Mozilla Version Control Tools

Release 0

October 05, 2018

Contents

Welcome to Mozilla's version control tools repository! Inside, you will find all kinds of code to support the development of code at Mozilla.

This repository contains the code that Mozilla uses in production to power hg.mozilla.org, Autoland, version control synchronization, and other sites and services.

The canonical repository is https://hg.mozilla.org/hgcustom/version-control-tools/.

CHAPTER 1

Goals

The primary goal of this repository and the code therein is to enable Mozillians to move faster and to be more productive, all while maximizing the quality of work being performed.

We help achieve that goal by providing robust tools that allow developers to spend more time using their brains instead of wrangling with tools, particularly in the area of version control and code management.

This repository aims to provide high-quality software that has a low barrier to change. We achieve this by having comprehensive, easy-to-write and run tests. Robust testing gives us the confidence that we can move fast without breaking things. Doing so allows us to serve the needs of Mozilla better and faster.

CHAPTER 2

Table of Contents

Developer Guide

Interested in contributing to Mozilla's version control tools? You've come to the right place!

Contributing

Find a bug? Interested in contributing a bug fix or enhancement? Read on!

Filing Bugs

Bugs against software in this repository can be filed in the Developer Services Bugzilla product. The component should be obvious. If you are unsure, use *General*.

Contacting Us

The people that maintain the code in this repository lurk in the following IRC channels on `irc.mozilla.org`:

#vcs Where everybody who maintains all the version control properties hangs out.

Just pop in a channel, ask a question, and wait for someone to answer. The channels can be quiet for hours at a time, so please stick around if nobody replies at first.

Submitting Patches for Review

We use https://phabricator.services.mozilla.com/ for conducting code review.

See https://moz-conduit.readthedocs.io/en/latest/phabricator-user.html for information on how to configure Phabricator at Mozilla. You will want to use the `rVCT` repository for review.

Before submitting patches for review, please *run the tests* and verify things still work. Please also read the following section on how to optimally create commits.

Commit Creation Guidelines

We prefer many, smaller and focused commits than fewer, larger commits. Please read Phabricator's Recommendations on Revision Control and apply the *One Idea Per Commit* practice to patches to this repository. Please also read their article on Writing Reviewable Code and tailor your commits appropriately.

It is recommended for commits to this repository to have the following commit message convention:

```
component: short description (bug xyz); r=reviewer

A sentence explaining the purpose of the patch. Another sentence
adding yet more detail.

Another paragraph adding yet more detail. We really like context to
exist in our patches rather than elsewhere.
```

The first line of the commit message begins with the component the patch is touching. Run `hg log` and see what others have used if you don't know what to put here. The short description should be pretty obvious.

If the patch is tracked in a bug, please enclose the bug in parenthesis at the end of the commit message. **This is different from the Firefox convention.** We do things differently because we want the beginning of the commit message to emphasize the thing that was changed. This improves discovery when filtering through commit messages, as it allows you to easily and cheaply find all commits that changed a specific component. Furthermore, we prefer to work with the mentality of code, not bugs, being first. We defer the bug to the end of the summary line to reflect that.

Bug and Review Requirements

We do not require that every commit have a bug association. If there isn't a bug on file, please don't waste time filing one just to write a patch.

We do not require that every commit be reviewed.

Please abide by the following rules before pushing without a bug or review:

- **A review is required** if you are modifying code that runs on a production service or we install or recommend installing on a user's machine (MozReview, Mercurial extensions, Mercurial hooks, etc).
- If you are adding new test code and you know what you are doing, you may **not** need a review. This exception is a little fuzzy around tests for production code (reviews are helpful to ensure the tests are accurate and proper).
- If you are adding or hacking on a miscellaneous tool that doesn't have test coverage or isn't widely used or relied upon, you may **not** need a review.
- You do **not** need a review to update documentation. If something is wrong, just fix it.
- When in doubt, ask someone on `#vcs` if you need a review before pushing.

Pushing Commits

When pushing commits to `ssh://hg.mozilla.org/hgcustom/version-control-tools`, it is important for you to set the `@` bookmark to the new tip.

Say you've created a new head or bookmark for your commit series. Assuming the working directory of your repository is on the commit you wish to make the new repository tip, here is how you should land your changes:

```
$ hg pull
pulling from default
requesting all changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
updating bookmark @

$ hg rebase -d @
```

```
$ hg bookmark @
moving bookmark '@' forward from abcdef012345

$ hg push -B @
pushing to default
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 4 files
exporting bookmark @
```

If you fail to update the remote `@` bookmark, nothing bad should happen. So don't worry too much if you forget to do it from time to time.

If you do forget, just perform a `hg push -B @` any time to update the remote bookmark. You can do this if you have no new changesets to push.

Creating and Maintaining a Development Environment

Requirements

To create a development and test environment, you'll need Linux or OS X host operating system. **Windows is currently not supported.**

You will need Python 2.7.

Many components use Docker. You'll need Docker to perform many tasks. Functionality requiring Docker should be skipped if Docker is not available.

Aside from the base requirements, the development and testing environment should be fully self-contained and won't pollute your system.

If you are on Windows or want to create a fully-isolated environment, the Vagrant configuration used by *Jenkins* provides a fully capable environment.

Ubuntu Requirements

On a fresh Ubuntu 16.04 install, the following packages need to be installed:

- build-essential
- git
- python-dev
- npm
- libcurl4-openssl-dev
- libffi-dev
- liblzma-dev
- libsasl2-dev
- libldap2-dev
- libssl-dev
- python3

- sqlite3
- zlib1g-dev
- mercurial (to clone version-control-tools)

Many of these dependencies are needed to compile binary Python extensions that are part of the virtualenv.

You can install these dependencies by running:

```
$ sudo apt-get install build-essential git python-dev npm \
  libcurl4-openssl-dev libffi-dev liblzma-dev \
  libsasl2-dev libldap2-dev libssl-dev python3 sqlite3 zlib1g-dev \
  mercurial
```

You will also need to install Docker for a number of test and dev environments to work. See the official Docker instructions for more.

Creating and Updating Your Environment

Development and testing requires the creation of a special environment containing all the prerequisites necessary to develop and test. This is accomplished by running the following command:

```
$ ./create-test-environment
```

Tip: You should periodically run `create-test-environment` to ensure everything is up to date. (Yes, the tools should do this automatically.)

Activating an Environment

Once you've executed `create-test-environment`, you'll need to *activate* it so your current shell has access to all its wonders:

```
$ source venv/bin/activate
```

boot2docker

If you are running OS X and have boot2docker installed to run Docker containers, you may want to increase the amount of memory available to the boot2docker VM.

Run the following to see how much memory is currently allocated to boot2docker:

```
$ boot2docker config | grep Memory
2048
```

The default is `2048` (megabytes). **We recommend at least 4096 MB.**

To adjust the amount of memory allocated to boot2docker, run the following:

```
$ VBoxManage modifyvm boot2docker-vm --memory 4096
```

Alternatively, if you haven't created a boot2docker VM yet, define the memory allocation when you create it:

```
$ boot2docker init --memory=4096
```

Testing

Methodology

Testing the code in this repository is taken very seriously. We want to facilitate confidence that any change will have the intended side-effects and won't regress behavior. We do this by providing a testing framework that is comprehensive and robust.

We currently support the following flavors of tests:

1. Python unit tests
2. Mercurial *t tests*
3. Mercurial .py tests

The test driver is responsible for identifying which flavor a particular file is.

Many tests interact with services running locally, commonly inside Docker containers. **Running actual services is encouraged over mocking.**

Running

Tests are executed by running the following in a built *environment*:

```
$ ./run-tests
```

To see help on options that control execution:

```
$ ./run-tests --help
```

Unknown script arguments will be proxied to Mercurial's `run-tests.py` testing harness.

Common tasks are described below.

Obtain code coverage results (makes tests run slower):

```
$ ./run-tests --cover
```

Use Docker images from last run (makes tests run faster):

```
$ ./run-tests --use-last-images
```

Test a single file:

```
$ ./run-tests path/to/test.t
```

Run all tests in a directory:

```
$ ./run-tests hgext/pushlog
```

Run a test in debug mode (see progress, interact with a debugger):

```
$ ./run-tests -d path/to/test.t
```

Run tests against all supported Mercurial versions:

```
$ ./run-tests --all-hg-versions
```

Run tests with a specific Mercurial installation:

```
$ ./run-tests --with-hg=/path/to/hg
```

Do not run Selenium tests:

```
$ ./run-tests --headless
```

Do not run tests that require Docker:

```
$ ./run-tests --no-docker
```

Run tests 1 at a time:

```
$ ./run-tests -j1
```

Authoring Tests

Test File Naming

Mercurial `.t` and `.py` tests will be automatically discovered from the following directories:

- `hgext/*/tests/`
- `hghooks/tests/`

Mercurial test filenames must be prefixed with `test-`. e.g. `test-foo.t`.

Python unit tests will be discovered from the following directories:

- `pylib/**`

Python unit test filenames must be prefixed with `test`. e.g. `test_foo.py`.

To write a new test, simply put the test file in one of the aforementioned directories and name it so that it will be discovered. If you run `run-tests path/test/test` and the specified filename wouldn't get discovered, an error will be raised saying so.

Choice of Test Flavor

Mercurial t tests are recommended for most tests.

Mercurial *t tests* are glorified shell scripts. Tests consist of a series of commands that will be invoked in a shell. However, they are much more than that. Expected output from commands is captured inline in the `.t` file. For example:

```
$ hg push
pushing to ssh://user@dummy/$TESTTMP/repos/test-repo
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
```

If the expected output differs from actual, the Mercurial test harness will print a diff of the changes.

.t tests are very useful for testing the behavior of command line programs.

Unless you are testing a headless Python module, you should probably be writing *t tests*.

Python APIs and Helper Scripts

Tests often want to instantiate services and interact with them. To facilitate this, there are various Python APIs and helper scripts.

The Python APIs are all available as part of the `vcttesting package`. There is typically a subpackage or module for each service you may want to interact with.

To facilitate testing from *t tests*, there are various command line tools for interacting with specific services. For example, the `bugzilla` tool allows you to start up and stop Bugzilla instances and perform common actions against them, such as create a bug.

These APIs and scripts exist only to support testing. Their APIs and arguments are not considered stable. They should not be relied on outside the context of the testing environment.

The CLI tools all use *mach* for command dispatching. Simply run `<tool> help` to see a list of what commands are available.

Jenkins Continuous Integration

This repository is continuously tested via Jenkins. You can find the canonical Jenkins job at https://ci.mozilla.org/job/version-control-tools/.

The Jenkins test environment is configured such that it can be executed by anyone, anywhere.

The `testing/jenkins` directory contains everything you need to reproduce the canonical Jenkins job.

In that directory are the following files:

Vagrantfile Defines the virtual machine used to run test automation.

run-main.py A script used to run the tests. This is what you'll configure your Jenkins job to execute to run the job.

run.sh Main script that runs inside the virtual machine to run test automation. This is invoked by `run-main.py`.

Configuring Jenkins

The Jenkins build only needs to consist of a single step: a shell script that executes:

```
testing/jenkins/run-main.py
```

For post-build actions, you have a number of options.

You can *Publish Cobertura Coverage Report* by using `**/coverage/coverage.xml` for the *Cobertura xml report pattern*.

You can *Publish JUnit test result reports* by using `coverage/results.xml` as the *Test Reports XML* value.

You can *Publish coverage.py HTML reports* by setting `coverage/html` as the *Report directory*.

You can *Publish HTML Reports* containing the generated Sphinx documentation by setting `sphinx-docs/html` as the *HTML directory to archive* and setting the *Index Page* to `index.html`.

Docker

Our development and test environment uses Docker extensively. The main goal of Docker is to facilitate developing and testing code in an environment that is as close to production as possible.

Image Management

We manage Docker images differently than most. We don't use Docker in production (yet). But, we also want our Docker images to match the server environment as closely as possible. If we were to use Dockerfiles to maintain the Docker environment and something *not Docker* to manage the servers, we would have duplication of effort and divergence between Docker and production and this would undermine the effectiveness of the Docker environment for mimicking production.

Instead of maintaining a Dockerfile for each image, we instead prefer to maintain Dockerfiles for base, bare bones images configured with Ansible. Then, we create and update Docker images by running an Ansible playbook on this base image.

Secure Base Images

Docker fails to take a secure approach to pulling images. For more, read Docker Insecurity. We take security seriously, so we've added secure image pulling into our Docker wrapper.

We introduce a special `FROM` syntax in Dockerfiles that initiates a secure pull of that base image. The syntax is as follows:

```
FROM secure:<repository>:<tag prefix>:<digest>:<URL>
```

e.g.:

```
FROM secure:mozsecure:ubuntu14042:sha256 e08475e91c8b342dce147346a11db16357619ed914aaad4d82be2e6ab743
```

When encountered, we will pull the image from the URL specified, verify its digest matches what is defined, then import the image, and finally associate it with the repository specified. The Dockerfile seen by Docker is dynamically rewritten to reference the securely, just-imported image.

Mercurial Customizations

This repository contains numerous customizations to the Mercurial version control tool.

Extensions

This repository contains a number of Mercurial extensions. Each is described in the sections below.

All extensions are located under the `hgext/` subdirectory.

bzexport

The bzexport extension provides commands for interacting with Bugzilla. It's known for its namesake `hg bzexport` command, which exports/uploads patches to Bugzilla. It also offers an `hg newbug` command to create new bugs from the command line.

This extension lives under `hgext/bzexport`.

bzpost

The bzpost extension will automatically update Bugzilla with comments containing the URL to pushed changesets after pushing changesets that reference bugs. The implementation is highly tailored towards the Firefox workflow.

firefoxtree

The firefoxtree extension makes working with the various Firefox repositories much more pleasant.

For more, read *it's documentation*.

mozext

mozext is a Swiss Army Knife for Firefox development. It provides a number of features:

- It defines aliases for known Firefox repositories. You can do `hg pull central`, etc.
- It provides a mechanism for tracking each repository via bookmarks, allowing you to more easily operate a unified repository.
- Changes to Python files are automatically checked for style.
- Pushlog data is synchronized to a local database.
- Bug data is extracted from commit messages and stored in a database.
- Many revision set and template functions are added.

If you are looking to turn Mercurial into a more powerful query tool or want to maintain a unified repository, *mozext* is very valuable.

This extension lives under `hgext/mozext`.

qbackout

The qbackout extension provides assitance to help perform code backouts the Mozilla way.

qimportbz

The qimportbz extension allows you to easily import patches from Bugzilla.

serverlog

The serverlog extension hacks up some Mercurial internals to record forensics that are useful for Mercurial server operators.

Hooks

The `hghooks` directory contains a number of Mercurial hooks used by Mozilla projects.

The content of this directory originally derived from its own repository. Changesets `e11fee681380` through `1f927bcba52c` contain the import of this repository.

This directory has its origins in the operation of the Mercurial server at Mozilla. It is an eventual goal to restructure the hooks to be usable on both client and server.

Available Hooks

changelog_correctness.py

Older versions of Mercurial had a bug where the set of modified files stored in the commit object were incomplete. Operations that relied on this cached set of changed files (hooks, some revset queries, log) could have inaccurate output if a buggy commit was present.

This hook looks for the presence of buggy metadata and rejects it.

commit-message.py

This hook attempts to enforce that commit messages are well-formed. It is targeted towards the Firefox commit message standard.

prevent_case_only_renames.py

This hook prevents file renames that only change the case of a file. e.g. renaming `foo` to `FOO` would be disallowed.

This hooks exists to prevent issues with case-insensitive filesystems.

prevent_string_changes.py

This hook is used to prevent changes to strings in string frozen release heads without the explicit approval from l10n-drivers.

prevent_webidl_changes.py

This hook prevents changes to WebIDL files that shouldn't be made.

All WebIDL changes must be reviewed by a DOM peer and this hook enforces that.

push_printurls.py

This hook prints relevant information about a push that just completed.

It will print the URL of the changesets on https://hg.mozilla.org/. It will also print TreeHerder URLs for Try pushes.

single_head_per_branch.py

This hook enforces that all Mercurial branches contain at most one head.

treeclosure.py

This hook prevents pushes to Firefox repositories that are currently closed.

trymandatory.py

This hook enforces the requirement that pushes to the Try repository contain Try job selection syntax.

whitelist_releng.py

This hook enforces a whitelist of accounts that are allowed to push to certain Release Engineer repositories.

Hook Development Standards

Hooks are written and loaded into Mercurial as Python modules. This goes against recommendations by the Mercurial project. However, we do this for performance reasons, as spawning new processes for hooks wastes valuable wall time during push. (Mercurial recommends against in-process hooks because they don't make promises about the stability of the internal API.)

Hooks should be unit tested via `.t tests` and should strive for 100% code coverage. We don't want any surprises in production. We don't want to have to manually test hooks when upgrading Mercurial. We should have confidence in our automated tests.

Pre-commit (notably `pretxnchangegroup`) hooks should filter the `strip` source and always return success for these. If an `hg strip` operation is running, the changesets already got into the repository, so a hook has no business checking them again.

Any hook change touching a Mercurial API should be reviewed by someone who knows Mercurial internals. You should default to getting review from `gps`.

Hooks connecting to external systems or performing process that could be deferred will be heavily scrutinized. We want `hg push` operations to be fast. Slow services, networks, or CPU or I/O intensive hooks all undermine that goal.

Creating and Maintaining a Development Environment

In order to run tests for the extensions and hooks, you'll need to create and activate an isolated environment.

From the root directory of a version-control-tools checkout:

```
$ ./create-environment hgdev
```

Or to create an environment with support for running Bugzilla tests requiring Docker:

```
$ ./create-environment hgdev --docker-bmo
```

This will create a Python virtualenv in `venv/hgdev`. Assuming all goes well, it will print instructions on how to *activate* that environment in your local shell and how to run tests.

If the command fails, a likely culprit is missing system package dependencies.

On Debian/Ubuntu based distros, install required system packages via:

```
$ apt-get install build-essential python-all-dev sqlite3
```

On RedHat/CentOS based distros:

```
$ yum install bzip2 gcc python-devel sqlite
```

Then try `./create-environment hgdev` again. If that fails, this documentation may need updated!

Headless Repositories

This document describes the architecture and state of Mozilla's headless repositories.

History Lesson

For the longest time, Mozilla operated a special Mercurial repository called `Try`. When people wanted to schedule a build or test job against Firefox's release automation, they would create a special Mercurial commit that contained metadata on what jobs to schedule. They would push this commit (and its ancestors) to a new head on the `Try` repository. Firefox release automation would continuously poll the repository via the pushlog data and would schedule jobs for new pushes / heads according to the metadata in the special commit.

This approach was simple and worked for a long time without significant issues. Unfortunately, Mercurial (and Git) have known scaling problems as the number of repository heads approaches infinity. For Mozilla, we start to encounter problems after a few thousand heads. Things started to get really bad after 10,000 heads or so.

While fixing Mercurial to scale to thousands of heads would be admirable, after talking with Mercurial core developers, it was apparent that this would be a lot of work and the success rate was not considered high, especially as we started talking about scaling to 1 million or more heads. The recommended solution was to avoid the *mega-headed* scaling problem alltogether and to limit ourselves to a bound number of heads.

Headless Repositories

A headless repository is conceptually a single repository with thousands, but with repository data stored outside the repository itself.

Clients still push to the repository as before. However, special code on the server intercepts the incoming data and siphons it off to an external store. A pointer to this external data is stored, allowing the repository to serve up this data to clients that request it.

Mozilla plans to use headless repositories for Try, which share similar models of many clients writing to a central server with limited, well-defined clients for that data.

Technical Details

A Mercurial extension on the push server will intercept incoming changegroup data and write a Mercurial bundle of that data to S3. This is tracked in https://bugzilla.mozilla.org/show_bug.cgi?id=1078916.

A relational database will record information on each bundle - the URL, what changesets it contains, etc. This database will be written to as part of push by the aforementioned Mercurial extension. This is tracked in https://bugzilla.mozilla.org/show_bug.cgi?id=1078920.

A Mercurial extension on the hgweb servers will serve requests for S3-backend changesets. Clients accessing the server will be able to request data in S3 as if it is hosted in the repository itself. This is tracked in https://bugzilla.mozilla.org/show_bug.cgi?id=1078918.

The hgweb servers will also expose an HTTP+JSON API that matches the existing pushlog API in order to allow clients to poll for new changes without having to change their client-side code.

Initially, a one-off server to run the headless repositories will be created. It will have one-off Mercurial versions, software stack, etc. We may revisit server topology once things are rolled out and proved. This is tracked in https://bugzilla.mozilla.org/show_bug.cgi?id=1057148.

Clients that pull Try data will need to either upgrade to Mercurial 3.2 or install a custom extension that facilitates following links to S3 bundles. This is because we plan to use Mercurial's bundle2 exchange format and a feature we want to use is only available in Mercurial 3.2.

Low-Level Details

1. Client performs *hg push ssh://hg.mozilla.org/try*

2. Mercurial queries remote and determines what missing changesets needs to be pushed.
3. Client streams changeset data to server.
4. Server applies public changesets to the repository and siphons draft changesets into a new bundle.
5. Public changesets are committed to the repository. Draft changesets are uploaded to S3 in a bundle file.
6. Server records metadata of S3-hosted files and push info into database.

Mercurial for Mozillians

Want to learn how to get the most from Mercurial at Mozilla? You've come to the right place!

Installing Mercurial

Having a modern Mercurial installed is important. Features, bug fixes, and performance enhancements are always being added to Mercurial. Staying up to date on releases is important to getting the most out of your tools.

Note: Mercurial has a strong commitment to backwards compatibility.

If you are scared that upgrading will break workflows or command behavior, don't be. It is very rare for Mercurial to intentionally break backwards compatibility.

Recommended Versions

Mozilla recommends running the latest stable release of Mercurial. The latest stable release is always listed at https://www.mercurial-scm.org/. **As of June 2018, the latest stable release is 4.6.1.**

Danger: Mercurial versions before 3.7.3 have known vulnerabilities that can lead to arbitrary code execution when pulling from repositories.
Mercurial versions before 4.4.1 have known vulnerabilities that can result in arbitrary client-side code execution when processing subrepositories.
Mercurial versions before 4.6.1 have possible vulnerabilities that could result in arbitrary client-side code execution when pulling from repositories.
Version 4.6.1 or newer should always be used.

Mercurial makes a major *X.Y* release every three months, typically around the first of the month. Release months are February, May, August, and November. A *X.Y.Z* point release is performed each month after or as needed (if a severe issue is encountered).

If you are conservative about software updates, it is OK to wait to upgrade until the *X.Y.1* point release following a major version bump.

Installing on Windows

If you are a Firefox developer, you should install Mercurial indirectly through MozillaBuild. Mercurial can be upgraded within MozillaBuild by running `pip install --upgrade Mercurial`.

If you are not a Firefox developer, download a Windows installer direct from the Mercurial project.

Installing on OS X

Mercurial is not installed on OS X by default. You will need to install it from a package manager or install it from source.

mach bootstrap

If you have a clone of a Firefox repository, simply run `mach bootstrap` to install/upgrade Mercurial. Keep in mind this will install all packages required for Firefox development. If this is not wanted, follow a set of instructions below.

Homebrew

Homebrew typically keeps their Mercurial package up to date. Install Mercurial through Homebrew by running:

```
$ brew install mercurial
```

You may want to run `brew update` first to ensure your package database is up to date.

MacPorts

MacPorts typically keeps their Mercurial package up to date. Install through MacPorts by running:

```
$ port install mercurial
```

From Source

See the section below about how to install Mercurial from source.

Installing on Linux, BSD, and other UNIX-style OSs

The instructions for installing Mercurial on many popular distributions are available on Mercurial's web site. However, many distros don't keep their Mercurial package reasonably current. You often need to perform a source install.

Installing from Pip

Mercurial source packages can be installed via `pip` - Python's preferred package management tool.

Installing Mercurial via pip is simple:

```
$ pip install Mercurial
```

To upgrade Mercurial:

```
$ pip install --upgrade Mercurial
```

By default, `pip install` may try to write to `/usr/local` or `/usr/lib` or other parts of your system that require elevated permissions to write to. To perform an install into your user directory:

```
$ pip install --user Mercurial
```

That *may* install Mercurial to a directory not in `PATH`, such as `~/.local/bin`. You may need to adjust your shell's startup file to add this directory to `PATH`. To see exactly where it installs things in `--user` mode, run `pip show Mercurial`. There will be a line like `Location: /home/myuser/.local/lib/python2.7/site-packages`. The `hg` executable is likely in a `bin` directory at the same level of the `lib/` directory.

Installing from Source

Installing Mercurial from source is simple and should not be dismissed because it isn't coming from a package.

Download a source archive from Mercurial. Alternatively, clone the Mercurial source code and check out the version you wish to install:

```
$ hg clone https://www.mercurial-scm.org/repo/hg
$ cd hg
$ hg up 4.6.1
```

Once you have the source code, run `make` to install Mercurial:

```
$ make install
```

If you would like to install Mercurial to a custom prefix:

```
$ make install PREFIX=/usr/local
$ make install PREFIX=/home/gps
```

Note: Mercurial has some Python C extensions that make performance-critical parts of Mercurial significantly faster. You may need to install a system package such as `python-dev` to enable you to build Python C extensions.

Tip: Are you concerned about a manual Mercurial install polluting your filesystem? Don't be.

A Mercurial source install is fully self-contained. If you install to a prefix, you only need a reference to the `PREFIX/bin/hg` executable to run Mercurial. You can create a symlink to `PREFIX/bin/hg` anywhere in `PATH` and Mercurial should *just work*.

Verifying Your Installation

To verify Mercurial is installed properly and has a basic configuration in place, run:

```
$ hg debuginstall
```

If it detects problems, correct them.

If you have a clone of the Firefox repository, you are highly encouraged to run *mach vcs-setup* to launch an interactive wizard that will help you optimally configure Mercurial for use at Mozilla.

Reasons to Upgrade

General Advice

Mercurial releases tend to be faster and have fewer bugs than previous releases. These are compelling reasons to stay up to date.

Avoid Mercurial versions older than 3.7.3 due to issues below.

Security Issues

Versions of Mercurial before 3.7.3 are vulnerable to multiple security issues that can lead to executing arbitrary code when cloning or pulling from repositories. Avoid versions older than 3.7.3!

Cloning and Pulling Performance

Mercurial 4.1 introduced supported for compression data over the wire protocol with zstandard. This is substantially faster than zlib and can result in faster clones and pulls due to faster compression and fewer bytes transferred over the wire.

Mercurial 3.6 contains a number of enhancements to performance of cloning and pull operations, especially on Windows. Clone times for mozilla-central on Windows can be several minutes faster with 3.6.

Revset Performance

Mercurial 3.5 and 3.6 contained a number of performance improvements to revision sets. If you are a user of `hg wip` or `hg smartlog`, these commands will likely be at least 4x faster on Mercurial 3.6.

Revsets are used internally by Mercurial. So these improvements result in performance improvements for a hodgepodge of operations.

Tags Cache Performance

Mercurial 3.4 contains improvements to the tags cache that prevent it from frequently doing CPU-intensive computations in some workflows.

Important: Users of evolve will have horrible performance due to the tags cache implementation in versions older than 3.4 and should upgrade to 3.4+.

Performance Issues with Large Repositories

Mercurial 3.0 through 3.1.1 contained a significant performance regression that manifests when cloning or pulling tens of thousands of changesets. These versions of Mercurial should be avoided when interacting with large repositories, such as mozilla-central.

Mercurial 3.3 introduced a class of performance regressions most likely encountered as part of running `hg blame` or `hg graft`. The regressions are largely fixed in 3.4.

CVE-2014-9390

Mercurial versions older than 3.2.3 should be avoided due to a security issue (CVE-2014-9390) impacting Windows and OS X users.

Supporting Old Versions

Mozilla has written a handful of Mercurial extensions. Supporting N versions of Mercurial is easier than supporting N+1 versions, especially as Mercurial's API is rapidly evolving. It is extra work to support old versions when new versions work just fine.

Newer Wire Protocol

Mercurial 3.5 featured a new wire protocol that performs pushes and pulls more efficiently.

Cloning from Pre-Generated Bundle Files

Mercurial 3.6 supports transparently cloning from pre-generated bundle files. When you clone from hg.mozilla.org, many of the larger repositories will be served from a CDN. This results in a faster and more reliable clone.

Mercurial 4.1 will download zstandard-compressed bundles from hg.mozilla.org by default. These are substantially smaller than gzip-based bundles.

Installing and Configuring Extensions

A vanilla install of Mercurial does not have many features. This is an intentional choice by Mercurial to keep the base install simple and free of foot guns.

Guided Installation of Extensions

If you have a copy of the Firefox source repository, you can run *./mach vcs-setup* to run an interactive wizard that will guide you through the process of configuring Mercurial.

Important: *mach vcs-setup* is the recommended way to configure Mercurial for use at Mozilla. The sections below effectively duplicate the work that *mach vcs-setup* does for you.

Installing Extensions

To install an extension, you'll need to add a line to your Mercurial configuration file.

As a user, you care about the following configuration files:

1. Your global hgrc
2. A per-repository hgrc

Your user-global configuration file is `~/.hgrc`. Settings in this file apply to all `hg` commands you perform everywhere on your system (for your current user).

Each repository that you clone or initialize has an optional `.hg/hgrc` file that provides repository-specific configurations.

Mercurial starts by loading and applying settings from global configuration files and then overlays configurations from each lesser-scoped files.

Tip: To learn more about Mercurial configuration files, run `hg help config`.

To install an extension, you add a line under the `[extensions]` section of a config file like the following:

```
[extensions]
foo=path/to/extension
```

This is saying *activate the **foo** extension whose code is present at **path/to/extension**.

Core Extensions That Almost Everyone Wants

Mercurial ships with a number of built-in extensions. Of these, every user will almost always want to install the following extensions:

fsmonitor Monitor the filesystem for changes using so operations querying filesystem state complete faster.

Important: fsmonitor is highly recommended when interacting with the Firefox repository. It will make Mercurial commands faster.

Since core extensions are bundled with Mercurial, they have a special syntax that makes them easier to install:

```
[extensions]
fsmonitor=
```

Core Extensions to Perform History Rewriting

Out of the box, Mercurial only allows commits operations to be additive. If you make a mistake, the solution is to create a new commit that fixes it. You can't rewrite old commits. You can't change the order of existing commits. You can't change the shape of the DAG of the commits.

These operations all have something in common: they rewrite history.

Note: Mercurial doesn't allow history rewriting by default because it is a major foot gun for people new to version control. A potential side-effect of history rewriting is data loss or confusion due to loss of state. Mercurial believes that these risks should be opt-in and has thus shipped without any history rewriting features enabled by default.

Mercurial ships with a number of built-in extensions that enable history rewriting:

histedit Enables the `hg histedit` command, which brings up a text editor listing commits, allowing you to change order and specify actions to perform.

The functionality is roughly equivalent to `git rebase -i`.

rebase Enables the `hg rebase` command, which allows you to splice commits across different chains in the DAG.

strip Enables you to delete changesets completely.

Core Extensions to Enable Different Workflows

Mercurial ships with some extensions that enable alternate workflows. These include:

shelve Enables uncommitted work to be saved to a standalone file without being committed to the repository.

chg

chg is a C wrapper for the `hg` command. Typically, when you type `hg`, a new Python process is created, Mercurial is loaded, and your requested command runs and the process exits.

With `chg`, a Mercurial *command server* background process is created that runs Mercurial. When you type `chg`, a C program connects to that background process and executes Mercurial commands.

chg can drastically speed up Mercurial. This is because the overhead for launching a new Python process is high (often over 50ms) and the overhead for loading Mercurial state into that process can also be high. With `chg`, you pay this cost once and all subsequent commands effectively eliminate the Python and Mercurial startup overhead. For example:

```
$ time hg --version
real    0m0.118s
user    0m0.100s
sys     0m0.015s

$ time chg --version
real    0m0.012s
user    0m0.000s
sys     0m0.004s

$ time hg export
real    0m0.137s
user    0m0.093s
sys     0m0.042s

$ time chg export
real    0m0.034s
user    0m0.000s
sys     0m0.004s
```

Here, we see ~100ms wall time improvement with chg activated. That may not sound like a lot, but you will notice.

If you have installed Mercurial through a package manager (such as Homebrew or APT) you may already have `chg` installed. For more information, see here.

3rd Party Extensions You Should Highly Consider

evolve

The evolve extension opens up new workflows that harness Mercurial's ability to record how changesets *evolve* over time.

Typically, when history is rewritten, new commits are created and the old ones are discarded. With the `evolve` extension enabled, Mercurial instead hides the old commits and writes metadata holding the relationship between old and new commits. This metadata can be transferred between clients, allowing clients to make intelligent decisions about how to recover from rewritten history. For example, if a force push is performed, a client will know exactly what rebase to perform to mimic what was done elsewhere.

The `evolve` extension also enables useful Mercurial commands such as `hg previous`, `hg next`, and `hg amend` (which is a shortcut for `hg commit --amend`).

githelp

Are you a Git user learning Mercurial for the first time? The githelp extension adds a `hg githelp` command that suggests Mercurial equivalent commands from Git commands. Just type a Git command and learn how to use Mercurial!

Configuring Authentication

SSH Configuration

Pushing to Mercurial is performed via SSH. You will need to configure your SSH client to talk appropriate settings to Mozilla's Mercurial servers.

Typically, the only setting that needs configured is your username. In your SSH config (likely `~/.ssh/config`), add the following:

```
Host hg.mozilla.org
  User me@mozilla.com
```

Tip: Be sure to replace `me@mozilla.com` with your Mozilla-registered LDAP account that is configured for SSH access to Mercurial.

The first time you connect, you will be asked to verify the host SSH key.

