snakebite Documentation

Release 2.11.0

Wouter de Bie

August 08, 2016

Contents

Snakebite is a python package that provides:

Client library

class `snakebite.client.`**`Client`**(*host*, *port=8020*, *hadoop_version=9*, *use_trash=False*, *effective_user=None*, *use_sasl=False*, *hdfs_namenode_principal=None*, *sock_connect_timeout=10000*, *sock_request_timeout=10000*, *use_datanode_hostname=False*)

A pure python HDFS client.

Example:

```
>>> from snakebite.client import Client
>>> client = Client("localhost", 8020, use_trash=False)
>>> for x in client.ls(['/']):
...     print x
```

Warning: Many methods return generators, which mean they need to be consumed to execute! Documentation will explicitly specify which methods return generators.

Note: `paths` parameters in methods are often passed as lists, since operations can work on multiple paths.

Note: Parameters like `include_children` and `recurse` are not used when paths contain globs.

Note: Different Hadoop distributions use different protocol versions. Snakebite defaults to 9, but this can be set by passing in the `hadoop_version` parameter to the constructor.

Parameters

- **`host`** (*`string`*) – Hostname or IP address of the NameNode
- **`port`** (*`int`*) – RPC Port of the NameNode
- **`hadoop_version`** (*`int`*) – What hadoop protocol version should be used (default: 9)
- **`use_trash`** (*`boolean`*) – Use a trash when removing files.
- **`effective_user`** (*`string`*) – Effective user for the HDFS operations (default: None - current user)
- **`use_sasl`** (*`boolean`*) – Use SASL authentication or not
- **`hdfs_namenode_principal`** (*`string`*) – Kerberos principal to use for HDFS

- **`sock_connect_timeout`** (*`int`*) – Socket connection timeout in seconds
- **`sock_request_timeout`** (*`int`*) – Request timeout in seconds
- **`use_datanode_hostname`** (*`boolean`*) – Use hostname instead of IP address to commuicate with datanodes

`cat` (*paths*, *check_crc=False*)
Fetch all files that match the source file pattern and display their content on stdout.

Parameters

- **`paths`** (*`list of strings`*) – Paths to display
- **`check_crc`** (*`boolean`*) – Check for checksum errors

Returns a generator that yields strings

`chgrp` (*paths*, *group*, *recurse=False*)
Change the group of paths.

Parameters

- **`paths`** (*`list`*) – List of paths to chgrp
- **`group`** – New group
- **`recurse`** (*`boolean`*) – Recursive chgrp

Returns a generator that yields dictionaries

`chmod` (*paths*, *mode*, *recurse=False*)
Change the mode for paths. This returns a list of maps containing the resut of the operation.

Parameters

- **`paths`** (*`list`*) – List of paths to chmod
- **`mode`** (*`int`*) – Octal mode (e.g. 0o755)
- **`recurse`** (*`boolean`*) – Recursive chmod

Returns a generator that yields dictionaries

Note: The top level directory is always included when *recurse=True*

`chown` (*paths*, *owner*, *recurse=False*)
Change the owner for paths. The owner can be specified as *user* or *user:group*

Parameters

- **`paths`** (*`list`*) – List of paths to chmod
- **`owner`** (*`string`*) – New owner
- **`recurse`** (*`boolean`*) – Recursive chown

Returns a generator that yields dictionaries

This always include the toplevel when recursing.

`copyToLocal` (*paths*, *dst*, *check_crc=False*)
Copy files that match the file source pattern to the local name. Source is kept. When copying multiple, files, the destination must be a directory.

Parameters

- **paths** (*list of strings*) – Paths to copy
- **dst** (*string*) – Destination path
- **check_crc** (*boolean*) – Check for checksum errors

Returns a generator that yields strings

count (*paths*)

Count files in a path

Parameters **paths** (*list*) – List of paths to count

Returns a generator that yields dictionaries

Examples:

```
>>> list(client.count(['/']))
[{'spaceConsumed': 260185L, 'quota': 2147483647L, 'spaceQuota': 18446744073709551615L, 'leng
```

delete (*paths*, *recurse=False*)

Delete paths

Parameters

- **paths** (*list*) – Paths to delete
- **recurse** (*boolean*) – Recursive delete (use with care!)

