AcidFS Documentation

Release 1.0

Chris Rossi

Apr 12, 2018

Contents

The filesystem on ACID

AcidFS allows interaction with the filesystem using transactions with ACID semantics. *Git* is used as a back end, and *AcidFS* integrates with the transaction package allowing use of multiple databases in a single transaction.

CHAPTER 1

Features

- Changes to the filesystem will only be persisted when a transaction is committed and if the transaction succeeds.
- Within the scope of a transaction, your application will only see a view of the filesystem consistent with that filesystem's state at the beginning of the transaction. Concurrent writes do not affect the current context.
- A full history of all changes is available, since files are stored in a backing *Git* repository. The standard *Git* toolchain can be used to recall past states, roll back particular changes, replicate the repository remotely, etc.
- Changes to a *AcidFS* filesystem are synced automatically with any other database making use of the *transaction* package and its two phase commit protocol, eg. *ZODB* or *SQLAlchemy*.
- Most common concurrent changes can be merged. There's even a decent chance concurrent modifications to the same text file can be merged.
- Transactions can be started from an arbitrary commit point, allowing, for example, a web application to apply the results of a form submission to the state of your data at the time the form was rendered, making concurrent edits to the same resource less risky and effectively giving you transactions that can span request boundaries.

CHAPTER 2

Motivation

The motivation for this package is the fact that it often is convenient for certain very simple problems to simply write and read data from a fileystem, but often a database of some sort winds up being used simply because of the power and safety available with a system which uses transactions and ACID semantics. For example, you wouldn't want a web application with any amount of concurrency at all to be writing directly to the filesystem, since it would be easy for two threads or processes to both attempt to write to the same file at the same time, with the result that one change is clobbered by another, or even worse, the application is left in an inconsistent, corrupted state. After thinking about various ways to attack this problem and looking at *Git's* datastore and plumbing commands, it was determined that *Git* was a very good fit, allowing a graceful solution to this problem.

CHAPTER 3

Limitations

In a nutshell:

- Only platforms where *fcntl* is available are supported. This excludes Microsoft Windows and probably the JVM as well.
- Kernel level locking is used to manage concurrency. This means *AcidFS* cannot handle multiple application servers writing to a shared network drive.
- The type of locking used only synchronizes other instances of *AcidFS*. Other processes manipulating the *Git* repository without using *AcidFS* could cause a race condition. A repository used by *AcidFS* should only be written to by *AcidFS* in order to avoid unpleasant race conditions.

All of the above limitations are a result of the locking used to synchronize commits. For the most part, during a transaction, nothing special needs to be done to manage concurrency since *Git's* storage model makes management of multiple, parallel trees trivially easy. At commit time, however, any new data has to be merged with the current head which may have changed since the transaction began. This last step should be synchronized such that only one instance of *AcidFS* is attempting this at a time. The mechanism, currently, for doing this is use of the *fcntl* module which takes advantage of an advisory locking mechanism available in Unix kernels.

CHAPTER 4

Usage

AcidFS is easy to use. Just create an instance of *acidfs.AcidFS* and start using the filesystem:

```
import acidfs

fs = acidfs.AcidFS('path/to/my/repo')
fs.mkdir('foo')
with fs.open('/foo/bar', 'w') as f:
    print >> f, 'Hello!'
```

If there is not already a *Git* repository at the path specified, one is created. An instance of *AcidFS* is not thread safe. The same *AcidFS* instance should not be shared across threads or greenlets, etc.

The transaction package is used to commit and abort transactions:

```
import transaction

transaction.commit()
# If no exception has been thrown, then changes are saved!  Yeah!
```

Note: If you're using Pyramid, you should use pyramid_tm. For other WSGI frameworks there is also repoze.tm2.

CHAPTER 5

Commit Metadata

The transaction package has built in support for providing metadata about a particular transaction. This metadata is used to set the commit data for the underlying git commit for a transaction. Use of these hooks is optional but recommended to provide meaningful audit information in the history of your repository. An example is the best illustration:

```
import transaction

current = transaction.get()
current.note('Added blog entry: "Bedrock Bro Culture: Yabba Dabba Dude!"')
current.setUser('Fred Flintstone')
current.setExtendedInfo('email', 'fred@bed.rock')
```

A users's name may also be set by using the `setExtendedInfo` method:

```
current.setExtendedInfo('user', 'Fred Flintstone')
```

