfortable enabling Watchman by default for our engineers.

For our repository, enabling Watchman integration has made Mercurial's status command more than 5x faster than Git's status command. Other commands that look for changed files--like diff, update, and commit—also became faster.



Working with large histories

The rate of commits and the sheer size of our history also pose challenges. We have thousands of commits being made every day, and as the repository gets larger, it becomes increasingly painful to clone and pull all of it. Centralized source control systems like Subversion avoid this by only checking out a single commit, leaving all of the history on the server. This saves space on the client but leaves you unable to work if the server goes down. More recent distributed source control systems, like Git and Mercurial, copy all of the history to the client which takes more time and space, but allows you to browse and commit entirely locally. We wanted a happy medium between the speed and space of a centralized system and the robustness and flexibility of a distributed one.

Improving clone and pull

Normally when you run a pull, Mercurial figures out what has changed on the server since the last pull and downloads any new commit metadata and file contents. With tens of thousands of files changing every day, downloading all of this history to the client every day is slow. To solve this problem we created the remotefilelog extension for Mercurial. This

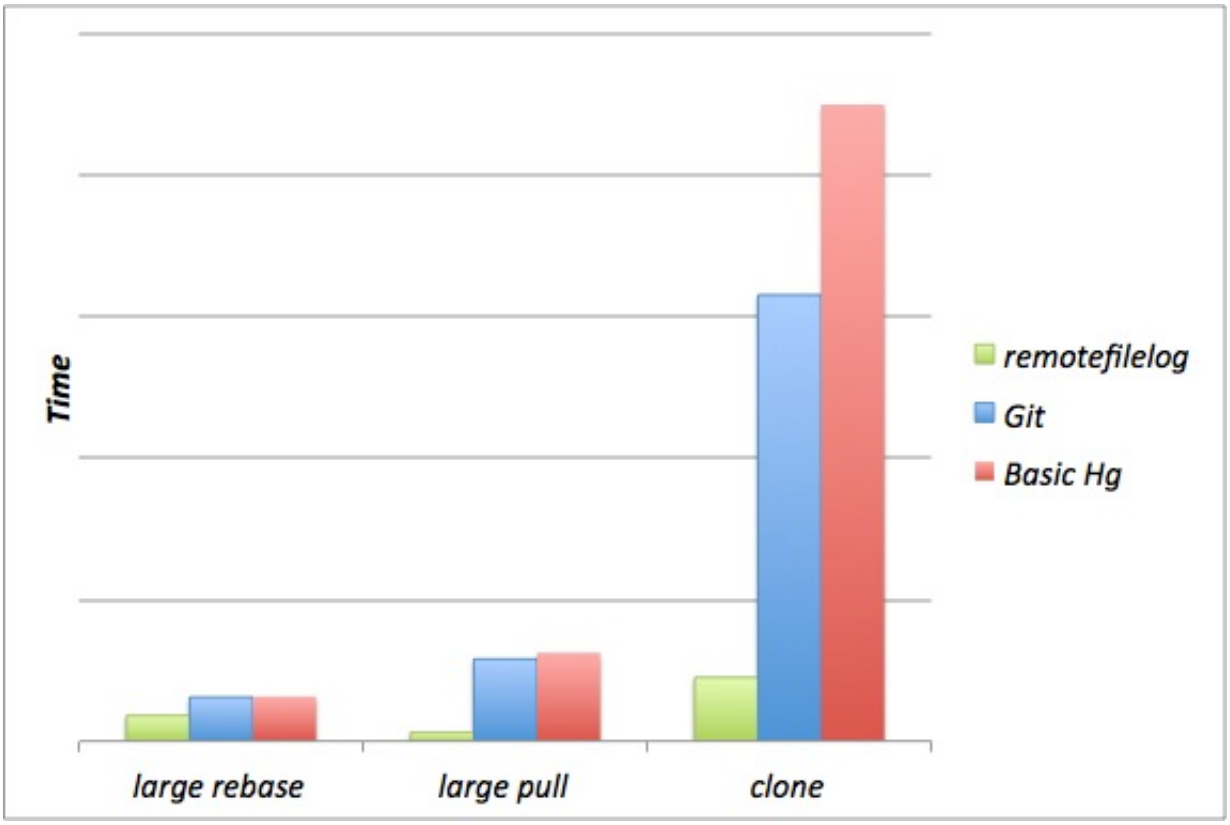
extension changes the clone and pull commands to download only the commit metadata, while omitting all file changes that account for the bulk of the download. When a user performs an operation that needs the contents of files (such as checkout), we download the file contents on demand using Facebook's existing memcache infrastructure. This allows clone and pull to be fast no matter how much history has changed, while only adding a slight overhead to checkout.

But what if the central Mercurial server goes down? A big benefit of distributed source control is the ability to work without interacting with the server. The remotefilelog extension intelligently caches the file revisions needed for your local commits so you can checkout, rebase, and commit to any of your existing bookmarks without needing to access the server. Since we still download all of the commit metadata, operations that don't require file contents (such as log) are completely local as well. Lastly, we use Facebook's memcache infrastructure as a caching layer in front of the central Mercurial server, so that even if the central repository goes down, memcache will continue to serve many of the file content requests.