Returns a generator that yields dictionaries

Note: Recursive deletion uses the NameNode recursive deletion functionality instead of letting the client recurse. Hadoops client recurses by itself and thus showing all files and directories that are deleted. Snakebite doesn't.

df ()

Get FS information

Returns a dictionary

Examples:

```
>>> client.df()
{'used': 491520L, 'capacity': 120137519104L, 'under_replicated': 0L, 'missing_blocks': 0L, '
```

du (*paths*, *include_toplevel=False*, *include_children=True*)

Returns size information for paths

Parameters

- **paths** (*list*) – Paths to du
- **include_toplevel** (*boolean*) – Include the given path in the result. If the path is a file, include_toplevel is always True.
- **include_children** (*boolean*) – Include child nodes in the result.

Returns a generator that yields dictionaries

Examples:

Children:

```
>>> list(client.du(['/']))
[{'path': '/Makefile', 'length': 6783L}, {'path': '/build', 'length': 244778L}, {'path': '/i
```

Directory only:

```
>>> list(client.du(['/'], include_toplevel=True, include_children=False))
[{'path': '/', 'length': 260185L}]
```

getmerge (*path*, *dst*, *newline=False*, *check_crc=False*)

Get all the files in the directories that match the source file pattern and merge and sort them to only one file on local fs.

Parameters

- **paths** (*string*) – Directory containing files that will be merged
- **dst** (*string*) – Path of file that will be written
- **nl** (*boolean*) – Add a newline character at the end of each file.

Returns string content of the merged file at dst

ls (*paths*, *recurse=False*, *include_toplevel=False*, *include_children=True*)

Issues 'ls' command and returns a list of maps that contain fileinfo

Parameters

- **paths** (*list*) – Paths to list
- **recurse** (*boolean*) – Recursive listing
- **include_toplevel** (*boolean*) – Include the given path in the listing. If the path is a file, include_toplevel is always True.
- **include_children** (*boolean*) – Include child nodes in the listing.

Returns a generator that yields dictionaries

Examples:

Directory listing

```
>>> list(client.ls(["/"]))
[{'group': u'supergroup', 'permission': 420, 'file_type': 'f', 'access_time': 1367317324982I
```

File listing

```
>>> list(client.ls(["/Makefile"]))
[{'group': u'supergroup', 'permission': 420, 'file_type': 'f', 'access_time': 1367317324982I
```

Get directory information

```
>>> list(client.ls(["/source"], include_toplevel=True, include_children=False))
[{'group': u'supergroup', 'permission': 493, 'file_type': 'd', 'access_time': 0L, 'block_rep
```

mkdir (*paths*, *create_parent=False*, *mode=493*)

Create a directoryCount

Parameters

- **paths** (*list of strings*) – Paths to create
- **create_parent** (*boolean*) – Also create the parent directories
- **mode** (*int*) – Mode the directory should be created with

Returns a generator that yields dictionaries

rename (*paths*, *dst*)
Rename (move) path(s) to a destination

Parameters

- **paths** (*list*) – Source paths
- **dst** (*string*) – destination

Returns a generator that yields dictionaries

rename2 (*path*, *dst*, *overwriteDest=False*)
Rename (but don't move) path to a destination

By only renaming, we mean that you can't move a file or folder out or in other folder. The renaming can only happen within the folder the file or folder lies in.

Note that this operation "always succeeds" unless an exception is raised, hence, the dict returned from this function doesn't have the 'result' key.

Since you can't move with this operation, and only rename, it would not make sense to pass multiple paths to rename to a single destination. This method uses the underlying rename2 method.

https://github.com/apache/hadoop/blob/ae91b13/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/protocol/ClientProtocol.java#L483-L523

Out of all the different exceptions mentioned in the link above, this method only wraps the FileAlreadyExistsException exception. You will also get a FileAlreadyExistsException if you have overwriteDest=True and the destination folder is not empty. The other exceptions will just be passed along.

Parameters

- **path** (*string*) – Source path
- **dst** (*string*) – destination

Returns A dictionary or None

rmdir (*paths*)
Delete a directory

Parameters **paths** (*list*) – Paths to delete

Returns a generator that yields dictionaries

serverdefaults (*force_reload=False*)
Get server defaults, caching the results. If there are no results saved, or the force_reload flag is True, it will query the HDFS server for its default parameter values. Otherwise, it will simply return the results it has already queried.

