HDFS ACLs Design

{cnauroth,sanjay}@hortonworks.com

Last Updated: 2014-02-07

Introduction

This document discusses use cases, requirements, and design of the HDFS ACLs feature. ACLs (access control lists) enhance the existing HDFS permission model to support controlling file access for arbitrary combinations of users and groups instead of a single owner, single group, and all other users. Development of this feature is tracked in Apache issue HDFS-4685.

Problem Statement

The current HDFS permission model is equivalent to traditional Unix permission bits. For each file or directory, permissions are managed for a set of 3 distinct user classes: owner, group, and others. There are 3 different permissions controlled for each user class: read, write, and execute. Thus, for any file system object, its permissions can be encoded in 3*3=9 bits. When a user attempts to access a file system object, HDFS enforces permissions according to the most specific user class applicable to that user. If the user is the owner, then HDFS checks the owner class permissions. If the user is not the owner, but is a member of the file system object's group, then HDFS checks the group class permissions. Otherwise, HDFS checks the others class permissions.

This model is sufficient to express a large number of security requirements. For example, consider a sales department that wants a single user, the department manager, to control all modifications to sales data. Other members of the department need to view the data, but must not be able to modify it. Everyone else in the company outside the sales department must not be able to view the data. This requirement can be implemented by running chmod 640 on the file, with the following outcome:

```
-rw-r-----  1 bruce  sales    22K Nov 18 10:55 sales-data
```

Only bruce may modify the file, only members of the sales group may read the file, and no one else may access the file in any way.

Now suppose there are new requirements. The sales department has grown such that it's no longer feasible for the manager, bruce, to control all modifications to the file. Instead, the new requirement is that bruce, diana, and clark are allowed to make modifications. Unfortunately, there is no way for permission bits to express this requirement, because there can be only one owner and one group, and the group is already used to implement the read-only requirement for the sales team. A typical workaround is to set the file owner to a synthetic user account, such as salesmgr, and allow bruce, diana, and clark to use the salesmgr account via sudo or similar impersonation mechanisms. The drawback to this workaround is that it forces complexity on to end users, because it requires them to use different accounts for different actions.

Also suppose that in addition to the sales staff, all executives in the company need to be able to

read the sales data. This is another requirement that cannot be expressed with permission bits, because there is only one group, and it's already used by sales. A typical workaround is to set the file's group to a new synthetic group, such as salesandexecs, and add all users of sales and all users of execs to that group. The drawback to this workaround is that it requires administrators to create numerous additional groups and to guarantee correct membership in those groups for all users.

We can generalize the above examples by stating that it is awkward to use permission bits to implement permission requirements that differ from the natural organizational hierarchy of users and groups. The goal of ACLs is to allow users to implement these requirements more naturally by stating that for any file system object, multiple users and multiple groups have different permissions.

Use Cases

This section discusses uses cases motivating the development of ACLs. Each use case describes how an ACL can be used to implement the requirement, using a textual representation of the ACL. The textual representation is intuitive in the context of the use case discussion. If the textual representation is unclear, see the following resources for more information:

http://users.suse.com/~agruen/acl/linux-acls/online/

https://www.cs.unc.edu/cms/help/help-articles/posix-acls-in-linux

UC1: Multiple Users

Multiple users require read access to a file. None of the users are the owner of the file. The users are not members of a common group, so it is impossible to use group permission bits.

This use case can be implemented by setting an access ACL containing multiple named user entries:

```
user:bruce:r--
user:diana:r--
user:clark:r--
```

UC2: Multiple Groups

Multiple groups require read and write access to a file. There is no group containing the union of all groups' members, so it is impossible to use group permission bits.

This use case can be implemented by setting an access ACL containing multiple named group entries:

```
group:sales:rw-
group:execs:rw-
```

UC3: Hive Partitioned Tables

Hive contains a partitioned table of sales data. The partition key is country. Hive persists partitioned tables using a separate sub-directory for each distinct value of the partition key, so the file system structure in HDFS looks like this:

```
user
`-- hive
    `-- warehouse
        `-- sales
            |-- country=CN
            |-- country=GB
            `-- country=US
```

A group named salesadmins is the owning group for all of these files. Members of this group may read or write all files. Separate country-specific groups may run Hive queries that read only data for a specific country, e.g. sales_CN, sales_GB, and sales_US. These groups do not have write access.

