Apache Hadoop Filesystem Contract

This is a list of expected behaviors of a Hadoop-compatible filesystem, including many that aren't in the `FileSystemContractBaseTest`.

Most of the operations are tested for HDFS in the Hadoop test suites, because MiniDFSCluster is used so ubiquitously. HDFS's actions have been modeled closely on the Posix Filesystem behavior -using the actions and return codes of Unix filesystem actions as a reference.

What is not so rigorously tested is how well other filesystems accessible from Hadoop behave.

- the bundled S3 filesystem makes Amazon's S3 blobstore accessible through the FileSystem API.
- The Swift filesystem driver provides similar functionality for the OpenStack Swift blobstore.
- The Azure object storage filesystem in branch-1-win talks to Microsoft's Azure equivalent.

All of these bind to blobstores, which do have different behaviours, especially regarding consistency guarantees, and atomicity of operations.

The Local filesystem provides access to the underlying filesystem of the platform -it's behaviour is defined by the operating system -and again, can behave differently from HDFS.

Finally, there are filesystems implemented by third parties, that assert compatibility with Apache Hadoop. There is no formal compatibility suite, and no way for anyone to certify compatibility.

This document does not attempt to formally define compatibility; passing the associated test suites does not guarantee correct behavior in MapReduce jobs. What the test suites do define is the expected set of actions -failing these tests will highlight potential issues.

Naming: This document follows RFC2119 rules regarding the use of MUST, MUST NOT, MAY, and SHALL -and MUST NOT be treated as normative. It also uses the term "iff" as shorthand for "if and only if".

Security: Except in the special section on security, this document assumes the client has full access to the filesystem

Core Assumptions and Requirements

Implicit Assumptions

"so obvious nobody bothered testing for them"

- It's a hierarchical directory structure with files and directories.
- Files contain data
- You cannot put files or directories under a file
- Directories contain 0 or more files
- A directory entry has no data itself
- You can write arbitrary binary data to a file -and when that file's contents are read in, from anywhere in or out the cluster -that data is returned.
- You can store many GB of data in a single file.
- The names of paths are at least upper and lower case ASCII, excluding "/" and possibly the space character ' '. (Windows doesn't allow a colon, as shown by HDFS-4470 (Azure blob store recommends not using a trailing "." as the .NET URI class strips that)
- `FileSystem.rename()` fails when attempting to rename a directory into a child (or n+1 grandchild) of itself
- `FileSystem.exists()` always returns true for the root directory.
- `FileStatus.isFile()` is true for a file of size 0
- `FileStatus.isFile()` is true for a file of size >0
- The root directory, "/", always exists, and cannot be renamed. It is always a directory, and cannot be overwritten by a file write operation. An attempt to recursively delete the root directory will delete it's contents (assuming permissions allow this), but will retain the root path itself.
- Security: If you don't have the permissions for an operation, it will fail with some kind of error.

Concurrency

- Rename is atomic/an ACID operation. *Blobstore services break this.*
- Delete of file, empty directory and recursive directory is atomic. *Blobstore services break this.*
- mkdir/mkdirs is atomic.
- If append is implemented, only one client can append. (RawLocalFS may break this, depending on locking in the native Filesystem).
- Only one writer can write to a file (does anything in MR/HBase use this for locks?)
- `FileSystem.listStatus()` does not appear to contain any claims of atomicity, though some uses in the MapReduce codebase (such as `FileOutputCommitter`) do assume that the listed directories do not get deleted between listing their status and recursive actions on the listed entries.

- A FileSystem instance MAY be cached and MAY be passed to other threads in the same process.

Consistency

The consistency model of a Hadoop filesystem is one-copy-update-semantics; that of a local Posix filesystem.

- **Create:** once the close() operation on an output stream writing a newly created file has completed, in-cluster operations querying the file metadata and contents MUST immediately see the file and its data.
- **Update:** Once the close() operation on an output stream writing a newly created file has completed, in-cluster operations querying the file metadata and contents MUST immediately see the new data.
- **Delete:** once a delete() operation is on a file has completed, stat(), open(), rename() and append() operations MUST fail.
- When file is deleted then overwritten, stat(), open(), rename() and append() operations MUST succeed: the file is visible.
- **Rename:** after a rename has completed, operations against the new path MUST succeed; operations against the old path MUST fail.
- The consistency semantics out of cluster MUST be the same as that in-cluster.
- All clients reading a file MUST see the same metadata and data until it is changed from a write(), append(), rename() or delete() operation.