Note: This function returns a copy of the results loaded from the server, so you can manipulate or change them as you'd like. If for any reason you need to change the results the client saves, you must access the property client._server_defaults directly.

Parameters **force_reload** (*bool*) – Should the server defaults be reloaded even if they already exist?

Returns dictionary with the following keys: blockSize, bytesPerChecksum, writePacketSize, replication, fileBufferSize, encryptDataTransfer, trashInterval, checksumType

Example:

```
>>> client.serverdefaults()
[{'writePacketSize': 65536, 'fileBufferSize': 4096, 'replication': 1, 'bytesPerChecksum': 51
```

setrep (*paths*, *replication*, *recurse=False*)
Set the replication factor for paths

Parameters

- **paths** (*list*) – Paths
- **replication** – Replication factor
- **recurse** (*boolean*) – Apply replication factor recursive

Returns a generator that yields dictionaries

stat (*paths*)
Stat a fileCount

Parameters **paths** (*string*) – Path

Returns a dictionary

Example:

```
>>> client.stat(['/index.asciidoc'])
{'blocksize': 134217728L, 'owner': u'wouter', 'length': 100L, 'access_time': 1367317326510L,
```

tail (*path*, *tail_length=1024*, *append=False*)
Show the end of the file - default 1KB, supports up to the Hadoop block size.

Parameters

- **path** (*string*) – Path to read
- **tail_length** (*int*) – The length to read from the end of the file - default 1KB, up to block size.
- **append** (*bool*) – Currently not implemented

Returns a generator that yields strings

test (*path*, *exists=False*, *directory=False*, *zero_length=False*)
Test if a path exist, is a directory or has zero length

Parameters

- **path** (*string*) – Path to test
- **exists** (*boolean*) – Check if the path exists
- **directory** (*boolean*) – Check if the path is a directory
- **zero_length** (*boolean*) – Check if the path is zero-length

Returns a boolean

Note: directory and zero length are AND'd.

text (*paths*, *check_crc=False*)
Takes a source file and outputs the file in text format. The allowed formats are gzip and bzip2

Parameters

- **paths** (*list of strings*) – Paths to display

- **check_crc** (*boolean*) – Check for checksum errors

Returns a generator that yields strings

touchz (*paths*, *replication=None*, *blocksize=None*)
Create a zero length file or updates the timestamp on a zero length file

Parameters

- **paths** (*list*) – Paths
- **replication** – Replication factor
- **blocksize** (*int*) – Block size (in bytes) of the newly created file

Returns a generator that yields dictionaries

class snakebite.client.**AutoConfigClient** (*hadoop_version=9*, *effective_user=None*, *use_sasl=False*)
A pure python HDFS client that support HA and is auto configured through the `HADOOP_HOME` environment variable.

HAClient is fully backwards compatible with the vanilla Client and can be used for a non HA cluster as well. This client tries to read `${HADOOP_HOME}/conf/hdfs-site.xml` and `${HADOOP_HOME}/conf/core-site.xml` to get the address of the namenode.

The behaviour is the same as Client.

Example:

```
>>> from snakebite.client import AutoConfigClient
>>> client = AutoConfigClient()
>>> for x in client.ls(['/']):
...     print x
```

Note: Different Hadoop distributions use different protocol versions. Snakebite defaults to 9, but this can be set by passing in the `hadoop_version` parameter to the constructor.

Parameters

- **hadoop_version** (*int*) – What hadoop protocol version should be used (default: 9)
- **effective_user** (*string*) – Effective user for the HDFS operations (default: None - current user)
- **use_sasl** (*boolean*) – Use SASL for authenication or not

class snakebite.client.**HAClient** (*namenodes*, *use_trash=False*, *effective_user=None*, *use_sasl=False*, *hdfs_namenode_principal=None*, *max_failovers=15*, *max_retries=10*, *base_sleep=500*, *max_sleep=15000*, *sock_connect_timeout=10000*, *sock_request_timeout=10000*, *use_datanode_hostname=False*)
Snakebite client with support for High Availability

HAClient is fully backwards compatible with the vanilla Client and can be used for a non HA cluster as well.