This use case can be implemented by setting an access ACL on each sub-directory containing an owning group entry and a named group entry:

```
country=CN
group::rwx
group:sales_CN:r-x

country=GB
group::rwx
group:sales_GB:r-x

country=US
group::rwx
group:sales_US:r-x
```

Note: The functionality of the owning group ACL entry (the group entry with no name) is equivalent to setting permission bits. The Design section discusses this in greater detail.

UC4: Default ACLs

A file system administrator or subtree owner wants to define an access policy applied to the entire subtree. This access policy must apply not only to the current set of files and directories, but also to new files and directories added in the future.

This use case can be implemented by setting a default ACL on the directory. The default ACL may contain any arbitrary combination of entries. For example:

```
default:user::rwx
default:user:bruce:rw-
default:user:diana:r--
default:user:clark:rw-
default:group::r--
default:group:sales::rw-
default:group:execs::rw-
default:others::---
```

During design discussions, there was an open question about how best to support the capability to override the default ACL at lower layers of the tree. There are two models possible:

1. **Umask-Default-ACL:** The default ACL of the parent is cloned to the ACL of the child at time of child creation. For new child directories, the default ACL itself is also cloned, so that the same policy is applied to sub-directories of sub-directories. Subsequent changes to the parent's default ACL will set a different ACL for new children, but will not alter existing children. This matches POSIX behavior. If the administrator wants to change policy on the sub-tree later, then this is performed by inserting a new more restrictive ACL entry at the appropriate sub-tree root (see UC6) and may also need to run a recursive ACL modification (analogous to chmod -R) since existing children are not effected by the new ACL.
2. **Inherited-Default-ACL:** A child that does not have an ACL of its own inherits its ACL from the nearest ancestor that has defined a default ACL. A child node that requires a different ACL can override the default (like the Umask-Default-ACL). Subsequent changes to the ancestor's default ACL will cause all children *that do not have an ACL* to inherit the new ACL regardless of child creation time (unlike Umask-Default-ACL). This model, like the ABAC ACLs (use case UC8), encourages the user to create fewer ACLs (typically on the root of specific subtrees) while the Posix-compliant Umask-Default-ACL is expected to results in larger number of ACLs in the system. It would also make a memory efficient implementation trivial. Note that this model is a deviation from POSIX behavior.

There are three subcases here

4a) Open up a child for wider access than the default.
4b) Restrict a child for narrower access than the default.
4c) Change the defaultAcl because you made a mistake originally.

Both models support use case 4a and 4b with equal ease. However, with the Inherited-Default-ACL, it is easy to identify children that have overridden the default-ACL - the existence of an ACL means that the user intended to override the default. Also 4c is a natural fit for Inherited-Default-ACL. For the UMask-Default-ACL, every child has an ACL and hence you have to walk down the subtree and compare the ACL with the default to see if the user had intended to override it.

Our conclusion is that we will implement Umask-Default-ACL, which maintains compliance with POSIX. This is a simpler extension of the existing HDFS permissions model, which does not rely on true inheritance, so we expect this will be less surprising for cluster administrators. If real-world usage demonstrates a strong need for true inheritance, then we can potentially consider implemented the NFSv4 model of ACLs in the future as a separate feature. POSIX ACLs and NFSv4 ACLs can coexist, because POSIX ACLs can be seen as a subset of the capabilities of NFSv4 ACLs.

UC5: Minimal ACL/Permissions Only

We maintain full support for deployments that want to use only permission bits and not ACLs with named user and named group entries. Permission bits are equivalent to a minimal ACL: a degenerate case ACL containing only 3 entries. For example:

```
user::rw-
group::r--
others::---
```

UC6: Block Access to a Sub-Tree for a Specific User

A deeply nested sub-tree of the file system was created as world-readable. Now there is a need to block access for a specific user to all files in that sub-tree.

This use case can be implemented by setting an ACL on the root of the sub-tree with a named user entry that strips all access from the user.

Assume this file system structure:

```
dir1
`-- dir2
    `-- dir3
        |-- file1
        |-- file2
```

```
        `-- file3
```

Setting the following ACL on dir2 blocks access for bruce to dir3, file1, file2, and file3:

```
user:bruce:---
```

More specifically, the removal of execute permissions on dir2 means that bruce cannot traverse into dir2, and therefore cannot see any of its children. This also means that access is blocked automatically for newly added files under dir2. If a file4 was created under dir3, bruce wouldn't be able to access it.

UC7: ACLs with Sticky Bit

Multiple named users or named groups require full access to a general-purpose shared directory, such as /tmp. However, write and execute permissions on the directory also give users the ability to delete or rename any files in the directory, even files created by other users. Users must be restricted so that they can delete or rename only files that they created.