Concurrency

- The data added to a file during a write or append MAY be visible during while the write operation is in progress.
- If a client opens a file for a read operation while another read() operation is in progress, the second operation MUST succeed. Both clients MUST have a consistent view of the same data.
- If a file is deleted while a read() operation is in progress, the delete() operation SHOULD complete successfully. Implementations MAY cause delete() to fail with an IOException instead.
- If a file is deleted while a read() operation is in progress, the read() operation SHOULD complete successfully. Implementations MAY cause read() operations to fail with an IOException instead.
- If an open for writing operation is attempted while a write operation is in progress, the open operation SHOULD fail. Implementations MAY succeed with the contents being exactly one of the sets of written data.
- *Undefined: action of delete() during write()*

Undefined limits

- Max # of files in a directory
- Max # of directories in a directory
- Max number of entries in a filesystem
- Max length of a filename (HDFS: 8000)
- MAX_PATH - total length of the entire directory tree referencing a file. Blobstores tend to stop at ~1024 characters
- max depth of a path (HDFS: 1000)
- Maximum filesize
- maximum completion time of blocking FS operations. MAPREDUCE-972 shows how distcp broke on slow s3 renames.
- timeout for idle HDFS read streams
- timeout for idle HDFS

Method-specific requirements

Reading

- An attempt to read a nonexistent file on a path MUST raise a FileNotFoundException.
- read() operations at the end of the file MUST return -1
- readFully() calls that attempt to read past the end of a file raises an IOException or subclass thereof. This SHOULD be an EOFException.
- A seek(offset) followed by a read operation MUST return the data saved into the file starting at byte[offset]

Rename

- Rename is atomic
- the parent directories of a the destination file/directory MUST exist for the rename to succeed
- rename(self, self) succeeds for a file but is a no-op: the file and its attributes are not updated.
- rename(self, self) fails for a directory.
- rename(path, path2) MUST fail if !exists(path)
- rename(path, path2) MUST fail if is-subdirectory-of(path, path2)
- rename(path, path2) MUST fail if path2 is a file
- rename("/",anything) MUST always fails. This implicitly be covered by the child directory rename rule.

Posix allows rename of an open file -is this something Hadoop relies on? Windows localfs *probably fails on this already.*

Delete

- Deleting a file is an atomic action.
- Deleting an empty directory is atomic.
- Deleting a directory tree is atomic.
- Deleting an empty directory MUST succeed, irrespective of the value of the recursive flag.
- Deleting a directory with child elements MUST succeed iff recursive==true.
- Deleting the root path, /, MUST, iff recursive==true, delete all entries in the filesystem -but leave "/' present.
- Deleting the root path, /, MUST fail iff recursive==false.
- After a delete operation completes successfully, attempts by clients to open the file, query its attributes, or locate it by enumerating the parent directory will fail. (i.e changes are immediately visible and consistent across all clients)

Caveats

- *Posix permits deletion of files that are open. This does not hold in Windows; a fact that has broken some tests. It's not clear whether real-world Hadoop applications depend upon this property.*
- *Testing the rm / command is going to be dangerous.*

LS operations

- `FileSystem.listStatus("/", filter)` MUST always succeed (networking/auth issues excluded) and MUST return the filtered child directory entries
- `FileSystem.listStatus("/existing-directory", filter)` succeeds and returns 0 or more children in the directory. Entries that match the filter are excluded.
- `FileSystem.listStatus("/existing-file", filter)` succeeds and returns the status of the file as the single element of the array, an element where `FileStatus.isDirectory()` returns false.
  Exception: if the file matches any filter, an empty array is returned.
- `FileSystem.listStatus("/non-existing-file", filter)` throws a `FileNotFoundException`
- After a file is created, all ls operations on the file and parent directory MUST find the file.
- After a file is created, all ls operations on the file and parent directory MUST not find the file.
- By the time the `listStatus()` operation returns to the caller, there is no guarantee that the information contained in the response is current. The details MAY be out of date -including the contents of any directory, the attributes of any files, and the existence of the path supplied.
- Security: if a caller has the rights to list a directory, it has the rights to list directories all the way up the tree.

getFileBlockLocations(filestatus status, int start, int len)

These conditions must be met in order

1. MUST return null if (file==null)
2. MUST throw an InvalidArgumentException if (src<0 || len<0)
3. MUST throw an IOException if status.isDirectory() is true
4. If the filesystem is not location aware SHOULD return "localhost","localhost:????" as the response.

create(Path)

1. MUST create parent directories according to the mkdir rules iff the path's parent directories are missing.

OutputStream.flush()

- client-side flush() SHOULD forward the data to the DFS.
- the DFS MAY flush that data to disk

OutputStream.close()

- Durability: Once a close() operation has successfully completed, the data MUST be persisted according to the durability guarantees of the filesystem.

Network Failures

- Any operation with a filesystem MAY signal an error by throwing an IOException or subclass thereof.
- The specific cause of the failure MAY be explicitly defined in the IOException subclass type
- The cause or details of the failure SHOULD be described in the exception's toString() value.
- Filesystem operations MUST not throw RuntimeException exceptions on the failures of remote operations, authentication or other operational problems.
- Stream read operations MAY fail if the read channel has been idle for a FileSystem-specific period of time.
- Stream write operations MAY fail if the write channel has been idle for a FileSystem-specific period of time.
- network and write failures MAY be raised in the Stream close() operation
- write streams