Example:

```
>>> from snakebite.client import HAClient
>>> from snakebite.namenode import Namenode
>>> n1 = Namenode("namenode1.mydomain", 8020)
>>> n2 = Namenode("namenode2.mydomain", 8020)
>>> client = HAClient([n1, n2], use_trash=True)
>>> for x in client.ls(['/']):
...     print x
```

Note: Different Hadoop distributions use different protocol versions. Snakebite defaults to 9, but this can be set by passing in the `version` parameter to the Namenode class constructor.

Parameters

- **`namenodes`** (*`list`*) – Set of namenodes for HA setup
- **`use_trash`** (*`boolean`*) – Use a trash when removing files.
- **`effective_user`** (*`string`*) – Effective user for the HDFS operations (default: None - current user)
- **`use_sasl`** (*`boolean`*) – Use SASL authentication or not
- **`hdfs_namenode_principal`** (*`string`*) – Kerberos principal to use for HDFS
- **`max_retries`** (*`int`*) – Number of failovers in case of connection issues
- **`max_retries`** – Max number of retries for failures
- **`base_sleep`** (*`int`*) – Base sleep time for retries in milliseconds
- **`max_sleep`** (*`int`*) – Max sleep time for retries in milliseconds
- **`sock_connect_timeout`** (*`int`*) – Socket connection timeout in seconds
- **`sock_request_timeout`** (*`int`*) – Request timeout in seconds
- **`use_datanode_hostname`** (*`boolean`*) – Use hostname instead of IP address to commuicate with datanodes

CHAPTER 2

CLI client

A command line interface for HDFS using `snakebite.client`.

2.1 Config

Snakebite CLI can accept configuration in a couple of different ways, but there's strict priority for each of them. List of methods, in priority order:

1. via path in command line - eg: `hdfs://namenode_host:port/path`
2. via `-n`, `-p`, `-V` flags in command line
3. via `~/.snakebiterc` file
4. via `/etc/snakebiterc` file
5. via `$HADOOP_HOME/core-site.xml` and/or `$HADOOP_HOME/hdfs-site.xml` files
6. via `core-site.xml` and/or `hdfs-site.xml` in default locations

More about methods from 3 to 6 below.

2.1.1 Config files

Snakebite config can exist in `~/.snakebiterc` - per system user, or in `/etc/snakebiterc` - system wide config.

A config looks like:

```
{
    "config_version": 2,
    "skiptrash": true,
    "namenodes": [
        {"host": "mynamenode1", "port": 8020, "version": 9},
        {"host": "mynamenode2", "port": 8020, "version": 9}
    ]
}
```

The version property denotes the protocol version used. CDH 4.1.3 uses protocol 7, while HDP 2.0 uses protocol 9. Snakebite defaults to 9. Default port of namenode is 8020. Default value of `skiptrash` is `true`.

2.1.2 Hadoop config files

Last two methods of providing config for snakebite is through hadoop config files. If `HADOOP_HOME` environment variable is set, snakebite will try to find `core-site.xml` and/or `hdfs-site.xml` files in `$HADOOP_HOME` directory. If `HADOOP_HOME` is not set, snakebite will try to find those files in a couple of default hadoop config locations:

- /etc/hadoop/conf/core-site.xml
- /usr/local/etc/hadoop/conf/core-site.xml
- /usr/local/hadoop/conf/core-site.xml
- /etc/hadoop/conf/hdfs-site.xml
- /usr/local/etc/hadoop/conf/hdfs-site.xml
- /usr/local/hadoop/conf/hdfs-site.xml

2.2 Bash completion

Snakebite CLI comes with bash completion file in /scripts. If snakebite is installed via debian package it will install completion file automatically. But if snakebite is installed via pip/setup.py it will not do that, as it would requite write access in /etc (usually root), in that case it's required to install completion script manually.