This use case can be implemented by combining an ACL with the sticky bit. The sticky bit is existing functionality that currently works with permission bits. It will continue to work as expected in combination with ACLs.

UC8: ABAC (Attribute-Based Access Control)

A file system administrator wants to define an access control policy in terms of arbitrary custom attributes attached to users and files. These attributes are free-form. There is no relation to concepts of file system hierarchy, file ownership, per-user permissions or per-group permissions.

For example, the administrator wants to tag a set of files as "confidential". The set of confidential files is scattered throughout the file system with no predictable hierarchical relationship between files. Only the users that also have the "confidential" property may access these files.

HDFS ACLs will not support this use case. This use case could be implemented by an alternative permissions model known as ABAC (Attribute-Based Access Control):

http://csrc.nist.gov/publications/drafts/800-162/sp800_162_draft.pdf

This project does not intend to implement ABAC, but this can be added later in a backwards-compatible way as a separate feature.

Requirements

HDFS must be able to associate an optional ACL with any file or directory. All HDFS operations that currently enforce permissions expressed in permission bits must also consider the ACL for the file or directory, if it is defined. Any existing logic that bypasses permission bits enforcement must also bypass ACLs. This includes the HDFS super-user and setting dfs.permissions to false in configuration.

HDFS must support new operations for setting and getting the ACL associated to a file or directory. These new operations must be accessible through all user-facing endpoints. This includes the FsShell CLI, programmatic manipulation through the FileSystem and FileContext classes, WebHDFS, libHDFS, and NFS. The file owner is allowed to set the ACL. Any user with read permissions on the file is allowed to get the ACL. Additionally, the HDFS super-user is allowed to call set and get the ACL of any file. (This is equivalent to the current logic for permission bits.)

The output of existing CLI commands must indicate the presence of ACLs. alongside the existing permission bits

The implementation of ACLs must be backwards-compatible with existing usage of permission bits. To achieve this, changes applied via permission bits (i.e. chmod) are also visible as changes in the ACL. Likewise, changes applied to ACL entries for the base user classes (owner, group, and others) are also visible as changes in the permission bits. In other words, permission bit operations and ACL operations manipulate a shared model, and the permission bit operations can be considered a subset of the ACL operations.

The addition of ACLs as a feature must not cause a detrimental impact to consumption of system resources in deployments that do not want to use ACLs. This includes CPU, memory, disk, and network bandwidth.

The number of entries in a single ACL is capped at a maximum of 32. Attempts to add ACL entries exceeding the maximum will fail with a user-facing error. This is done for 2 reasons: to simplify management, and to limit resource consumption. ACLs with a very high number of entries tend to become difficult to understand and may indicate that the requirements are better implemented by defining additional groups or users. ACLs with a very high number of entries also would require more memory and storage and take longer to evaluate on each permission check. The number 32 is chosen for consistency with the maximum number of ACL entries enforced by the ext family of file systems.

http://users.suse.com/~agruen/acl/linux-acls/online/#tab:acl_entries

Symlinks do not have ACLs of their own. The ACL of a symlink is always seen as the default permissions (777 permission bits). Operations that modify the ACL of a symlink instead modify

the ACL of the symlink's target. This is consistent with current treatment of permissions for symlinks.

Within a snapshot, all ACLs are frozen at the moment that the snapshot was created. ACL changes in the parent of the snapshot are not applied to the snapshot.

Tooling that currently propagates permission bits will not propagate ACLs. This includes the cp -p shell command and distcp -p. This would be difficult to guarantee, particularly in the case of trying to propagate ACLs from a local file system or running distcp between 2 different HDFS versions, one of which may not support ACLs.

Design

Our design selects the POSIX ACL model for the sake of simplicity and familiarity. As an alternative, we also considered the NFSv4 ACL model. We assert that the POSIX model is sufficient to meet known use cases, and there is a risk that the additional complexity of the NFSv4 model could be a source of confusion for end users.