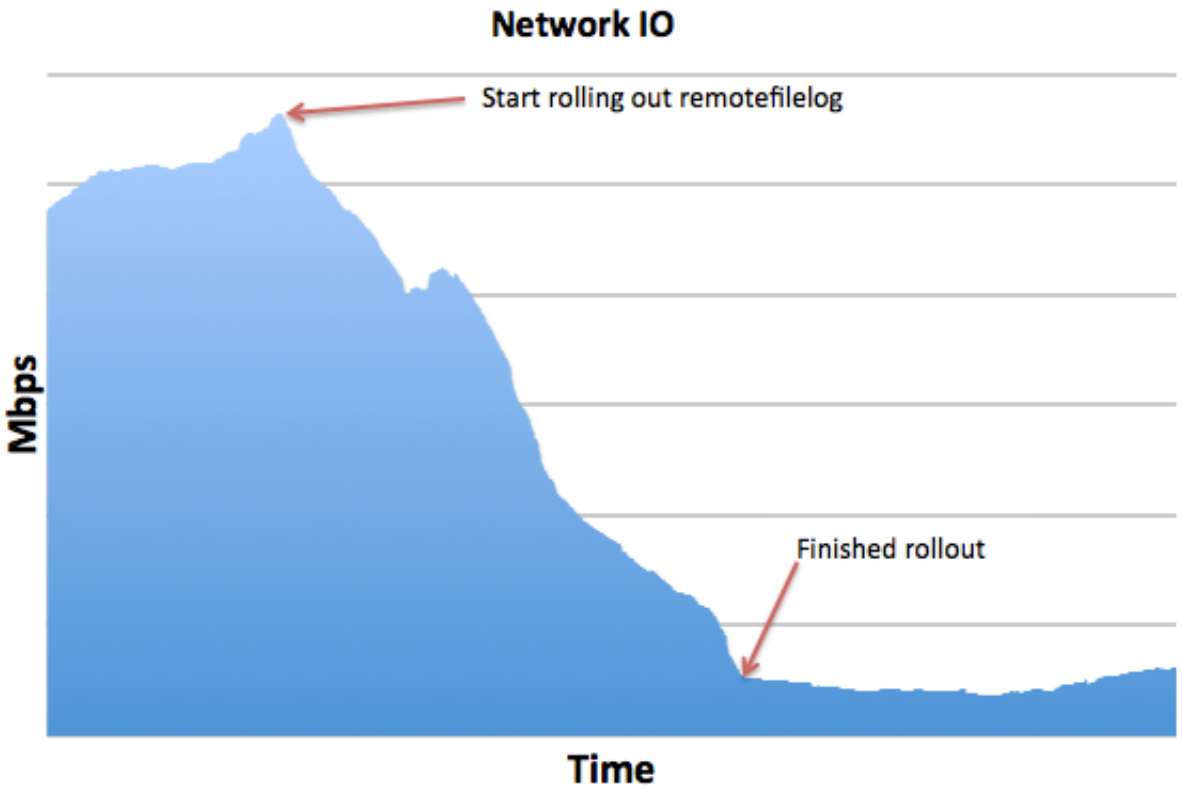
This type of setup is of course not for everyone—it's optimized for work environments that have a reliable Mercurial server and that are always connected to a fast, low-latency network. For work environments that don't have a fast, reliable Internet connection, this extension could result in Mercurial commands being slow and failing unexpectedly when the server is congested or unreachable.

Clone and pull performance gains

Enabling the remotefilelog extension for employees at Facebook has made Mercurial clones and pulls 10x faster, bringing them down from minutes to seconds. In addition, because of the way remotefilelog stores its local data on disk, large rebases are 2x faster. When compared with our previous Git infrastructure, the numbers remain impressive. Achieving these types of performance gains through extensions is one of the big reasons we chose Mercurial.



Finally, the remotefilelog extension allowed us to shift most of the request traffic to memcache, which reduced the Mercurial server's network load by more than 10x. This will make it easier for our Mercurial infrastructure to keep scaling to meet growing demand.



How it works

Mercurial has several nice abstractions that made this extension possible. The most notable is the filelog class. The filelog is a data structure for representing every revision of a particular file. Each version of a file is identified by a unique hash. Given a hash, the filelog can reconstruct the requested version of a file. The remotefilelog extension replaces the filelog with an alternative implementation that has the same interface. It accepts a hash, but instead of reconstructing the version of the file from local data, it fetches that version from either a local cache or the remote server. When we need to request a large number of files from the server, we do it in large batches to avoid the overhead of many requests.

Open Source

Together, the **hgwatchman** and **remotefilelog** extensions have improved source control performance for our developers, allowing them to spend more time getting stuff done instead of waiting for their tools. If you have a large deployment of a distributed revision control system, we encourage you to take a look at them. They've made a difference for our developers, and we hope they will prove valuable to yours, too.

More to Read

react-devtools

Recommended

Scaling memcached at Facebook

Scaling memcache at Facebook

2013: A Year of Open Source at Facebook

Speaking a whole new language: DConf 2013 at Facebook

Want to work with us?

Join the team, we're hiring! Here are some of our current open positions:

Software Engineer, Full Stack
Partner Engineering Manager, APAC
Software Engineer, Mobile

More Engineering Positions

Connect

Keep Updated

Stay up-to-date via RSS with the latest open source project releases from Facebook, news from our Engineering teams, and upcoming events.

Subscribe