2.3 Usage

```
snakebite [general options] cmd [arguments]
general options:
  -D --debug                     Show debug information
  -V --version                   Hadoop protocol version (default:9)
  -h --help                      show help
  -j --json                      JSON output
  -n --namenode                  namenode host
  -p --port                      namenode RPC port (default: 8020)
  -v --ver                       Display snakebite version

commands:
  cat [paths]                    copy source paths to stdout
  chgrp <grp> [paths]            change group
  chmod <mode> [paths]           change file mode (octal)
  chown <owner:grp> [paths]      change owner
  copyToLocal [paths] dst        copy paths to local file system destination
  count [paths]                  display stats for paths
  df                             display fs stats
  du [paths]                     display disk usage statistics
  get file dst                   copy files to local file system destination
  getmerge dir dst               concatenates files in source dir into destination local file
  ls [paths]                     list a path
  mkdir [paths]                  create directories
  mkdirp [paths]                 create directories and their parents
  mv [paths] dst                 move paths to destination
  rm [paths]                     remove paths
  rmdir [dirs]                   delete a directory
  serverdefaults                 show server information
  setrep <rep> [paths]           set replication factor
  stat [paths]                    stat information
  tail path                       display last kilobyte of the file to stdout
  test path                       test a path
  text path [paths]               output file in text format
  touchz [paths]                  creates a file of zero length
  usage <cmd>                     show cmd usage

to see command-specific options use: snakebite [cmd] --help
```

CHAPTER 3

Development

3.1 How to start

We try to make it as easy as possible to start development on snakebite. We recommend to use virtualenv (+ virtualenvwrapper) for development purposes, it's not required to but highly recommended. To install, and create development environment for snakebite:

1. install virtualenvwrapper: `$ pip install virtualenvwrapper` 2. create development environment: `$ mkvirtualenv snakebite_dev`

More about virtualenvwrapper and virtualenv here

Below is the list of recommended steps to start development:

1. clone repo: `$ git clone git@github.com:spotify/snakebite.git` 2. fetch all developer requirements: `$ pip install -r requirements-dev.txt` 3. run tests: `$ python setup.py test`

If tests succeeded you are ready to hack! Remember to always test your changes and please come back with a PR <3

3.2 Open issues

If you're looking for open issues please take a look here.

Thanks!

CHAPTER 4

Testing

Snakebite provides integration and unit tests for its functionalities. To be able to truly test integration with HDFS, we provide wrapper around `snakebite.minicluster`, and base class for integration tests `MiniClusterTestBase` - on setup for such test class minicluster is started, when tests are done minicluster is destroyed. There's some performance overhead - but it's not a problem (yet).

Snakebite by default uses nose and tox for testing. Tests are integrated with *setup.py*, so to start tests one can simply: `$ python setup.py test`

Because we require minicluster to fully test snakebite, java needs to be present on the system.

Note: It's possible to run snakebite tests inside snakebite Docker test image - to learn more see section Fig below. Note that it's not default testing method as it requires Docker to be present.

4.1 Tox

Tox allow us to create automated isolated python test environments. It's also a place where we can prepare environment for testing - like download hadoop distributions, set environment variables etc. Tox configuration is available in `tox.ini` file in root directory.

There are 4 test environments:

- python 2.6 + CDH
- python 2.7 + CDH
- python 2.6 + HDP
- python 2.7 + HDP

We bootstrap environment with `pip install -r requirements-dev.txt` (deps section) And then we setup environment via `/scripts/ci/setup_env.sh` script. `setup_env.sh` script downloads hadoop distribution tar, and extracts it. Help for `setup_env.sh`:: Setup environment for snakebite tests

options:

-h, --help	show brief help
-o, --only-download	just download hadoop tar(s)
-e, --only-extract	just extract hadoop tar(s)
-d, --distro	select distro (hdp|cdh)b

When environment is ready we actually run tests via: `/scripts/ci/run_tests.sh`

One can run tests manually via `/scripts/ci/run_tests.sh` but make sure that `HADOOP_HOME` environment variable exists so that it knows where to find minicluster jar file. This way it's possible to test snakebite against custom Hadoop distributions. `run_tests.sh` script uses `nose` for testing, so that if you wish to pass anything to nose, just add parameters to `run_tests.sh`.

One can pass parameters to tox/nose through setup.py via `--tox-args` flag:

```
$ python setup.py test --tox-args="--recreate -e py26-hdp '--quiet'"
```

Will test py26-hdp tox environment, make sure it will be recreated, and also through `run_tests.sh` script instruct nose to be quite.

```
$ python setup.py test --tox-args="-e py26-hdp test/test_test.py
```

Will use py26-hdp tox environment and also instruct nose to run only tests from test/test_test.py.