The rest of the design document assumes the reader is familiar with the POSIX ACL domain model. Here are several resources that discuss POSIX ACLs:

http://users.suse.com/~agruen/acl/linux-acls/online/

https://www.cs.unc.edu/cms/help/help-articles/posix-acls-in-linux

Enforcement

All existing HDFS operations that enforce permissions must also enforce ACLs, if defined on the inode. The following is pseudo-code that describes calculation of the client's effective permissions in the presence of ACLs. Every client operation executes in the context of exactly one user and zero or more group memberships. It is possible that multiple ACL entries are a match for the user. This logic disambiguates selection of the applicable ACL entries and uses the entries to calculate the final effective permissions to enforce. The pseudo-code uses a mix of Java-like syntax and set notation.

```
let user = the requesting user
let userGroups = { the set of all of user's groups }
let fileOwner = the file's owner
let fileGroup = the file's group
let aclEntries = { the set of ACL entries for the file }
let userGroupsInAcl = userGroups ∩ (fileGroup ∪ aclEntries.getNamedGroups())

if (user == fileOwner) {
  effectivePermissions = aclEntries.getOwnerPermissions()
```

```
} else if (user ∈ aclEntries.getNamedUsers()) {
  effectivePermissions = aclEntries.getNamedUserPermissions(user)
} else if (userGroupsInAcl != ∅) {
  effectivePermissions = ∅
  if (fileGroup ∈ userGroupsInAcl) {
    effectivePermissions = effectivePermissions ∪
      aclEntries.getGroupPermissions()
  }
  for ({group | group ∈ userGroupsInAcl}) {
    effectivePermissions = effectivePermissions ∪
      aclEntries.getNamedGroupPermissions(group)
  }
} else {
  effectivePermissions = aclEntries.getOthersPermissions()
}
```

To summarize several key points of this logic:

1. The file owner entry is always the most specific match. If the user is the file owner, then the owner entry defines the effective permissions completely. Even if there is also a matching named user entry, the owner entry still takes precedence, and the named user entry is not considered.
2. If the user is a member of the file's group or at least one group for which there is a named group entry in the ACL, then effective permissions are calculated from groups. This is the union of the file group permissions (if the user is a member of the file group) and all named group entries matching the user's groups. For example, consider a user that is a member of 2 groups: sales and execs. The user is not the file owner, and the ACL contains no named user entries. The ACL contains named group entries for both groups as follows: group:sales:r--, group:execs:-w-. In this case, the user's effective permissions are rw-.
3. This logic is compatible with deployments that do not wish to use ACLs. By substituting the empty set (∅) for aclEntries and tracing through the logic, we can see that the conditional paths for user entries and named group entries simply never execute. The logic degenerates to selection of owner, group, or others, which is the same way that permissions work today.

FsShell CLI

Two new sub-commands are added to FsShell: setfacl and getfacl. These are modeled after the same Linux shell commands, though fewer flags are implemented. Support for the additional flags could be added later as an enhancement if required.

```
-setfacl [-bkR] {-m|-x} <acl_spec> <path>
-setfacl --set <acl_spec> <path>

Sets Access Control Lists (ACLs) of files and directories.
```

```
-b:         Remove all but the base ACL entries.  The entries for user, group,
            and others are retained for compatibility with permission bits.
-k:         Remove the default ACL.
-R:         Apply operations to all files and directories recursively.
-m:         Modify ACL.  New entries are added to the ACL, and existing
            entries are retained.
-x:         Remove specified ACL entries.  Other ACL entries are retained.
--set:      Fully replace the ACL, discarding all existing entries.  The
            <acl_spec> must include entries for user, group, and others for
            compatibility with permission bits.
<acl_spec>: Comma-separated list of ACL entries.
<path>:     File or directory to modify.

-getfacl [-R] <path>

Displays the Access Control Lists (ACLs) of files and directories.  If a
directory has a default ACL, then getfacl also displays the default ACL.

-R:     List the ACLs of all files and directories recursively.
<path>: File or directory to list.
```

The output of the ls command is changed to append the character '+' to the permissions of any file or directory that has an ACL. This is typical of ACL implementations on Linux.

http://en.wikipedia.org/wiki/File_system_permissions#Traditional_Unix_permissions

Web UI

The file system browser is changed to append the '+' character to the permissions of any file or directory that has an ACL. This is done for parity with the CLI changes.

There is no intention to build new web UI functionality for full display of ACLs at this time. This could be done later as a separate feature if desired.

File System API

The following public API methods are added to FileSystem and FileContext to support setting and getting ACLs.

All methods accept an EnumSet of flags that can control the behavior of the method. This reduces the overall API footprint and prevents the need to add method overloads later if additional flags are required.