The fingerprints of the host keys for `hg.mozilla.org` are as follows:

```
ED25519 (server preferred key)
256 SHA256:7MBAdqLe8+aSYkv+5/2LUUxd+WdgYcVSV+ZQVEKA7jA hg.mozilla.org
256 SHA1:Ft++OU96cvaREKNFCJ6AiuCpGac hg.mozilla.org
256 MD5:96:eb:3b:78:f5:ca:19:e2:0c:a0:95:ea:04:28:7d:26 hg.mozilla.org

RSA
4096 SHA256:RX2OK8A1KNWdxyu6ibIPeEGLBzc5vyQW/wd7RKjBehc hg.mozilla.org
4096 SHA1:p2MGe4wSw8ZnQ5J9ShBk/6VA+Co hg.mozilla.org
4096 MD5:1c:f9:cf:76:de:b8:46:d6:5a:a3:00:8d:3b:0c:53:77 hg.mozilla.org
```

A GPG signed document stating asserting the validity of these keys can be verified:

```
curl https://hg.mozilla.org/hgcustom/version-control-tools/raw-file/tip/docs/vcs-server-info.asc > mo
gpg --verify mozilla-vcs-info.asc
```

Verify your SSH settings are working by attempting to SSH into a server. Your terminal output should resemble the following:

```
 $ ssh hg.mozilla.org
 A SSH connection has been successfully established.

 Your account (me@example.com) has privileges to access Mercurial over
 SSH.

You are a member of the following LDAP groups that govern source control
access:

   scm_level_1

This will give you write access to the following repos:

   Try

You will NOT have write access to the following repos:

   Autoland (integration/autoland), Firefox Repos (mozilla-central, releases/*), ...

 You did not specify a command to run on the server. This server only
 supports running specific commands. Since there is nothing to do, you
 are being disconnected.
 Connection to hg.mozilla.org closed.
```

Authenticating with Services

Various Mercurial extensions interface with services such as Bugzilla. In order to do so, they often need to send authentication credentials as part of API requests. This document explains how this is done.

`mozhg.auth` contains a unified API for any Mercurial extension or hook wishing to obtain authentication credentials. New extensions are encouraged to use or add to this module instead of rolling their own code.

Finding Bugzilla Credentials

`mozhg.auth.getbugzillaauth()` is the API used to request credentials for `bugzilla.mozilla.org`. It will attempt to find credentials in the following locations:

1. The `bugzilla.userid` and `bugzilla.cookie` values from the active Mercurial config.
2. The `bugzilla.username` and `bugzilla.password` values from the active Mercurial config.
3. Login cookies from a Firefox profile.
4. Interactive prompting of username and password credentials.

Credential Extraction from Firefox Profiles

As mentioned above, authentication credentials are searched for in Firefox profiles. For example, Bugzilla login cookies are looked for in Firefox's cookie database.

The first step of this is finding available Firefox profiles via the current user's `profiles.ini` file.

By default, the available profiles are sorted. The default profile is searched first. Remaining profiles are searched according to the modification time of files in the profile - the more recent the profile was used, the earlier it is searched.

If the `bugzilla.firefoxprofile` config option is present, it will explicitly control the Firefox profile search order. If the value is a string such as `default`, only that profile will be considered. If the value is a comma-delimited list, only the profiles listed will be considered and profiles will be considered in the order listed.

The DAG and Mercurial

Distributed version control systems (DVCS) like Mercurial (and Git) utilize a directed acyclic graph (DAG) for representing commits. This DAG is how the *history* of a repository is stored and often represented.

To help understand how this works, we'll be using Mercurial commands to show and manipulate the DAG.

We start by creating an empty Mercurial repository with an empty graph:

```
$ hg init repo
$ cd repo
```

The Mercurial command for inspecting the repository history (and the DAG by extension) is `hg log`. Let's run it on our empty repository:

```
$ hg log -G
```

No output. This is confirmation that the graph is empty.

Note: We use `-G` throughout this article because it is necessary to print a visual representation of the DAG.

Adding Nodes

We'll need to create nodes to make our DAG interesting. The way we do this is by *committing*. But first, we need something to commit. Let's create a file:

```
$ echo 1 > file
```

Then we tell Mercurial that we're interested in storing the history of this file:

```
$ hg add file
```

We then use `hg commit` to create a new node (*changeset* in Mercurial parlance) and add it to the DAG:

```
$ hg commit -m A
```

(`-m A` says to use `A` for the commit message).

Now let's take a look a the DAG:

```
$ hg log -G -T '{desc}'
@  A
```

Note: `-T '{desc}'` tells Mercurial to only print the *description* or *commit message* from the changeset/node. Without it, output would be much more verbose.

It looks like we have a single node - denoted by `@` and `A`.

This graph / repository state isn't very interesting. So, we perform another commit:

```
$ echo 2 >> file
$ hg commit -m B
```

And then inspect the DAG:

```
$ hg log -G -T '{desc}'
@  B
|
o  A
```

This is slightly more interesting!

The `B` node has been introduced. That is pretty obvious.

`A` has its node represented by a `o`. `B` has its node represented by `@`. Mercurial uses `@` to represent the node currently attached to the *working directory*. This will be discussed later.

The first column has grown a vertical pipe character (`|`) between the two entries. In graph terms, this is an *edge*. While the visualization doesn't indicate it, `B` internally stores a pointer back to `A`. This constitutes the *edge* and since the edge is directional (`B` to `A`), that makes the graph *directed* (the *D* from *DAG*).

Directed graphs borrow terminology from biology to represent relationships between nodes:

parent node A node that came before another (is referred to by another node).

child node A node that derives directly from another.

root node A node that has no parents.

head node A node that has no children.

descendants All the children, the children's children, the children's children's children, and so on of a node.

ancestors The parents, parents' parents, parents' parents' parents, and so on of a node.

In our graph so far:

- `A` is the parent of `B`
- `B` is a child of `A`
- `A` is a root
- `B` is a head
- The descendants of `A` are just `B`.
- The ancestors of `B` are just `A`.

If you don't understand this, let's try committing a few more nodes to help your understanding.:

```
$ echo 3 >> file
$ hg commit -m C
$ echo 4 >> file
$ hg commit -m D

$ hg log -G -T '{desc}'
@ D
|
o C
|
o B
|
o A
```

`A` is still the root node. Since `B` has children, it is no longer a *head*. Instead, `D` is now our head node.

If all you do is `hg commit` like we've been doing so far, your repository's DAG will be a linear chain of nodes, just like we have constructed above. 1 head. Every node has 1 parent (except the root).

Important: The important takeaway from this section is that the *history* of Mercurial repositories is stored as a DAG. `hg commit` creates a changeset and appends a node to a graph. A DAG node and a Mercurial changeset are effectively the same thing.

Nodes are Hashes of Content

Up to this point, we've been using our single letter commit messages (`A`, `B`, etc) to represent nodes in our DAG. This is good for human understanding, but it hides an important detail of how Mercurial actually works.

Mercurial uses a SHA-1 digest to identify each node/changeset in the repository/DAG. The SHA-1 digest is obtained by hashing the content of the changeset.

Essentially, Mercurial changesets consist of various pieces of data including but not limited to:

1. The parent node(s)
2. The set of files and their state
3. The author
4. The date
5. The commit message

Mercurial assembles all these pieces of data in a well-defined manner, feeds the result into a SHA-1 hasher, and uses the digest of the result as the node/changeset ID.

SHA-1 digests are 20 bytes or 40 hex characters. They look like `835dbd9444dbed0cdc2ca27e23839f05a58e1dc1`. For readability, these are almost always abbreviated to 12 characters in user-facing interfaces. e.g. `835dbd9444db`.

We can ask Mercurial to render these SHA-1 digests instead of the commit messages:

```
$ hg log -G -T '{node}'
@  2bf9b23b2d0379540038866a72699a8ce5e92e84
|
o  0f165760af41ddde6470860088f421c1efcc5a5f
|
o  7175417717e87c88e4cf61ab2f76f2c54c76fa4b
|
o  8febb2b7339e5843832ab893ca2a002cd4394a03
```

Or we can ask for the short versions:

```
$ hg log -G -T '{node|short} {desc}'
@  2bf9b23b2d03 D
|
o  0f165760af41 C
|
o  7175417717e8 B
|
o  8febb2b7339e A
```

Note: We start to use some more capabilities of Mercurial's *templates* feature. This allows output from Mercurial commands to be customized. See `hg help templates` for more.

Because SHA-1s (even their short versions) are difficult to remember, we'll continue using commit messages and single letters throughout this article to aid comprehension.

Important Properties from Using Hashing

Since node IDs are derived by hashing content, this means that changing **any** of that content will result in the node ID changing.

Change a file: new node ID.

Change the commit message: new node ID.

Change the parent of a node: new node ID.

Changing the content of a changeset and thus its node ID is referred to as *history rewriting* because it changes the *history* of a repository/DAG. *History rewriting* is an important topic, but it won't be discussed quite yet. The important thing to know is that if you change anything that's part of the changeset, the node ID changes.

Moving Between Nodes

Looking at the state of our Mercurial repository on the filesystem, we see two entries:

```
$ ls -A
.hg/
file
```

The `.hg` directory contains all the files managed by Mercurial. It should be treated as a black box.

Everything else in this directory (currently just the `file` file and the current directory) is referred to as the *working directory* or *working copy* (both terms used interchangeably).

The *working directory* is based on the state of the files in a repository at a specific changeset/node. We say *based on* because you can obviously change file contents. But initially, the *working directory* matches exactly what is stored in a specific changeset/node.

The `hg update` (frequently `hg up`) command is used to change which node in the DAG the *working directory* corresponds to.

If you `hg up 7175417717e8`, the *working directory* will assume the state of the files from changeset/node `7175417717e8`. . . . If you `hg up 2bf9b23b2d03`, state will be changed to `2bf9b23b2d03`. . . .

The ability to move between nodes in the DAG introduces the possibility to...

Creating DAG Branches

Up until this point, we've examined perfectly linear DAGs. As a refresher:

```
$ hg log -G -T '{node|short} {desc}'
@  2bf9b23b2d03 D
|
o  0f165760af41 C
|
o  7175417717e8 B
|
o  8febb2b7339e A
```

Every node (except the root, `A`/`8febb2b7339e`) has 1 parent node. And the graph as a whole has a single head (`D`/`2bf9b23b2d03`).

Let's do something a bit more advanced. We start by switching the *working directory* to a different changeset/node:

```
$ hg up 7175417717e8
1 files updated, 0 files merged, 0 files removed, 0 files unresolved

$ hg log -G -T '{node|short} {desc}'
o  2bf9b23b2d03 D
|
o  0f165760af41 C
|
@  7175417717e8 B
|
o  8febb2b7339e A
```

(Note how `@` - the representation of the active changeset/node in the *working directory* - moved from `D` to `B`)

Now let's commit a new changeset/node:

```
$ echo 5 >> file
$ hg commit -m E
created new head
```

That *created new head* message is a hint that our DAG has changed. Can you guess what happened?

Let's take a look:

```
$ hg log -G -T '{node|short} {desc}'
@  4a3687e9313a E
|
| o  2bf9b23b2d03 D
```

```
| |
| o  0f165760af41 C
|/
o  7175417717e8 B
|
o  8febb2b7339e A
```

`B` now has multiple direct children nodes, `C` and `E`. In graph terminology, we refer to this as a *branch point*.

`E` has no children, so it is a *head* node (`D` is still a head node as well).

Because the visualization of the graph can resemble a tree (from nature, not your computer science textbooks), small *protrusions* from the main *trunk* are referred to as *branches* from the perspective of the DAG. (Mercurial has overloaded *branch* to convey additional semantics, so try not to confuse a *DAG branch* with a *Mercurial branch*.)

The *created new head* message was Mercurial telling us that we created not only a new *DAG head* but also a new *DAG branch*.

Because your commit is taking the repository in a different *direction* (very non-scientific word), this act of creating new DAG branches is sometimes referred to as *divergence* or *diverging*.

DAG branches turn out to be an excellent way to work on separate and isolated units of change. These are often referred to as *feature branches* because each DAG branch consists of a specific feature. For more, see *Workflows*.

It's worth noting that `hg commit` **always** produces a new head node because the node being created never has any children. However, it may not create a new DAG branch: a new DAG branch is only created when the parent node of the commit isn't a head node.

Before we go on let's commit a new changeset on top of `E` to make the DAG branch more pronounced:

```
$ echo 6 >> file
$ hg commit -m F

$ hg log -G -T '{node|short} {desc}'
@  da36621d7a94 F
|
o  4a3687e9313a E
|
| o  2bf9b23b2d03 D
| |
| o  0f165760af41 C
|/
o  7175417717e8 B
|
o  8febb2b7339e A
```

Merging DAG Branches

Now that we have multiple DAG branches, it is sometimes desirable to *merge* them back into one. The Mercurial command for performing this action is `hg merge`.

Let's change our working directory to the changeset that we want to merge *into*. We choose `D`, since it was our original head.:

```
$ hg up 2bf9b23b2d03
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Now we tell Mercurial to bring the changes from `F`'s head into `D`'s:

```
$ hg merge da36621d7a94
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)

$ hg commit -m G
```

Visualizing the result:

```
$ hg log -G -T '{node|short} {desc}'
@    19c6c94d7bb2 G
|\
| o  da36621d7a94 F
| |
| o  4a3687e9313a E
| |
o |  2bf9b23b2d03 D
| |
o |  0f165760af41 C
|/
o  7175417717e8 B
|
o  8febb2b7339e A
```

`G/19c6c94d7bb2` is what is referred to as a *merge commit*. It is the result of a commit operation that merged 2 nodes. From the perspective of the DAG, it is a node with 2 parents, not 1.

Conclusion

These are the basics of how Mercurial uses a directed acyclic graph (DAG) to represent repository history.

If you would like to learn more about how distributed version control systems (like Mercurial) use DAGs, please read this article.

For more on workflows that build upon this knowledge, see *Workflows*.

Workflows

Mercurial is a flexible tool that allows you to choose from several workflows and variations thereof.

Important: Before reading on, it is important to have a firm grasp on the concepts described in *The DAG and Mercurial*. That article describes how Mercurial models repository history as a directional acyclic graph (DAG) and understanding of this is critical for many workflows.

Feature Branches and Head-Based Development

As mentioned in *The DAG and Mercurial*, DAG branches are commonly used to work on isolated units of change. DAG branches used this way are called *feature branches* because each DAG branch tracks a specific *feature* or line of work.

Another way to think about this is as *head-based development*. Each DAG branch has its own head node (this is a basic property of directed acyclic graphs). So, working on different DAG branches is effectively working on different heads.

The general way feature branches/head-based development work is:

1. Set your starting point via `hg up <starting node>`.
2. Modify files
3. `hg commit`
4. Repeat #2 and #3 until work is done
5. Integrate DAG branch somehow (typically a rebase or merge)

To help understand this, let's start with the following state:

```
$ hg log -G -T '{node|short} {desc}'
@  2bf9b23b2d03 D
|
o  0f165760af41 C
|
o  7175417717e8 B
|
o  8febb2b7339e A
```

The working directory is based on `D`. But we don't like the state of `D`, so we decide to start working from `B` instead:

```
$ hg up 7175417717e8
$ echo changes > file
$ hg commit -m E
created new head
$ echo 'more changes' > file
$ hg commit -m F
```

You get bored working on that feature. Or, you run into some obstacles and want to try a fresh approach. So, you decide to start a new *feature branch*:

```
$ hg up 7175417717e8
$ echo 'new feature' > file
$ hg commit -m G
$ echo 'more new features' > file
$ hg commit -m H
```

Then you have a revelation about the first feature branch you were working on and go back to make a change:

```
$ hg up 82123e512a06
$ echo revelation > file
$ hg commit -m I
```

Our repository now looks like:

```
$ hg log -G -T '{node|short} {desc}'
@  bcb7c3592ba2 I
|
| o  2ca760d1e4fe H
| |
| o  d6248a455a1b G
| |
o |  82123e512a06 F
| |
o |  bed01724d682 E
|/
| o  2bf9b23b2d03 D
| |
| o  0f165760af41 C
|/
```

```
o  7175417717e8 B
|
o  8febb2b7339e A
```

To the uninitiated, this view can look complicated because it kind of is. You've got a number of commits and lines going every which way. And, you can imagine how complicated things can become if you are working on several heads and/or the repository history is large and/or fast moving. We need tools beyond `hg log -G` to help us sort through these commits.

Finding Heads

The `hg heads` command can be used to quickly see all repository heads. Running it on our current repository will reveal something like:

```
$ hg heads
changeset:   8:bcb7c3592ba2
tag:         tip
parent:      5:82123e512a06
user:        Gregory Szorc <gps@mozilla.com>
date:        Wed Aug 12 12:57:28 2015 -0700
summary:     I

changeset:   7:2ca760d1e4fe
user:        Gregory Szorc <gps@mozilla.com>
date:        Wed Aug 12 12:52:26 2015 -0700
summary:     H

changeset:   3:2bf9b23b2d03
user:        Gregory Szorc <gps@mozilla.com>
date:        Wed Aug 12 11:57:08 2015 -0700
summary:     D
```

Tip: `hg heads` is roughly equivalent to `hg log -r 'head()'`, which uses the `head()` revision set function to only select head changesets/nodes.

`hg heads` can be useful to get a quick overview of all *unmerged* DAG branches. If the canonical repository only has a single head, then `hg heads` will be a good approximation for *what work hasn't been merged yet*. But if the canonical repository has many heads (this is frequently the case), then `hg heads` may lose some of its utility because it will display **all** heads, not just the ones you care about.

Read on for some ways to deal with this.

Labeling

Up until this point, all our Mercurial commands were interacting with the 12 character hex abbreviation of the full SHA-1 changeset. These values are effectively random, opaque, and difficult to memorize. It can be annoying and possibly difficult for humans to grasp with them. This is why Mercurial provides facilitites for *labeling* heads and changesets. There are many forms of labels in Mercurial.

Bookmarks

Bookmarks are specially behaving labels attached to changesets. When you commit when a bookmark is *active*, the active label/bookmark automatically moves to the just-committed changeset.

For more on bookmarks, see *Using Bookmarks*.

Bookmark users may find the `hg bookmarks` command useful, as it prints a concise summary of all bookmarks. This is arguably a better version of `hg heads`, which we learned about above. However, a downside of `hg bookmarks` is that it only shows the changesets with bookmarks: it doesn't show other changesets in that head or the overall DAG. For that, we'll need more powerful tools. Keep reading to learn more.

Branches

Mercurial branches (not to be confused with generic *DAG branches*) are a more heavyweight label that can be applied to changesets. Unlike bookmarks whose labels move as changesets are committed, branches are stored inside the changeset itself and are permanent.

When you make a Mercurial branch active, all subsequent commits will forever be associated with that branch.

Branches are useful for long-running heads, such as tracking releases. However, their utility for short-lived feature development is widely considered to be marginal. And for large repositories, the presence of hundreds or even thousands of branches over time or from hundreds of developers can lead to a lot of clutter and confusion.

Important: The use of Mercurial branches for feature development is highly discouraged. For Firefox, Mercurial branches are never used for tracking features.

Because the use of Mercurial branches is discouraged, we won't describe how they are used.

MQ

Mercurial Queues (MQ) is a workflow extension that focuses on interacting with stacks of labeled patches. Contrast this with head-based workflows, where you are interacting with nodes and heads on the repository DAG.

Some like MQ because it hides the complexity of the DAG. It takes a simple and easily comprehended approach to working on things. However, it also has numerous setbacks:

- MQ doesn't perform 3-way merges and thus merge conflicts (in the form of *.rej* files) are much more common.
- Managing labels for every single changeset can be cumbersome, introducing overhead that encourages fewer, larger, and harder-to-review commits.
- Performance on large repositories can be horrible.
- The extension isn't actively developed and bugs often go unfixed.

Important: The Mercurial project doesn't recommend MQ, especially for new Mercurial users. At Mozilla, we also recommend not using MQ. Use a head-based workflow instead.

Refining What Changesets are Shown

`hg heads`, `hg bookmarks`, `hg branches`, `hg qseries`, and other commands meant to summarize common entities within the repository each suffer from the limitation that they often show too little information. When doing development, you often want to see all the changesets in a head or want to see the shape of the DAG. We need a way to view the important information from the aforementioned commands, without the overload that `hg log -G` gives us. Fortunately, Mercurial has an answer.

The output from the `hg log` command can be highly configurable via the use of *revision sets (revsets)* and *templates*. The former determines what to show and the latter how to show it.

When we run `hg log -G`, Mercurial will display information for **all** changesets and render it according to the default command line template. As you'll quickly learn, this is far from an ideal way to find changesets you care about.

For a fast and information rich display of changesets relevant to you - a view on the heads/features you've been working on - we highly recommend the `hg wip` command described at Customizing Mercurial Like a Pro.

To Label or Not to Label

Before we learned about bookmarks, branches, and MQ patches, we learned how to create label-less DAG branches. Various Mercurial workflows use labels because they are more human friendly than SHA-1 fragments. But, they aren't required.

Note: The concept of label-less heads does not exist in Git: Git requires all heads to have a label (a Git branch name) or the head and the commits unique to it will eventually be deleted via garbage collection.

Because Git requires labels and Mercurial does not, it is accurate to say that Mercurial has lighter weight DAG branches than Git!

Since Mercurial doesn't require labels, it raises an interesting question: should you use labels?

The answer, like most things, depends.

Custom and powerful query and rendering tools like the aforementioned `hg wip` command are sufficient for many to simply not need labels and to use anonymous, unlabeled changesets and heads for everything. A benefit to this approach is less overhead interacting with and managing labels: you don't need to make a bookmark or branch active: you just update to a changeset, make changes, and commit. You don't need to clean up labels when you are done. It's all very low-level and feels fast. It also contributes to understanding of the DAG and its concepts.

A downside of label-less workflows is you have to interact with SHA-1s or SHA-1 fragments all the time. There is a lot of copying and pasting of these values in order to run commands. And, this is simply too much for some people. Some just need human-friendly labels.

Using Bookmarks

The Mercurial project recommends the use of bookmarks for doing development.

At its core, bookmarks are a labeling mechanism. Instead of a numeric revision ID or alphanumeric SHA-1 (fragments), bookmarks provide human-friendly identifiers to track and find changesets or lines of work.

Tip: If you are a Git user, bookmarks are similar to Git branches. Although they don't behave exactly the same.

Bookmarks and Feature Development

Bookmarks are commonly used to track the development of something - a *feature* in version control parlance. The workflow is typically:

1. Create a bookmark to track a feature
2. Commit changes for that feature against that bookmark
3. Land the changes

Bookmarks typically exist from the time you start working on a feature to the point that feature lands, at which time you delete the bookmark, for it is no longer necessary.

Creating and Managing Bookmarks

Numerous guides exist for using bookmarks. We will not make an attempt at reproducing their work here.

Recommending reading for using bookmarks includes:

- The official Mercurial wiki
- Bookmarks Kick Start Guide
- A Guide to Branching in Mercurial

The following sections will expand upon these guides.

Getting the Most out of Bookmarks

Use Mercurial 3.2 or Newer

Mercurial 3.2 adds notification messages when entering or leaving bookmarks. These messages increase awareness for when bookmarks are active.

Integrate the Active Bookmark into the Shell Prompt

If you find yourself forgetting which bookmark is active and you want a constant reminder, consider printing the active bookmark as part of your shell prompt. To do this, use the prompt extension or the *scm-prompt.sh* script from Facebook's hg-experimental repository.

Sharing Changesets

Once you have a changeset (or several!) that you'd like to get checked into a Mozilla repository, you'll need to share them with others in order to get them reviewed and landed.

When working with mozilla-central, pushing your changesets to MozReview is the primary method of sharing. See the MozReview documentation for more information.

Collaborating / Sharing Bookmarks

Say you have multiple machines and you wish to keep your bookmarks in sync across all of them. Or, say you want to publish a bookmark somewhere for others to pull from. For these use cases, you'll need a server accessible to all parties to push and pull from.

If you have Mozilla commit access, you can create a user repository to hold your bookmarks.

If you don't have Mozilla commit access or don't want to use a user repository, you can create a repository on Bitbucket.

> **Warning:** The Firefox repository may be larger than what Bitbucket allows you to store. If you want to share bookmarks for the Firefox repository, a user repository is your best bet.

If neither of these options work for you, you can run your own Mercurial server.

Pushing and Pulling Bookmarks

`hg push` by default won't transfer bookmark updates. Instead, you need to use the `-B` argument to tell Mercurial to push a bookmark update. e.g.:

```
$ hg push -B my-bookmark user
pushing to user
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
exporting bookmark my-bookmark
```

Tip: When pushing bookmarks, it is sufficient to use `-B` instead of `-r`.

When using `hg push`, it is a common practice to specify `-r <rev>` to indicate which local changes you wish to push to the remote. When pushing bookmarks, `-B <bookmark>` implies `-r <bookmark>`, so you don't need to specify `-r <rev>`.

Unlike `hg push`, `hg pull` will pull all bookmark updates automatically. If a bookmark has been added or updated since the last time you pulled, `hg pull` will tell you so. e.g.:

```
$ hg pull user
pulling from user
pulling from $TESTTMP/a (glob)
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
updating bookmark my-bookmark
```

Things to Watch Out For

Mercurial repositories are publishing by default. If you push to a publishing repository, your Mercurial client won't let you modify pushed changesets.

As of February 2015, user repository on hg.mozilla.org are non-publishing by default, so you don't have to worry about this. However, if you use a 3rd party hosting service, this could be a problem. Some providers have an option to mark repositories as non-publishing. This includes Bitbucket. **If you plan on sharing bookmarks and rewriting history, be sure you are using a non-publishing repository.**

Performing Common Tasks

Revive a Commit That Was Backed Out

Say your repository has the following history:

```
changeset:   263002:dfc93c68f9c7
user:        Gijs Kruitbosch <gijskruitbosch@gmail.com>
date:        Mon Dec 22 15:05:06 2014 +0000
summary:     Bug 1113299 - hide tab mirroring feature if unavailable, r=jaws

changeset:   263001:268dfa4925ec
```

```
user:        Brian Grinstead <bgrinstead@mozilla.com>
date:        Tue Jan 13 12:25:57 2015 -0800
summary:     Backed out changeset 291e3a83a122 (bug 1042619)

changeset:   263000:492134f929e2
user:        Tim Nguyen <ntim.bugs@gmail.com>
date:        Tue Jan 13 09:51:00 2015 -0500
summary:     Bug 1121048 - Add back round corners on perf tool icon glyphs. r=jsantell
```

Commit `291e3a83a122` was backed out by `263001:268dfa4925ec` and you want to *revive* it, either to reland it or to work on it again.

`hg graft` should be used to revive old commits. In this case:

```
$ hg graft -f 291e3a83a122
grafting 262999:291e3a83a122 "Bug 1042619 - Change 'width x height' letter x to × in devtools fronter
merging browser/devtools/canvasdebugger/canvasdebugger.js
merging browser/devtools/layoutview/view.js
merging browser/devtools/responsivedesign/responsivedesign.jsm

$ hg log
changeset:   263004:3c6672d2df85
tag:         tip
parent:      263002:dfc93c68f9c7
user:        Aaron Raimist <aaronraimist@protonmail.ch>
date:        Tue Jan 13 11:59:01 2015 -0800
summary:     Bug 1042619 - Change 'width x height' letter x to × in devtools frontend;r=bgrins
```

As you can see, `hg graft` recreated the original commit. The `merging` lines in the output above indicate that Mercurial invoked its merge resolution algorithm to as part of grafting. What this means is that the listed files were changed between when the commit was originally performed and where the new commit resides. Mercurial was able to automatically merge the differences. Had it not been able to do so, it would entered the merge resolution workflow and asked you to run `hg graft --continue` to finish the graft.

Note: The `-f` in this example is important: it allows grafting of commits that are already ancestors of their destination. Without it, Mercurial sees that you are attempting to recreate a commit that has already been applied and will prevent you from probably shooting yourself in the foot.

If you are familiar with Git, `hg graft` is roughly equivalent to `git cherry-pick`.

Upgrading Repository Storage

Mercurial periodically makes changes to its on-disk storage that require a one-time *upgrade* of repository data to take advantage of the new storage format. Mercurial doesn't do this automatically because the backwards compatibility guarantees of Mercurial say that the version of Mercurial that created a repo should always be able to read from it, even if common repo operations are performed by a newer version.

You have 2 options for upgrading repository storage:

1. Re-clone the repo
2. Run `hg debugupgraderepo`

Upgrading Storage via Clone

A fresh Mercurial clone will usually use optimal/recommended storage for the Mercurial version being used. However, depending on how the clone is performed and where it is cloned from, this may not always work as expected.

To achieve an optimal clone with efficient storage, **always clone from https://hg.mozilla.org/ - not from a local repo**. By cloning from hg.mozilla.org, your clone will inherit the optimal storage used by the server. If you clone from anywhere else, you may inherit sub-optimal storage.

Say you have a copy of *mozilla-central* in a local *mozilla-central* directory. Perform a clone-based upgrade by running the following:

```
# Grab pristine copy of repo.
$ hg clone -U https://hg.mozilla.org/mozilla-central mozilla-central.new

# Copy over .hgrc
$ cp mozilla-central/.hg/hgrc mozilla-central.new/.hg/hgrc

# Pull your unpublished work from your local clone into the new clone.
$ hg --config phases.publish=false -R mozilla-central.new pull mozilla-central

# Now rename/remove your repos as appropriate.
```

Important: That config adjustment for `phases.publish=false` is important. Without it, *draft* changesets will become *public* and Mercurial won't let you edit them. To guard against, it is a good practice to add the following to your per-repo `.hg/hgrc` file immediately after a clone:

```
[phases]
publish = false
```

If you accidentally *publish* your *draft* changesets, you can reset phases by running the following commands:

```
# Reset all phases to draft.
$ hg phase --draft --force -r 0:tip

# Synchronize phases from a publishing repo.
$ hg pull https://hg.mozilla.org/...
```

Upgrading Storage via `debugupgraderepo`

(Requires Mercurial 4.1 or newer)

Upgrading repository storage in-place is relatively easy: just use `hg debugupgraderepo`. This command (which is strictly still an experimental command but shouldn't corrupt your data) essentially does an in-place `hg clone` while applying various data and storage optimizations along the way. **The command doesn't make any permanent changes until the very end and makes a backup of your original data, so there should be a low risk of data loss.**

In its default mode of execution, `hg debugupgraderepo` simply converts storage to the latest storage format: it doesn't heavily process data to optimize it. So, to get the benefits of data optimization (which will shrink the size of the repo and make operations faster), you need to pass some flags to the command.

The first time you upgrade a repo, run as follows:

```
$ hg debugupgraderepo --optimize redeltaparent --optimize redeltamultibase --run
```

`redeltaparent` tells Mercurial to recalculate the internal deltas in storage so a logical parent is used. The first time this runs, it will significantly slow down execution but it can result in significant space savings on a Firefox repos. If

you specify this on a repo where data is already efficiently stored, it is almost a no-op.

`redeltamultibase` tells Mercurial to calculate for merges against both parents and to use the smallest. This always adds significant processing time to repos with lots of merges. It can also drastically reduce the repository size (by several hundred megabytes for Firefox repos).

On a Firefox repository, it could take 2-3 hours to perform data optimizations if the repository isn't already optimized. If you clone from hg.mozilla.org, you will get these optimizations automatically because the server performs them.

Analyzing Pushlog Data Offline

The `mozext` extension replicates pushlog data to a local SQLite database during `hg pull` operations. The extension also exposes some *revsets* and *template* features that allow querying and formatting of pushlog data.

To enable `mozext`, add the following to your hgrc:

```
[extensions]
firefoxtree = /path/to/version-control-tools/hgext/firefoxtree
mozext = /path/to/version-control-tools/hgext/mozext
```

Then, in a Firefox clone, run an `hg pull` to grab pushlog data:

```
$ hg pull central
$ hg pull inbound
$ hg pull autoland
```

Now, you can run revsets to query pushlog data:

```
# Find all changesets initially pushed to autoland
$ hg log -r 'firstpushtree(autoland)' -T '{rev} {node|short} {desc|firstline}\n'

# Find revisions that were heads at time of push (this means they should have CI
# results available)
$ hg log -r 'pushhead(central)'
```

firefoxtree Extension

The `firefoxtree` Mercurial extension enhances the interaction with Firefox repositories.

Background on Multiple Repositories

Firefox developers typically interact with multiple repositories. There is the canonical head of Firefox development, mozilla-central. There are landing repositories such as mozilla-inbound and fx-team. Then there are release repositories like mozilla-aurora, mozilla-beta, and mozilla-release.

All of these repositories share the same initial commit and thus are one logical repository. However, for historical and continuity reasons, the separate topological heads of this conceptual single repository are all stored in separate repositories on Mozilla's servers.

Consolidating the Repositories Locally

Traditionally, Mozilla developers maintain separate clones of each repository. There is thus a one to one mapping between local and remote repositories. For example, you may have separate `mozilla-central` and

`mozilla-inbound` directories/clones to track the different *upstream* repositories. This practice is grossly inefficient. The shared repository data is fetched and stored multiple times. This creates more load for the server, occupies more space on disk, and adds overhead to common tasks such as rebasing from central to inbound.

The *firefoxtree* extension allows you to easily combine the separate remote repositories into a local, single, unified repository.

When you `hg pull` from a known Firefox repository, *firefoxtree* will automatically create a local-only *label* corresponding to the name of the remote repository. These labels will manifest as tags. For example:

```
$ hg pull https://hg.mozilla.org/mozilla-central
pulling from https://hg.mozilla.org/mozilla-central
searching for changes
adding changesets
adding manifests
adding file changes
added 130 changesets with 651 changes to 329 files
(run 'hg update' to get a working copy)

$ hg log -r tip
changeset:   248552:d380166816dd
tag:         central
tag:         tip
user:        ffxbld
date:        Sat Nov 08 03:20:23 2014 -0800
summary:     No bug, Automated blocklist update from host bld-linux64-spot-144 - a=blocklist-update
```

You can see from the output of `hg log` that changeset `d380166816dd` has the `central` *tag* associated with it.

The following example demonstrates how to pull various Firefox repositories into a single local repository and then how to navigate between commits.:

```
$ hg pull https://hg.mozilla.org/integration/mozilla-inbound
pulling from https://hg.mozilla.org/integration/mozilla-inbound
searching for changes
adding changesets
adding manifests
adding file changes
added 34 changesets with 140 changes to 113 files (+1 heads)
(run 'hg heads .' to see heads, 'hg merge' to merge)

$ hg pull https://hg.mozilla.org/integration/b2g-inbound
pulling from https://hg.mozilla.org/integration/b2g-inbound
searching for changes
adding changesets
adding manifests
adding file changes
added 21 changesets with 119 changes to 13 files (+1 heads)
(run 'hg heads .' to see heads, 'hg merge' to merge)