4.2 Fig

Note: Fig is experimental testing method, it's very promising though.

Fig is "fast, isolated development environments using Docker". It abstracts away whole test environment, create completely fresh and isolated test environments using Docker.

Currently we use base testing image `ravwojdyla/snakebite_test:base`, it was created using `/scripts/build-base-test-docker.sh` and `/scripts/Dockerfile`. Base test image is a Ubuntu Trusty with: * oracle java 7 * python 2.6 * python 2.7 * pip * CDH distribution * HDP distribution

Base docker image doesn't change, to create new test image with current working tree, based on `ravwojdyla/snakebite_test:base`:

```
$ fig build
```

Fig will create new image based on `ravwojdyla/snakebite_test:base`, with current working tree, that can be used for tests. Fig currently specifies 4 tests: * `testPy26cdh`: python 2.6 + CDH * `testPy26hdp`: python 2.6 + HDP * `testPy27cdh`: python 2.7 + CDH * `testPy27hdp`: python 2.7 + HDP

To run specific test (eg. testPy26cdh):

```
$ fig run testPy26cdh
```

The biggest value in Fig is that tests are completely isolated, all the snakebite dependencies are present on test image. Unfortunately Fig depends on Docker - which is quite a big dependency to have, and that's why it's default method of testing for snakebite. It's worth to mention that Fig still uses Tox inside test container.

CHAPTER 5

Minicluster

class `snakebite.minicluster.`**`MiniCluster`**(*testfiles_path*, *start_cluster=True*, *nnport=None*)

Class that spawns a hadoop mini cluster and wrap hadoop functionality

This class requires the `HADOOP_HOME` environment variable to be set to run the `hadoop` command. It will search `HADOOP_HOME` for `hadoop-mapreduce-client-jobclient<version>-tests.jar`, but the location of this jar can also be supplied by the `HADOOP_JOBCLIENT_JAR` environment variable.

Since the current minicluster interface doesn't provide for specifying the namenode post number, and chooses a random one, this class parses the output from the minicluster to find the port numer.

All supplied methods (like *`put()`*, *`ls()`*, etc) use the hadoop command to perform operations, and not the snakebite client, since this is used for testing snakebite itself.

All methods return a list of maps that are snakebite compatible.

Example without `snakebite.client`

```
>>> from snakebite.minicluster import MiniCluster
>>> cluster = MiniCluster("/path/to/test/files")
>>> ls_output = cluster.ls(["/"])
```

Example with `snakebite.client`

```
>>> from snakebite.minicluster import MiniCluster
>>> from snakebite.client import Client
>>> cluster = MiniCluster("/path/to/test/files")
>>> client = Client('localhost', cluster.port)
>>> ls_output = client.ls(["/"])
```

Just as the snakebite client, the cluster methods take a list of strings as paths. Wherever a method takes `extra_args`, normal hadoop command arguments can be given (like -r, -f, etc).

More info can be found at http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/CLIMiniCluster.html

Note: A minicluster will be started at instantiation

Note: Not all hadoop commands have been implemented, only the ones that were necessary for testing the snakebite client, but please feel free to add them

Parameters

- **`testfiles_path`** (*`string`*) – Local path where test files can be found. Mainly used for `put()`
- **`start_cluster`** (*`boolean`*) – start a MiniCluster on initialization. If False, this class will act as an interface to the `hadoop fs` command

`count`(*src*)
Perform `count` on a path

`df`(*src*)
Perform `df` on a path

`du`(*src*, *extra_args=[]*)
Perform `du` on a path

`exists`(*path*)
Return True if <src> exists, False if doesn't

`is_directory`(*path*)
Return True if <path> is a directory, False if it's NOT a directory

`is_files`(*path*)
Return True if <path> is a file, False if it's NOT a file

`is_greater_then_zero_bytes`(*path*)
Return True if file <path> is greater than zero bytes in size, False otherwise

`is_zero_bytes_file`(*path*)
Return True if file <path> is zero bytes in size, else return False

`ls`(*src*, *extra_args=[]*)
List files in a directory

`mkdir`(*src*, *extra_args=[]*)
Create a directory

`put`(*src*, *dst*)
Upload a file to HDFS

This will take a file from the `testfiles_path` supplied in the constuctor.