AclSpec is a specification of how to change ACL entries. In some cases, an AclSpec may not be a complete ACL. For example, when deleting an ACL entry for a named user or named group, the ACL spec contains the named user or group, but it does not include permissions. (This information would be useless in this context, because the ACL entry is being removed.)

getAcls is a new method capable of returning an iterator over requested ACLs. The Acl object contains members that fully describe the path, the current permission bits, and the ACL if present. An alternative implementation would have been to add Acl as a member field of the existing FileStatus, FsPermission or PermissionStatus classes. This is rejected for compatibility reasons. If a new version of the client (post-ACLs) deserializes an old version of the FileStatus (pre-ACLs), then this could cause bugs. If we assume a null Acl if we encounter EOF during deserialization, then there is a risk of null pointer dereferences in downstream code. Modifying FsPermission would have caused an impact to storage and network bandwidth, because the current FsPermission is encoded compactly as a 16-bit integer.

```
/**
 * Modifies ACL entries of files and directories.  This method can add new ACL
 * entries or modify the permissions on existing ACL entries.  All existing
 * ACL entries that are not specified in this call are retained without
 * changes.  (Modifications are merged into the current ACL.)
 *
 * @param path Path to modify
 * @param aclSpec List<AclEntry> describing modifications
 * @throws IOException if an ACL could not be modified
 */
public void modifyAclEntries(Path path, List<AclEntry> aclSpec)
  throws IOException

/**
 * Removes ACL entries from files and directories.  Other ACL entries are
 * retained.
 *
 * @param path Path to modify
 * @param aclSpec List<AclEntry> describing entries to remove
 * @throws IOException if an ACL could not be modified
 */
public void removeAclEntries(Path path, List<AclEntry> aclSpec)
  throws IOException

/**
 * Removes all default ACL entries from files and directories.
 *
 * @param path Path to modify
 * @throws IOException if an ACL could not be modified
 */
public void removeDefaultAcl(Path path) throws IOException
```

```
/**
 * Removes all but the base ACL entries of files and directories.  The
 * entries for user, group, and others are retained for compatibility
 * with permission bits.
 *
 * @param path Path to modify
 * @throws IOException if an ACL could not be removed
 */
public void removeAcl(Path path) throws IOException

/**
 * Fully replaces ACL of files and directories, discarding all existing
 * entries.
 *
 * @param path Path to modify
 * @param aclSpec List<AclEntry> describing modifications, must include
 *   entries for user, group, and others for compatibility with permission
 *   bits.
 * @throws IOException if an ACL could not be modified
 */
public void setAcl(Path path, AclSpec aclSpec) throws IOException

/**
 * Gets the ACL of a file or directory.
 *
 * @param path Path to get
 * @return AclStatus describing the ACL of the file or directory
 * @throws IOException if an ACL could not be read
 */
public AclStatus getAclStatus(Path path) throws IOException
```

libHDFS

The following functions are added to libHDFS. Each function is a straightforward C binding of the corresponding Java FileSystem method. This binding follows prior established patterns in libHDFS:

- The first argument is an hdfsFS handle.
- New parameters and return values are opaque data types: forward-declared structs that the caller accesses by pointer. This enables backwards-compatible future changes in the data layout.
- Callers interact with the new structures by passing them to getters and setters. (These functions are straightforward and omitted here for brevity.)
- Callers are responsible for deallocation by passing the structures to a corresponding hdfsFree method. (These functions are straightforward and omitted here for brevity.)
- Java exceptions are mapped to int return codes.

```
/**
 * Modifies ACL entries of files and directories.  This method can add new ACL
 * entries or modify the permissions on existing ACL entries.  All existing
 * ACL entries that are not specified in this call are retained without
 * changes.  (Modifications are merged into the current ACL.)
 *
 * @param fs the configured filesystem handle
 * @param path path to modify
 * @param aclSpec hdfsAclSpec describing modifications
 * @return 0 on success else -1
 */
int hdfsModifyAclEntries(hdfsFS fs, const char *path,
  hdfsAclSpec aclSpec);

/**
 * Removes ACL entries from files and directories.  Other ACL entries are
 * retained.
 *
 * @param fs the configured filesystem handle
 * @param path path to modify
 * @param aclSpec hdfsAclSpec describing modifications
 * @return 0 on success else -1
 */
int hdfsRemoveAclEntries(hdfsFS fs, const char *path,
  hdfsAclSpec aclSpec);

/**
 * Removes all default ACL entries from files and directories.
 *
 * @param fs the configured filesystem handle
 * @param path path to modify
 * @return 0 on success else -1
 */
int hdfsRemoveDefaultAcl(hdfsFS fs, const char *path);

/**
 * Removes all but the base ACL entries of files and directories.  The
 * entries for user, group, and others are retained for compatibility
 * with permission bits.
 *
 * @param fs the configured filesystem handle
 * @param path path to modify
 * @return 0 on success else -1
 */
int hdfsRemoveAcl(hdfsFS fs, const char *path);

/**
```

```
 * Fully replaces ACL of files and directories, discarding all existing
 * entries.
 *
 * @param fs the configured filesystem handle
 * @param path path to modify
 * @param aclSpec hdfsAclSpec describing modifications
 * @return 0 on success else -1
 */
int hdfsSetAcl(hdfsFS fs, const char *path, hdfsAclSpec aclSpec);

/**
 * Gets the ACL of a file or directory.
 *
 * @param path Path to get
 * @return hdfsAclStatus describing the ACL of the file or directory.  This
 *   data structure is dynamically allocated, and it must be deallocated by
 *   calling hdfsAclStatusFree when done.  Returns NULL on error.
 */
hdfsAclStatus hdfsGetAclStatus(hdfsFS fs, const char *path);
```