The keys `acidfs_user` and `acidfs_email` are available in extended info in case you are sharing a transaction with a system that has a different notion of what user and email should be set to. Substance D, for examples, sets the user to an integer OID that represents the user in its system, but that might not be what you want to see in the Git log for your repository:

```
current.setExtendedInfo('acidfs_user', 'Fred Flintstone')
current.setExtendedInfo('acidfs_email', 'fred@bed.rock')
```

The transaction might look something like this in the git log:

```
commit 3aa61073ea755f2c642ef7e258abe77215fe54a2
Author: Fred Flintstone <fred@bed.rock>
Date:   Sun Sep 16 22:08:08 2012 -0400

    Added blog entry: "Bedrock Bro Culture: Yabba Dabba Dude!"
```

CHAPTER 6

API

class acidfs.**AcidFS**(*repo*, *head='HEAD'*, *create=True*, *bare=False*, *user_name=None*, *user_email=None*, *name='AcidFS'*, *path_encoding='ascii'*)

An instance of *AcidFS* exposes a transactional filesystem view of a *Git* repository. Instances of *AcidFS* are not threadsafe and should not be shared across threads, greenlets, etc.

Paths

Many methods take a *path* as an argument. All paths use forward slash / as a separator, regardless of the path separator of the underlying operating system. The path / represents the root folder of the repository. Paths may be relative or absolute: paths beginning with a / are absolute with respect to the repository root, paths not beginning with a / are relative to the current working directory. The current working directory always starts at the root of the repository. The current working directory can be changed using the `chdir()` and `cd()` methods.

Constructor Arguments

`repo`

The path to the repository in the real, local filesystem.

`head`

The name of a branch to use as the head for this transaction. Changes made using this instance will be merged to the given head. The default, if omitted, is to use the repository's current head.

`create`

If there is not a Git repository in the indicated directory, should one be created? The default is *True*.

`bare`

If the Git repository is to be created, create it as a bare repository. If the repository is already created or *create* is False, this argument has no effect.

`user_name`

If the Git repository is to be created, set the user name for the repository to this value. This is the same as creating the repository and running *git config user.name "<user_name>"*.

`user_email`

If the Git repository is to be created, set the user email for the repository to this value. This is the same as creating the repository and running *git config user.email "<user_email>"*.

`name`

Name to be used as a sort key when ordering the various databases (datamanagers in the parlance of the transaction package) during a commit. It is exceedingly rare that you would need to use anything other than the default, here.

`path_encoding`

Encode paths with this encoding. The default is *ascii*.

open(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
Open a file for reading or writing.

Implements the semantics of the *open* function in Python's io module, which is the default implementation in Python 3. Opening a file in text mode will return a file-like object which reads or writes unicode strings, while opening a file in binary mode will return a file-like object which reads or writes raw bytes.

Because the underlying implementation uses a pipe to a *Git* plumbing command, opening for update (read and write) is not supported, nor is seeking.

cwd()
Returns the path to the current working directory in the repository.

chdir(*path*)
Change the current working directory in repository.

cd(*path*)
A context manager that changes the current working directory only in the scope of the 'with' context. Eg:

```
import acidfs

fs = acidfs.AcidFS('myrepo')
with fs.cd('some/folder'):
    fs.open('a/file')    # relative to /some/folder
fs.open('another/file') # relative to /
```

listdir(*path=''*)
Return list of files in indicated directory. If *path* is omitted, the current working directory is used.

mkdir(*path*)
Create a new directory. The parent of the new directory must already exist.

mkdirs(*path*)
Create a new directory, including any ancestors which need to be created in order to create the directory with the given *path*.

rm(*path*)
Remove a single file.

rmdir(*path*)
Remove a single directory. The directory must be empty.

rmtree(*path*)
Remove a directory and any of its contents.

mv(*src*, *dst*)
Move a file or directory from *src* path to *dst* path.

exists(*path*)
Returns boolean indicating whether a file or directory exists at the given *path*.

isdir(*path*)
Returns boolean indicating whether the given *path* is a directory.

empty(*path*)
Returns boolean indicating whether the directory indicated by *path* is empty.

get_base()
Returns the id of the commit that is the current base for the transaction.

set_base(*commit*)
Sets the base commit for the current transaction. The *commit* argument may be the SHA1 of a commit or the name of a reference (eg. branch or tag). The current transaction must be clean. If any changes have been made in the transaction, a ConflictError will be raised.

hash(*path=''*)
Returns the sha1 hash of the object referred to by *path*. If *path* is omitted the current working directory is used.

Python Module Index

a

Index