Permissions Guide

Table of contents

1. Overview

The Hadoop Distributed File System (HDFS) implements a permissions model for files and directories that shares much of the POSIX model. Each file and directory is associated with an *owner* and a *group*. The file or directory has separate permissions for the user that is the owner, for other users that are members of the group, and for all other users. For files, the *r* permission is required to read the file, and the *w* permission is required to write or append to the file. For directories, the *r* permission is required to list the contents of the directory, the *w* permission is required to create or delete files or directories, and the *x* permission is required to access a child of the directory.

In contrast to the POSIX model, there are no *sticky*, *setuid* or *setgid* bits for files as there is no notion of executable files. For directories, there are no *setuid* or *setgid* bits directory as a simplification. Collectively, the permissions of a file or directory are its *mode*. In general, Unix customs for representing and displaying modes will be used, including the use of octal numbers in this description. When a file or directory is created, its owner is the user identity of the client process, and its group is the group of the parent directory (the BSD rule).

Each client process that accesses HDFS has a two-part identity composed of the *user name*, and *groups list*. Whenever HDFS must do a permissions check for a file or directory `foo` accessed by a client process,

- If the user name matches the owner of `foo`, then the owner permissions are tested;
- Else if the group of `foo` matches any of member of the groups list, then the group permissions are tested;
- Otherwise the other permissions of `foo` are tested.

If a permissions check fails, the client operation fails.

2. User Identity

As of CDH3, Hadoop supports two different modes of operation to determine the user's identity, specified by the `hadoop.security.authentication` property:

`simple`
In this mode of operation, the identity of a client process is determined by the host operating system. On Unix-like systems, the user name is the equivalent of `` `whoami` ``.

`kerberos`
In Kerberized operation, the identity of a client process is determined by its Kerberos credentials. For example, in a Kerberized environment, a user may use the `kinit` utility to obtain a Kerberos ticket-granting-ticket (TGT) and use `klist`

to determine their current principal. When mapping a Kerberos principal to an HDFS username, all *components* except for the *primary* are dropped. For example, a principal `todd/foobar@CORP.COMPANY.COM` will act as the simple username `todd` on HDFS.

Regardless of the mode of operation, the user identity mechanism is extrinsic to HDFS itself. There is no provision within HDFS for creating user identities, establishing groups, or processing user credentials.

3. Group Mapping

Once a username has been determined as described above, the list of groups is determined by a *group mapping service*, configured by the `hadoop.security.group.mapping` property. The default implementation, `org.apache.hadoop.security.ShellBasedUnixGroupsMapping`, will shell out to the Unix `bash -c groups` command to resolve a list of groups for a user.

For HDFS, the mapping of users to groups is performed on the NameNode. Thus, the host system configuration of the NameNode determines the group mappings for the users.

Note that HDFS stores the user and group of a file or directory as strings; there is no conversion from user and group identity numbers as is conventional in Unix.

4. Understanding the Implementation

Each file or directory operation passes the full path name to the name node, and the permissions checks are applied along the path for each operation. The client framework will implicitly associate the user identity with the connection to the name node, reducing the need for changes to the existing client API. It has always been the case that when one operation on a file succeeds, the operation might fail when repeated because the file, or some directory on the path, no longer exists. For instance, when the client first begins reading a file, it makes a first request to the name node to discover the location of the first blocks of the file. A second request made to find additional blocks may fail. On the other hand, deleting a file does not revoke access by a client that already knows the blocks of the file. With the addition of permissions, a client's access to a file may be withdrawn between requests. Again, changing permissions does not revoke the access of a client that already knows the file's blocks.

5. Changes to the File System API

All methods that use a path parameter will throw `AccessControlException` if permission checking fails.

New methods:

- `public FSDataOutputStream create(Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication, long blockSize, Progressable progress) throws IOException;`
- `public boolean mkdirs(Path f, FsPermission permission) throws IOException;`
- `public void setPermission(Path p, FsPermission permission) throws IOException;`
- `public void setOwner(Path p, String username, String groupname) throws IOException;`
- `public FileStatus getFileStatus(Path f) throws IOException;` will additionally return the user, group and mode associated with the path.

The mode of a new file or directory is restricted my the `umask` set as a configuration parameter. When the existing `create(path, …)` method (*without* the permission parameter) is used, the mode of the new file is `666&^umask`. When the new `create(path,` *permission*`, …)` method (*with* the permission parameter *P*) is used, the mode of the new file is `P&^umask&666`. When a new directory is created with the existing `mkdirs(path)` method (*without* the permission parameter), the mode of the new directory is `777&^umask`. When the new `mkdirs(path,` *permission* `)` method (*with* the permission parameter *P*) is used, the mode of new directory is `P&^umask&777`.

6. Changes to the Application Shell

New operations:

- `chmod [-R]` *mode file …*
  Only the owner of a file or the super-user is permitted to change the mode of a file.
- `chgrp [-R]` *group file …*
  The user invoking `chgrp` must belong to the specified group and be the owner of the file, or be the super-user.
- `chown [-R]` *[owner][:[group]] file …*
  The owner of a file may only be altered by a super-user.
- `ls` *file …*
- `lsr` *file …*
  The output is reformatted to display the owner, group and mode.

7. The Super-User

The super-user is the user with the same identity as name node process itself. Loosely, if you started the name node, then you are the super-user. The super-user can do anything in that permissions checks never fail for the super-user. There is no persistent notion of who *was* the super-user; when the name node is started the process identity determines who is the super-user *for now*. The HDFS super-user does not have to be the super-user of the name node host, nor is it necessary that all clusters have the same super-user. Also, an experimenter running HDFS on a personal workstation, conveniently becomes that installation's super-user without any configuration.

In addition, the administrator my identify a distinguished group using a configuration parameter. If set, members of this group are also super-users.

8. The Web Server

By default, the identity of the web server is a configuration parameter. That is, the name node has no notion of the identity of the *real* user, but the web server behaves as if it has the identity (user and groups) of a user chosen by the administrator. Unless the chosen identity matches the super-user, parts of the name space may be inaccessible to the web server.

9. Configuration Parameters

- `dfs.permissions = true`
  If `yes` use the permissions system as described here. If `no`, permission *checking* is turned off, but all other behavior is unchanged. Switching from one parameter value to the other does not change the mode, owner or group of files or directories.
  Regardless of whether permissions are on or off, `chmod`, `chgrp` and `chown` *always* check permissions. These functions are only useful in the permissions context, and so there is no backwards compatibility issue. Furthermore, this allows administrators to reliably set owners and permissions in advance of turning on regular permissions checking.
- `dfs.web.ugi = webuser,webgroup`
  The user name to be used by the web server. Setting this to the name of the super-user allows any web client to see everything. Changing this to an otherwise unused identity allows web clients to see only those things visible using "other" permissions. Additional groups may be added to the comma-separated list.
- `dfs.permissions.supergroup = supergroup`
  The name of the group of super-users.
- `dfs.upgrade.permission = 0777`
  The choice of initial mode during upgrade. The *x* permission is *never* set for files. For configuration files, the decimal value *51110* may be used.
- `dfs.umask = 022`
  The `umask` used when creating files and directories. For configuration files, the decimal

value *1810* may be used.