WebHDFS/HttpFS

The following additions are made to the REST API of both WebHDFS and HttpFS. Each operation is equivalent to the method with the same name on the FileSystem API.

An <ACLSPEC> is a URL-encoded string describing how to change the ACL. The string consists of one or more comma-separate entries. In general, ACL spec entries match the syntax for ACL entries shown earlier in the problem statement and use cases. For example:

```
user:bruce:rw-,user:diana:r--
```

However, in some cases, the ACL spec contains entries that are not complete ACL entries. In particular, requests to remove ACL entries do not include the permission portion in the ACL spec entries. It would be meaningless to include permission information, because the entry is being removed. Here is an example of an ACL spec sent with a REMOVEACLENTRIES request:

```
user:bruce

curl -i -X PUT 'http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=MODIFYACLENTRIES
  &aclspec=<ACLSPEC>'

curl -i -X PUT 'http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=REMOVEACLENTRIES
  &aclspec=<ACLSPEC>'

curl -i -X PUT 'http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=REMOVEDEFAULTACL'
```

```
curl -i -X PUT 'http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=REMOVEACL'

curl -i -X PUT 'http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=SETACL
  &aclspec=<ACLSPEC>'
```

For each modification operation, the client receives a response with zero content length:

```
HTTP/1.1 200 OK
Content-Length: 0
```

For get, the client receives a response with an `AclStatus` JSON object:

```
curl -i 'http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=GETACLSTATUS'

{
  "AclStatus":
  {
    "owner": "bruce",
    "group": "sales",
    "stickyBit": false,
    "entries":
    [
      "user::rwx",
      "user:bruce:rw-",
      "group::r-x",
      "group:execs:rw-",
      "other::---"
    ]
  }
}
```

All operations obey REST semantics: PUT operations are idempotent and the GET operation is safe.

NFS

HDFS currently implements NFSv3. The NFSv3 base protocol does not include support for ACLs. However, it is possible to support getting and setting ACLs via NFS by implementing the NFSACL protocol extension for NFSv3:

http://lwn.net/Articles/120338/

NameNode Persistence of ACLs

In-Memory Representation

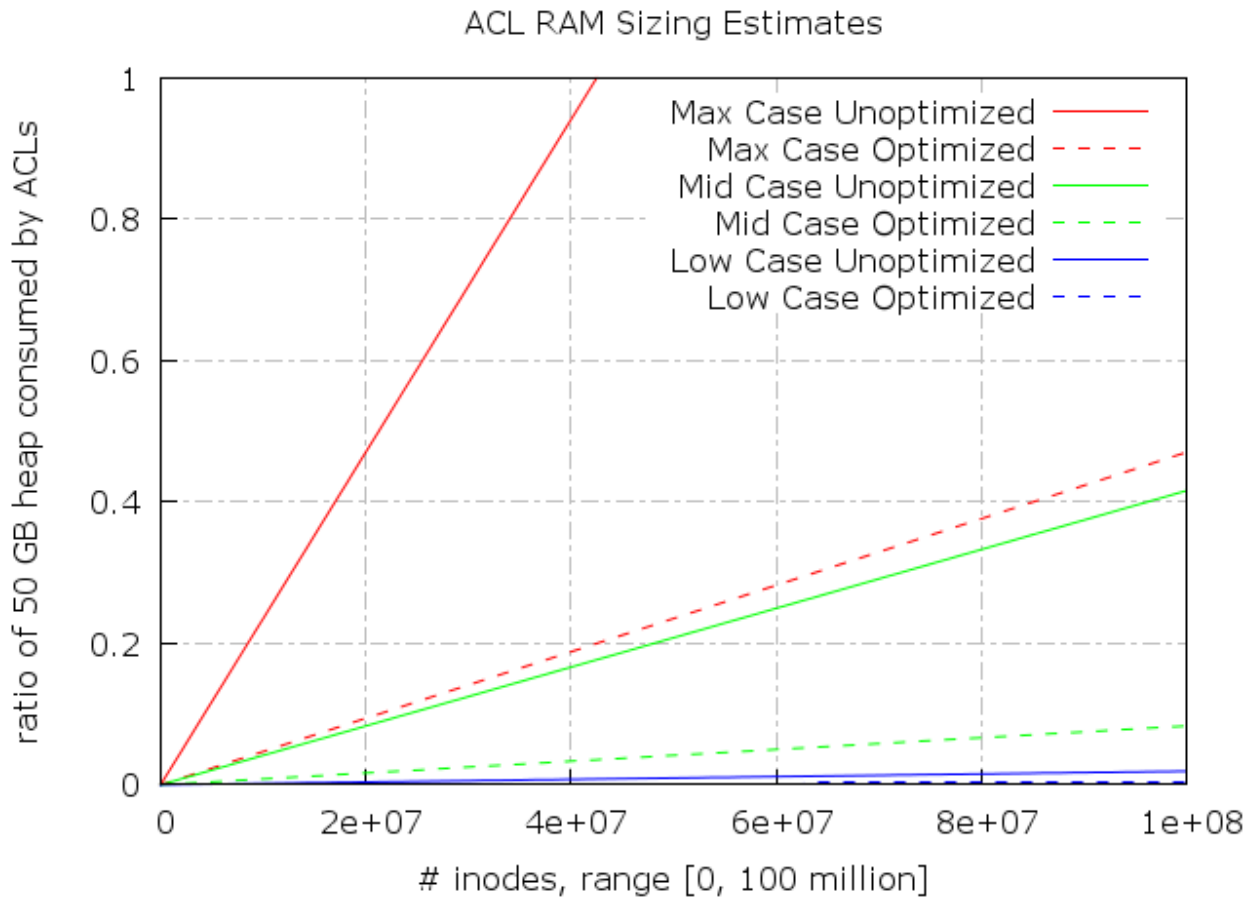
One goal of the design is to minimize increase in the NameNode memory footprint. In particular,

deployments that do not wish to use ACLs must not suffer a large increase in memory consumption as a side effect of introducing this feature.

The most straightforward implementation would attach a nullable ACL field to every inode, in the FsPermission, PermissionStatus, or INode classes. This is rejected for failure to meet the above goal. The addition of a reference field would increase memory consumption per inode in all deployments, even where ACLs are not used. Additionally, there would be backwards-compatibility issues in changing FsPermission or FsPermissionStatus. (See discussion in FileSystem API.)