$ hg up central
327 files updated, 0 files merged, 10 files removed, 0 files unresolved

$ hg log -r .
changeset:   248552:d380166816dd
tag:         central
tag:         tip
user:        ffxbld
date:        Sat Nov 08 03:20:23 2014 -0800
summary:     No bug, Automated blocklist update from host bld-linux64-spot-144 - a=blocklist-update
```

```
$ hg up inbound
118 files updated, 0 files merged, 2 files removed, 0 files unresolved

$ hg log -r .
changeset:   248586:e021487d1297
tag:         inbound
user:        Connor <cojojennings@gmail.com>
date:        Wed Oct 29 23:58:03 2014 -0400
summary:     Bug 575094 - Modify how prefservice is accessed so that it's from the parent process and
```

Tip: If you are familiar with Git, it may help to think of these repository labels as *remote refs*.

To view a summary of which repositories are on which changesets, run `hg fxheads`:

```
$ hg fxheads
248607:a7a2bacecce7 b2ginbound tip Bumping manifests a=b2g-bump
248552:d380166816dd central No bug, Automated blocklist update from host bld-linux64-spot-144 - a=blo
246125:c742dcb56135 fx-team Bug 1088729 - Only allow http(s) directory links and https/data images [r
248586:e021487d1297 inbound Bug 575094 - Modify how prefservice is accessed so that it's from the par
```

Tip: The output of `hg fxheads` is only current from the last time you pulled from each repository. Given the frequency of pushes to the Firefox repositories, at least one of your labels will likely be out of date.

Pre-defined Repository Paths

Typically, if you are pulling from multiple remotes, you need to define the names and URLs of those remotes in the `[paths]` section of the repository's `.hg/hgrc` file. The names and URLs of Firefox repositories are well-known, so *firefoxtree* does this for you.

Simply type `hg pull <tree>` to pull from a known Firefox repository. For example:

```
$ hg pull central
$ hg pull inbound
```

Or type `hg push <tree>` to push to a Firefox repository.:

```
$ hg push inbound
$ hg push aurora
```

Tip: The registered name aliases should be intuitive. Try a name of a popular Firefox repository. It should *just work*. If you get stumped or want to see the full list of names, read the source.

The Special *fxtrees* Path

The `fxtrees` path is special. If you `hg pull fxtrees`, firefoxtree will pull from all Firefox repositories that you have previously pulled from.

Typically, developers only care about a subset of all the Firefox repositories. `hg pull fxtrees` provides a convenient mechanism to only pull the repositories you have previously expressed an interest in interacting with.

Other Special Paths

There are some special paths that expand to multiple repositories. If you run `hg pull` with one of these paths, firefoxtree will iterate through each of these repositories and pull from all of them. These special paths include:

integration Pull from all integration/landing repositories (inbound, fx-team, b2g-inbound)

releases Pull from all repositories that Firefox is released from (central, aurora, beta, release, esr, etc)

Safer Push Defaults

The default behavior of `hg push` is to want to transfer all non-remote changesets to the remote. In other words, if you have pulled mozilla-central and mozilla-aurora into the same repository and you `hg push ssh://hg.mozilla.org/mozilla-central`, Mercurial will want to transfer all of mozilla-aurora's changesets to central!

The way you are supposed to do this is to always pass a `--rev` or `-r` argument to `hg push` to tell Mercurial exactly what changesets to push. Commonly, you want to push the working copy's commit, so the command to use would be `hg push -r . <remote>`.

Since `hg push -r .` is almost always what is wanted when pushing to a Firefox repository, *firefoxtree* automatically changes `hg push` to behave like `hg push -r .` when pushing to a Firefox repository.

Working with Unified Repositories and Repository Labels

Astute readers may have noticed that Mercurial is reporting the repository labels as *tags*. However, they don't behave like *tags*. The `.hgtags` file is not updated and `hg push` won't transfer them. Under the hood, the extension is using an extension-only feature of Mercurial to supplement the tags list. The labels are being reported as tags, but have almost nothing to do with actual tags.

The repository labels can only be modified by *firefoxtree*. Furthermore, they are only modified when running `hg pull`. Unlike bookmarks or branches, user actions such as committing will **not advance the labels**.

Working With a Unified Firefox Repository

Traditionally, the various Firefox repositories (`mozilla-central`, `mozilla-inbound`, `mozilla-aurora`, etc) have been separate repositories. They all share the same root commit (`8ba995b74e18334ab3707f27e9eb8f4e37ba3d29`), so it is possible to combine them locally. This has several benefits:

- You can easily switch between heads using `hg up`
- You can easily compare changes across different heads using `hg log`, `hg diff`, and other tools.
- Landing a patch is as simple as `hg rebase`.
- You only have to fetch the data associated with each commit exactly once (with separate repositories, you transfer down each commit *N* times).

Unified Repository on hg.mozilla.org

`https://hg.mozilla.org/mozilla-unified` is a *read-only* unified repository containing all of the commits on the `default` branch of the various Firefox repositories (mozilla-central, inbound, aurora, beta, release, esr, etc) in chronological order by push time.

Advantages of the Unified Repo

If you pull from multiple Firefox repositories or maintain multiple clones, pulling from the unified repository will be faster and require less local storage than pulling from or cloning *N* repositories.

The unified repository **does not include extra branches**, notably the `*_RELBRANCH` branches. If you've ever pulled the mozilla-beta or mozilla-release repositories, you know how annoying the presence of these branches can be.

The unified repository features **bookmarks that track each canonical repository's head**. For example, the `central` bookmark tracks the current head of mozilla-central. If you naively pulled all the Firefox repositories into a local Mercurial repository, you would have multiple *anonymous* heads on the `default` branch and you wouldn't know which head belonged to which Firefox repository. The bookmarks solve this problem.

In a nutshell, the unified repository solves many of the problems with Firefox's multi repository management model in a way that doesn't require client-side workarounds like the *firefoxtree extension*.

Working with the Unified Repo (Vanilla Mercurial)

Here is the basic workflow for interacting with the unified repo when using a vanilla Mercurial configuration (no `firefoxtree` extension).

First, clone the repo:

```
$ hg clone --uncompressed https://hg.mozilla.org/mozilla-unified
```

Update to a bookmark you want to base work off of:

```
$ hg up central
42 files updated, 0 files merged, 0 files removed, 0 files unresolved
(activating bookmark central)
```

Then start a new bookmark to track your work:

```
$ hg bookmark myfeature
```

Then make changes and commit:

```
<edit some files>
$ hg commit
```

If you want to rebase:

```
$ hg pull
$ hg rebase -b myfeature -d central
```

Be sure you've activated your own bookmark or deactivated the Firefox bookmark before committing or you may move the bookmark from the server. The easiest way to do this is:

```
$ hg up .
(leaving bookmark central)
```

Tip: Facebook's scm-prompt.sh implements shell prompt integration for both Mercurial and Git. It displays the currently active bookmark, which is useful to prevent accidentally committing on bookmark belonging to a Firefox repo.

Working with the Unified Repo (firefoxtree)

If you have the `firefoxtree` extension installed, the behavior of the unified repository changes slightly. Instead of pulling bookmarks, the `firefoxtree` extension stores labels of the same name in a read-only namespace instead.

Not using bookmarks means you can't accidentally update your local bookmarks corresponding to Firefox repo state and drift out of sync with reality. It also means you don't have to activate and deactivate bookmarks locally: enabling you to engage in simpler workflows.

Here is how workflows typically look like with `firefoxtree` installed.

First, clone the repo:

```
$ hg clone --uncompressed https://hg.mozilla.org/mozilla-unified
```

Update to the repo/head you want to work on:

```
$ hg up central
42 files updated, 0 files merged, 0 files removed, 0 files unresolved
(activating bookmark central)
```

Optionally create a new bookmark to track your work:

```
$ hg bookmark myfeature
```

Then make changes and commit:

```
<edit some files>
$ hg commit
```

If you want to rebase:

```
$ hg pull
$ hg rebase -b myfeature -d central
```

The `firefoxtree` extension will also print the number of new commits to each repo since last pull.:

```
$ hg pull
pulling from https://hg.mozilla.org/mozilla-unified
searching for changes
adding changesets
adding manifests
adding file changes
added 39 changesets with 309 changes to 235 files
updated firefox tree tag beta (+2 commits)
updated firefox tree tag esr52 (+2 commits)
updated firefox tree tag inbound (+34 commits)
updated firefox tree tag release (+1 commits)
(run 'hg update' to get a working copy)
```

generaldelta and the Unified Repo

The unified repository is encoded using Mercurial's *generaldelta* storage mechanism. This results in smaller repositories and faster repository operations.

Important: Mercurial repositories created before Mercurial 3.7 did not use generaldelta by default. Pulling from the repository to a non-generaldelta clone will result in **slower** operations.

It is highly recommended to create a new clone of the unified repository with Mercurial 3.7+ to ensure your client is using generaldelta.

To check whether your existing Firefox clone is using generaldelta:

```
$ grep generaldelta .hg/requires
```

If there is no `generaldelta` entry in that file, you will need to create a new repo that has generaldelta enabled. **Adding ‘‘generaldelta‘‘ to the requires file does not enable generaldelta on an existing repo, so don’t do it.** See *Upgrading Repository Storage* for instructions on how to do this.

incompatible Mercurial client; bundle2 required

Does this happen to you?:

```
$ hg clone https://hg.mozilla.org/mozilla-unified firefox
requesting all changes
abort: remote error:
incompatible Mercurial client; bundle2 required
(see https://www.mercurial-scm.org/wiki/IncompatibleClient)
```

This message occurs when the Mercurial client is not speaking the modern *bundle2* protocol with the server. For performance reasons, we require *bundle2* to clone or pull the unified repository. This decision is non-negotiable because removing this restriction could result in excessive CPU usage on the server to serve data to legacy clients.

If you see this message, one of the following is true:

- Your Mercurial client is too old. You should *upgrade*.

Uplifting / Backporting Commits

Often times there are commits that you want to uplift to other projects branches. e.g. a commit from `mozilla-central` should be uplifted to `mozilla-aurora`. This operation is typically referred to as a *backport* or a *cherry-pick*.

The `hg graft` command should be used to perform these kinds of operations.

Say you wish to backport `77bbac61cd5e` from *central* to *aurora*.:

```
# Ensure your destination repository is up to date.
$ hg pull aurora
...

# Update to the destination where commits should be applied.
$ hg up aurora

# Perform the backport.
$ hg graft -r 77bbac61cd5e
```

When `hg graft` is executed, it will essentially *rebase* the specified commits onto the target commit. If there are no merge conflicts or other issues, it will commit the new changes automatically, preserving the original commit message.

If you would like to edit the commit message on the new commit (e.g. you want to add `a=`), simply add `--edit`:

```
$ hg graft --edit -r 77bbac61cd5e
```

If Mercurial encounters merge conflicts during the operation, you’ll see something like the following:

```
$ hg graft -r 77bbac61cd5e
warning: conflicts during merge.
merging foo incomplete! (edit conflicts, then use 'hg resolve --mark')
abort: unresolved conflicts, can't continue
(use hg resolve and hg graft --continue)
```

Read Mercurial's conflict docs for how to resolve conflicts. When you are done resolving conflicts, simply run `hg graft --continue` to continue the graft where it left off.

If you wish to backport multiple commits, you can specify a range of commits to process them all at once:

```
$ hg graft -r 77bbac61cd5e::e8f80db57b48
```

Tip: `hg graft` is superior to other solutions like `hg qimport` because `hg graft` will perform a 3-way merge and will use Mercurial's configured merge tool to resolve conflicts. This should give you the best possible merge conflict outcome.

Maintaining Multiple Checkouts With a Unified Repository

Developers often maintain multiple checkouts / working directories of Firefox. For example, you may do all your day-to-day work on `mozilla-central` but also have a `mozilla-beta` checkout around for testing patches against Firefox Beta.

A common reason why developers do this is because updating to different commits frequently requires a build system clobber. This is almost always true when updating between different Gecko versions.

Some people may say *I prefer maintaining separate clones because it means I don't have to clobber as often.* What they are really saying is *I want to maintain separate working directories that are independent.*

The solution to use is to use `hg share`. `hg share` allows you to create a new working copy of a repository that *shares* the backing repository store with another.

Add the following to your Mercurial configuration file:

```
[extensions]
share =
```

Then, create a shared store as follows:

```
$ hg share /path/to/existing/clone /path/to/new/checkout
```

Now, you can `hg up` inside both repositories independently! If you commit to one, that commit will be available in the other checkouts using that shared store.

Tip: Mercurial 3.3 and newer support sharing bookmarks with repositories created with `hg share`. To activate bookmark sharing, you'll need to add `-B` to `hg share`. e.g. `hg share -B existing new-checkout`

Caution: Users of MQ should exercise extreme caution when using shared stores.
MQ operates at a low-level in Mercurial: every MQ operation is essentially creating or deleting commits from the store. Deleting commits from large repositories like Firefox's can be a very expensive operation. You not only pay a penalty at operation time, but all the shared repositories may have expensive computations to perform the next time the repository is accessed.
MQ users are advised to not use `hg share`.
MQ users are advised to switch to head/bookmark-based development to avoid these limitations.

Firefox Workflow

This article outlines the **recommended** workflow for interacting with the Firefox repository (mozilla-unified).

Optimally Configure Mercurial

When you run `bootstrap.py` or `mach bootstrap` (if you already have a clone), the bootstrapper will prompt you to run a Mercurial configuration wizard. You should run this wizard and make sure it is happy about your Mercurial state.

You should also run *mach bootstrap* periodically to ensure Mercurial support files are up-to-date.

Important: The instructions in this article assume the *firefoxtree extension* is installed. Please activate it when the wizard prompts you to!

Cloning the Repository

Clone the Firefox repository by running:

```
$ hg clone https://hg.mozilla.org/mozilla-unified firefox
$ cd firefox
```

Feature Development

So you want to start work on a new Firefox feature? This section is for you.

Start by obtaining the latest code so you aren't working on old and possibly stale code:

```
$ hg pull
```

Then update to the tip of `mozilla-central`:

```
$ hg up central
```

Now, change some stuff. We assume you know how to do this.

Commit your changes:

```
$ hg commit
<type commit message in editor>
```

Make more changes and keep committing:

```
$ hg commit
<another commit message>
```

Push your changes to MozReview to initiate code review:

```
$ hg push review
```

Important: We assume you've followed the mozreview_user to configure MozReview.

OK. Progress on that feature is blocked waiting on review. It could take a while for that to happen. Let's start working on something else. We always start by pulling the latest code so our change isn't out of date before we've event started.:

```
$ hg pull
$ hg up central
<change stuff>
$ hg commit
<change stuff>
$ hg commit
$ hg push review
```

Changing Code After Reviews

A review comes back. Unfortunately review was not granted and you need to make changes. No worries.

We need to update the working directory to the changeset to be modified so we can edit them. How you do this depends on how you are *tracking* commits. For most workflows, we recommend `hg wip` (see *Workflows* for more on this command) to find them. e.g.:

```
$ hg wip
o   6139:5060abe260e9 gps  vcsreplicator: explicitly record obsolescence markers in pushkey messages
: o   5940:e16f6960cdeb gps  hgmo: update automationrelevance for Mercurial 3.8; r?smacleod
: o   5939:e62f4eb60ef3 gps  mozhg: fix test output for Mercurial 3.8; r?glob
: o   5938:f71be022e59c gps  global: upgrade Mercurial to 3.8.3 (bug 1277714)
:/
o   5937:5a8623230b7a gps  pushlog: convert user and nodes to bytes (bug 1295724); r=smacleod
: o   5717:9c2ca05479e9 gps  hgmo: handle obsolete changesets (bug 1286426); r?glandium
```

If you want to edit `e16f6960cdeb`, you would `hg up e16f6960cdeb`.

Or if you are using bookmarks, you can update directly to the bookmark:

```
$ hg up my-bookmark
4 files updated, 0 files merged, 0 files removed, 0 files unresolved
(activating bookmark my-bookmark)
```

Now that your working directory is updated, you can start making changes.

You have several options available to you. If you know the changes are small and won't conflict if reordered, go ahead and make them now and commit:

```
<make changes>
$ hg commit
<make more changes>
$ hg commit
```

Then squash the changesets together:

```
$ hg histedit
```

Note: For `hg histedit` to work without arguments, you'll need Mercurial 3.7 or newer.

You'll then need to:

1. Reorder your *fixup changesets* to occur immediately after (below) the changesets they will be modifying.
2. Set the action on these *fixup changesets* to `roll` so they are fully absorbed into the changeset that came before.

Alterantively, you can edit changes directly. Again, use `hg histedit`. But this time, change the action of the changesets you want to modify to `edit`. Mercurial will print some things and will leave you with a shell. The *working directory* will have been updated to the state of the commit you are editing. If you run `hg status` or `hg`

`diff` you will see that this changesets's changes are applied to files already. Make your changes to the files then run `hg histedit --continue` to continue with the history editing.

Note: Advanced users can use the *evolve extension <https://bitbucket.org/marmoute/mutable-history>* to edit changesets in place. Because this is still an experimental feature, it isn't documented here.

Once all the changes are made, you'll want to submit for review again:

```
$ hg push review
```

Then we're back to waiting.

Autolanding

You finally get review and can land your changes!

The easiest way to do this is through the use of Autoland. You can access Autoland through the `Land Commits` option of the `Automation` menu. Clicking this button displays a dialog containing a list of commits to be landed. MozReview will attempt to automatically rewrite the commit messages to reflect who reviewed which commit. If everything looks good, click the `OK` button and the autolander will land your commits for you.

Autoland will attempt to rebase you commits on the head of the `autoland` repo for you automatically. If it can't do this (say there was a file merge conflict during the base), an error will (eventually) be displayed in MozReview and you will have to rebase yourself and push the result back to MozReview and try the autoland request again.

Note: Only landing to the `autoland` repo is supported. This is because we will be removing *integration repos* in the future so the history of mozilla-central isn't linear and free of merge commits.

If Autoland succeeds, *Pulsebot* will comment in your bug that your changes have landed. Unfortunately, there is not currently any notification that Autoland has failed outside of MozReview, so if the trees are open and your changes have not landed within a few minutes, please check back in MozReview to see if any errors have occurred.

Manual Reviewer Attribution and Landing

Unable to use Autoland? Follow these instructions.

Update to the tip-most changeset that will land (often a head) after finding the changesets using the technique in the previous section:

```
$ hg up <SHA-1 or label>
```

Before landing, we need to rebase our unlanded changesets on top of the latest changeset from an integration branch:

```
$ hg pull
$ hg rebase -d inbound
```

If you need to add `r=` reviewer attribution to the commit message, do that now:

```
$ hg histedit
```

Change the action to `m` for all the changesets and proceed to update commit messages accordingly.

And finally we land:

```
$ hg push -r . inbound
```

Using Mercurial in Automation

Are you looking to consume Mercurial as part of an automated system? This article attempts to answer common questions and to suggest best practices.

Best Practices

Read the Scripting Help Topic

`hg help scripting` (introduced in Mercurial 3.5) gives a Mercurial generic overview of how machines should consume Mercurial.

Get Design and Code Review from Someone Who Knows Mercurial

Before you write any code or deploy anything, it might be a good idea to get design review or code review from someone who knows Mercurial.

Pop in `#vcs` on `irc.mozilla.org` or send an email to dev-version-control@lists.mozilla.org.

Consider Using the Command Server

Mercurial has a *command server* mode where a persistent `hg` process is created and individual commands are dispatched to the server one at a time.

There exists client libraries for communicating with this command server python-hglib is the Python client and it can be installed via `pip install python-hglib`.

Use of the command server is preferred over direct `hg` process invocation because:

1. It is faster.
2. A written client API will handle dispatching and parsing responses automatically, freeing you up to write meaningful code.

Python + hg process startup overhead is non-trivial (~50 ms). If you are performing many Mercurial commands, use of the command server could have a profound impact on performance.

Use Templates

Most Mercurial commands accept a `--template/-T` argument to control how output is formatted. You should use this capability when calling commands to ensure output is exactly how you intend. This can often result in making the output easier for machines to parse.

Tip: Use `-T json` to produce JSON output from commands.

Add –traceback to All Commands

In the rare case Mercurial crashes, it is valuable to have crash information to help with debugging. Adding `--traceback` to all commands will have Mercurial print a stack trace when it crashes.

As an alternative to adding `--traceback` to every command, add the following to your hgrc:

```
[ui]
traceback = on
```

Always Check Exit Codes

Always check that the exit code from `hg` commands is what is expected (probably `0`).

Specify an Explicit hgrc

Mercurial will automatically inherit the system-wide and per-user `hgrc` files. This can have unintended consequences, such as the enabling of an extension or defining of specific user credentials.

When invoking the `hg` command, set the `HGRCPATH` environment variable to that of a known good config file.

Tip: To disallow loading of an external hgrc file, set `HGRCPATH` to the value `/dev/null`.

Note: If executing from within a Mercurial repository, the `hg` process will automatically load the `.hg/hgrc` file. This is not always intended, especially if you are running commands that don't interact with a specific repository.

To prevent this, execute your `hg` process from outside any Mercurial repository. e.g. `/`.

You can always pass `-R /path/to/repo` to have Mercurial operate on a specific repository - you don't need to have `cwd` be from within the repository.

Other Tips

Debugging

When debugging Mercurial commands, consider adding `--verbose` or `--debug` to the command invocation to get Mercurial to print more information about what it is doing. This output can be especially when reporting bugs.

Like `--traceback`, these options can be enabled via `hgrc` files:

```
[ui]
debug = True
verbose = True
```

Reporting Issues with Mercurial

Are you having a bad experience with Mercurial or with hg.mozilla.org? Read on to learn how to report it and (hopefully) get it resolved.

Mercurial and Mozilla

The Mercurial project cares about Mozilla because the Firefox repository is one of the largest open source Mercurial repositories both in terms of repository size and number of users. Because of that scale, Mozillians tend to notice issues with Mercurial before others. As such, the Mercurial project is very keen to learn about and address the problems Mozillians may have.

How to Report Issues

With hg.mozilla.org

Notice something weird on hg.mozilla.org (including performance problems)? Please file a bug against Developer Services :: hg.mozilla.org.

If you want to talk with someone before filing a bug, hop in to #vcs on irc.mozilla.org. That is a low traffic channel and your question should be answered eventually.

With Core Mercurial

Found a bug in Mercurial? Have a performance concern? File a bug in Mercurial's Bugzilla. Choose the *Mercurial* component if you are unsure what component to use.

Before filing bugs, ensure you are using the latest Mercurial release. If not, the first thing people will ask you is to try to reproduce with the latest release.

When filing bugs, practice good bug filing etiquite and try to include steps to reproduce.

Tip: Many Mercurial developers have a copy of mozilla-central for performance testing. Bug reports that reference mozilla-central (or any public repository for that matter) are acceptable.

If you are unsure whether something is a bug, hop in to #mercurial on the Freenode IRC network and ask around. Or, post to one of the Mercurial mailing lists.

With a Mozilla Extensions

Mozillians have authored a handful of Mercurial extensions. If you find a bug in one, file a bug against that extension's Bugzilla component. These components all exist in the *Developer Services* product on bugzilla.mozilla.org.

Before filing bugs, ensure you are using the latest Mercurial release and that your version-control-tools repo (the repo containing most of the extensions) is fully up to date. Otherwise, you may be reporting a bug that's been fixed already.

For the Impatient

In a hurry and don't want to spend the time to report a proper bug? Ping *gps* on irc.mozilla.org and he'll do the right thing if he can.

Ways to Make Bug Reports More Useful

All Mercurial commands accept the following options, which may prove useful when reporting bugs:

`--debug` When this option is present, Mercurial will print debug info to output. This may aid debugging hangs and other issues.

`--traceback` When this option is present, Mercurial will dump its execution stack when aborted. Using –traceback plus ctrl+c is a good way to see which method is spinning the CPU.

`--profile` Profile Mercurial execution and print a summary after execution. This is useful for debugging performance issues to see where Mercurial is spending most of its time.

Including the output with one or more of these options can make bugs reports (especially performance issues) much more meaningful.

Another tool to aid debugging is the *blackbox* extension. Simply add the following to your hgrc:

```
[extensions]
blackbox =

[blackbox]
track = *
maxsize = 10 MB
maxfiles = 2
```

A `.hg/blackbox.log` file will now exist in each repository. This log will capture forensic details that may aid debugging and performance analysis.

How Not to Report Issues

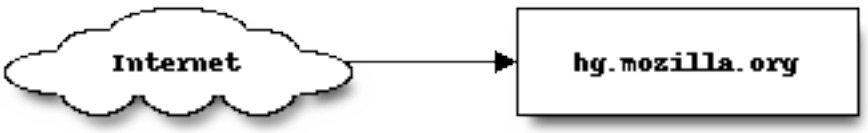
Please do not complain about issues (e.g. on #developers) without telling someone who can do something about it. Otherwise, you have effectively complained to a black hole and your problems will likely persist because someone empowered to do something about them doesn't know of them.

If you see something, file something! Don't be just a complainer: be an enabler.

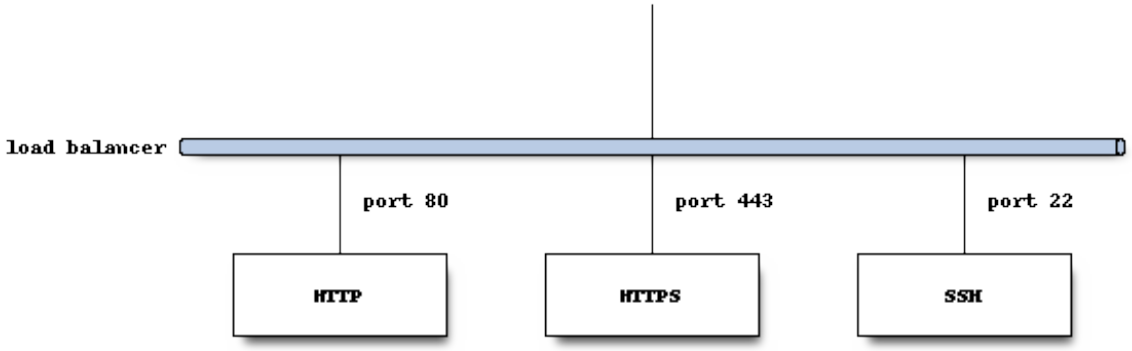
hg.mozilla.org

Architecture

hg.mozilla.org is an Internet connected service.

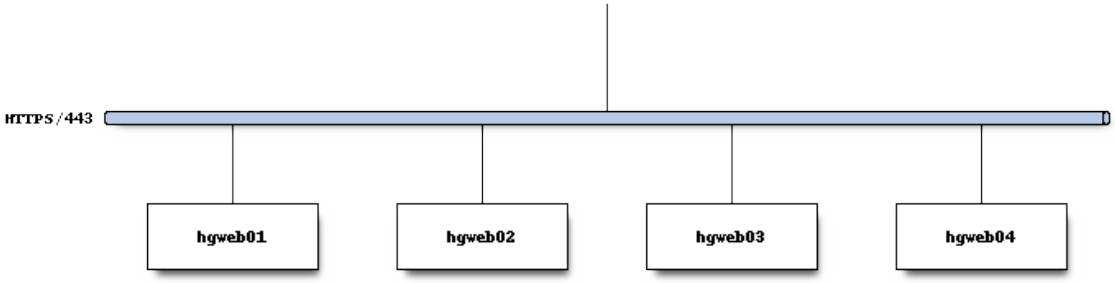


hg.mozilla.org exposes HTTP and SSH service endpoints all running on standard ports. Connections to these services connect to a load balancer.



HTTP requests to port 80 are **always** HTTP 301 redirected to https://hg.mozilla.org/ and come back in via port 443.

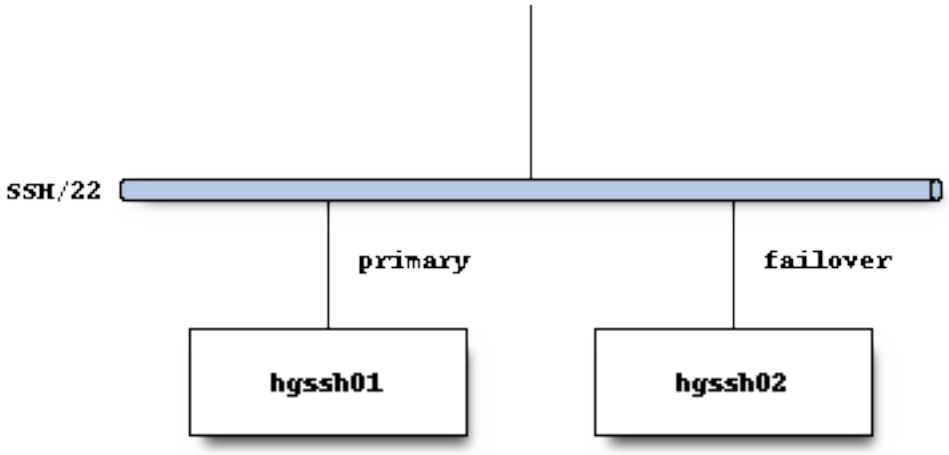
HTTPS connections to port 443 are routed to a pool of read-only mirrors. We call these the *hgweb* servers.



(Server names are representative.)

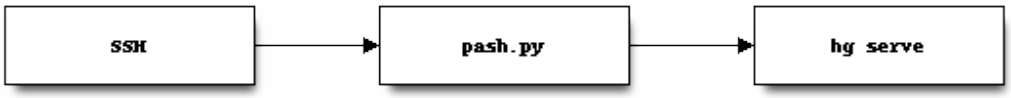
Each server is identically configured and behaves like the others. HTTP connections/requests are routed to any available *hgweb* server.

SSH connections to port 22 are routed to a single *primary* server. There is a warm standby for the *primary* server that is made active should the primary server fail. We collectively refer to these as the *hgssh* servers.

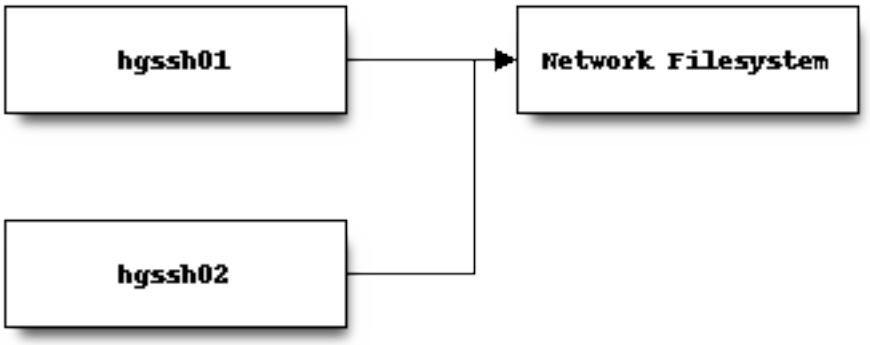


SSH connections use public key authentication. User access control and storage of SSH public keys is stored in an LDAP server.

An authenticated SSH connection spawns the `pash.py` script from the version-control-tools repository. This script performs additional checking of the incoming *request* before eventually spawning an `hg serve` process, which allows the Mercurial client to communicate with a repository on the server.



The *hgssh* servers hold the canonical repository data on a network appliance exporting a mountable read-write filesystem.



All repository write operations are performed via SSH and are handled by the *primary hgssh* server.

Various Mercurial hooks and extensions run on the *hgssh* server when repository events occur. Some of these verify incoming changes are acceptable and reject them if not. Others perform actions in reaction to the incoming change.

In terms of service architecture, the most important action taken in reaction to pushes is writing events into the replication system.

Repository Replication Mechanism

See *Replication* for the architecture of the replication system.

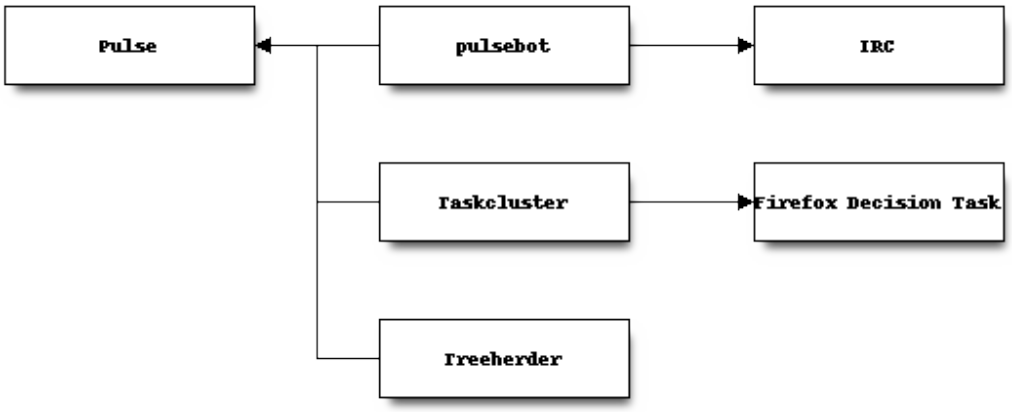
Notifications

As described in the replication documentation, there exists a `pushdataaggregator` Kafka topic exposing a stream of fully-replicated repository change messages.

Various services on the active primary *hgssh* server consume this topic and do things with the messages. One consumer notifies Pulse of repository changes. Another sends messages to AWS SNS.

See *Change Notifications* for more.

From there, various other services not part of the hg.mozilla.org infrastructure react to events. For example, *pulsebot* creates IRC notifications, Taskcluster schedules Firefox CI, and Treeherder records the push.



HTTP Repository Serving

Nearly every consumer of hg.mozilla.org consumes the service via HTTP: only pushes should be using SSH.

The HTTP service is pretty standard for a Python-based service: there's an HTTP server running on each *hgweb* server that converts the requests to WSGI and sends them to a Python process running Mercurial's built-in *hgweb* server. *hgweb* handles processing the request and generating a response.

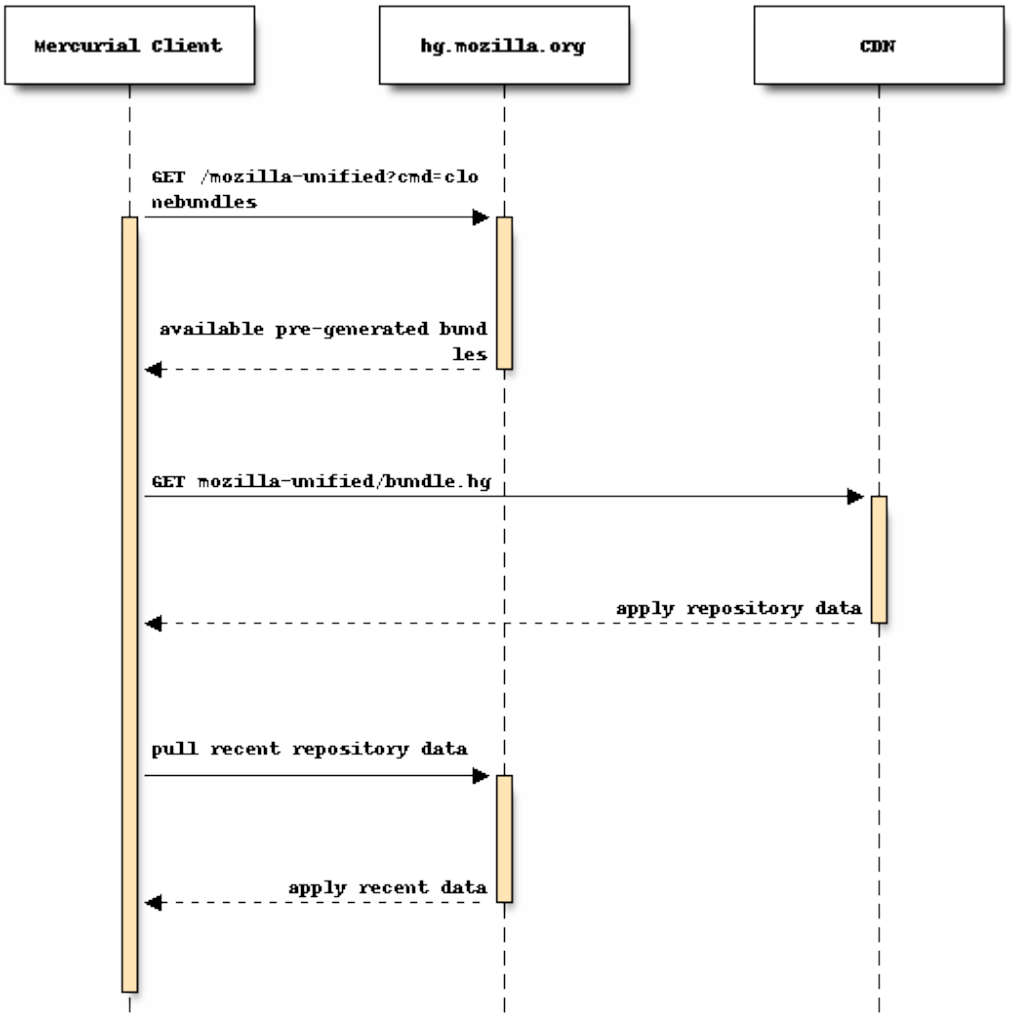
hgweb serves mixed content types (HTML, JSON, etc) for web browsers and other agents. In addition, *hgweb* also services Mercurial's custom *wire protocol* for communicating with Mercurial clients.

When a client executes an `hg` command that needs to talk to hg.mozilla.org, the client process establishes an HTTP connection with hg.mozilla.org and issues commands to a repository there via HTTP. Run `hg help internals.wireproto` for details of how this works.

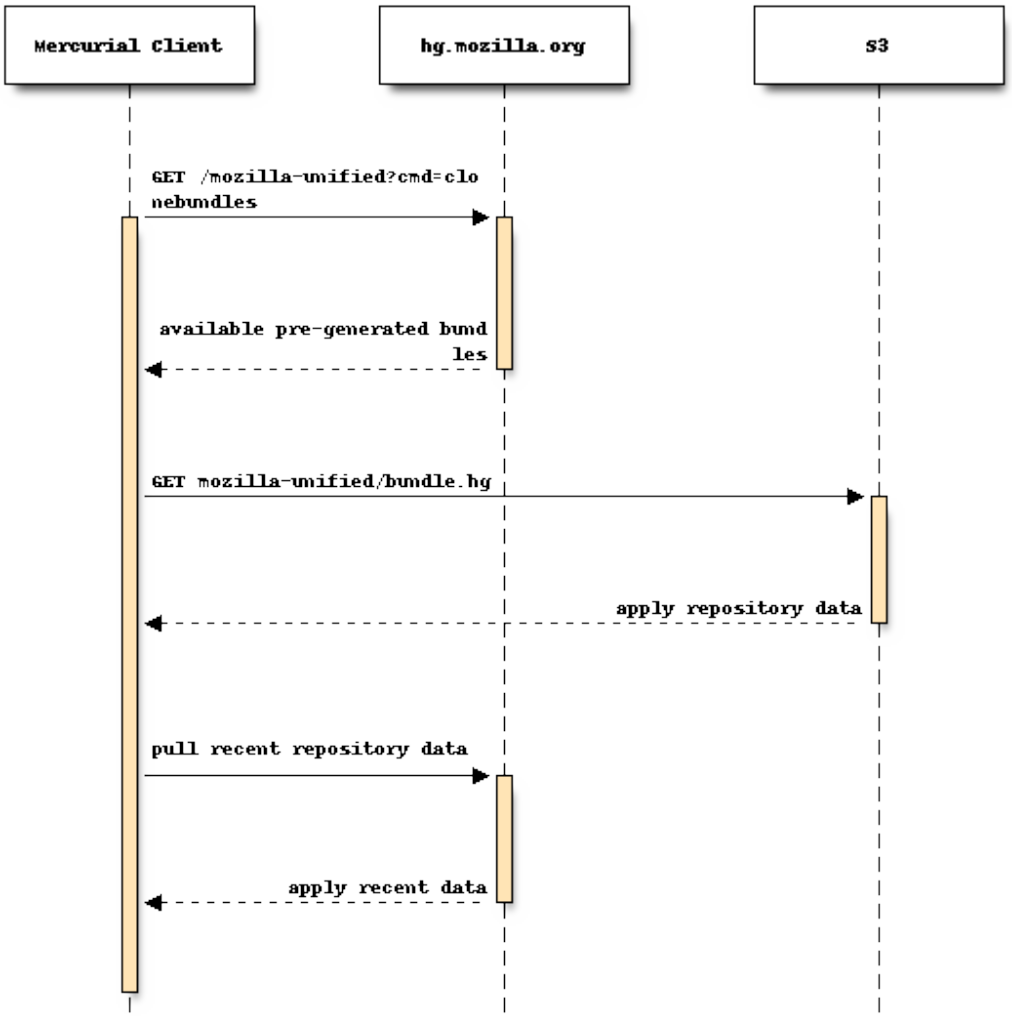
Clone Offload

In order to alleviate server-side CPU and network load, frequently accessed repositories on hg.mozilla.org use Mercurial's *clone bundles* feature so `hg clone` operations download most repository data from a pre-generated static *bundle* file hosted on a scalable HTTP server.

Most Mercurial clients will fetch a bundle from the CloudFront CDN.



If the client can't connect to CloudFront (requires SNI) or if the client is connecting from an AWS IP belonging to an AWS region where we have an S3 bucket containing repository data, the client will connect to S3 instead.



Offloading the bulk of expensive `hg clone` operations to pre-generated files hosted on highly scalable services results in faster clones for clients and drastically reduces the server requirements for the hg.mozilla.org service.

See *Cloning from Pre-Generated Bundles* and the Cloning from S3 blog post for more on *clone bundles*.

Custom APIs

In addition to the APIs and services that Mercurial provides out-of-the-box, hg.mozilla.org offers a handful of custom APIs for services to consume.

Pushlog

The Pushlog records when pushes are made to a repository and who made them. See *Pushlog* for more.

Firefox Releases

Firefox repositories are able to expose information about the set of Firefox releases tied to Mercurial changesets in that repository. See *Firefox Release Data* for more.

Repository Metadata

The `repoinfo` web command provides additional information about a repository, such as the group membership required to push to that repository.

See e.g. https://hg.mozilla.org/mozilla-central/repoinfo and https://hg.mozilla.org/mozilla-central/json-repoinfo.

Automation Relevance

The `automationrelevance` web command provides information on what changesets are *relevant* to automated consumers given a source changeset.

URLs have the form `<repo>/json-automationrelevance/<changeset>`. e.g. https://hg.mozilla.org/integration/autoland/json-automationrelevance/67342f5762dd6e7ea3a783876cd9d35517ff3386.

The data contains a set of changesets and corresponding metadata (such as pushlog info and the set of files changed). The *relevant* changesets are the changesets that should be used to influence scheduling of tasks that examine the correctness of those changesets.

For most repositories, the *relevant* set consists of changesets in the same push as the queried changeset.

For special repositories (notably the *try* repositories), the *relevant* set is supplemented by non-public ancestors of the queried changeset. e.g. if you push a series of changesets to Try and then iterate on just the last changeset, the *relevant* set on subsequent pushes will include the base changesets from previous push(es).

Pushlog

Mozilla has taught Mercurial how to record who pushes what where and when. This is called the *pushlog*. It is essentially a log of pushes to repositories.

Technical Details

All pushes to `hg.mozilla.org` occur via SSH. When clients talk to the server, the authenticated username from SSH is stored in the `USER` environment variable. When a push occurs, our custom `pushlog` Mercurial extension will record the username, the current time, and the list of changesets that were pushed in a SQLite database in the repository.

Installing

The *pushlog* extension (source in `hgext/pushlog`) contains the core data recording and data replication code. When installed, a `pretxnchangegroup` hook inserts pushlog entries when changesets are introduced. To install this extension, add the following line to your hgrc:

```
[extensions]
pushlog = /path/to/version-control-tools/hgext/pushlog
```

No additional configuration is necessary.

The web components for pushlog are separate from the core extension and require a bit more effort to configure. This code lives in `hgext/pushlog/feed.py`. It is our intention to eventually aggregate this code into a single pushlog extension so there is a unified pushlog experience.

The web component will require the following extension:

```
[extensions]
pushlog-feed = /path/to/version-control-tools/hgext/pushlog/feed.py
```

`pushlog/feed.py` exposes some hgweb endpoints that expose pushlog data.

Templates

It isn't enough to activate the `pushlog/feed.py` extension: you'll also need to configure some Mercurial theming to render pushlog data.

The Atom output will require the existence of an `atom` style. You are encouraged to copy the files in `hgtemplates/atom` to your Mercurial styles directory.

The `pushloghtml` page will render the `pushlog` template. This is something you'll need to define. Look for `pushlog.tmpl` files in `hgtemplates/` in this repository for examples.

Pushlog templates typically make use of a named `pushlogentry` entity. You may also need to define this. Searching for `pushlog` in `hgtemplates` to find all references is probably a good idea.

Pushlog Wire Protocol Command

The `pushlog` extension exposes a `pushlog` command and capability to the Mercurial wire protocol. This enables Mercurial clients to retrieve pushlog data directly from the wire protocol.

For more details, read the source in `hgext/pushlog/__init__.py`.

The Push ID

Entries in the pushlog have an incrementing integer key that uniquely identifies them. It is guaranteed that push ID `N + 1` occurs after `N`.

hgweb Commands

There are a couple custom hgweb commands that expose pushlog information.

For reference, an *hgweb command* is essentially a per-repository handler in hgweb (Mercurial's HTTP interface). URLs have the form `https://hg.mozilla.org/<repository>/<command>/<args>`.

json-pushes Command

The `json-pushes` command exposes JSON representation of pushlog data.

pushlog Command

The `pushlog` command exposes an ATOM feed of pushes to the repository.

It behaves similarly to `json-pushes` in terms of what parameters it can accept.

pushloghtml Command

The `pushloghtml` command exposes HTML show pushlog data.

Query Parameters

Various hgweb pushlog commands accept query string parameters to control what data is returned.

The following parameters control selection of the lower bound of pushes. Only 1 takes effect at a time. The behavior of specifying multiple parameters is undefined.

startdate A string defining the start date to query pushes from. Only pushes after this date (non-inclusive) will be returned.

fromchange Only return pushes that occurred after the push that introduced this changeset. The value can be any changeset identifier that Mercurial can resolve. This is typically a 40 byte changeset SHA-1.

startID Only return pushes whose ID is greater than the integer specified.

The following parameters control selection of the upper bound of pushes. Behavior is similar to the parameters that control the lower bound.

enddate A string defining the end date for pushes. Only pushes before this date (non-inclusive) will be returned.

tochange Only return pushes up to and including the push that introduced the specified changeset.

endID Only return pushes up to and including the push with the specified push ID.

Only parameters that control behavior include:

user Only show pushes performed by the specified user.

changeset Only show pushes that introduced the specified changeset.

tipsonly If the value is `1`, only return info from the tip-most changeset in the push. The default is to return info for all changesets in a push.

full If this parameter is present (the value is ignored), responses will contain more verbose info for each changeset.

version Format of the response. `1` and `2` are accepted. `1` is the default (for backwards compatibility).

This is only used by `json-pushes`.

Dates can be specified a number of ways. However, using seconds since UNIX epoch is highly preferred.

JSON Payload Formats

Version 1 Version 1 (the default) consists of a JSON object with keys corresponding to push IDs and values containing metadata about just the push. e.g.:

```
{
  "16": {
    "changesets": [
    "91826025c77c6a8e5711735adaa9766dd4eac7fc",
```

```
    "25f2a69ac7ac2919ef35c0b937b862fbb9e7e1f7"
   ],
   "date": 1227196396,
   "user": "gszorc@mozilla.com"
  }
}
```

An optional `obsoletechangesets` key may also be present in each push. Read below for more.

Version 2 Version 2 introduces a container for pushes so that additional metadata can be communicated in the main object in the payload. Here is an example payload:

```
{
  "lastpushid": 21,
  "pushes": {
    "16": {
      "changesets": [
        "91826025c77c6a8e5711735adaa9766dd4eac7fc",
        "25f2a69ac7ac2919ef35c0b937b862fbb9e7e1f7"
      ],
      "date": 1227196396,
      "user": "gszorc@mozilla.com"
    }
  }
}
```

The top-level objects contains the following properties:

pushes An object containing push information.

This is the same object that constitutes version 1's response.

lastpushid The push ID of the most recent push known to the database.

This value can be used by clients to determine if more pushes are available. For example, clients may query for N changesets at a time by specifying `endID`. The value in this property can tell these clients when they have exhausted all known pushes.

Push Objects The value of each entry in the pushes object is an object describing the push and the changesets therein.

The following properties are always present:

changesets An array of changeset entries.

By default, entries are 40 character changeset SHA-1s included in the push. If `full` is specified, entries are objects containing changeset metadata (see below).

Changesets are in DAG/revlog order with the tip-most changeset last.

The array may be empty. This can occur if changesets from this push are now hidden/obsolete.

obsoletechangesets (optional) An array of 40 character changeset SHA-1s of now obsolete changesets included in the push.

The DAG order relationship between `changesets` and `obsoletechangesets` is strictly speaking undefined.

This key is only present if the repository has obsolescence data and the push has changesets that are now obsolete.

date Integer seconds since UNIX epoch that the push occurred.

For pushes that take a very long time (more than a single second), the data will be recorded towards the end of the push, just before the transaction is committed to Mercurial. Although, this is an implementation details.

There is no guarantee of strict ordering between dates. i.e. the `date` of push ID `N + 1` could be less than the `date` of push ID `N`. Such is how clocks work.

user The string username that performed the push.

If `full` is specified, each entry in the `changesets` and `obsoletechangesets` array will be an object instead of a string. Each object will have the following properties:

node The 40 byte hex SHA-1 of the changeset.

parents An array of 1 or 2 elements containing the 40 byte hex SHA-1 of the parent changesets. Merges have 2 entries. Root changesets have the value `0000000000000000000000000000000000000000`.

author The author string from the changeset.

desc The changeset's commit message.

branch The branch the changeset belongs to.

`default` is the default branch in Mercurial.

tags An array of string tags belonging to this changeset.

files An array of filenames that were changed by this changeset.

precursors (optional) An array of 40 character hex SHA-1 nodes identifying *precursor* nodes.

Precursor nodes are essentially previously versions of this changeset.

Precursor nodes come from obsolescence data. This key won't exist if there are no precursor nodes for this changeset.

The precursor changesets are hidden and not available to normal Mercurial operations. However, querying the pushlog for their info *may* return results.

Here's an example:

```
{
  "author": "Eugen Sawin <esawin@mozilla.com>",
  "branch": "default",
  "desc": "Bug 1110212 - Strong randomness for Android DNS resolver. r=sworkman",
  "files": [
    "other-licenses/android/res_init.c"
  ],
  "node": "ee4fe2ec168e719e822dabcdd797c0cff9ce2407",
  "parents": [
    "803bc910c45a875d9d76dc689c45dd91a1e02e23"
  ],
  "precursors": [
    "d313a202a85e114000f669c2fcb49ad42376ac04"
  ],
  "tags": []
}
```

Writing Agents that Consume Pushlog Data

It is common to want to write tools or services that consume pushlog data. For example, you may wish to perform processing of new commits as they arrive.

Before you consider using the pushlog for this, you should consider the *change notification services* on hg.mozilla.org instead. If those aren't sufficient, you should request one that is.

If you must consume the pushlog for monitoring for new pushes, you will need to periodically poll each repository separately. The following best practices should be used:

1. Query by push ID, not by changeset or date.
2. Always specify a `startID` and `endID`.
3. Try to avoid `full` if possible.
4. Always use the latest format version.
5. Don't be afraid to ask for a new pushlog feature to make your life easier.

Querying by push ID is preferred because date ordering is not guaranteed (due to system clock skew) and because changesets can occur in multiple pushes in *Headless Repositories*. If a changeset occurs in multiple pushes, using the changeset as an identifier is ambiguous! Push IDs are the only guaranteed stable method for selecting pushes.

We recommend that `startID` and `endID` always be specified so response sizes are bound. If they are omitted, the server may generate a very large payload. We've seen clients request **all** push data from the server and the response JSON is over 100 MB!

Specifying `full` will incur an additional lookup on the server. Without `full`, the response JSON is generated purely from the SQLite database. With `full`, data needs to be read from Mercurial. This adds overhead to the lookup and to the transfer. If you don't need the extra data, please don't request it.

Managing Repositories

`hg.mozilla.org` has a self-service mechanism for managing some repositories on the server.

Overview

You can SSH into `hg.mozilla.org` and execute specific commands to perform repository management. You don't get a full shell. Instead, you will be interfacing with an interactive wizard that will guide you through available options.

What Repositories Can Be Managed

Currently, the self-service interface only allows management of *user repositories*. These are repositories under https://hg.mozilla.org/users/.

For management of non-user repositories, please file a Developer Services :: hg.mozilla.org bug.

Who Has Access

Any account with level 1 commit access can push to user repositories and can manage their own user repositories.

When to Use a User Repository

`hg.mozilla.org` server operators have a reluctant tolerance towards the existence of user repositories on hg.mozilla.org.

In the ideal world, Mozilla is not a generic repository hosting service and hosting of non-critical repositories would be pushed off to free service hosting providers, such as Bitbucket. One reason is performance: user repositories take resources that could otherwise be put towards making other services and repositories faster.

However, there are some scenarios where third party hosting providers won't fulfill developer needs.

Important: Unless you have a specific need that requires hosting on hg.mozilla.org, you should consider hosting your repository on another service instead of hg.mozilla.org.

Configuring

You will need to configure your SSH client to talk to `hg.mozilla.org`. See *SSH Configuration* for instructions.

Creating a User Repository

A new repository can be created by running the `clone <repo>` command. e.g.:

```
$ ssh hg.mozilla.org clone my-new-repo
```

An interactive wizard will guide you through the process that should be self-expanatory. You'll have the option of creating an empty repository or cloning (forking) from an existing repository.

Important: Creating repositories can sometimes take many minutes. Do not ctrl+c the operation or the servers may be left in an inconsistent state.

If you encounter a problem, file a bug or say something in `#vcs` on IRC.

Editing a User Repository

It is possible to configure a number of options on repositories. The common interface for all actions is to run `edit <repo>`. e.g.:

```
$ ssh hg.mozilla.org edit my-repo
```

An interactive wizard will guide you through available options. Some of those options are described below.

Repository Description

It is possible to edit the repository's *description*. This is text that appears in the repository listing at https://hg.mozilla.org/.

Tip: We recommend always defining a description so people know what your repository is used for.

Publishing and Non-Publishing Repositories

Mercurial has a feature called *phases* that prevents changesets that have been *published* from being mutated (deleted, rewritten, etc). It prevents a foot gun where you could accidentally rewrite history that has been shared with others, which would result in divergent branches and confusion.

By default, Mercurial repositories are *publishing*, which means that anything pushed gets published (bumped to the *public* phase in technical terms) and Mercurial clients won't let you change those commits.

Important: User repositories on hg.mozilla.org are non-publishing by default.

If you would like to change whether your repository is publishing or non-publishing, use the `edit <repo>` command and select the appropriate option.

Obsolescence

It is possible to toggle the *obsolescence* feature on individual user repositories.

Obsolescence is an **experimental** feature of Mercurial that records how changesets evolve over time and allows old, obsoleted commits to disappear from the repository.

Obsolescence is an aspect of changeset evolution.

There are downsides to enabling obsolescence:

- The transfer of obsolescence markers during push isn't optimal and may significantly increase push times.
- Obsolescence doesn't yet integrate very well with hgweb (the HTTP/HTML repository viewer).
- We may have to disable this feature or incur data loss due to its experimental nature.

Important: Due to the experimental state of the obsolescence feature, we may have to disable this feature or incur data loss at any time.

Warning: Enable obsolescence at your own risk.

Tip: You do not need to enable obsolescence unless a user of your repository is using the *evolve* extension.

Deleting a User Repository

To delete a user repository, run `edit <repo>` and select the `delete` option.

User Repository URLs

Your own user repositories are accessible under the following URLs:

ssh://hg.mozilla.org/users/<username>/<repo> (read/write) https://hg.mozilla.org/users/<username>/<repo> (read only)

Your SSH/LDAP username is normalized. Specifically, the `@` in your email address is normalized to _. e.g. `mary@example.com` becomes `mary_example.com`.

When you create a user repository, you probably want to set up some paths in your hgrc. Here is an example `.hg/hgrc`:

```
[paths]
default = https://hg.mozilla.org/users/me_example.com/my-repo
default-push = ssh://hg.mozilla.org/users/me_example.com/my-repo
```

Change Notifications

hg.mozilla.org publishes notifications when various events occur, such as a push to a repository.

Common Properties of Notifications

Notifications are sent after an event has been completely replicated to all active HTTPS replication mirrors.

Note: Unlike polling the pushlog, reacting to data advertised by these notifications is not susceptible to race conditions when one mirror may have replicated data before another.

hg.mozilla.org guarantees chronological writing of notifications within individual repositories only. Assume we have 2 repositories, `X` and `Y` with pushes occurring in the order of `X1`, `Y1`, `X2`, `Y2`. We only guarantee that `X1` is delivered before `X2` and `Y1` is delivered before `Y2`. In other words, it is possible for `Y1` (or even `Y2`) to be delivered before `X1` (or `X2`).

Delivered messages have a *type*. The sections that follow describe the format/schema of each message type.

changegroup.1

This message type describes a push that introduced changesets (commits) on a repository.

The fields of this message type constitute the root fields of messages publishes to the `exchange/hgpushes/v1` exchange.

repo_url The URL of the repository that was pushed to.

heads A list of 40 character SHA-1 changesets that are DAG heads resulting from this push. Typically, there is only 1 entry (because most pushes only push 1 head).

pushlog_pushes A list of dicts describing each *Pushlog* entry related to changesets in this push. This list *should* be a single item. But it can be empty. If you see multiple entries, please say something in `#vcs`.

The list should only be empty for special repositories, such as the experimental unified repositories.

The composition of this dict is described below.

source Where this changeset came from.

The value will almost always be `push`.

(Not present in ‘‘exchange/hgpushes/v1‘‘ messages)

Each `pushlog_pushes` entry consists of the following keys:

pushid Integer pushlog ID.

time Integer UNIX timestamp the push occurred, as recorded by the pushlog.

user The authenticated user that performed the push. Typically an e-mail address.

push_json_url URL of JSON endpoint that describes this push in more detail.

push_full_json_url URL of JSON endpoint that describes this push in even more detail (lists files that changed, etc). See *Pushlog* for what the `json-pushes` JSON API returns.

newrepo.1

This message is sent when a new repository is created.

This message has the following fields:

repo_url URL of the created repository.

obsolete.1

This message describes obsolescence markers added on a repository.

Obsolescence markers tell when a changeset was *obsoleted* and should no longer be exposed to the outside world, effectively hiding it from history.

Essentially, an obsolescence marker contains a *precursor* node and a list of 0 or more *successor* nodes. The *precursor* node is hidden as a result of the creation of a marker. The *successor* nodes are the nodes that replaced the *precursor* node. If there is no replacement (the changeset was dropped), the list of *successors* is empty.

This message has the following fields:

markers A list of dicts describing each obsolescence marker in detail. The format of these entries is described below.

repo_url The URL of the repository this marker applies to.

Each `markers` entry is a dict with the following fields:

precursor Dict describing the *precursor* node.

successors List of dicts describing the *successor* nodes.

user String user that produced this marker (this comes from Mercurial's `ui.username` config option).

time Float corresponding to number of seconds since UNIX epoch time when this marker was produced.

The fields of a `precursor` or `successors` dict are as follows:

node 40 character SHA-1 of changeset.

known Bool indicating whether the changeset is known to the repo. Sometimes obsolescence markers reference changesets not pushed to the repo. This flag helps consumers know whether they might be able to query the repo for more info about this changeset.

visible Bool indicating whether the changeset is visible to the repository at the time the message was created. If `false`, the changeset is known but hidden. Value is `null` if the changeset is known `known`.

Even if the value is `true`, there is no guarantee a consumer of this message will be able to access changeset metadata from the repository, as a subsequent obsolescence marker could have made this changeset hidden by the time the consumer sees this message and queries the repository. This is one reason why this data structure contains changeset metadata that would normally be obtained by the consumer.

desc String of commit message for the changeset. May be null if the changeset is not known to the repo.

push Dict describing the pushlog entry for this changeset.

Will be `null` if the changeset is not known or if there isn't a pushlog entry for it.

The content of this dict matches the entries from `pushlog_pushes` from `changeset.1` messages.

Examples

An example message payload for is as follows:

```
{
  "type": "changegroup.1",
  "data": {
    "repo_url": "https://hg.mozilla.org/try",
    "heads": ["eb6d9371407416e488d2b2783a5a79f8364330c8"],
    "pushlog_pushes": {
      "time": 14609750810,
      "pushid": 120040,
```

```
      "user": "tlin@mozilla.com",
      "push_json_url": "https://hg.mozilla.org/try/json-pushes?version=2&startID=120039&endID=120040"
      "push_full_json_url": "https://hg.mozilla.org/try/json-pushes?version=2&full=1&startID=120039&e
    }
  }
}
```

Pulse Notifications

hg.mozilla.org guarantees at least once delivery of Pulse messages when a push is performed to the server.

Pulse messages are written to the following exchanges:

- exchange/hgpushes/v1
- (experimental) exchange/hgpushes/v2

The routing key for each message is the relative path of the repository on hg.mozilla.org (e.g. `mozilla-central` or `integration/mozilla-inbound`).

The `payload` of the JSON messages published to Pulse depend on the exchange.

The `exchange/hgpushes/v1` exchange only supported publishing *push events* that described a push. The `exchange/hgpushes/v2` exchange supports publishing multiple event types.

Important: New message types can be added to the `exchange/hgpushes/v2` exchange at any time.

Consumers should either ignore unknown message types or fail fast when encountering one.

The `exchange/hgpushes/v2` exchange has a payload with the following keys:

`type` String denoting the message type.

`data` Dictionary holding details about the event.

The message types and their data are described later in this document.

SNS Notifications

Change events for hg.mozilla.org are published to Amazon Simple Notification Service (SNS).

Messages are published to SNS topic `arn:aws:sns:us-west-2:699292812394:hgmo-events`.

The message is JSON with the following keys:

`type` String denoting the message type.

`data_url` URL where JSON describing the event can be obtained.

`data` (optional) Dictionary holding details about the event.

`external` (optional) Boolean indicating whether data is only available externally. If this key is present, `data` will not be present and the only way to obtain data is to query `data_url`.

`repo_url` (optional) URL of repository from which this data originated. This key is only present if `data` is not present, as this value is already recorded inside `data`. The main purpose of this key is to facilitate message filtering without having to query `data_url` to determine which repository the message belongs to.

The message types and their data are described later in this document.

At least once delivery is guaranteed. And, new message types may be introduced at any time.

Automated Clients

Do you have or want to run an automated consumer of hg.mozilla.org? Then this document is for you.

Read on to learn about the should and should nots of automatically consuming the hg.mozilla.org service.

When reading sections that recommend *not* doing something, keep in mind that https://hg.mozilla.org/ is ultimately a service in support of various Mozilla projects. The recommendation to *not* do something is not a hard or fast *no*: it is more of a preference so we can avoid operational issues. If you find yourself having to perform a lot of work to abide by the guidelines in this document, you should approach the hg.mozilla.org service operators to discuss alternatives to make your job easier. The hgmo-service-discuss@mozilla.com mailing list (contents are private) is a good way to get in touch.

Identifying Clients

Automated requests to https://hg.mozilla.org/ that belong to a well-defined service should use a custom *User-Agent* HTTP request header to identify themselves. This is so server operators can better assess the impact that individual clients have on the server.

Important: Server operators reserve the right to block generic *User-Agent* values that are contributing abnormally high load. Using a custom *User-Agent* lessens the possibility of being blocked due to *User-Agent* name conflict with a malicious agent also not using a custom *User-Agent*.

Limiting Concurrent Requests

https://hg.mozilla.org/ does not have infinite capacity and clients issuing several concurrent HTTP requests can cause capacity problems on the server.

In general, any single logical client should not attempt to issue more than 10 concurrent HTTP requests. This limit applies whether the requests are issued from a single IP or across several IPs.

The fewer concurrent requests that are issued, the lesser the load impact on the server. However, this obviously increases the time for the client to finish all the requests. So it's a trade-off.

Limiting Request Volume

Again, https://hg.mozilla.org/ does not have infinite capacity. Clients issuing thousands of HTTP requests can contribute significant server load, especially to certain endpoints.

Clients should be respectful of server capacity limitations and try to limit the total number of requests. This especially holds true for requests that take more than 1.0s to complete or transfer a lot of data from server to client.

Repository data on https://hg.mozilla.org/ (including pushlog and other custom data) can often be cloned to a local machine. Such is the nature of distributed version control. Once on the local machine, one can use *hg serve* to run a local HTTP server which exposes much of the same data as https://hg.mozilla.org/. Or one can use *hg* commands to access repository data.

Clients needing to query large parts of the repository (e.g. to collect information on every changeset or file) are highly encouraged to query data offline, if possible.

Obtaining Recent Changes

It is common for clients to want to know when a change occurs on hg.mozilla.org so that they can do something in reaction to it.

There are two mechanisms for determining when changes occur: subscription-based notifications and polling HTTP-based APIs on hg.mozilla.org.

For the push-based notifications, clients can subscribe to e.g. SNS or Pulse queues and receive messages milliseconds after they are published by hg.mozilla.org.

For polling, typically the *pushlog* HTTP+JSON API is used. Clients need to periodically query the pushlog and see if anything has changed since the last query.

The recommend mechanism for learning about recent changes is to subscribe to one of the push-based notifications (e.g. SNS or Pulse). This allows clients to see changes to *any* repository milliseconds after it occurs.

While it shouldn't occur frequently, subscription-based mechanisms can be lossy. e.g. if the subscriber is detached - possibly for a long-enough time - it may lose an event. For that reason, robust consumers without a tolerance for data loss should supplement subscription-based monitoring with pushlog polling. If subscriptions *lose* an event, the polling fallback should catch any events that fell through. Because subscriptions shouldn't be lossy, the pushlog polling interval can be relatively high - say every 5 minutes.

If clients want to react to an event in an expedient manner (say under a minute), they should use subscription-based notifications.

If latency is not important, it is acceptable to use polling.

Important: Clients should not poll repository endpoints on https://hg.mozilla.org/ more frequently than once every minute, as this can contribute significant load to the servers.

If clients really need to react to events with that little latency, they should be using subscription-based notification mechanisms.

Cloning from Pre-Generated Bundles

`hg.mozilla.org` supports offloading clone requests to pre-generated bundle files stored in a CDN and Amazon S3. **This results in drastically reduced server load (which helps prevent outages due to accidental, excessive load) and frequently results in faster clone times.**

How It Works

When a Mercurial client clones a repository, it looks to see if the server is advertising a list of available, pre-generated bundle files. If it is, it looks at the list, finds the most appropriate entry, downloads and applies that bundle, then does the equivalent of an `hg pull` against the original Mercurial server to fetch new data since the time the bundle file was produced. The end result is a faster clone with drastically reduced load on the Mercurial server.

Enabling

If you are running Mercurial 3.7 or newer, support for cloning from pre-generated bundles is built-in to Mercurial itself and enabled by default.

If you are running Mercurial 3.6, support is built-in but requires enabling a config option:

```
[experimental]
clonebundles = true
```

If you are running a Mercurial older than 3.6, upgrade to leverage the clone bundles feature.

Mercurial 4.1 is required to support zstd bundles, which are smaller and faster than bundles supported by earlier versions.

Configuring

hg.mozilla.org will advertise multiple bundles/URLs for each repository. Each listing varies by:

- Bundle type
- Server location

By default, Mercurial uses the first entry in the server-advertised bundles list that the client supports.

The *clone bundles* feature allows the client to define preferences of which bundles to fetch. The way this works is the client defines some key-value pairs in its config and bundles having these attributes will be upweighted.

Bundle Attributes on hg.mozilla.org

On `hg.mozilla.org`, following attributes are defined in the manifest:

BUNDLESPEC This defines the type of bundle.

We currently generate bundles with the following specifications: `zstd-v2`, `gzip-v1`, `gzip-v2`, `none-packed1`.

REQUIRESNI Indicates whether the URL requires SNI (a TLS extension). This is set to `true` for URLs where multiple certificates are installed on the same IP and SNI is required. It is undefined if SNI is not required.

ec2region The EC2 region the bundle file should be served from. We support `us-west-1`, `us-west-2`, `us-east-1`, `eu-central-`. You should prefer the region that is closest to you.

cdn Indicates whether the URL is on a CDN. Value is `true` to indicate the URL is a CDN. All other values or undefined values are to be interpretted as not a CDN.

Example Manifests

Here is an example *clone bundles* manifest:

```
https://hg.cdn.mozilla.net/mozilla-unified/82c75fd3a2de796351296592c459ab4aa4cd0baf.zstd-max.hg BUNDL
https://s3-us-west-2.amazonaws.com/moz-hg-bundles-us-west-2/mozilla-unified/82c75fd3a2de796351296592c
https://s3-us-west-1.amazonaws.com/moz-hg-bundles-us-west-1/mozilla-unified/82c75fd3a2de796351296592c
https://s3-external-1.amazonaws.com/moz-hg-bundles-us-east-1/mozilla-unified/82c75fd3a2de79635129659
https://s3-eu-central-1.amazonaws.com/moz-hg-bundles-eu-central-1/mozilla-unified/82c75fd3a2de7963512
https://hg.cdn.mozilla.net/mozilla-unified/82c75fd3a2de796351296592c459ab4aa4cd0baf.gzip-v2.hg BUNDLE
https://s3-us-west-2.amazonaws.com/moz-hg-bundles-us-west-2/mozilla-unified/82c75fd3a2de796351296592c
https://s3-us-west-1.amazonaws.com/moz-hg-bundles-us-west-1/mozilla-unified/82c75fd3a2de796351296592c
https://s3-external-1.amazonaws.com/moz-hg-bundles-us-east-1/mozilla-unified/82c75fd3a2de79635129659
https://s3-eu-central-1.amazonaws.com/moz-hg-bundles-eu-central-1/mozilla-unified/82c75fd3a2de7963512
https://hg.cdn.mozilla.net/mozilla-unified/82c75fd3a2de796351296592c459ab4aa4cd0baf.packed1-gd.hg BUN
https://s3-us-west-2.amazonaws.com/moz-hg-bundles-us-west-2/mozilla-unified/82c75fd3a2de796351296592c
https://s3-us-west-1.amazonaws.com/moz-hg-bundles-us-west-1/mozilla-unified/82c75fd3a2de796351296592c
https://s3-external-1.amazonaws.com/moz-hg-bundles-us-east-1/mozilla-unified/82c75fd3a2de79635129659
https://s3-eu-central-1.amazonaws.com/moz-hg-bundles-eu-central-1/mozilla-unified/82c75fd3a2de7963512
```

As you can see, listed bundle URLs vary by bundle type (compression and format) and location. For each repository we generate bundles for, we generate:

1. A zstd bundle (either default compression or maximum compression depending on repo utilization)
2. A gzip bundle (the default compression format)
3. A *streaming* bundle file (larger but faster)

For each of these bundles, we upload them to the following locations:

1. CloudFront CDN
2. S3 in us-west-2 region
3. S3 in us-west-1 region
4. S3 in us-east-1 region
5. S3 in eu-central-1 region

Which Bundles to Prefer

The zstd bundle hosted on CloudFront is the first entry and is thus preferred by clients by default.

zstd bundles are the smallest bundles and for most people they are the ideal bundle to use.

Note: Mercurial 4.1 is required to use zstd bundles. If an older Mercurial client is used, larger, non-zstd bundles will be used.

If you have a super fast internet connection, you can prefer the *packed/streaming* bundles. This will transfer 30-40% more data on average, but will require almost no CPU to apply. If you can fetch from S3 or CloudFront at 1 Gbps speeds, you should be able to clone Firefox in under 60s.:

```
# HG 3.7+
[ui]
clonebundleprefers = VERSION=packed1

# HG 3.6
[experimental]
clonebundleprefers = VERSION=packed1
```

Manifest Advertisement to AWS Clients

If a client in Amazon Web Services (e.g. EC2) is requesting a bundle manifest and that client is in an AWS region where bundles are hosted in S3, the advertised manifest will only show S3 URLs for the same AWS region. In addition, stream clone bundles are the highest priority bundle.

This behavior ensures that AWS transfer are intra-region (which means they are fast and don't result in a billable AWS event) and that `hg clone` completes as fast as possible (stream clone bundles are faster than gzip bundles).

Important: If you have machinery in an AWS region where we don't host bundles, please let us know. There's a good chance that establishing bundles in your region is cheaper than paying the cross-region transfer costs (intra-region transfer is free).

Manifest Advertisements to Mozilla Offices

If the client request appears to originate from a Mozilla office network, we make the assumption that the network speed and bandwidth are sufficient to always prefer the high-speed streamed clone bundles.

Which Repositories Have Bundles Available

Bundles are automatically generated for repositories that are high volume (in terms of repository size and clone frequency) or have a need for bundles.

The list of repositories with bundles enabled can be found at https://hg.cdn.mozilla.net/. A JSON document describing the bundles is available at https://hg.cdn.mozilla.net/bundles.json.

If you think bundles should be made available for a particular repository, let a server operator know by filing a `Developer Services :: hg.mozilla.org` bug or by asking in #vcs on irc.mozilla.org.

Contributing

Repositories and Relevant Files

Most code for hg.mozilla.org and related services can be found in the version-control-tools repo. In this repo, the following directories are relevant:

ansible/roles/hg-ssh Ansible role for the hg.mozilla.org SSH/master servers.

ansible/roles/hg-ssh-server Shared Ansible role for Mercurial SSH servers.

ansible/roles/hg-web Ansible role for the hg.mozilla.org HTTP/mirror servers.

docs/hgmo Sphinx documentation related to hg.mozilla.org (you are reading this now).

hgext Various Mercurial extensions. Many of which are used on hg.mozilla.org.

hgext/hgmo Contains hodgepodge of Mercurial customizations specific to hg.mozilla.org.

hghooks Mercurial hooks that run (primarily) on hg.mozilla.org.

hgserver Mercurial code and tests specific to Mercurial servers.

hgtemplates Mercurial templates. Contains both upstream templates and Mozilla customizations.

hgtemplates/gitweb_mozilla Fork of the `gitweb` Mercurial web style with Mozilla customizations. If you want to hack the theming on hg.mozilla.org, this is what you modify.

hgwsgi WSGI and Mercurial config files for repos on hg.mozilla.org.

pylib/mozautomation Python package containing utility code for things like defining names and URLs of common repos, parsing commit messages, etc.

pylib/mozhg Python package containing shared Mercurial code (code using Mercurial APIs and therefore covered by the GPL).

pylib/vcsreplicator Code for managing replication of data from Mercurial master server to read-only mirrors.

Terraform code for managing associated AWS infrastructure can be found in the devservices-aws repo under the `hgmo` directory.

Hacking the Theming

The version-control-tools repository contains all the files necessary to run a local `hgweb` server that behaves close enough to the actual server to facilitate hacking on the theming (the visual layout of the site).

To run a local server, run the following:

```
$ hg --config extensions.hgmo=/path/to/vct/hgext/hgmo serve --hgmo
```

Among other things, the `hgmo` extension adds a flag to `hg serve` that activates *hg.mozilla.org mode*. This will activate some other extensions and configure the style settings to run out of the version-control-tools repository.

Styles for Mercurial are checked into the `hgtemplates/` directory in version-control-tools. The default style for Mercurial is `paper`. However, hg.mozilla.org runs the `gitweb_mozilla` theme. This theme is based off of the `gitweb` theme.

The `hgtemplates/gitweb_mozilla/map` file is the main file mapping template names to their values. Some templates are large and split into their own `.tmpl` file.

Hint: To figure out what templates are used for various URLs, read `hg help hgweb`.

Hint: If you modify a template file, changes should be visible on next page load: no server restart is necessary. However, some pages are cached, so you may need to force reload the page in your browser via shift-reload or similar.

moz.build Metadata

hg.mozilla.org has the ability to extract and render metadata from moz.build files.

Web Command

On repositories that have this feature enabled, a new web command is available: `mozbuildinfo`. Requests to this web command have the form:

```
/<repo>/json-mozbuildinfo[/<rev>][?p=<path1>[&p=<path2>]]
```

By default, requests return moz.build info for all files changed by the current `tip` changeset. To specify which changeset should be returned, the hash of that changeset can be specified as a URL component. To change which files have their metadata resolved, pass the `p` query string argument 1 or more times with the filename(s) you are interested in.

The response from this web command is JSON. It looks something like:

```
{
  "aggregate": {
    "bug_component_counts": [
      [
        [
          "Core",
          "Build Config"
        ],
        1
      ]
    ],
    "recommended_bug_component": [
      "Core",
      "Build Config"
    ]
  },
  "files": {
    "Makefile.in": {
      "bug_component": [
        "Core",
        "Build Config"
      ]
```

```
      }
    }
}
```

The JSON object typically contains a `files` object containing per-file metadata and an `aggregate` object containing, well, aggregate data from all files.

If an error occurs, the object will have an `error` property containing a string error message.

The data inside the JSON is generated by the mozbuild Python package, whose canonical home lives in the python/mozbuild directory of mozilla-central. The mozbuild.frontend.context module is likely of the most interest.

Security of moz.build Evaluation

moz.build files are Python files. By evaluating moz.build files checked into version control, we are essentially enabling remote code execution. The security of the moz.build evaluation implementation is thus critically important.

It is essentially impossible to properly sandbox Python code from within (C)Python itself. Many have tried. They have all failed. So while moz.build files themselves are executed within a limited sandbox, this sandbox only reduces functionality initially available to moz.build files and doesn't comprise a security sandbox.

When the `mozbuildinfo` web command is requested, the following occurs on the server:

1. A WSGI process running as the *hg* user receives the request.
2. The `mozbuildinfo` web command handler is invoked
3. We verify that moz.build evaluation is enabled for the repository (it isn't enabled by default).
4. We verify that a wrapper script is installed (we don't allow executing moz.build files in the same process as the WSGI handler).
5. We verify additional request parameters.
6. We invoke the `mozbuild-eval` executable using `sudo`, passing JSON describing the request to it. The executable initially runs as root.
7. `mozbuild-eval` calls the `clone()` system call to create a new process with new namespaces for most Linux primitives (IPC, network, mount, pid, UTC).
8. The new process is moved to a special control group which has resource and device access limits in place.
9. The mounts in the new process are reset and all mounts besides a read-only loopback mount containing repository contents are removed. This includes `procfs` and `/dev`.
10. We `chroot()` into a new directory, which contains only the Python code needed to evaluate moz.build files.
11. The real, effective, and saved GID and UID of the process are changed to a low-privileged user.
12. We fork a new Python process to evaluate the moz.build files metadata.

The process evaluating moz.build files:

- Is running as a regular user/group
- That doesn't have write access to repositories via normal filesystem permissions
- That doesn't have write access to repositories via a read-only bind mount of the repositories
- Is executing inside a chroot with minimal files available
- Is executing in a control group that doesn't allow access to any devices except `urandom`.
- Is executing in a control group that limits CPU, memory, and I/O usage

- Doesn't have access to `procfs`
- Doesn't have a handle on any mounts from the host except the bind mount

Firefox Release Data

hg.mozilla.org has facilities for aggregating and exposing information about Firefox releases performed from Mercurial changesets.

Important: This feature is currently experimental and implementation details may change. If you would like the feature to stabilize, please ping *gps* in `#vcs` on irc.mozilla.org.

This feature is currently only enabled on the mozilla-central repository.

Features

Release Info on Changeset Pages

Changeset pages like https://hg.mozilla.org/mozilla-central/rev/1362c0928dc1 display information on Firefox releases in relation to that changeset.

Most pages should have a *first release with* and *last release without* section. Exceptions include changesets in the very early and very modern repository history. (This info isn't displayed unless we can find a release in both directions.)

If a release was made from that changeset, there will also be information on those releases. e.g. https://hg.mozilla.org/mozilla-central/rev/09a4282d1172. This includes a link to the pushlog containing changesets landed between two releases.

Listing of Known Releases

The `firefoxreleases` web command renders known Firefox releases from changesets in a repo. e.g. https://hg.mozilla.org/mozilla-central/firefoxreleases.

A JSON view is available by using the `json-firefoxreleases` web command. e.g. https://hg.mozilla.org/mozilla-central/json-firefoxreleases.

Filtering by *platform* is available by specifying the `platform` query string argument. e.g. https://hg.mozilla.org/mozilla-central/firefoxreleases?platform=win32.

Note: If you want to read this data from machines, a better source might be https://mozilla-services.github.io/buildhub/.

Development Info

Please file bugs and feature requests against the ‘‘hg.mozilla.org‘ Bugzilla component.

For real time support, make noise in `#vcs` on `irc.mozilla.org`.

Replication

As described at *Architecture*, `hg.mozilla.org` consists of a read/write master server available via SSH and a set of read-only mirrors available over HTTP. As changes are pushed to the master server, they are replicated to the read-only mirrors.

VCSReplicator Introduction

Version Control System Replicator (vcsreplicator - `pylib/vcsreplicator`) is a modern system for replicating version control data.

vcsreplicator is built on top of a distributed transaction log. When changes (typically pushes) are performed, an event is written to the log. Downstream consumers read from the log (hopefully in near real time) and replay changes that were made upstream.

The distributed transaction log is built on top of *Apache Kafka <https://kafka.apache.org/>*. Unlike other queueing and message delivery systems (such as AMQP/RabbitMQ), Kafka provides guarantees that satisfy our requirements. Notably:

- Kafka is a distributed system and can survive a failure in a single node (no single point of failure).
- Clients can robustly and independently store the last fetch offset. This allows independent replication mirrors.
- Kafka can provide delivery guarantees matching what is desired (order preserving, no message loss for acknowledged writes).
- It's fast and battle tested by a lot of notable companies.

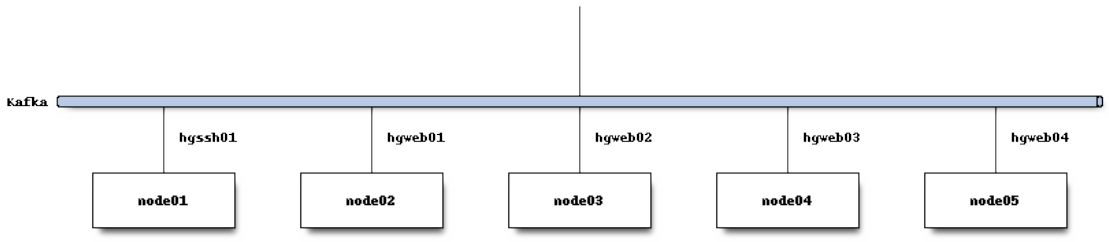
For more on the subject of distributed transaction logs, please read this excellent article from the people behind Kafka.

vcsreplicator is currently designed to replicate changes from a single *leader* Mercurial server to several, independent *mirror* servers. It supports replicating:

- changegroup data
- pushkey data (bookmarks, phases, obsolescence markers, etc)
- hgrc config files
- repository creation

Architecture

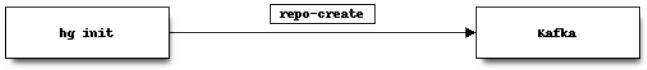
A cluster of servers form a Kafka cluster to expose a single logical network service. On hg.mozilla.org, that cluster looks like the following:



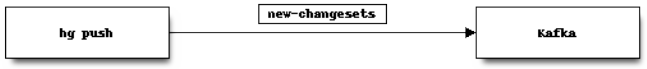
A Kafka topic (i.e. a message queue) is created for messages describing repository change events.

A Mercurial extension is installed on the leader server (the server where writes go). When a Mercurial repository is changed, a new message is published to Kafka.

For example, when a repository is created:



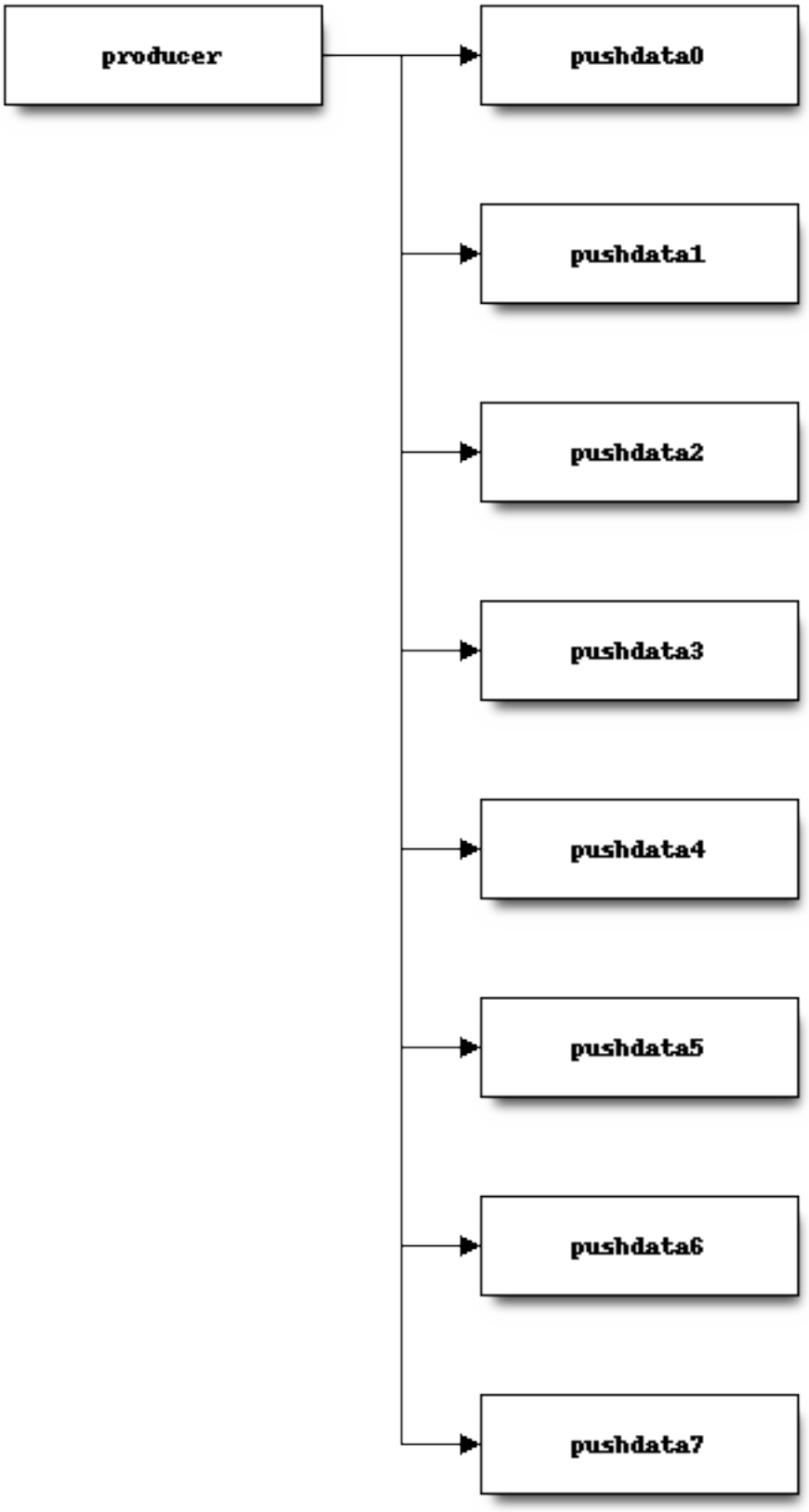
And when a new push occurs:



The published messages contain the repository name and details about the change that occurred.

Publishing to Kafka and Mercurial's transaction are coupled: if we cannot write messages to Kafka, the repository change operation is prevented or *rolled back*.

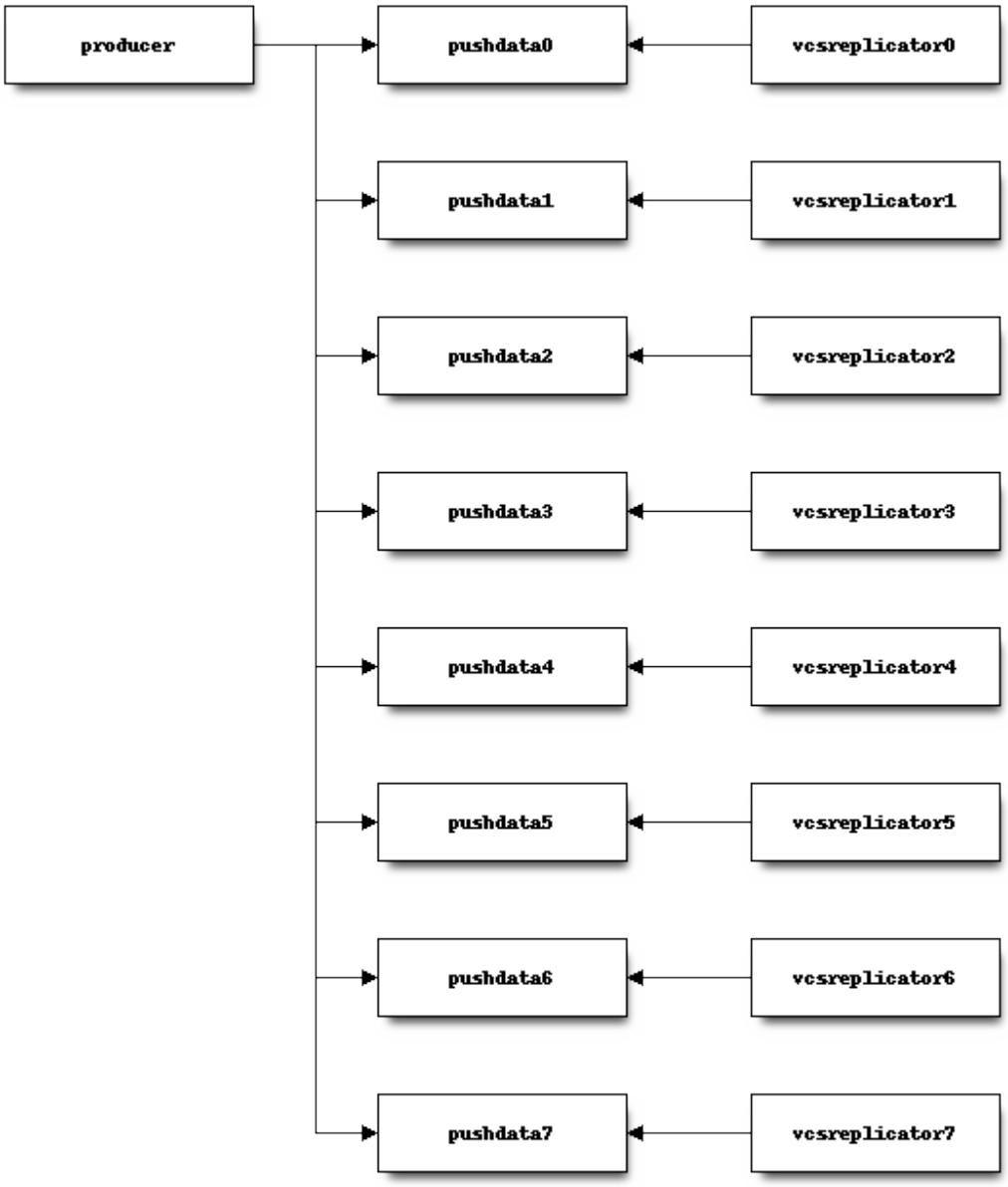
At a lower level on hg.mozilla.org, the `pushdata` Kafka topic contains 8 partitions and repository change messages are written into 1 of 8 of those partitions.


producer
pushdata0
pushdata1
pushdata2
pushdata3
pushdata4
pushdata5
pushdata6
pushdata7

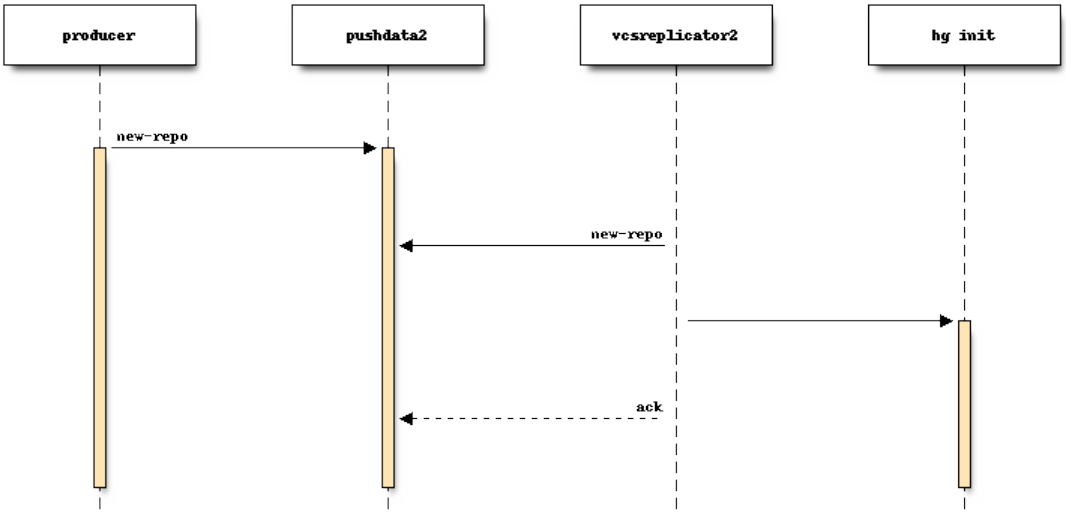
Each repository deterministically writes to the same partition via a name-based routing mechanism plus hashing. For example, the *foo* repository may always write to partition `1` but the *bar* repository may always write to partition `6`.

Messages published to Kafka topics/partitions are ordered: if message `A` is published before message `B`, consumers will always see `A` before `B`. This means the messages for a given repository are always consumed in chronological order.

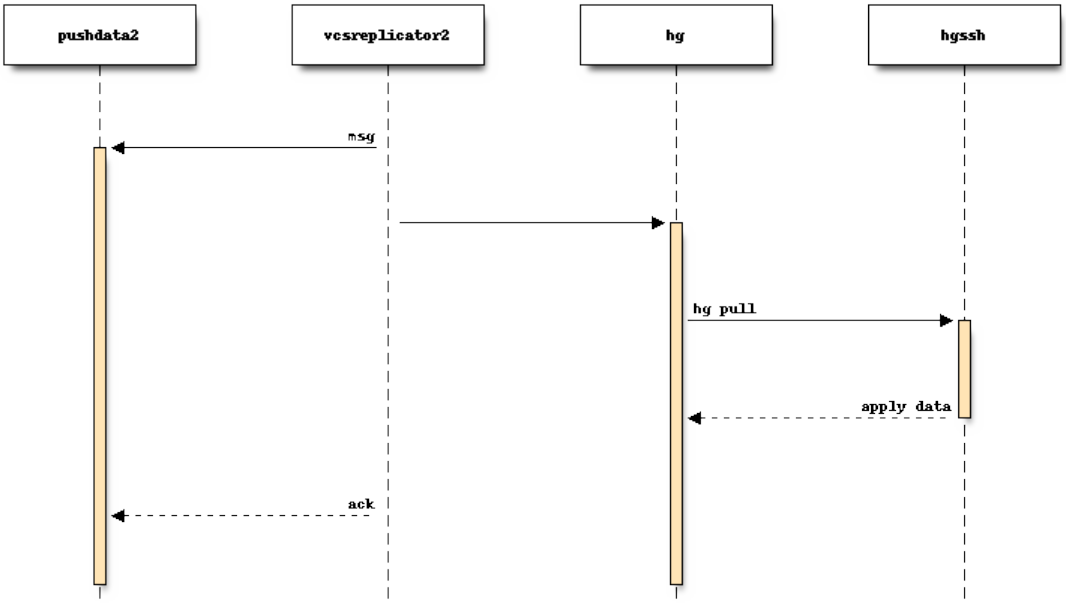
On each mirror, a `vcsreplicator-consumer` daemon process is bound to each partition in the `pushdata` topic. These processes essentially monitor each partition for new messages.



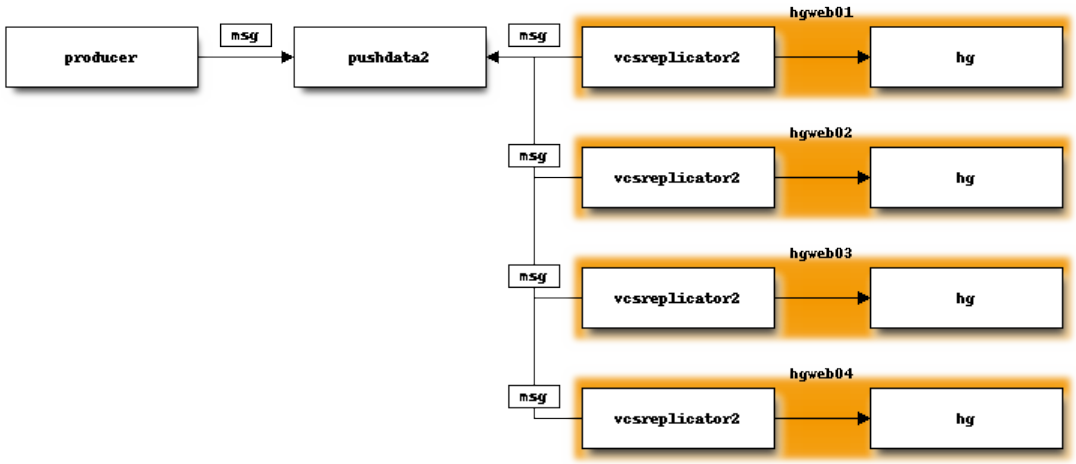
When a new message is written to the partition, the consumer daemon reacts to that message. The consumer daemon then takes an appropriate action for each message, often by invoking an `hg` process to complete an action. e.g. when a repository is created:



Sometimes the replicated data is too large to fit in a Kafka message. In that case, the consumer will connect to the leader server to obtain data.



Consumers react to new messages within milliseconds of them being published. And the same activity is occurring on each consumer simultaneously and independently.



Consumers typically fully process a message within a few seconds. Events corresponding to *big* changes (such as cloning a repository, large pushes, etc) can take longer - sometimes minutes.

We rely on repository change messages to have deterministic side-effects. i.e. independent consumers starting in the same state that apply the same stream of messages should end up in an identical state. In theory, a consumer could start from the very beginning of a Kafka topic, apply every message, and arrive at an identical state as the leader.

Consumers only process a single message per topic-partition simultaneously. This is to ensure complete ordering of messages for a given repository and to ensure that messages are successfully processed at most once.

After a consumer successfully performs actions in reaction to a published message, it acknowledges that Kafka message. Once a consumer has acknowledged a message, that message will never be delivered to that consumer again.

Kafka tracks acknowledged messages by recording the *offset* of the last acknowledged message within a given topic-partition.

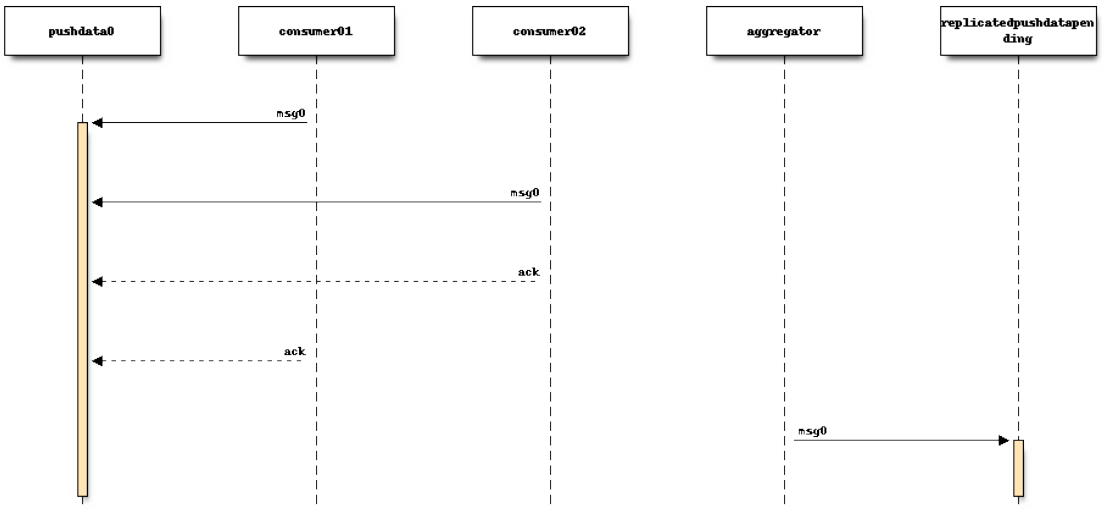
Each mirror maintains its own offsets into the various topic-partitions. If a mirror goes offline, Kafka will durably store messages. When the consumer process comes back online, it will resume consuming messages at the last acknowledged offset, picking up where it left off when it disconnected from Kafka.

Aggregated Push Data

Repository change messages may be written into multiple partitions to facilitate parallel consumption. Unfortunately, this loses total ordering of messages since there is no ordering across Kafka partitions.

In addition, consumers - being on separate servers - don't react to and acknowledge messages at exactly the same time. i.e. there is always a window of time between message publish and it being fully consumed where different consumers have different repository states due to being in different phases of processing a message. As an example, a *fast* server may take 1s to process a push to a repository but a *slow* server may take 2s. There is a window of 1s where one server has new state and another has old state. Exposing inconsistent state can confuse repository consumers.

The leader server runs a daemon that monitors the partition consumer offsets for all active consumers. When all active consumers have acknowledged a message, the daemon re-published that fully-consumed message in a separate Kafka topic - `replicatedpushdatapending` on hg.mozilla.org.



There is a single partition in the `replicatedpushdatapending` topic, which means all messages for all repositories are available in a single, ordered stream. And those messages aren't exposed until all active consumers have processed them.

Each mirror runs a daemon that subscribes to this single topic-partition. This daemon will take appropriate actions for each received message before acknowledging it.

The leader server also monitors consumer offsets in the `replicatedpushdatapending` topic-partition. When a message is fully consumed, it is re-published to the `replicatedpushdata` Kafka topic.

The stream of messages in the `replicatedpushdata` Kafka topic represents all fully-replicated repository changes acknowledged by all consumers.

This stream of fully-replicated messages can be consumed by consumers wish to react to events. e.g. on hg.mozilla.org, we have a daemon that publishes messages to Pulse (a RabbitMQ broker) and Amazon SNS so 3rd party consumers can get a notification when a repository even occurs and then implement their own derived actions (such as triggering CI against new commits).

Minimizing the Inconsistency Window for Exposed Data

As mentioned above, there is a race condition between the `vcsreplicator-consumer` processes on different servers fully processing and acknowledging a message written to the `pushdata` topic. This can lead to different servers exposing different repository state at any given time.

We minimize this window by employing a multi-stage *commit* to expose new repository data.

The stream of repository change messages being written by the leader includes a message containing the set of revisions that are DAG heads that should be exposed by servers. When this message is initially published to the `pushdata` topic, the `vcsreplicator-consumer` process performs a no-op when it sees this message. But the message is copied into the `replicatedpushdatapending` topic when its offset is acknowledged.

The *hgweb* servers run a `vcsreplicator-headsconsumer` process that is similar to the `vcsreplicator-consumer` process, except it only performs meaningful action for these messages containing repository DAG heads.

When `vcsreplicator-headsconsumer` sees a *heads* message, it atomically writes out a file containing the set of repository heads. Then it acknowledges the message.

A Mercurial extension loaded into the *hgweb* processes teaches Mercurial to use the file containing repository heads to determine what data to publicly expose on the server. Even if the repository contains new repository data, unless the new data is listed in the *heads* file, it won't be visible to repository consumers.

The `vcsreplicator-headsconsumer` process is running on each mirror and the actions it is performing are occurring independently: there is no central/shared data store recording which heads to expose. This means there is still a race condition where each server writes out *heads* files at a different pace. However, because the `vcsreplicator-headsconsumer` process is only writing out (usually) small files and because the process runs at elevated priority, the timings on different servers is usually very tight and the inconsistency window is thus very short - often just a few milliseconds. This window is so short as to not pose a significant problem.

Known Deficiencies

Shared Replication Log and Sequential Consumption Consumer processes can only process 1 event at a time. Events from multiple repositories are written to a shared replication log. Therefore, replication of repository X may be waiting on an event in repository Y to finish processing. This can add unwanted replication latency. Or, if a consuming processes crashes or gets in an endless loop trying to apply an event, consuming stalls.

Ideally, each repository would have its own replication event log and a pool of processes could consume events from any available replication log. There would need to be locking on consumers to ensure multiple processes aren't operating on the same repository. Such a system may not be possible with Kafka since apparently Kafka does not scale to thousands of topics and/or partitions. Although, hg.mozilla.org might be small enough for this to work. Alternate message delivery systems could potentially address this drawback. Although many message delivery systems don't provide the strong guarantees about delivery and ordering that Kafka does.

Reliance on hg pull Currently, pushing of new changegroup data results in `hg pull` being executed on mirrors. `hg pull` is robust and mostly deterministic. However, it does mean that mirrors must connect to the leader server to perform the replication. This means the leader's availability is necessary to perform replication.

A replication system more robust to failure of the leader would store all data in Kafka. As long as Kafka is up, mirrors would be able to synchronize. Another benefit of this model is that it would likely be faster: mirrors would have all to-be-applied data immediately available and wouldn't need to fetch it from a central server. Keep in mind that fetching large amounts of data can add significant load on the remote server, especially if several machines are connecting at once.

Another benefit of having all data in the replication log is that we could potentially store this *bundle* data in a key-value store (like S3) and leverage Mercurial's built in mechanism for serving bundles from remote URLs. The Mercurial server would essentially serve `hg pull` requests by telling clients to fetch data from a scalable, possibly distributed key-value store (such as a CDN).

A benefit of relying on `hg pull` based replication is it is simple: we don't need to reinvent Mercurial data storage. If we stop using `hg pull`, various types of data updates potentially fall through the cracks, especially if 3rd party extensions are involved. Also, storing data in the replication log could explode the size of the replication log, leading to its own scaling challenges.

Inconsistency Window on Mirrors Mirrors replicate independently. And data applied by mirrors is available immediately. Therefore, there is a window (hopefully small) where mirrors have inconsistent state of a repository.

If 2 mirrors are behind the same load balancer and requests are randomly routed to each of the mirrors, there is a chance a client may encounter inconsistent state. For example, a client may poll the pushlog to see what changesets are available then initiate a `hg pull -r <rev>` to fetch a just-pushed changeset. The pushlog from an in sync mirror may expose the changeset. But the `hg pull` hits an out-of-date mirror and is unable to find the requested changeset.

There are a few potential mechanisms to rectify this problem.

Mirrors could use shared storage. Mercurial's built-in transaction semantics ensure that clients don't read data that hasn't been fully committed yet. This is done at the filesystem level so any networked filesystem (like NFS) that honors atomic file moves should enable consistent state to be exposed to multiple consumers. However, networked filesystems have their own set of problems, including performance and possibly single points of failure. Not all environments are able to support networked filesystems either.

A potential (and yet unexplored) solution leverages ZooKeeper and Mercurial's *filtered repository* mechanism. Mercurial's repository access layer goes through a *filter* that can hide changesets from the consumer. This is frequently encountered in the context of obsolescence markers: obsolescence markers hide changesets from normal view. However, the changesets can still be accessed via the *unfiltered* view, which can be accessed by calling a `hg` command with the `--hidden` argument.

It might be possible to store the set of fully replicated heads for a given repository in ZooKeeper. When a request comes in, we look up which heads have been fully replicated and only expose changesets up to that point, even if the local repository has additional data available.

We would like to avoid an operational dependency on ZooKeeper (and Kafka) for repository read requests. (Currently, reads have no direct dependency on the availability of the ZooKeeper and Kafka clusters and we'd like to keep it this way so points of failure are minimized.) Figuring out how to track replicated heads in ZooKeeper so mirrors can expose consistent state could potentially introduce a read-time dependency.

Related to this problem of inconsistent state of mirrors is knowing when to remove a failing mirror from service. If a mirror encounters a catastrophic failure of its replication mechanism but the Mercurial server is still functioning, we would ideally detect when the mirror is drifting out of sync and remove it from the pool of mirrors so clients don't encounter inconsistent state across the mirror pool. This sounds like an obvious thing to do. But automatically removing machines can be dangerous, as being too liberal in yanking machines from service could result in removing machines necessary to service current load. When you consider that replication issues tend to occur during periods of high load, you can imagine what bad situations automatic decisions could get us in. Extreme care must be practiced when going down this road.

Data Loss Data loss can occur in a few scenarios.

Depending on what data is changed in the push, a single push may result in multiple replication messages being sent. For example, there could be a changegroup message and a pushkey message. The messages aren't written to Kafka as an atomic unit. Therefore, it's possible for 1 message to succeed, the cluster to fail, and the next message to fail, leaving the replication log in an inconsistent state.

In addition, messages aren't sent until *after* Mercurial closes the transaction committing data to the repository. It's therefore possible for the transaction to succeed but the message send to fail.

Both scenarios are mitigated by writing a no-op *heartbeat* message into the replication log as one of the final steps before transaction close. If this heartbeat can't be send, the transaction is aborted. The reasoning here is that by testing the replication log before closing the transaction, we have a pretty good indication whether the replication log will be writeable after transaction close. However, there is still a window for failure.

In the future, we should write a single replication event to Kafka for each push (requires bundle2 on the client) or write events to Kafka as a single unit (if that's even supported). We should also support rolling back the previous transaction in Mercurial if the post transaction close message(s) fails to write.

Installation and Configuring

vcsreplicator requires Python 2.7+, access to an Apache Kafka cluster, and an existing Mercurial server or repository.

For now, we assume you have a Kafka cluster configured. (We'll write the docs eventually.)

Mercurial Extension Installation

On a machine that is to produce or consume replication events, you will need to install the vcsreplicator Python package:

```
$ pip install /version-control-tools/pylib/vcsreplicator
```

On the leader machine, you will need to install a Mercurial extension. Assuming this repository is checked out in `/version-control-tools`, you will need the following in an hgrc file (either the global one or one inside a repository you want replicated):

```
[extensions]
# Load it by Python module (assuming it is in sys.path for the
# Mercurial server processes)
vcsreplicator.hgext =

# Load it by path.
vcsreplicator = /path/to/vcsreplicator/hgext.py
```

Producer hgrc Config

You'll need to configure your hgrc file to work with vcsreplicator:

```
[replicationproducer]

# Kafka host(s) to connect to.
hosts = localhost:9092

# Kafka client id
clientid = 1

# Kafka topic to write pushed data to
topic = pushdata

# How to map local repository paths to partions. You can:
#
# * Have a single partition for all repos
# * Map a single repo to a single partition
# * Map multiple repos to multiple partitions
#
# The partition map is read in sorted order of the key names.
# Values are <partition>:<regexp>. If the partitions are a comma
# delimited list of integers, then the repo path will be hashed and
# routed to the same partition over time. This ensures that all
# messages for a specific repo are routed to the same partition and
# thus consumed in a strict first in first out ordering.
#
# Map {repos}/foo to partition 0
# Map everything else to partitions 1, 2, 3, and 4.
partitionmap.0foo = 0:\{repos\}/foo
partitionmap.1bar = 1,2,3,4:.*

# Required acknowledgement for writes. See the Kafka docs. -1 is
# strongly preferred in order to not lose data.
reqacks = -1

# How long (in MS) to wait for acknowledgements on write requests.
```

```
# If a write isn't acknowledged in this time, the write is cancelled
# and Mercurial rolls back its transaction.
acktimeout = 10000

# Normalize local filesystem paths for representation on the wire.
# This both enables replication for listed paths and enables leader
# and mirrors to have different local filesystem paths.
[replicationpathrewrites]
/var/repos/ = {repos}/
```

Consumer Config File

The consumer daemon requires a config file.

The `[consumer]` section defines how to connect to Kafka to receive events. You typically only need to define it on the follower nodes. It contains the following variables:

hosts Comma delimited list of `host:port` strings indicating Kafka hosts.

client_id Unique identifier for this client.

connect_timeout Timeout in milliseconds for connecting to Kafka.

topic Kafka topic to consume. Should match producer's config.

group Kafka group the client is part of.

You should define this to a unique value.

The `[path_rewrites]` section defines mappings for how local filesystem paths are normalized for storage in log messages and vice-versa.

This section is not required. Presence of this section is used to abstract storage-level implementation details and to allow messages to define a repository without having to use local filesystem paths. It's best to explain by example. e.g.:

```
[path_rewrites]
/repos/hg/ = {hg}/
```

If a replication producer produces an event related to a repository under `/repos/hg/` - let's say `/repos/hg/my-repo`, it will normalize the path in the replication event to `{hg}/my-repo`. You could add a corresponding entry in the config of the follower node:

```
[path_rewrites]
{hg}/ = /repos/mirrors/hg/
```

When the consumer sees `{hg}/my-repo`, it will expand it to `/repos/mirrors/hg/my-repo`.

Path rewrites are very simple. We take the input string and match against registered rewrites in the order they were defined. Only a leading string search is performed - we don't match if the first character is different. Also, the match is case-insensitive (due to presence of case-insensitive filesystems that may report different path casing) but case-preserving. If you have camelCase in your repository name, it will be preserved.

The `[pull_url_rewrites]` section is used to map repository paths from log messages into URLs suitable for pulling from the leader. They work very similarly to `[path_rewrites]`.

The use case of this section is that it allows consumers to construct URLs to the leader repositories at message processing time rather than message produce time. Since URLs may change over time (don't tell Roy T. Fielding) and since the log may be persisted and replayed months or even years later, there needs to be an abstraction to redefine the location of a repository later.

Note: The fact that consumers perform an `hg pull` and need URLs to pull from is unfortunate. Ideally all repository data would be self-contained within the log itself. Look for a future feature addition to vcsreplicator to provide self-contained logs.

Aggregator Config File

The aggregator daemon (the entity that copies fully acknowledged messages into a new topic) has its own config file.

All config options are located in the `[aggregator]` section. The following config options are defined.

hosts Comma delimited list of `host:port` strings indicating Kafka hosts.

client_id Unique identifier for this client.

connect_timeout Timeout in milliseconds for connecting to Kafka.

monitor_topic The Kafka topic that will be monitored (messages will be copied from).

monitor_groups_file Path to a file listing the Kafka groups whose consumer offsets will be monitored to determine the most recent acknowledged offset. Each line in the file is the name of a Kafka consumer group.

ack_group The consumer group to use in `monitor_topic` that the aggregator daemon will use to record which messages it has copied.

aggregate_topic The Kafka topic that messages from `monitor_topic` will be copied to.

Upgrading Kafka

It is generally desirable to keep Kafka on an up-to-date version, to benefit from bugfixes and performance improvements. Upgrading the hg.mozilla.org Kafka instances can be done via a rolling upgrade, allowing the replication system to continue working with no downtime. Kafka also uses Apache Zookeeper as a distributed configuration service, and you may wish to upgrade Zookeeper at the same time as Kafka. The steps to do so are as follows:

Note: While these steps will likely cover all upgrade requirements, there is no guarantee that Apache will not change the upgrade process in the future. Make sure to check the release notes for any breaking changes to both application code and the upgrade procedure before moving forward.

Note: You should perform the Zookeeper and Kafka upgrades independently, to minimize the risk of failure and avoid debugging two applications if something goes wrong. See https://kafka.apache.org/documentation/#upgrade and https://wiki.apache.org/hadoop/ZooKeeper/FAQ#A6

Steps for both applications

Both Kafka and Zookeeper are deployed from a tarball which is uploaded to a Mozilla owned Amazon S3 bucket. We need to upload our new Kafka/Zookeeper tarballs to this bucket and keep their SHA 256 hashsum to ensure the correct file is downloaded.

1. **Upload new package archives to** https://s3-us-west-2.amazonaws.com/moz-packages/<package name>
2. Calculate the sha256 hash of the uploaded archives.
3. Update version-control-tools/ansible/roles/kafka-broker/tasks/main.yml with the new package names and hashes.

Zookeeper

To upgrade Zookeeper, simply deploy the updated code to hg.mozilla.org and then run *systemctl restart zookeeper* serially on each host to perform the upgrade. Easy!

Kafka

To upgrade Kafka, some additional steps must be performed if message format or wire protocol changes have been made between your current and updated versions. For example, when upgrading from 0.8.2 to 1.1.0, both message formats and wire protocol changes were made, so all of the following changes must be made.

4. Before updating the code, advertise the message format and protocol versions used by the current Kafka instance in the server config (kafka-server.properties) using the following properties: - inter.broker.protocol.version=CURRENT_KAFKA_VERSION - log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION
5. Deploy code in the previously updated archive to hg.mozilla.org.
6. Serially run *systemctl restart kafka.service* on each broker to update the code.
7. Update inter.broker.protocol.version to the version of Kafka you are upgrading to and run *systemctl restart kafka.service* serially once again.
8. Update log.message.format.version to the version of Kafka you are upgrading to and run *systemctl restart kafka.service* serially once again.

Note: After running each of the restart commands, you should *tail* the Kafka logs on the newly updated server, as well as a server that has yet to be updated, to make sure everything is working smoothly.

History of Replication Systems

This document aims to describe the historical approaches to repository replication used by hg.mozilla.org. Reading this document should leave readers with an understanding of what replication approaches were used, what worked, what didn't, why we switched strategies, etc.

Shared Filesystem / No Replication

When hg.mozilla.org was initially deployed, the SSH and HTTP servers used a shared filesystem via NFSv3. This was architecturally simple. All writes were available atomically on the read-only HTTP servers.

However, it suffered from a few serious limitations.

First, Mercurial makes heavy use of multiple files for repository storage. This I/O access model combined with NFS's overhead for I/O operations meant that many Mercurial operations were slow. Therefore hg.mozilla.org was slow. Fixing this would require overhauling how Mercurial repository storage worked. Not an easy endeavor!

Second, the lack of a writable filesystem meant that Mercurial could not populate various repository cache files and this made some Mercurial operations very slow. (The HTTP servers mounted the filesystem read-only for security reasons.) This issue could have been worked around by writing a Mercurial extension to store cache files in a separate location than `.hg/cache`. However, we never did so. Another workaround would have been to populate all the necessary caches on the SSH server at push time so an HTTP server wouldn't need to write caches since they were already up-to-date.

Because NFS was making read-only operations on the HTTP servers substantially slower than they had the potential to be, we decided to have the HTTP servers store repositories on their local filesystem and to use a replication system to keep everything in sync.

Push-Time Synchronous Replication

The push-time replication system was our first replication system. It was very crude yet surprisingly effective:

1. A hook fired during the `changegroup` and `pushkey` hooks as part of `hg push` operations.
2. This hook executed the `repo-push.sh` script, which iterated through all mirrors and effectively ran `ssh -l hg <mirror> <repo>`.
3. On each mirror, the SSH session effectively ran the `mirror-pull` scripts. This scripts essentially ran `hg pull ssh://hg.mozilla.org/<repo>`.

Each mirror performed its replication in parallel. So the number of mirrors could be scaled without increasing replication time proportionally.

Replication was performed synchronously with the push. So, the client's `hg push` command doesn't finish until all mirrors had completed their replication. This added a few seconds of latency to pushes.

There were several downsides with this replication method:

- Replication was synchronous with push, adding latency. This was felt most notably on the Try repository, which took 9-15s to replicate. Other repositories typically would take 1-8s.
- If a mirror was slow, it was a long pole and slowed down replication for the push, adding yet more latency to the push.
- If a mirror was down, the system was not intelligent enough to automatically remove the mirror from the mirrors list. The master would retry several times before failing. This added latency to pushes.
- If a mirror was removed from the replication system, it didn't re-sync when it came back online. Instead, it needed to be manually re-synced by running a script. If a server rebooted for no reason, it could become out of sync and someone may or may not re-sync it promptly.
- Each mirror synced and subsequently exposed data at different times. There was a window during replication where mirror A would advertise data that mirror B did not yet have. This could lead to clients seeing inconsistent repository state due to hitting different servers behind the load balancer.

In addition:

- There was no mechanism for replicating repository creation or deletion events.
- This was no mechanism for replicating hgrc changes.
- This replication system was optimized for a low-latency, high-availability intra-datacenter environment and wouldn't work well with a future, globally distributed hg.mozilla.org service (which would be far more prone to network events such as loss of connectivity).

Despite all these downsides, the legacy replication system was surprisingly effective. Mirrors getting out of sync was rare. Historically the largest problem was the increased push latency due to synchronous replication.

Kafka-Based Replication System

We wanted a replication system with fewer deficiencies than the push-time synchronous replication system described in the section above. Notably, we wanted:

- Replication to be asynchronous with the `hg push` operation so people wouldn't have to wait as long for their operation to complete.

- Mirrors that were down or slow wouldn't slow down `hg push` operations.
- Mirrors that went down would recover and catch up on replication backlog automatically when they return to service.
- Repository creation and deletion events could be replicated.
- hgrc changes could be replicated.
- The window of inconsistency across the HTTP servers would be reduced.

We devised a replication system built on top of message queues backed by Apache Kafka to implement such a system.

This system is described in detail at *Replication*.

Essentially, Kafka provides a distributed transaction log. During `hg push` operations, the server writes messages into Kafka describing the changes that were made to the repository. Daemons on mirrors react to new messages within milliseconds, triggering the replication of repository data. The stream of messages is ordered and consumers record their consume offset. This allows consumers to go offline and resume replication at the last consumed offset when they come back online.

Operational Guide

Deploying to hg.mozilla.org

All code running on the servers behind hg.mozilla.org is in the version-control-tools repository.

Deployment of new code to hg.mozilla.org is performed via an Ansible playbook defined in the version-control-tools repository. To deploy new code, simply run:

```
$ ./deploy hgmo
```

Important: Files from your local machine's version-control-tools working copy will be installed on servers. Be aware of any local changes you have performed, as they might be reflected on the server.

For minor deployments, pre-announcement of changes is not necessary: just do the deployment. As part of the deployment, an IRC notification will be issued in #vcs by Ansible.

For major upgrades (upgrading the Mercurial release or other major changes such as reconfiguring SSH settings or other changes that have a higher chance of fallout, pre-announcement is highly recommended.

Pre-announcements should be made to dev-version-control.

Deployment-time announcements should be made in `#vcs`. In addition, the on-duty Sheriff (they will have `|Sheriffduty` appended to their IRC nick) should be notified. Anyone in `#ci` with `|buildduty` in their IRC nick should also be notified. Sending an email to `sheriffs@mozilla.org` can't also hurt.

If extra caution is warranted, a bug should be filed against the Change Advisory Board. This board will help you schedule the upgrade work. Details can be found at https://wiki.mozilla.org/IT/ChangeControl.

Deployment Gotchas

Not all processes are restarted as part of upgrades. Notably, the `httpd` + WSGI process trees are not restarted. This means that Python or Mercurial upgrades may not been seen until the next `httpd` service restart. For this reason, deployments of Mercurial upgrades should be followed by manually restarting `httpd` when each server is out of the load balancer.

Restarting httpd/wsgi Processes

Note: this should be codified in an Ansible playbook

If a restart of the `httpd` and WSGI process tree is needed, perform the following:

1. Log in to the Zeus load balancer at https://zlb1.ops.scl3.mozilla.com:9090
2. Find the `hgweb-http` pool.
3. Mark as host as `draining` then click `Update`.
4. Poll the *draining* host for active connections by SSHing into the host and curling `http://localhost/server-status?auto`. If you see more than 1 active connection (the connection performing server-status), repeat until it goes away.
5. `systemctl restart httpd.service`
6. Put the host back in service in Zeus.
7. Repeat 3 to 6 until done with all hosts.

Forcing a hgweb Repository Re-clone

It is sometimes desirable to force a re-clone of a repository to each hgweb node. For example, if a new Mercurial release offers newer features in the repository store, you may need to perform a fresh clone in order to *upgrade* the repository on-disk format.

To perform a re-clone of a repository on hgweb nodes, the `hgmo-reclone-repos` deploy target can be used:

```
$ ./deploy hgmo-reclone-repos mozilla-central releases/mozilla-beta
```

The underlying Ansible playbook integrates with the load balancers and will automatically take machines out of service and wait for active connections to be served before performing a re-clone. The re-clone should thus complete without any client-perceived downtime.

Repository Mirroring

The replication/mirroring of repositories is initiated on the master/SSH server. An event is written into a distributed replication log and it is replayed on all available mirrors. See *Replication* for more.

Most repository interactions result in replication occurring automatically. In a few scenarios, you'll need to manually trigger replication.

The `vcsreplicator` Mercurial extension defines commands for creating replication messages. For a full list of available commands run `hg help -r vcsreplicator`. The following commands are common.

hg replicatehgrc Replicate the hgrc file for the repository. The `.hg/hgrc` file will effectively be copied to mirrors verbatim.

hg replicatesync Force mirrors to synchronize against the master. This ensures the repo is present on mirrors, the hgrc is in sync, and all repository data from the master is present.

Run this if mirrors ever get out of sync with the master. It should be harmless to run this on any repo at any time.

Important: You will need to run `/var/hg/venv_tools/bin/hg` instead of `/usr/bin/hg` so Python package dependencies required for replication are loaded.

Marking Repositories as Read-only

Repositories can be marked as read-only. When a repository is read-only, pushes are denied with a message saying the repository is read-only.

To mark an individual repository as read-only, create a `.hg/readonlyreason` file. If the file has content, it will be printed to the user as the reason the repository is read-only.

To mark all repositories on hg.mozilla.org as read-only, create the `/repo/hg/readonlyreason` file. If the file has content, it will be printed to the user.

Retiring Repositories

Users can *delete their own repositories* - this section applies only to non-user repositories.

Convention is to retire (aka delete) repositories by moving them out of the user accessible spaces on the master and deleting from webheads.

This can be done via ansible playbook in the version-control-tools repository:

```
$ cd ansible
$ ansible-playbook -i hosts -e repo=relative/path/on/server hgmo-retire-repo.yml
```

Managing Repository Hooks

It is somewhat common to have to update hooks on various repositories.

The procedure for doing this is pretty simple:

1. Update a `.hg/hgrc` file on the SSH master
2. Replicate hgrc to mirrors

Generally speaking, `sudo vim` to edit `.hg/hgrc` files is sufficient. Ideally, you should use `sudo -u hg vim .hg/hgrc`.

To replicate hgrc changes to mirrors after updating an hgrc, simply run:

```
$ /var/hg/venv_tools/bin/hg replicatehgrc
```

Note: `hg replicatehgrc` operates on the repo in the current directory.

The definition of hooks is somewhat inconsistent. Generally speaking, hook entries are cargo culted from another repo.

Try Head Management

The Try repository continuously grows new heads as people push to it. There are some version control operations that scale with the number of heads. This means that the repository gets slower as the number of heads increases.

To work around this slowness, we periodically remove old heads. We do this by performing dummy merges. The procedure for this is as follows:

```
# Clone the Try repo. This will be very slow unless --uncompressed is used.
hg clone --uncompressed -U https://hg.mozilla.org/try
cd try
# Verify heads to merge (this could take a while on first run)
```

```
hg log -r 'head() and branch(default) and not public()'
# Capture the list of heads to merge
hg log -r 'head() and branch(default) and not public()' -T '{node}\n' > heads
# Update the working directory to the revision to be merged into. A recent
# mozilla-central revision is typically fine.
hg up <revision>
# Do the merge by invoking `hg debugsetparents` repeatedly
for p2 in `cat heads`; do echo $p2; hg debugsetparents . $p2; hg commit -m 'Merge try head'; done
```

Clonebundles Management

Various repositories have their content *snapshotted* and uploaded to S3. These snapshots (*bundles* in Mercurial parlance) are advertised via the Mercurial server to clients and are used to seed initial clones. See *Cloning from Pre-Generated Bundles* for more.

From an operational perspective, bundle generation is triggered by the `hg-bundle-generate.service` and `hg-bundle-generate.timer` systemd units on the master server. This essentially runs the `generate-hg-s3-bundles` script. Its configuration lives in the script itself as well as `/repo/hg/bundles/repos` (which lists the repos to operate on and their bundle generation settings).

The critical output of periodic bundle generation are the objects uploaded to S3 (to multiple buckets in various AWS regions) and the advertisement of these URLs in per-repo `.hg/clonebundles.manifest` files. Essentially for each repo:

1. Bundles are generated
2. Bundles are uploaded to multiple S3 buckets
3. `clonebundles.manifest` is updated to advertise newly-uploaded URLs
4. `clonebundles.manifest` is replicated from hgssh to hgweb mirrors
5. Clients access `clonebundles.manifest` as part of `hg clone` and start requesting referenced URLs.

If bundle generation fails, it isn't the end of the world: the old bundles just aren't as up to date as they could be.

Important: The S3 buckets have automatic 7 day expiration of objects. The assumption is that bundle generation completes successfully at least once a week. If bundle generation doesn't run for 7 days, the objects referenced in `clonebundles.manifest` files will expire and clients will encounter HTTP 404 errors.

In the event that a bundle is *corrupted*, manual intervention may be required to mitigate to problem.

As a convenience, a backup of the `.hg/clonebundles.manifest` file is created during bundle generation. It lives at `.hg/clonebundles.manifest.old`. If a new bundle is corrupt but an old one is valid, the mitigation is to restore from backup:

```
$ cp .hg/clonebundles.manifest.old .hg/clonebundles.manifest
$ /var/hg/venv_tools/bin/hg replicatesync
```

If a single bundle or type of bundle is corrupted or causing problems, it can be removed from the `clonebundles.manifest` file so clients stop seeing it.

Inside the `clonebundles.manifest` file are *N* types of bundles uploaded to *M* S3 buckets (plus a CDN URL). The bundle types can be identified by the `BUNDLESPEC` value of each entry. For example, if *stream clone* bundles are causing problems, the entries with a `BUNDLESPEC` containing `none-packed` could be removed.

Danger: Removing entries from a `clonebundles.manifest` can be dangerous.
The removal of entries could shift a lot of traffic from S3/CDN to the hgweb servers themselves - possibly overloading them.
The removal of a particular entry type could have performance implications for Firefox CI. For example, removing *stream clone* bundles will make `hg clone` take several minutes longer. This is often acceptable as a short-term workaround and is preferred to removing *clone bundles* entirely.

Important: If modifying a `.hg/clonebundles.manifest` file, remember to run `/repo/hg/venv_tools/bin/hg replicatesync` to trigger the replication of that file to hgweb mirrors. Otherwise clients won't see the changes!

Corrupted fncache File

In rare circumstances, a `.hg/store/fncache` file can become corrupt. This file is essentially a cache of all known files in the `.hg/store` directory tree.

If this file becomes corrupt, symptoms often manifest as *stream clones* being unable to find a file. e.g. during working directory update there will be an error:

```
abort: No such file or directory: '<path>'
```

You can test the theory that the fncache file is corrupt by grepping for the missing path in the `.hg/store/fncache` file. There should be a `<path>.i` entry in the `fncache` file. If it is missing, the fncache file is corrupt.

To rebuild the `fncache` file:

```
$ sudo -u <user> /var/hg/venv_tools/bin/hg -R <repo> debugrebuildfncache
```

Where `<user>` is the user that owns the repo (typically `hg`) and `<repo>` is the local filesystem path to the repo to repair.

`hg debugrebuildfncache` should be harmless to run at any time. Worst case, it effectively no-ops. If you are paranoid. make a backup copy of `.hg/store/fncache` before running the command.

Important: Under no circumstances should `.hg/store/fncache` be removed or altered by hand. Doing so may result in further repository damage.

Mirrors in `pushdataaggregator_groups` File

On the SSH servers, the `/repo/hg/pushdataaggregator_groups` file lists all hgweb mirrors that must have acknowledged replication of a message before that message is re-published to `replicatedpushdata` Kafka topic. This topic is then used to publish events to Pulse, SNS, etc.

When adding or removing hgweb machines from active service, this file needs to be **manually** updated to reflect the current set of active mirrors.

If an hgweb machine is removed and the `pushdataaggregator_groups` file is not updated, messages won't be re-published to the `replicatedpushdata` Kafka topic. This should eventually result in an alert for lag of that Kafka topic.

If an hgweb machine is added and the `pushdataaggregator_groups` file is not updated, messages could be re-published to the `replicatedpushdata` Kafka topic before the message has been acknowledged by all replicas. This could result in clients seeing inconsistent repository state depending on which hgweb server they access.

Verifying Replication Consistency

The replication service tries to ensure that repositories on multiple servers are as identical as possible. But testing for this using standard filesystem comparison tools is difficult because some bits on disk may vary even though Mercurial data is consistent.

The `hg mozrepohash` command can be used to display hashes of important Mercurial data. If the output from this command is identical across machines, then the underlying repository stores should be identical.

To mass collect hashes of all repositories, you can run something like the following on an hgssh host:

```
$ /var/hg/version-control-tools/scripts/find-hg-repos.py /repo/hg/mozilla/ | \
  sudo -u hg -g hg parallel --progress --res /var/tmp/repohash \
  /var/hg/venv_tools/bin/hg -R /repo/hg/mozilla/{} mozrepohash
```

or the following on an hgweb host:

```
$ /var/hg/version-control-tools/scripts/find-hg-repos.py /repo/hg/mozilla/ | \
  sudo -u hg -g hg parallel --progress --res /var/tmp/repohash \
  /var/hg/venv_replication/bin/hg -R /repo/hg/mozilla/{} mozrepohash
```

This command will use GNU parallel to run `hg mozrepohash` on all repositories found by the `find-hg-repos.py` script and write the results into `/var/tmp/repohash`.

You can then `rsync` those results to a central machine and compare output:

```
$ for h in hgweb{1,2,3,4}.dmz.mdc1.mozilla.com; do \
    rsync -avz --delete-after --exclude stderr $h:/var/tmp/repohash/ $h/ \
  done