`terminate`()
Terminate the cluster

Since the minicluster is started as a subprocess, this method has to be called explicitely when your program ends.

CHAPTER 6

Hadoop RPC protocol description

Snakebite currently implements the following protocol in `snakebite.channel.SocketRpcChannel` to communicate with the NameNode.

6.1 Connection

The Hadoop RPC protocol works as described below. On connection, headers are sent to setup a session. After that, multiple requests can be sent within the session.

Function	Type	Default
Header	`bytes`	"hrpc"
Version	`uint8`	7
Auth method	`uint8`	80 (Auth method `SIMPLE`)
Serialization type	`uint8`	0 (`protobuf`)
IpcConnectionContextProto length	`uint32`	
IpcConnectionContextProto	`bytes`	

6.2 Sending messages

When sending a message, the following is sent to the sever:

Function	Type
Length of the next two parts	`uint32`
RpcPayloadHeaderProto length	`varint`
RpcPayloadHeaderProto	`protobuf serialized message`
HadoopRpcRequestProto length	`varint`
HadoopRpcRequestProto	`protobuf serialized message`

`varint` is a Protocol Buffer variable int.

Note: The Java protobuf implementation uses `writeToDelimited` to prepend the message with their lenght, but the python implementation doesn't implement such a method (yet).

Next to an `rpcKind` (snakebites default is `RPC_PROTOCOL_BUFFER`), an `rpcOp` (snakebites default is `RPC_FINAL_PAYLOAD`), the `RpcPayloadHeaderProto` message defines a `callId` that is added in the RPC response (described below).

The `HadoopRpcRequestProto` contains a `methodName` field that defines what server method is called and a has a property `request` that contains the serialized actual request message.

6.3 Receiving messages

After a message is sent, the response can be read in the following way:

Function	Type
Length of the RpcResponseHeaderProto	`varint`
RpcResponseHeaderProto	`bytes`
Length of the RPC response	`uint32`
Serialized RPC response	`bytes`

The `RpcResponseHeaderProto` contains the `callId` of the request and a status field. The status can be `SUCCESS`, `ERROR` or `FAILURE`. In case `SUCCESS` the rest of response is a complete protobuf response.

In case of `ERROR`, the response looks like follows:

Function	Type
Length of the RpcResponseHeaderProto	`varint`
RpcResponseHeaderProto	`bytes`
Length of the RPC response	`uint32`
Length of the Exeption class name	`uint32`
Exception class name	`utf-8 string`
Length of the stack trace	`uint32`
Stack trace	`utf-8 string`

- A pure python HDFS client library that uses protobuf messages over Hadoop RPC to communicate with HDFS.
- A command line interface (CLI) for HDFS that uses the pure python client library.
- A hadoop minicluster wrapper.
- Hadoop RPC specification.

CHAPTER 7

Background

Since the ‘normal’ Hadoop HDFS client (`hadoop fs`) is written in Java and has a lot of dependencies on Hadoop jars, startup times are quite high (> 3 secs). This isn’t ideal for integrating Hadoop commands in python projects.

At Spotify we use the luigi job scheduler that relies on doing a lot of existence checks and moving data around in HDFS. And since calling `hadoop` from python is expensive, we decided to write a pure python HDFS client that only relies on protobuf. The current `snakebite.client` library uses protobuf messages and implements the Hadoop RPC protocol for talking to the NameNode.

During development, we needed to verify `snakebite.client` behavior against the real client and for that we implemented a `minicluster` that wraps a Hadoop Java mini cluster. Obviously this `minicluster` can be used in different projects, so we made it a part of snakebite.

And since it’s nice to have a CLI that uses `snakebite.client` we’ve implemented a CLI client as well.

Warning: all methods that read data from a data node are able to check the CRC during transfer, but this is disabled by default because of performance reasons. This is the opposite behaviour from the stock Hadoop client.

LICENSE

Code in `channel`, `logger` and `service` was borrowed from https://code.google.com/p/protobuf-socket-rpc/ and carries it’s respective license.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index