$ diff -r hgweb1.dmz.mdc1.mozilla.com hgweb2.dmz.mdc1.mozilla.com
```

SSH Server Services

This section describes relevant services running on the SSH servers.

An SSH server can be in 1 of 2 states: *master* or *standby*. At any one time, only a single server should be in the *master* state.

Some services always run on the SSH servers. Some services only run on the active master.

The *standby* server is in a state where it is ready to become the master at any time (such as if the master crashes).

Important: The services that run on the active master are designed to only have a single global instance. Running multiple instances of these services can result in undefined behavior or event data corruption.

Master Server Management

The current active master server is denoted by the presence of a `/repo/hg/master.<hostname>` file. e.g. the presence of `/repo/hg/master.hgssh4.dmz.scl3.mozilla.com` indicates that `hgssh4.dmz.scl3.mozilla.com` is the active master.

All services that should have only a single instance (running on the master) have systemd unit configs that prevent the unit from starting if the `master.<hostname>` file for the current server does not exist. So, as long as only a single `master.<hostname>` file exists, it should not be possible to start these services on more than one server.

The hg-master.target systemd unit provides a common target for starting and stopping all systemd units that should only be running on the active master server. The unit only starts if the /repo/hg/master.<hostname> file is present.

Note: The hg-master.target unit only tracks units specific to the master. Services like the sshd daemon processing Mercurial connections are always running and aren't tied to hg-master.target.

The /repo/hg/master.<hostname> file is monitored every few seconds by the hg-master-monitor.timer and associated /var/hg/version-control-tools/scripts/hg-master-start-sto script. This script looks at the status of the /repo/hg/master.<hostname> file and the hg-master.target unit and reconciles the state of hg-master.target with what is wanted.

For example, if /repo/hg/master.hgssh4.dmz.scl3.mozilla.com exists and hg-master.target isn't active, hg-master-start-stop will start hg-master.target. Similarly, if /repo/hg/master.hgssh4.dmz.scl3.mozilla.com is deleted, hg-master-start-stop will ensure hg-master.target (and all associated services by extension) are stopped.

So, the process for transitioning master-only services from one machine to another is to delete one master.<hostname> file then create a new master.<hostname> for the new master.

Important: Since hg-master-monitor.timer only fires every few seconds and stopping services may take several seconds, one should wait at least 60s between removing one master.<hostname> file and creating a new one for a server server. This limitation could be improved with more advanced service state tracking.

sshd_hg.service

This systemd service provides the SSH server for accepting external SSH connections that connect to Mercurial.

This is different from the system's SSH service (sshd.service). The differences from a typical SSH service are as follows:

- The service is running on port 222 (not port 22)
- SSH authorized keys are looked up in LDAP (not using the system auth)
- All logins are processed via pash, a custom Python script that dispatches to Mercurial or performs other adminstrative tasks.

This service should always be running on all servers, even if they aren't the master. This means that hg-master.target does not control this service.

hg-bundle-generate.timer and hg-bundle-generate.service

These systemd units are responsible for creating Mercurial bundles for popular repositories and uploading them to S3. The bundles it produces are also available on a CDN at https://hg.cdn.mozilla.net/.

These bundles are advertised by Mercurial repositories to facilitate *bundle-based cloning*, which drastically reduces the load on the hg.mozilla.org servers.

This service only runs on the master server.

pushdataaggregator-pending.service

This systemd service monitors the state of the replication mirrors and copies fully acknowledged/applied messages into a new Kafka topic (replicatedpushdatapending).

The `replicatedpushdatapending` topic is watched by the `vcsreplicator-headsconsumer` process on the hgweb machines.

This service only runs on the master server.

pushdataaggregator.service

This systemd service monitors the state of the replication mirrors and copies fully acknowledged/applied messages into a new Kafka topic (`replicatedpushdata`).

The `replicatedpushdata` topic is watched by other services to react to repository events. So if this service stops working, other services will likely sit idle.

This service only runs on the master server.

pulsenotifier.service

This systemd service monitors the `replicatedpushdata` Kafka topic and sends messages to Pulse to advertise repository events.

For more, see *Change Notifications*.

The Pulse notifications this service sends are relied upon by various applications at Mozilla. If it stops working, a lot of services don't get notifications and things stop working.

This service only runs on the master server.

snsnotifier.service

This systemd service monitors the `replicatedpushdata` Kafka topic and sends messages to Amazon S3 and SNS to advertise repository events.

For more, see *Change Notifications*.

This service is essentially identical to `pulsenotifier.service` except it publishes to Amazon services, not Pulse.

unifyrepo.service

This systemd service periodically aggregates the contents of various repositories into other repositories.

This service and the repositories it writes to are currently experimental.

This service only runs on the master server.

Monitoring and Alerts

hg.mozilla.org is monitored by Nagios.

check_hg_bundle_generate_age

This check monitors the last generation time of *clone bundles*. The check is a simple wrapper around the `check_file_age` check. It monitors the age of the `/repo/hg/bundles/lastrun` file. This file should be touched every ~24h when the `hg-bundle-generate.service` unit completes.

Remediation If this alert fires, it means the `hg-bundle-generate.service` unit hasn't completed in the past 1+ days. This failure is non-urgent. But the failure needs to be investigated within 5 days.

A bug against the hg.mozilla.org service operator should be filed. The alert can be acknowledged once a bug is on file.

If the alert turns critical and an hg.mozilla.org service operator has not acknowledged the alert's existence, attempts should be made to page a service operator. The paging can be deferred until waking hours for the person being paged, as the alert does not represent an immediate issue. The important thing is that the appropriate people are made aware of the alert so they can fix it.

check_zookeeper

check_zookeeper monitors the health of the ZooKeeper ensemble running on various servers. The check is installed on each server running ZooKeeper.

The check verifies 2 distinct things: the health of an individual ZooKeeper node and the overall health of the ZooKeeper ensemble (cluster of nodes). Both types of checks should be configured where this check is running.

Expected Output When everything is functioning as intended, the output of this check should be:

```
zookeeper node and ensemble OK
```

Failures of Individual Nodes A series of checks will be performed against the individual ZooKeeper node. The following error conditions are possible:

NODE CRITICAL - not responding "imok": <response> The check sent a `ruok` request to ZooKeeper and the server failed to respond with `imok`. This typically means the node is in some kind of failure state.

NODE CRITICAL - not in read/write mode: <mode> The check sent a `isro` request to ZooKeeper and the server did not respond with `rw`. This means the server is not accepting writes. This typically means the node is in some kind of failure state.

NODE WARNING - average latency higher than expected: <got> > <expected> The average latency to service requests since last query is higher than the configured limit. This node is possibly under higher-than-expected load.

NODE WARNING - open file descriptors above percentage limit: <value> The underlying Java process is close to running out of available file descriptors.

We should never see this alert in production.

If any of these node errors is seen, `#vcs` should be notified and the on call person for these servers should be notified.

Failures of Overall Ensemble A series of checks is performed against the ZooKeeper ensemble to check for overall health. These checks are installed on each server running ZooKeeper even though the check is seemingly redundant. The reason is each server may have a different perspective on ensemble state due to things like network partitions. It is therefore important for each server to perform the check from its own perspective.

The following error conditions are possible:

ENSEMBLE WARNING - node (HOST) not OK: <state> A node in the ZooKeeper ensemble is not returning `imok` to an `ruok` request.

As long as this only occurs on a single node at a time, the overall availability of the ZooKeeper ensemble is not compromised: things should continue to work without service operation. If the operation of the ensemble is compromised, a different error condition with a critical failure should be raised.

ENSEMBLE WARNING - socket error connecting to HOST: <error> We were unable to speak to a host in the ensemble.

This error can occur if ZooKeeper is not running on a node it should be running on.

As long as this only occurs on a single node at a time, the overall availability of the ZooKeeper ensemble is not compromised.

ENSEMBLE WARNING - node (HOST) is alive but not available A ZooKeeper server is running but it isn't healthy.

This likely only occurs when the ZooKeeper ensemble is not fully available.

ENSEMBLE CRITICAL - unable to find leader node; ensemble likely not writable We were unable to identify a leader node in the ZooKeeper ensemble.

This error almost certainly means the ZooKeeper ensemble is down.

ENSEMBLE WARNING - only have X/Y expected followers This warning occurs when one or more nodes in the ZooKeeper ensemble isn't present and following the leader node.

As long as we still have a quorum of nodes in sync with the leader, the overall state of the ensemble should not be compromised.

ENSEMBLE WARNING - only have X/Y in sync followers This warning occurs when one or more nodes in the ZooKeeper ensemble isn't in sync with the leader node.

This warning likely occurs after a node was restarted or experienced some kind of event that caused it to get out of sync.

check_vcsreplicator_lag

`check_vcsreplicator_lag` monitors the replication log to see if consumers are in sync.

This check runs on every host that runs the replication log consumer daemon, which is every *hgweb* machine. The check is only monitoring the state of the host it runs on.

The replication log consists of N independent partitions. Each partition is its own log of replication events. There exist N daemon processes on each consumer host. Each daemon process consumes a specific partition. Events for any given repository are always routed to the same partition.

Consumers maintain an offset into the replication log marking how many messages they've consumed. When there are more messages in the log than the consumer has marked as applied, the log is said to be *lagging*. A lagging consumer is measured by the count of messages it has failed to consume and by the elapsed time since the first unconsumed message was created. Time is the more important lag indicator because the replication log can contain many small messages that apply instantaneously and thus don't really constitute a notable lag.

When the replication system is working correctly, messages written by producers are consumed within milliseconds on consumers. However, some messages may take several seconds to apply. Consumers do not mark a message as consumed until it has successfully applied it. Therefore, there is always a window between event production and marking it as consumed where consumers are out of sync.

Expected Output When a host is fully in sync with the replication log, the check will output the following:

```
OK - 8/8 consumers completely in sync

OK - partition 0 is completely in sync (X/Y)
OK - partition 1 is completely in sync (W/Z)
...
```

This prints the count of partitions in the replication log and the consuming offset of each partition.

When a host has some partitions that are slightly out of sync with the replication log, we get a slightly different output:

```
OK - 2/8 consumers out of sync but within tolerances

OK - partition 0 is 1 messages behind (0/1)
OK - partition 0 is 1.232 seconds behind
OK - partition 1 is completely in sync (32/32)
...
```

Even though consumers are slightly behind replaying the replication log, the drift is within tolerances, so the check is reporting OK. However, the state of each partition's lag is printed for forensic purposes.

Warning and Critical Output The monitor alerts when the lag of any one partition of the replication log is too great. As mentioned above, lag is measured in message count and time since the first unconsumed message was created. Time is the more important lag indicator.

When a partition/consumer is too far behind, the monitor will issue a **WARNING** or **CRITICAL** alert depending on how far behind consumers are. The output will look like:

```
WARNING - 2/8 partitions out of sync

WARNING - partition 0 is 15 messages behind (10/25)
OK - partition 0 is 5.421 seconds behind
OK - partition 1 is completely in sync (34/34)
...
```

The first line will contain a summary of all partitions' sync status. The following lines will print per-partition state.

The check will also emit a warning when there appears to be clock drift between the producer and the consumer.:

```
WARNING - 0/8 partitions out of sync
OK - partition 0 is completely in sync (25/25)
WARNING - clock drift of -1.234s between producer and consumer
OK - partition 1 is completely in sync (34/34)
...
```

Remediation to Consumer Lag If everything is functioning properly, a lagging consumer will self correct on its own: the consumer daemon is just behind (due to high load, slow network, etc) and it will catch up over time.

In some rare scenarios, there may be a bug in the consumer daemon that has caused it to crash or enter a endless loop or some such. To check for this, first look at systemd to see if all the consumer daemons are running:

```
$ systemctl status vcsreplicator@*.service
```

If any of the processes aren't in the `active (running)` state, the consumer for that partition has crashed for some reason. Try to start it back up:

```
$ systemctl start vcsreplicator@*.service
```

You might want to take a look at the logs in the journal to make sure the process is happy:

```
$ journalctl -f --unit vcsreplicator@*.service
```

If there are errors starting the consumer process (including if the consumer process keeps restarting due to crashing applying the next available message), then we've encountered a scenario that will require a bit more human involvement.

Important: At this point, it might be a good idea to ping people in #vcs or page Developer Services on Call, as they are the domain experts.

If the consumer daemon is stuck in an endless loop trying to apply the replication log, there are generally two ways out:

1. Fix the condition causing the endless loop.
2. Skip the message.

We don't yet know of correctable conditions causing endless loops. So, for now the best we can do is skip the message and hope the condition doesn't come back:

```
$ /var/hg/venv_replication/bin/vcsreplicator-consumer /etc/mercurial/vcsreplicator.ini --skip --parti
```

Note: The `--partition` argument is semi-important: it says which Kafka partition to pull the to-be-skipped message from. The number should be the value from the systemd service that is failing / reporting lag.

Important: Skipping messages could result in the repository replication state getting out of whack.

If this only occurred on a single machine, consider taking the machine out of the load balancer until the incident is investigated by someone in #vcs.

If this occurred globally, please raise awareness ASAP.

Important: If you skip a message, please file a bug in Developer Services :: hg.mozilla.org with details of the incident so the root cause can be tracked down and the underlying bug fixed.

---

check_vcsreplicator_pending_lag

`check_vcsreplicator_pending_lag` monitors the replication log to see whether the `vcsreplicator-headsconsumer` process has processed all available messages.

This check is similar to `vcsvcsreplicator_lag` except it is monitoring the processing of the `replicatedpushdatapending` topic as performed by the `vcsreplicator-headsconsumer` process.

Expected Output When a host is fully in sync with the replication log, the check will output the following:

```
OK - 1/1 consumers completely in sync

OK - partition 0 is completely in sync (X/Y)
```

When a host has some partitions that are slightly out of sync with the replication log, we get a slightly different output:

```
OK - 1/1 consumers out of sync but within tolerances

OK - partition 0 is 1 messages behind (0/1)
OK - partition 0 is 1.232 seconds behind
```

Even though consumers are slightly behind replaying the replication log, the drift is within tolerances, so the check is reporting OK. However, the state of each partition's lag is printed for forensic purposes.

Warning and Critical Output The monitor alerts when the lag of the replication log is too great. Lag is measured in message count and time since the first unconsumed messaged was created. Time is the more important lag indicator.

When a partition/consumer is too far behind, the monitor will issue a **WARNING** or **CRITICAL** alert depending on how far behind consumers are. The output will look like:

```
WARNING - 1/1 partitions out of sync

WARNING - partition 0 is 15 messages behind (10/25)
OK - partition 0 is 5.421 seconds behind
```

The check will also emit a warning when there appears to be clock drift between the producer and the consumer.:

```
WARNING - 0/1 partitions out of sync
OK - partition 0 is completely in sync (25/25)
WARNING - clock drift of -1.234s between producer and consumer
```

Remediation to Consumer Lag Because of the limited functionality performed by the `vcsreplicator-headsconsumer` process, this alert should never fire.

If this alert fires, the likely cause is the `vcsreplicator-headsconsumer` process / `vcsreplicator-heads.service` daemon has crashed. Since this process operates mostly identically across machines, it is expected that a failure will occur on all servers, not just 1.

First check the status of the daemon process:

```
$ systemctl status vcsreplicator-heads.service
```

If the service isn't in the `active (running)` state, the consumer daemon has crashed for some reason. Try to start it:

```
$ systemctl start vcsreplicator-heads.service
```

You might want to take a look at the logs in the journal to make sure the process is happy:

```
$ journalctl -f --unit vcsreplicator-heads.service
```

If there are errors starting the consumer process (including if the consumer process keeps restarting due to crashing applying the next available message), then we've encountered a scenario that will require a bit more human involvement.

Important: If the service is not working properly after restart, escalate to VCS on call.

---

check_pushdataaggregator_pending_lag

`check_pushdataaggregator_pending_lag` monitors the lag of the aggregated replication log (the `pushdataaggregator-pending.service` systemd service).

The check verifies that the aggregator service has copied all fully replicated messages to the `replicatedpushdatapending` Kafka topic.

The check will alert if the number of outstanding ready-to-copy messages exceeds configured thresholds.

Important: If messages aren't being copied into the aggregated message log, recently pushed changesets won't be exposed on https://hg.mozilla.org/.

Expected Output Normal output will say that all messages have been copied and all partitions are in sync or within thresholds:

```
OK - aggregator has copied all fully replicated messages

OK - partition 0 is completely in sync (1/1)
OK - partition 1 is completely in sync (1/1)
OK - partition 2 is completely in sync (1/1)
OK - partition 3 is completely in sync (1/1)
OK - partition 4 is completely in sync (1/1)
OK - partition 5 is completely in sync (1/1)
OK - partition 6 is completely in sync (1/1)
OK - partition 7 is completely in sync (1/1)
```

Failure Output The check will print a summary line indicating total number of messages behind and a per-partition breakdown of where that lag is. e.g.:

```
CRITICAL - 2 messages from 2 partitions behind

CRITICAL - partition 0 is 1 messages behind (1/2)
OK - partition 1 is completely in sync (1/1)
CRITICAL - partition 2 is 1 messages behind (1/2)
OK - partition 3 is completely in sync (1/1)
OK - partition 4 is completely in sync (1/1)
OK - partition 5 is completely in sync (1/1)
OK - partition 6 is completely in sync (1/1)
OK - partition 7 is completely in sync (1/1)

See https://mozilla-version-control-tools.readthedocs.io/en/latest/hgmo/ops.html
for details about this check.
```

Remediation to Check Failure If the check is failing, first verify the Kafka cluster is operating as expected. If it isn't, other alerts on the hg machines should be firing. **Failures in this check can likely be ignored if the Kafka cluster is in a known bad state.**

If there are no other alerts, there is a chance the daemon process has become wedged. Try bouncing the daemon:

```
$ systemctl restart pushdataaggregator-pending.service
```

Then wait a few minutes to see if the lag decreased. You can also look at the journal to see what the daemon is doing:

```
$ journalctl -f --unit pushdataaggregator-pending.service
```

If things are failing, escalate to VCS on call.

check_pushdataaggregator_lag

`check_pushdataaggregator_lag` monitors the lag of the aggregated replication log (the `pushdataaggregator.service` systemd service).

The check verifies that the aggregator service has copied all fully replicated messages to the unified, aggregate Kafka topic.

The check will alert if the number of outstanding ready-to-copy messages exceeds configured thresholds.

Important: If messages aren't being copied into the aggregated message log, derived services such as Pulse notification won't be writing data.

Expected Output Normal output will say that all messages have been copied and all partitions are in sync or within thresholds:

```
OK - aggregator has copied all fully replicated messages

OK - partition 0 is completely in sync (1/1)
```

Failure Output The check will print a summary line indicating total number of messages behind and a per-partition breakdown of where that lag is. e.g.:

```
CRITICAL - 1 messages from 1 partitions behind

CRITICAL - partition 0 is 1 messages behind (1/2)

See https://mozilla-version-control-tools.readthedocs.io/en/latest/hgmo/ops.html
for details about this check.
```

Remediation to Check Failure If the check is failing, first verify the Kafka cluster is operating as expected. If it isn't, other alerts on the hg machines should be firing. **Failures in this check can likely be ignored if the Kafka cluster is in a known bad state.**

If there are no other alerts, there is a chance the daemon process has become wedged. Try bouncing the daemon:

```
$ systemctl restart pushdataaggregator.service
```

Then wait a few minutes to see if the lag decreased. You can also look at the journal to see what the daemon is doing:

```
$ journalctl -f --unit pushdataaggregator.service
```

If things are failing, escalate to VCS on call.

check_pulsenotifier_lag

`check_pulsenotifier_lag` monitors the lag of Pulse *Change Notifications* in reaction to server events.

The check is very similar to `check_vcsreplicator_lag`. It monitors the same class of thing under the hood: that a Kafka consumer has read and acknowledged all available messages.

For this check, the consumer daemon is the `pulsenotifier` service running on the master server. It is a systemd service (`pulsenotifier.service`). Its logs are in `/var/log/pulsenotifier.log`.

Expected Output There is a single consumer and partition for the pulse notifier Kafka consumer. So, expected output is something like the following:

```
OK - 1/1 consumers completely in sync

OK - partition 0 is completely in sync (159580/159580)

See https://mozilla-version-control-tools.readthedocs.io/en/latest/hgmo/ops.html
for details about this check.
```

Remediation to Check Failure There are 3 main categories of check failure:

1. pulse.mozilla.org is down
2. The `pulsenotifier` daemon has crashed or wedged

3. The hg.mozilla.org Kafka cluster is down

Looking at the last few lines of `/var/log/pulsenotifier.log` should indicate reasons for the check failure.

If Pulse is down, the check should be acked until Pulse service is restored. The Pulse notification daemon should recover on its own.

If the `pulsenotifier` daemon has crashed, try restarting it:

```
$ systemctl restart pulsenotifier.service
```

If the hg.mozilla.org Kafka cluster is down, lots of other alerts are likely firing. You should alert VCS on call.

In some cases, `pulsenotifier` may repeatedly crash due to a malformed input message, bad data, or some such. Essentially, the process encounters bad input, crashes, restarts via systemd, encounters the same message again, crashes, and the cycle repeats until systemd gives up. This scenario should be rare, which is why the daemon doesn't ignore *bad* messages (ignoring messages could lead to data loss).

If the daemon becomes wedged on a specific message, you can tell the daemon to skip the next message by running:

```
$ /var/hg/venv_tools/bin/vcsreplicator-pulse-notifier --skip /etc/mercurial/notifications.ini
```

This command will print a message like:

```
skipped hg-repo-init-2 message in partition 0 for group pulsenotifier
```

Then exit. You can then restart the daemon (if necessary) via:

```
$ systemctl start pulsenotifier.service
```

Repeat as many times as necessary to clear through the *bad* messages.

Important: If you skip messages, please file a bug against `Developer Services :: hg.mozilla.org` and include the systemd journal output for `pulsenotifier.service` showing the error messages.

---

check_snsnotifier_lag

`check_snsnotifier_lag` monitors the lag of Amazon SNS *Change Notifications* in reaction to server events.

This check is essentially identical to `check_pulsenotifier_lag` except it monitors the service that posts to Amazon SNS as opposed to Pulse. Both services share common code. So if one service is having problems, there's a good chance the other service is as well.

The consumer daemon being monitored by this check is tied to the `snsnotifier.service` systemd service. Its logs are in `/var/log/snsnotifier.log`.

Expected Output Output is essentially identical to *check_pulsenotifier_lag*.

Remediation to Check Failure Remediation is essentially identical to *check_pulsenotifier_lag*.

The main differences are the names of the services impacted.

The systemd service is `snsnotifier.service`. The daemon process is `/var/hg/venv_tools/bin/vcsreplicator-sns-notifier`.

Adding/Removing Nodes from Zookeeper and Kafka

When new servers are added or removed, the Zookeeper and Kafka clusters may need to be *rebalanced*. This typically only happens when servers are replaced.

The process is complicated and requires a number of manual steps. It shouldn't be performed frequently enough to justify automating it.

Adding a new server to Zookeeper and Kafka

The first step is to assign a Zookeeper ID in Ansible. See https://hg.mozilla.org/hgcustom/version-control-tools/rev/da8687458cd1 for an example commit. Find the next available integer **that hasn't been used before**. This is typically `N+1` where `N` is the last entry in that file.

Note: Assigning a Zookeeper ID has the side-effect of enabling Zookeeper and Kafka on the server. On the next deploy, Zookeeper and Kafka will be installed.

Deploy this change via `./deploy hgmo`.

During the deploy, some Nagios alerts may fire saying the Zookeeper ensemble is missing followers. e.g.:

```
hg is WARNING: ENSEMBLE WARNING - only have 4/5 expected followers
```

This is because as the deploy is performed, we're adding references to the new Zookeeper server before it is actually started. These warnings should be safe to ignore.

Once the deploy finishes, start Zookeeper on the new server:

```
$ systemctl start zookeeper.service
```

Nagios alerts for the Zookeeper ensemble should clear after Zookeeper has started on the new server.

Wait a minute or so then start Kafka on the new server:

```
$ systemctl start kafka.service
```

At this point, Zookeeper and Kafka are both running and part of their respective clusters. Everything is in a mostly stable state at this point.

Rebalancing Kafka Data to the New Server

When the new Kafka node comes online, it will be part of the Kafka cluster but it won't have any data. In other words, it won't really be used (unless a cluster event such as creation of a new topic causes data to be assigned to it).

To have the new server actually do something, we'll need to run some Kafka tools to rebalance data.

The tool used to rebalance data is `/opt/kafka/bin/kafka-reassign-partitions.sh`. It has 3 modes of operation, all of which we'll use:

1. Generate a reassignment plan
2. Execute a reassignment plan
3. Verify reassignments have completed

All command invocations require a `--zookeeper` argument defining the Zookeeper servers to connect to. The value for this argument should be the `zookeeper.connect` variable from `/etc/kafka/server.properties`. e.g. `localhost:2181/hgmoreplication`. **If this value doesn't match exactly, the "–generate" step may emit empty output and other operations may fail.**

The first step is to generate a JSON document that will be used to perform data reassignment. To do this, we need a list of broker IDs to move data to and a JSON file listing the topics to move.

The list of broker IDs is the set of Zookeeper IDs as defined in `ansible/group_vars/hgmo` (this is the file you changed earlier to add the new server). Simply select the servers you wish for data to exist on. e.g. `14,15,16,17,20`.

The JSON file denotes which Kafka topics should be moved. Typically every known Kafka topic is moved. Use the following as a template:

```
{
  "topics": [
    {"topic": "pushdata"},
    {"topic": "replicatedpushdata"},
    {"topic": "replicatedpushdatapending"},
  ],
  "version": 1
}
```

Hint: You can find the set of active Kafka topics by doing an `ls /var/lib/kafka/logs` and looking at directory names.

Once you have all these pieces of data, you can run `kafka-reassign-partitions.sh` to generate a proposed reassignment plan:

```
$ /opt/kafka/bin/kafka-reassign-partitions.sh \
  --zookeeper <hosts> \
  --generate \
  --broker-list <list> \
  --topics-to-move-json-file topics.json
```

This will output 2 JSON blocks:

```
Current partition replica assignment

{...}
Proposed partition reassignment configuration

{...}
```

You'll need to copy and paste the 2nd JSON block (the proposed reassignment) to a new file, let's say `reassignments.json`.

Then we can execute the data reassignment:

```
$ /opt/kafka/bin/kafka-reassign-partitions.sh \
  --zookeeper <hosts> \
  --execute \
  --reassignment-json-file reassignments.json
```

Data reassignment can take up to several minutes. We can see the status of the reassignment by running:

```
$ /opt/kafka/bin/kafka-reassign-partitions.sh \
  --zookeeper <hosts> \
  --verify \
  --reassignment-json-file reassignments.json
```

If your intent was to move Kafka data off a server, you can verify data has been removed by looking in the `/var/lib/kafka/logs` data on that server. If there is no topic/partition data, there should be no sub-directories

in that directory. If there are sub-directories (they have the form `topic-<N>`), adjust your `topics.json` file, generate a new `reassignments.json` file and execute a reassignment.

Removing an old Kafka Node

Once data has been removed from a Kafka node, it can safely be turned off.

The first step is to remove the server from the Zookeeper/Kafka list in Ansible. See https://hg.mozilla.org/hgcustom/version-control-tools/rev/adc5024917c7 for an example commit.

Deploy this change via `./deploy hgmo`.

Next, stop Kafka and Zookeeper on the server:

```
$ systemctl stop kafka.service
$ systemctl stop zookeeper.service
```

At this point, the old Kafka/Zookeeper node is shut down and should no longer be referenced.

Clean up by disabling the systemd services:

```
$ systemctl disable kafka.service
$ systemctl disable zookeeper.service
```

Kafka Nuclear Option

If Kafka and/or Zookeeper lose quorum or the state of the cluster gets *out of sync*, it might be necessary to *reset* the cluster.

A hard *reset* of the cluster is the *nuclear option*: full data wipe and starting the cluster from scratch.

A full reset consists of the following steps:

1. Stop all Kafka consumers and writers
2. Stop all Kafka and Zookeeper processes
3. Remove all Kafka and Zookeeper data
4. Define Zookeeper ID on each node
5. Start Zookeeper 1 node at a time
6. Start Kafka 1 node at a time
7. Start all Kafka consumers and writers

To stop all Kafka consumers and writers:

```
# hgweb*
$ systemctl stop vcsreplicator@*.service

# hgssh*
$ systemctl stop hg-master.target
```

You will also want to make all repositories read-only by creating the `/repo/hg/readonlyreason` file (and having the content say that pushes are disabled for maintenance reasons).

To stop all Kafka and Zookeeper processes:

```
$ systemctl stop kafka.service
$ systemctl stop zookeeper.service
```

To remove all Kafka and Zookeeper data:

```
$ rm -rf /var/lib/kafka /var/lib/zookeeper
```

To define the Zookeeper ID on each node (the `/var/lib/zookeeper/myid` file), perform an Ansible deploy:

```
$ ./deploy hgmo
```

Note: The deploy may fail to create some Kafka topics. This is OK.

Then, start Zookeeper one node at a time:

```
$ systemctl start zookeeper.service
```

Then, start Kafka one node at a time:

```
$ systemctl start kafka.service
```

At this point, the Kafka cluster should be running. Perform an Ansible deploy again to create necessary Kafka topics:

```
$ ./deploy hgmo
```

At this point, the Kafka cluster should be fully capable of handling hg.mo events. Nagios alerts related to Kafka and Zookeeper should clear.

You can now start consumer daemons:

```
# hgweb
$ systemctl start vcsreplicator@*.service

# hgssh
$ systemctl start hg-master.target
```

When starting the consumer daemons, look at the journal logs for any issues connecting to Kafka.

As soon as the daemons start running, all Nagios alerts for the systems should clear.

Finally, make repositories pushable again:

```
$ rm /repo/hg/readonlyreason
```

Upgrading Mercurial

This document describes how to upgrade the Mercurial version deployed to hg.mozilla.org.

Managing Mercurial Packages

We generally don't use Mercurial packages provided by upstream or from distros because they aren't suitable or aren't new enough. So, we often need to build them ourselves.

Building RPMs

Mercurial RPMs are created by invoking `make` targets in Mercurial's build system. From a Mercurial source checkout:

```
$ make docker-centos7
```

This will build Mercurial RPMs in isolated Docker containers and store the results in the `packages/` directory. `.rpm` files can be found under `packages/<distro>/RPMS/*.rpm`. e.g. `packages/centos7/RPMS/x86_64/mercurial-3.9-1.x86_64.rpm`.

Building .deb Packages

The process for producing Debian .deb packages is similar: run Mercurial's make targets for building packages inside Docker:

```
$ make docker-ubuntu-xenial
```

`.deb` files will be available in the `packages/` directory.

Uploading Files to S3

Built packages are uploaded to the `moz-packages` S3 bucket.

CentOS 6 packages go in the `CentOS6` folder. CentOS 7 packages in the `CentOS7` folder.

When uploading files, they should be marked as world readable, since we have random systems downloading from this bucket.

Upgrading And Modifying Templates

The repository contains a vendored copy of Mercurial's templates plus modifications in the `hgtemplates/` directory. These templates are used by the hgweb server.

We have made several modifications to the templates. The most significant modification is the addition of the *gitweb_mozilla* theme, which is a fork of the *gitweb* theme. We have also made a number of changes to the *json* theme to facilitate rendering additional data not exposed by Mercurial itself.

Modifying Templates

When modifying a template, it isn't enough to simply change a template file in `hgtemplates/`: you must also track that change by recording it somewhere in `hgtemplates/.patches/`.

Most modifications are tracked by patch files. Essentially, there exists a standalone patch file describing the change. To modify a template via patch file, do the following:

1. Create a new Mercurial changeset like you normally would. i.e. make file changes and `hg commit` the result.
2. Export the just-created changeset to a standalone patch file. e.g. `hg export . > hgtemplates/.patches/my-change.patch`.
3. Track the new patch file via `hg add hgtemplates/.patches/<name>.patch`.
4. Modify the `hgtemplates/.patches/series` file and add the new patch file to the list.
5. Run `run-tests /hgserver/tests/test-template-sync.t` and verify the test passes.

`test-template-sync.t` verifies that the current state of the checkout matches what would be obtained if all modifications were performed on a fresh copy of the templates. In other words, it verifies we can reproduce the current state of the templates.

If the test passes, `hg commit --amend` or `hg histedit` your changesets so the modifications to `hgtemplates/.patches` are part of the changeset that modifies files in `hgtemplates/`. Then submit that for review and land when acceptable.

For other modifications (such as adding or removing a file), see the file lists at the top of `hgtemplates/.patches/mozify-templates.py` to influence behavior.

Upgrading Templates

When Mercurial is upgraded, we need to synchronize our vendored templates with the new templates from upstream.

To do that, we run the `hgtemplates/.patches/mozify-templates.py` script. This script will:

1. Wipe away `hgtemplates/`.
2. Copy the canonical templates from upstream into `hgtemplates/`.
3. Perform special modifications to templates (notably adding and removing certain files and performing hard-coded template transforms).
4. Attempt to apply and commit each patch listed in `hgtemplates/.patches/series`.

Important: Ensure your working directory is clean and `hgtemplates/` is free of untracked files before continuing. Run `hg revert -C hgtemplates/` and `hg purge hgtemplates/` to do this.

To perform an upgrade:

```
$ hgtemplates/.patches/mozify-templates.py /path/to/mercurial/templates \
    hgtemplates hgtemplates
```

e.g.:

```
$ hgtemplates/.patches/mozify-templates.py venv/mercurials/4.6.2/lib/python2.7/site-packages/mercuria
    hgtemplates hgtemplates
```

This tells the script to copy templates from the 1st argument, to grab patches and files from the 2nd argument, and to write the result into the 3rd argument.

If everything is successful, several commits would have been made. You can use e.g. `hg show stack` (assuming the `show` extension is enabled) to see them. These changesets can be removed via `hg prune` or `hg strip` without causing harm.

If the script fails, chances are it failed to run `hg import` to apply a patch. Your working directory may or may not be in a good state. Check that with `hg status` and resolve via `hg revert` etc as appropriate.

To recover from a non-working patch file, you'll need to update the failed patch file until it applies cleanly. To do that, look at the process output for the name of the patch file that failed to apply. Next, you'll attempt to apply it manually. e.g. if the `foo.patch` file fails:

```
$ hg import --partial hgtemplates/.patch/foo.patch
```

You will then need to resolve any conflicts, fix the files until they are in the state you want, etc. Then `hg commit --amend` the result. This will produce a new changeset with a working version of the patch.

Next, you will update the failing `.patch` file with the new version and commit the result. e.g.

> $ hg export > hgtemplates/.patch/foo.patch $ hg commit -m 'hgtemplates: update foo.patch for Mercurial 4.7 upgrade'

Then you need to start the template upgrade process over from the beginning with the modified `.patch` file in place. e.g.:

```
$ hg up @
$ hg rebase -s tip -d .
$ hgtemplates/.patches/mozify-templates.py venv/mercurials/4.6.2/lib/python2.7/site-packages/mercuria
    hgtemplates hgtemplates
```

You can also safely `hg prune` or `hg strip` the changesets produced by `mozify-templates.py`.

Once you've repeated this process and `mozify-templates.py` completes without error, `hgtemplates/` now contains the upstream templates plus our modifications.

Then, modify `hgserver/tests/test-template-sync.t` so it picks up the Mercurial templates from the appropriate Mercurial version in its `mozify-templates.py` invocation. And run this test and verify all is happy. Then commit that change.

At this point, the repository has several commits. There could be modifications to `hgtemplates/.patches/`. There will be changesets tracking the upstream changes to `hgtemplates/` and changes made by each patch. And there should be a changeset for the change to `test-template-sync.t`.

At this point, it is recommended to run `hg histedit` and roll all the changesets together. This will produce a unified changeset containing every change. It should effectively be a diff of the upstream changes plus whatever changes to patches were needed to accommodate upstream changes. This changeset should be suitable for review.

GitHub Webhooks

Mozilla collects and republishes GitHub Webhooks for a number of Mozilla's organizations and projects.

Overall Architecture

GitHub Webhooks are configured at the organizational or project level to publish `application/json` payloads to https://3abyt2fapj.execute-api.us-west-2.amazonaws.com/prod/webhook.

These HTTP requests are delivered to an Amazon API Gateway service operated by the Developer Productivity team. Each webhook request invokes an AWS Lambda function which does the following:

- Publishes the record to an AWS Kinesis Firehose
- Publishes the record to an *all* AWS SNS topic and optionally a a *public* AWS SNS topic if the event is non-private.

Data published to the Kinesis Firehose is flushed to Amazon S3 for long-term storage and to facilitate analytics.

Additional AWS Lambda functions consume the *public* SNS topic and republish events to other channels, such as Pulse.

Private Events

While the ingestion server often receives *all* events for an organization or repository, not all events are republished in public channels.

The following events are excluded from the public:

- Any event belonging to a *private* repository
- Team membership changes (`membership` and `team_add` events)
- Transition of repository from private to public (`public` event)
- Repository creation, deletion, or public/private transitions (`repository` event)

- Any new events GitHub adds that aren't in a list of allowed events

Pulse Notifications

Pulse is a RabbitMQ exchange operated by Mozilla that serves as a nexus of event publishing for various systems.

GitHub Webhook events are republished to the exchange/github-webhooks/v1 exchange.

The routing key for each message is of the form `<repository>/<event>` where `<repository>` is the GitHub account/organization + repository and `<event>` is the GitHub event name. e.g. `mozilla/gecko-dev/push` or `servo/servo/issues`.

The JSON message published to pulse has the following relevant keys:

`event` GitHub event name. e.g. `push`, `issues`, or `status`.

`request_id` UUID uniquely identifying this message. The ID is generated by GitHub.

`payload` The payload of the GitHub event. The formats are documented at https://developer.github.com/v3/activity/events/types/.

Delivery of GitHub events to Pulse is best effort. If Pulse is down, data may fail to publish.

SNS Topic

Non-private GitHub events are published to the `arn:aws:sns:us-west-2:699292812394:github-webhooks-public` AWS SNS topic.

Kinesis Firehose and S3 Access

Access to the streaming GitHub data in Kinesis Firehose and the historical data retained in S3 can be granted on a per-case basis. If interested, email `developer-services@mozilla.org`.

Version Control Synchronization

This section of the documentation describes services, utilities, and accumulated knowledge for *synchronizing* the content of version control repositories. This involves replicating/mirroring repositories, converting repositories from one format to another (e.g. between Mercurial and Git), rewriting commits within and across repositories, etc.

Development Guide

Repositories and Relevant Code

Most code related to version control syncing is in the version-control-tools repo. In this repository, the following directories are relevant:

ansible/roles/vcs-sync Ansible role for version control sync server.

docs/vcssync Sphinx documentation for everything related to version control synchronization (you are reading this now).

vcssync Python package containing code and services for everything related to version control synchronization.

Code reviews for everything in version-control-tools should be conducted with Phabricator. Reviews should be directed at `gps` or `glob`.

Terraform configs for AWS infrastructure for VCS Sync lives in the devservices-aws repo, specifically in the `vcssync` directory. Change proposals are submitted via GitHub Pull Requests. Reviews are typically handled by someone on the Ops Team, namely `fubar`, `dividehex`, or `dhouse`. For new infrastructure, ``gps or `glob` should likely take a look.

Creating a Development and Testing Environment

From a fresh `version-control-tools` checkout, run the following to create a development and testing environment:

```
$ ./create-environment vcssync
```

Then activate the environment in your shell via:

```
$ source activate venv/vcssync/bin/activate
```

Running Tests

To run the vcssync tests, run:

```
$ ./run-tests
```

Using Betamax for HTTP Request Replaying

We use the Betamax Python package to facilitate testing HTTP requests against various services, such as the GitHub API.

When the `BETAMAX_LIBRARY_DIR` and `BETAMAX_CASSETTE` environment variables are defined, Betamax is configured to use the cassette (recording of HTTP interactions) specified. Betamax's record mode is set to `none`, which means that only HTTP interactions saved in the cassette are allowed.

The `vcssync/tests/record_cassettes.py` script is used to record cassettes (read: perform actual interactions with real servers and save the results). Run this script from an activated virtualenv to create/update/re-record cassettes. Minor changes in the cassettes (such as dates and request IDs) are expected to change. Other things may change over time and changes should be scrutinized during review.

`record_cassettes.py` requires a GitHub API token. Go to https://github.com/settings/tokens/new to generate one. It should only need minimal privileges. While the cassettes shouldn't save your token, it is a good practice to delete the token once you're done recording cassettes.

Servo Repository Syncing

Aspects of development on the Servo and Firefox/Gecko web platforms are closely related, with some components shared between the two projects. Developers often want to make changes for and test against both projects at once. For this reason, Mozilla has an automated mechanism for *synchronizing* the two code repositories.

Architecture

Note: This section details the architecture as it is implemented today. It does not (yet) detail the final, planned architecture.

Servo is canonically developed on GitHub using the Git version control tool. Servo makes heavy use of the GitHub workflow receiving changes (*Pull Requests*). When a Git branch is accepted, it is merged into the *master* branch of the Servo Git repository.

Firefox is canonically developed on hg.mozilla.org using the Mercurial version control tool. There are multiple repositories that are periodically merged into one another.

The commit history of the Servo projected is *vendored* into the *servo/* directory of the Firefox Mercurial repository. The mechanism by which this happens is roughly as follows:

1. Git commits are normalized to provide a sanitized, suitable-for-Firefox representation of history.
2. The rewritten Git commits are converted to Mercurial changesets.
3. The Mercurial changesets are essentially replayed onto the Firefox repository into the *servo/* directory.

Git History Rewriting

Before Servo's history is converted to Mercurial, it goes through a significant rewriting and normalization process. The following transforms are performed:

- Merge commits are removed, leaving only the first-parent ancestry (commits on non-first-parent DAG branches tend to not be very useful in the context of Firefox and pollute the history of Firefox, which tries to not use merge commits).
- Certain directories (like web platform tests) are removed from history because they are large and/or redundant with files already in the Firefox repo.
- Non-useful annotations from commit messages (such as the boilerplate markdown for reviewable.io) are removed.
- The commit message summary line is prefixed with `servo:` and contains the GitHub pull request number, title and source (data obtained from GitHub API).
- The Git committer field is set to the value of the author field.
- Annotations to the commit's original source URL and revision are added to the commit message.

The end result is a Git head with 100% linear history (no merge commits) and consistently formatted commit messages.

Mercurial Conversion

The rewritten Git history is converted to a Mercurial repository using `hg convert`. As part of the conversion:

- All data related to Git submodules is dropped (Servo stopped using submodules in 2015).
- Aggressive copy detection is performed matching at 75% similarly.

Repository Overlay into Firefox

After the rewritten Servo repository is converted to Mercurial, the next step is to incorporate those changesets into the Firefox repository.

This is accomplished through a process called *overlaying*. Essentially, this process builds a list of files (*manifest* in Mercurial terminology) that is a union of files in the Firefox and Servo repositories, prefixing files from the Servo repository with the path `servo/`. A changeset copying details from the corresponding source changeset is committed. This result looks like someone took all the diffs from the Mercurial Servo repository and applied them to the Firefox repository.

Operational Guide

The processes for converting the Servo repository and overlaying it into the Firefox repository live on a *vcs-sync* server. The processes are largely stateless. All that's needed to run an instance (beside the code of course) are some credentials to access various authenticated services.

The service configuration lives in the `vcs-sync` Ansible role in the version-control-tools repository. There is a standalone `servo-sync.yml` playbook for configuring just the Servo pieces.

The logical service is composed of multiple systemd services and related units, each responsible for one part of the pipeline.

servo-linearize.service

The `servo-linearize.service` systemd service is responsible for rewriting Git history and converting it to Mercurial.

This one-shot service runs in response to a detected push to the Servo Git repository and periodically via a timer (`servo-linearize.timer`).

When executed, this process:

1. Fetches changes from the Servo Git repository to a local clone.
2. Rewrites any Git commits from the origin that haven't already been rewritten.
3. Mirrors the Git repo (contains original and converted refs/heads) to https://github.com/mozilla/converted-servo.
4. Converts any unconverted Git commits to Mercurial changesets
5. Pushes the Mercurial repository to https://hg.mozilla.org/projects/converted-servo-linear
6. Exits.

The service should be safe to run at any time. If there is no work to do, it no-ops.

servo-overlay.service

The `servo-overlay.service` systemd service is responsible for overlaying the Mercurial Servo repository onto a Firefox repository. This one-shot service runs in response to a detected push to the Mercurial Servo repository and periodically via a timer (`servo-overlay.timer`).

When executed, this process:

1. Pulls the latest revision of the Firefox Mercurial repository onto which changesets should be based.
2. Pulls the Mercurial Servo repository.
3. Finds changesets from the Mercurial Servo repository not yet applied in the Firefox repository.
4. Overlays each Servo changeset into the Firefox repository until done.
5. Attempts to push the result.

The service should be safe to run at any time. If there is no work to do, it no-ops.

In some situations, the operation may fail. For example, the Firefox repository is closed and pushes aren't being allowed. When this happens, a subsequent invocation will delete what was left over from the previous failed attempt and redo the process.

servo-pulse-monitor.service

The `servo-pulse-monitor.service` systemd service is a daemon that subscribes to Pulse notifications for hg.mozilla.org and GitHub. When it sees a push to the Servo Git repository, it starts `servo-linearize.service`. When it sees a push to the `converted-servo-linear` Mercurial repository, it starts `servo-overlay.service`.

The purpose of the Pulse monitor daemon is to react to repository change events with minimal delay. This allows commits to be *synchronized* from Servo to Firefox in a matter of seconds. This is both faster and more efficient than polling servers for activity.

Neither Pulse nor GitHub have highly robust delivery guarantees. So it is possible change notification messages may be lost. For this reason, systemd timers periodically trigger `servo-linearize.service` and `servo-overlay.service`. In the event of a Pulse notification failure, the maximum time to handle is effectively capped at the period of these timers (as opposed to when the next Pulse notification is delivered - which could be hours or more).

servo-sync.target

For convenience, the `servo-sync.target` systemd unit can be used to start and stop all services associated with Servo VCS syncing.

servo-backout-pr

The `servo-backout-pr` systemd service is responsible for creating GitHub Pull Requests on the `servo/servo` repository when changes are backed out of the `/servo` directory on the integration-autoland repository. This service runs in response to any push to the integration-autoland repository, and periodically via a timer (`servo-backout-pr.timer`).

When executed it:

1. Fetches changes to the integration-autoland repository to a local clone.
2. Scans for backout commits pushed since the last time the service ran.
3. Fetches upstream changes to a GitHub fork of servo/servo.
4. Applies the backout to the servo/servo fork, stages and pushes to GitHub.
5. Creates a Pull Request for merging into servo/servo.

In order to ensure the two repositories (servo and autoland) are kept in sync, servo's autoland bot `homu` will merge Pull Requests generated by this service with a higher priority than any other request.

Errors and Issues

Error running mach: ['vendor', 'rust']

```
Error running mach:  ['vendor', 'rust']
all possible versions conflict with previously selected versions of ..
```

Occasionally changes to servo's dependencies result in `mach vendor rust` failing (https://bugzilla.mozilla.org/show_bug.cgi?id=1361005).

A stylo developer is required to resolve this issue; ask for assistance in the `#servo` channel on IRC. They will have to manually vendor the correct versions and push the changes to `integration-autoland`. The next time `servo-vcs-sync` runs it will automatically correct itself.

"Out of sequence" servo commit This describes a situation where a commit was pushed to servo/servo after a change was backed out of integration-autoland, but before the servo-backout-pr service created the backout pull request on GitHub.

If the servo commit doesn't conflict with the backout: Stylo devs push the servo commit directly to integration-autoland, after the backout commit, with extra data set that signals to servo-vcs-sync to skip the commit.

This requires installation of the mercurial extension by adding the following to your `~/.hgrc` file. The extension is part of the `version-control-tools` repo.

```
[extensions]
manualoverlay=~/.mozbuild/version-control-tools/hgext/manualoverlay
```

Apply the servo commit to your local integration-autoland clone, and commit with a `--manualservosync` argument which points to the corresponding servo/servo pull request.

```
hg commit --manualservosync https://github.com/servo/servo/pull/12345
```

Finally push the changes to hg.mozilla.org from the command line.

Once this commit has been pushed with the correct extra data, `servo-vcs-sync` should resume normal operations. The ordering of the commits will be swapped with regards to the two repositories (autoland will be backout→commit, servo will be commit→backout), however as the final contents of the files will be identical, the `servo-vcs-sync` process can resume and process the next servo commit.

Work is underway to detect this non-conflicting state and automatically resolve it.

If the servo commit conflicts with the backout: Should one or many of the out-of-sequence commits conflict with the backout extra work is required to bring the two repositories back into alignment.

- The pull request for the gecko backout will not be able to be merged automatically by homu.
- The servo devs will have to push fixup commits to the backout PR to make it mergeable.
- The stylo devs will also have to push the servo commit directly to integration-autoland, with a different set of fixups to address the merge conflicts there. This commit needs to be committed with the `manualservosync` switch as per the non-conflicting workflow.
- It is imperative that the final results of the two fixup changes result in identical file contents.

Merge conflicts within the backout service The backout service performs a merge from the master branch to its backout branch when creating / updating backout pull requests. It is possible (in a rare race condition) that this merge could have merge conflicts, causing the backout service to fail.

Provisioning a New Instance

The Servo VCS Syncing *appliance* can be provisioned in a relatively turn-key manner. Generally speaking, it should be safe to destroy the existing instance and provision a new one at any time.

The EC2 instance and other supporting AWS infrastructure is managed by Terraform. From the `devservices-aws` Git repo, go to the `vcssync` directory and run `terraform plan` then `terraform apply` if the proposed changed check out.

After a minute or two, you should be able to SSH into `servo-vcs-sync.mozops.net` via the bastion host in us-west-2.

Note: The instance reboots after initialization to apply any system package updates that may require a reboot.

Once you have a fresh instance, you'll need to provision it.

The first step is to install the secrets on the host. These include SSH keys, passwords, and other tokens. The secrets file is encrypted in a *vault*. Have a friendly Ops friend decrypt the file then run `ansible-playbook -i hosts vcssync-seed-secrets.yml` from `ansible/` in `version-control-tools`. This will copy the secrets file to the host.

Once the secrets file is in place on the server, Ansible can do the reset. From `version-control-tools`:

```
$ ./deploy vcs-sync
```

This will take a while on initial provision because it needs to install system packages, Python virtualenvs, and pre-clone various repositories.

Accumulated Knowledge

Over the years, Mozilla has had to deploy numerous solutions for rewriting and synchronizing the history of various version control repositories. Synchronization is an inherently difficult problem and as one can expect, we've learned a lot through trial and error. This document serves to capture some of that knowledge.

Thoughts on `git filter-branch`

`git filter-branch` is Git's built-in tool for complex repository history rewriting. It accepts as arguments *filters* - or executables or scripts - to perform actions at specific stages. e.g. *rewrite the commit message* or *modify the files in a commit*.

While `git filter-branch` generally gets the job done for simple, one-time rewrites, we've found that it isn't suitable for a) rewriting tree content (read: files in commits) of large repositories b) use in incremental conversion scenarios c) where robustness or complete control is needed or d) where performance is important.

The following are some of the deficiencies we've encountered with `git filter-branch`:

Index update performance Rewriting files in history using `git filter-branch` requires either a `--tree-filter` or an `--index-filter` (there is a `--subdirectory-filter` but it is internally implemented as an index filter of sorts).

`--tree-filter` should be avoided at all costs because having to perform a working copy *sync* on every commit adds a lot of overhead. This is especially true in scenarios where you are removing a large directory from history. e.g. if you are rewriting history to remove 10,000 files from a directory, each commit processed by `--tree-filter` will need to repopulate the files from that directory on disk. That's **extremely** slow.

`--index-filter` is superior to `--tree-filter` in that it only needs to populate the Git index on every commit (as opposed to the working copy). This is substantially faster. But, our experience is that `--index-filter` is still a bit slow.

Writing the index requires I/O (it is a file). Some index update operations also require I/O (to e.g. `stat()` paths). For this reason, if using an `--index-filter`, it is highly recommended to perform operations on a

tmpfs volume (`-d` argument). Failure to do so could result in significant slowdown due to waiting on filesystem I/O.

When rewriting large parts of the index, we found the performance of `git update-index` against the existing index to be a bit slow. This is even when using `--assume-unchanged` to prevent verifying changes with the filesystem. In some cases (including one where we deleted 90% of the files in a repository), we found that writing a new index file from scratch (by setting the `$GIT_INDEX_FILE` environment variable combined with `git update-index --index-info` to produce a new index file) then replacing the existing index was filter than updating the index in place.

We also found the best way to load entries into an index was via `git update-index -z --index-info`.

Overhead of filter invocation Every `--filter-*` argument passed to `git filter-branch` invokes a process for every commit. If you have 4 filters and 10,000 commits, that's 40,000 new processes. If your process startup overhead is 10ms (typical for Python processes), that's 400s right there - and your processes haven't even done any real work yet! By the time you factor in the filter processes doing something, you could be spending dozens of minutes in filters for large repositories.

Complexity around incremental rewriting We often want to perform incremental, ongoing rewriting of a repository. For example, we want to remove a directory and publish the result to a separate repo. `git filter-branch` can be coerced to do this, but it requires a bit of work.

`git filter-branch` is given a *rev-list* of commits to operate on. When doing an incremental rewrite, you need to specify the base commits to *anchor* how far back processing should go. For simple histories, specifying `base..head` *just works*. However, things quickly fall apart in more complicated scenarios. Imagine this history:

```
E
| \
D  F
|  |
C  |
| /
B
|
A
```

If we initially converted `C`, the next conversion could naively specify a *rev-list* of `C..E`. This would include `F` since it is an ancestor of `E`. It would also pull in `B` and `A` since those are ancestors of `F`. This would mean that `git filter-branch` would redundantly operate on `A` and `B`! In the best case, this would lead to overhead and slow down incremental operations. In the worst case it would lead to divergent history. Not good.

This problem can be avoided by using the `^COMMIT` syntax in the *rev-list* to exclude a commit and any of its ancestors. If your repository has very complicated history, you may need to specify `^COMMIT` multiple times, one for each known root in the unconverted *incoming* set of commits.

Another problem with incremental operations is grafting *incoming* commits onto the appropriate commit from the last run. Unless you take action, `git filter-branch` will parent your new commit in the *source* DAG, which is not what you want for incremental conversions!

While you can solve this problem with a `--parent-filter` to rewrite parents of processed commits, we found this approach too complicated. Instead, before incremental conversion, we walked the DAG of the to-be-processed commits. For each root node in that sub-graph, we created a Git graft (using the `info/grafts` file) mapping the old parent(s) to the already-converted parents. The `info/grafts` file was only modified for the duration of `git filter-branch`. A benefit of this approach over `--parent-filter` was you only need to process the graft mapping once before conversion, as opposed for every commit. This mattered for performance.

Ignoring commits from outside first parent ancestry One of our common repository rewriting scenarios is stripping out merge commits from a repository (we like linear history). It is possible to do this with `git filter-branch` by using a `--parent-filter` that simply only returns the first parent.

However, there is no easy way to tell `git filter-branch` to only convert the first parent ancestry. While `git log` has a `--first-parent` argument, there is no *rev-list* syntax to do this. And, listing each first parent commit explicitly will exhaust argument length for large repositories.

So, you either have to call `git filter-branch` in batches with single commits or have to live with `git filter-branch` converting commits not in the first parent ancestry. The latter can have major performance implications (e.g. you process 80% more commits than you need to).

Control over refs `git filter-branch` automatically updates the source ref it is converting. This is slightly annoying.

`git filter-branch` seems like an appropriate tool for systematic repository rewriting. But our experiences tell us otherwise. If you are a developer and need it for a quick one-off or if you are performing a one-time rewrite, it's probably fine. But for ongoing, robust rewriting, it's far from our first choice.

A case study demonstrating our lack of content for `git filter-branch` is converting the history of the Servo repository to Mercurial so we could vendor it into Firefox. This conversion had a few requirements:

- We wanted to strip a few directories containing 100,000+ files
- We wanted to *linearize* the history so there were no merges
- We wanted to rewrite the commit message
- We wanted to insert hidden metadata in the commit object so `hg convert` would treat it properly

This was initially implemented with `git filter-branch` using 4 filters: *parent*, *msg*, *index*, and *commit*. The *parent* filter was implemented with `sed`. The rest were Python scripts. Rewriting ~23,000 commits with `git filter-branch` took almost 2 hours. That was after spending considerable time to optimize the index filter to run as fast as possible (including doing nothing if the current commit wasn't in first parent ancestry). Without these optimizations and tmpfs, run-time was 5+ hours!

After realizing that we were working around `git filter-branch` more than it was helping us, we rewrote all the functionality in Python, building on top of the *Dulwich* package - a Python implementation of the Git file formats and protocols - *Dulwich*:

- Gave us full control over which commits were processed. No more complexity around incremental operations!
- Allowed us to perform all operations against rich data structures (as opposed to parsing state from filter arguments, environment variables, or by running `git` commands). This was drastically simpler (assume you have knowledge of Git's object types and how they work) and faster to code.
- Allowed us to use a single Python process for rewriting. This eliminated all new process overhead from `git filter-branch`.
- Allowed us to bypass the index completely. Instead, we manipulated Git *tree* objects in memory. While more complicated, this cut down on significant overhead.
- Drastically reduced I/O. Most of this was from avoiding the index. With Dulwich, the only I/O was object reads and writes, which are pretty fast.
- Guaranteed better consistency. When using `git` commands, things like environment variables and `~/.gitconfig` files matter. With Dulwich, this magic wasn't in play and execution is much more tolerable of varying environments.

It took ~4 hours to rewrite the `git filter-branch` based solution to use Dulwich. This was made far easier by the fact that our filters were implemented in Python before. The effort was worth it: **Python + Dulwich performed an identical conversion of the Servo repository in ~10s versus ~2 hours** - a ~700x speedup.

Converting from Git to Mercurial

Git and Mercurial have remarkably similar concepts for structuring commit data. Essentially, both have commit objects a) with a link to a tree or manifest of all files as they exist in that commit b) links to parent commits. Not only is conversion between Git and Mercurial repositories possible, but numerous tools exist for doing it!

While there are several tools that can perform conversions, each has its intended use cases and gotchas.

In many cases `hg convert` for performing an unidirectional conversion of Git to Mercurial *just works* and is arguably the tool best suited for the job (on the grounds that Mercurial itself knows the best way for data to be imported into it). That being said, we've run into a few scenarios where `hg convert` on its own isn't sufficient:

Removing merges from history We sometimes want to remove merge commits from Git history as part of converting to Mercurial. `hg convert` doesn't handle this case well.

In theory, you can provide `hg convert` a splice map that instructs the conversion to remove parents from a merge. And, `hg convert` happily parses this and starts converting away. But it will eventually explode in a few places where it assumes all parents of a source commit exist in the converted history. This could likely be fixed upstream.

Copy/rename detection performance Mercurial stores explicit copy and rename metadata in file history. Git does not. So when converting from Git to Mercurial, `hg convert` asks Git to resolve copy and rename metadata, which it then stores in Mercurial. This more or less *just works*.

A problem with resolving copy and rename metadata is it is very computationally expensive. When Git's `--find-copies-harder` flag is used, Git examines *every* file in the commit/tree to find a copy source. For repositories with say 100,000 files, you can imagine how slow this can be.

Sometimes we want to remove files as part of conversion. If doing the removal inside `hg convert`, `hg convert` will have Git perform the copy and rename detection *before* those discarded files are removed. This means that Git does a lot of throwaway work for files that aren't relevant. When removing tens of thousands of files, the overhead can be staggering.

Copy/rename metadata and deleted files As stated above, Mercurial stores explicit copy and rename metadata in file history. When files are being deleted by `hg convert`, there appears to be some problems where `hg convert` gets confused if the copy or rename source resides in a deleted file. This is almost certainly a correctable bug in `hg convert`.

Behavior for empty changesets When removing files from history (including ignoring Git submodules), it is possible for the converted Git commit to be empty (no file changes).

`hg convert` has (possibly buggy) behavior where it automatically discards empty changesets, but only if a `--filemap` is being used. This means that empty Git commits are preserved unless `--filemap` is used. (A workaround is to specify `--filemap /dev/null`.)

When these scenarios are in play, we've found that it is better to perform the Git to Mercurial conversion in 2 phases:

1. Perform a Git->Git rewrite
2. Convert the rewritten Git history to Mercurial

In cases where lots of files are being removed from Git history, this approach is *highly* recommended because of the performance overhead of processing the unwanted files during `hg convert`.

Vendoring Projects in a Monorepo

A monolithic repository (*monorepo*) is a version control repository containing multiple projects and/or everything related to a project, team, or company. Contrast with an approach where every logical unit exists in its own repository.

While a monorepo may contain content for a project, that doesn't mean that that project is canonically developed in that monorepo. For example, the Firefox monorepo contains the source code for Servo and WebRTC, but neither are

canonically developed against mozilla-central. Instead, commits to the canonical *upstream* repo make their way to the monorepo through various means.

When it comes to vendoring projects in a monorepo, there are several decisions that need to be made. This document attempts to describe them.

Synchronization Frequency

If a project is canonically being developed outside the monorepo, you'll need to decide how frequently upstream changes should be incorporated in the monorepo.

The least structured approach is an *as needed* frequency. Essentially, a human determines when a new version from upstream should be vendored and then action is taken.

The other end of the spectrum is a 1:1 mapping: whenever a change is made upstream, that change is essentially replayed in the monorepo. This doesn't mean commits align exactly (you can squash commits together for example). It does mean that every time a new *push* is made upstream that something attempts to synchronize that change (if relevant) to the monorepo.

Rewriting and Normalization

Monorepos like mozilla-central have standards for what commits should look like. e.g. it favors linear history with commit-level bisectability. And commit messages should be formatted in a certain way.

Unless the monorepo and the upstream project share similar commit authoring *style* conventions or the monorepo doesn't care about consistency, it is likely that upstream commits will be rewritten as part of vendoring in the monorepo. There are various rewriting mechanisms that can be employed:

- Linearizing history to avoid merge commits
- Removing unwanted content from commit messages
- Prefixing commit message summary lines
- Adding annotations in commit message to denote upstream source
- Removing files or directories
- Renaming files or directories
- Running a transformation on file content
- Rewriting author/committer names and times

Each vendored project needs to decide what rewriting and normalization to perform.

Trusting and Auditing Content

Any code or files have the potential to cause harm. For example:

- If a project's build system is invoked as part of building a common project in the monorepo, then from the perspective of that upstream project, you essentially have remote code execution privileges on the machines of developers relying on you.
- A project could relicense as GPLv3 (or similar liberal license), contaminating your code.
- A trademarked or patent protected file could be added, making you liable by extension.
- An attacker may target a vendored project because of lax code review standards in an attempt to attack something in your monorepo or its users.

Since every vendored project in a monorepo represents a point of vulnerability, it is important that a trust relationship or expectation be established. This especially holds true if content is being vendored automatically in near real time. A risk analysis should be performed and an auditing process should be established for all vendored content.

Initial Import

Vendoring is a continuous endeavor. But before you get there, you need to decide how the initial import should be performed.

The big decision that needs to be made is whether to do a bulk, single commit import or whether to import with history. There are pros and cons to each approach.

Single commit import pros:

- Simple to perform (just copy files)
- Minimal overhead for version control storage (no extra history to store)

History import pros:

- Code archeology over the imported project is simpler
- It's possible to bisect over history of the project
- The data will always be in the monorepo and can't disappear if the upstream project goes away
- Forces you to solve rewriting and normalization before any import is done and therefore helps identify mistakes and sub-optimal choices before they are a permanent part of history of the monorepo

Single commit import cons:

- Examining history of the imported project requires using a separate repo and/or tools
- No ability to bisect over history of imported project
- You run the risk of wanting project history in the monorepo later and not having it (this has happened a number of times in the Firefox monorepo, such as with mozharness)
- Data hosted for the *external* project may go away or be tampered with later

History import cons:

- Requires a lot of up-front work and verification
- Can introduce overhead to the repository (e.g. thousands of now-deleted files taking up megabytes of space)
- Can *pollute* history of the monorepo
- Can interfere with bisection operations against other projects in the monorepo

If a history import is performed, there is also the choice of how those (possibly heavily rewritten) commits should be *joined* with the history of the monorepo. The choices are between replaying the commits onto an existing head or introducing a new *root* in the DAG and merging it with an existing head.

The new *root* approach makes *bisect* more manageable by not introducing a range of potentially thousands of commits in the middle of the monorepo's history. In other words, when you naively bisect every commit in the monorepo, you won't spend potentially several iterations in the middle of the history of the imported project, which has no relevant changes to the monorepo in the context of the bisect operation.

The new *root* approach can also make some archeology operations simpler and faster.

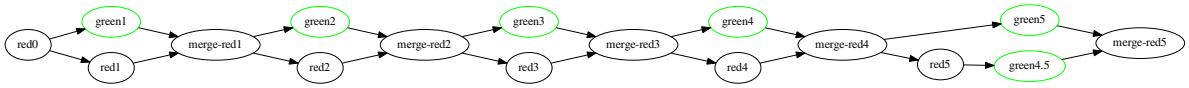
The main downside to the *root* approach is it is somewhat hacky. Some tools can be confused by multiple DAG root nodes in a repository. In practice, we don't think this is a huge issue. The history of the Servo project vendored in the Firefox monorepo was added as a new root node, for example.

Cross-channel L10n

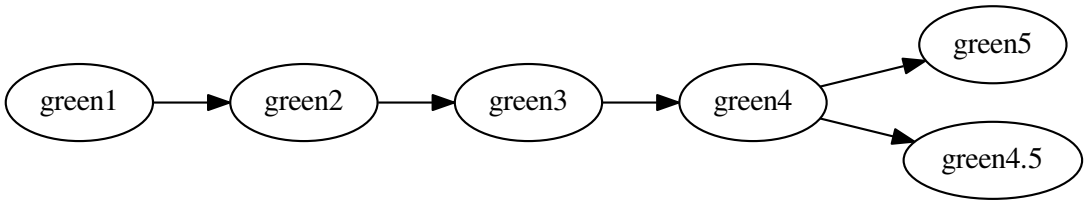
Architecture

Sparse Commit Graph

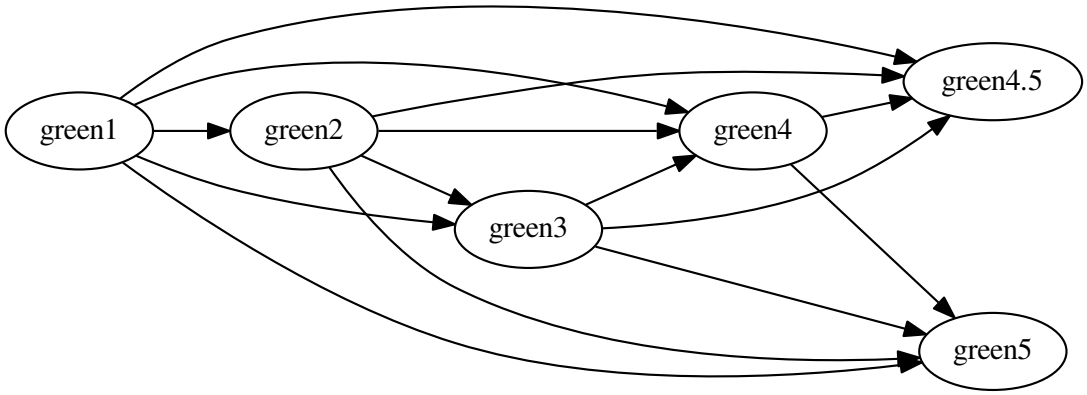
One challenge in the creation of the generated repository is correctly ordering the incoming changesets in a commit graph. Common graphs in mozilla-central look like the following, with "green" nodes being the ones affecting localization that we want to transform.



What we'd expect to get would be a graph that just goes through our nodes, in this case effectively a linear graph.



Natively, mercurial creates a graph that shows all paths from each node to another that can be taken in the full graph, creating, in this case, a graph that connects all nodes.



The code generating our target repository needs to either strip that graph down, or create a sparse graph from scratch.

Creating the cross-channel l10n repository consists of the following architectural components:

1. Creating initial content
2. Updating content per push of original repository

3. Updating content for configuration changes

The intent is that the initial content creation is actually the same as a configuration change that adds new files. One could choose to do otherwise, but this is also a good way to ensure that the change handling configuration changes won't just fail if the incoming content shows unexpected complexity.

Both the creation of initial content as well as the updates per push create commits that match the history of the original commits. This way, the generated diffs match the original diffs best. To do so, we need to create a DAG that only has the commits we're interested in. Details on the creation of the graph for a subset of commits are described in Sparse Commit Graph.

Note: This documentation is for code yet to be written. Its intention is to serve as a guideline for what could and probably should be, not as something that is.

Cross-channel Localization (aka x-channel L10n) allows us to create one localization for all currently shipping versions of Firefox, Firefox for Android, as well as Thunderbird and SeaMonkey.

At any point in time, in en-US we have anywhere between 8 and 10 versions in two conceptual DAGs involved. To have one localization for all, we need to provide the superset of all the strings on all those version in each localization.

The way we do that is by creating a repository containing those English strings, and expose that to localization.

To write this out, say we had three upstream repositories, `mozilla-central`, `releases/mozilla-beta`, and `comm-central`. In comm-central, we have `mail/locales/en-US/mail.properties` with the following content

```
send-mail = Send Mail
```

In mozilla-central, we have `browser/locales/en-US/browser.properties` with

```
open-window = Open Window
close-window = Close Window
open-tab = Open Tab
```

and the corresponding file on the beta branch in releases/mozilla-beta is

```
open-window = Open Window
open-new-window = Open New Window
close-window = Close Window
```

The generated repository should have two files, which paths correspond to those in the upstream repositories, but don't have the `locales/en-US` in them. The contents of `browser/browser.properties` are

```
open-window = Open Window
open-new-window = Open New Window
close-window = Close Window
open-tab = Open Tab
```

with the two shared strings from central and beta, and each of the two unique strings from central and aurora, in a good order in the resulting file. For comm-central, it also has `mail/mail.properties`

```
send-mail = Send Mail
```

For simplicity, we didn't create a diverging branch of this file in this example, but it'd work exactly like in `browser.properties`.

As part of the generation, we also rewrite the configuration files from the upstream repository to be valid configuration files in the generated repository.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search