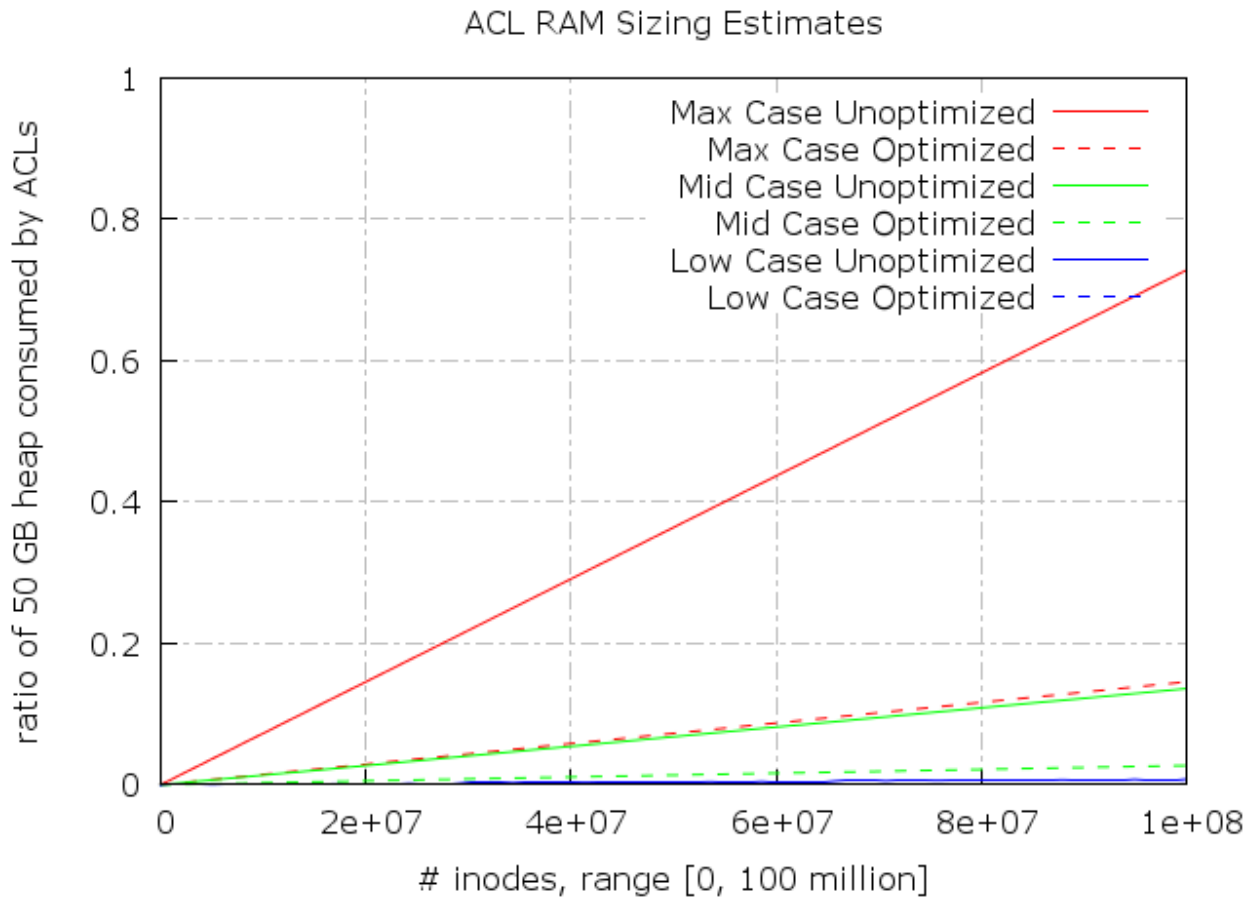
Instead, the design moves storage of ACLs into a new inode feature. The new feature contains a list of ACL entries, and it is attached to an inode only if that inode has an ACL. In this way, the feature imposes no additional cost in memory on deployments that choose not to make use of ACLs.

A new data structure called the **Global ACL Set** can store every distinct ACL that is in use by files or directories in the NameNode. Thus, multiple inodes with the exact same ACL entries can share representation of that ACL in memory. There is no need to store duplicate copies. When a client sets an ACL, the Global ACL Set is responsible for either creating a new ACL (if the same entries are not used on existing inodes) or simply returning an existing ACL (if the same entries are already used on existing inodes). New ACL instances are created in memory only when needed, so the Global ACL Set provides copy-on-write semantics. Additionally, the entries of the Global ACL Set are referenced-counted. File deletions and ACL removals decrement the reference count. When the reference count drops to zero, the Global ACL Set can delete the ACL completely from memory.



The above plot estimates memory consumption by ACLs in a presumed 50 GB NameNode, trying to store up to 100 million inodes. The plot includes 3 different usage scenarios:
max usage: All inodes have an ACL, and each one has the maximum ACL entries (32).
mid usage: Half the inodes have an ACL, and each ACL has 10 entries.
low usage: 10% of inodes have an ACL, and each ACL has 2 entries.
For each scenario, the plot shows unoptimized memory consumption and also the optimized memory consumption, assuming that only 20% of the ACLs are unique. Each scenario uses a different line color. The unoptimized version uses a solid line, and the optimized version uses a dashed line.

The plot demonstrates that ACL usage really needs to get quite high (very large number of inodes with a very high proportion of them having ACLs) before this memory optimization starts to provide a benefit.



This plot is similar to the above, but estimates an additional optimization technique applied in conjunction with the Global ACL Set. Instead of storing ACLs in their natural object model representation, a more compact in-memory encoding is possible. The AclEntry data structure is:

```
private final AclEntryType type; // 4 values: user, group, mask, other
private final String name; // user name or group name
private final FsAction permission; // 8 values: all, read_write, etc.
private final AclEntryScope scope; // 2 values: access, default
```

This data structure takes up 32 bytes (4 8-byte pointers on a 64-bit architecture). Instead, we could pack each entry into an array of 32-bit ints, using 2 bits for type, 3 bits for permission, 1 bit for scope, and 25 bits for user name or group name (like PermissionStatusFormat). Using both optimization techniques shows that it's possible to achieve small memory consumption, even for deployments making very heavy use of ACLs.

Neither the Global ACL Set nor the compact encoding are implemented in the current version, but they can be implemented in future versions if real-world usage justifies it.

An earlier design alternative for in-memory representation considered repurposing the existing permission bits on the inode to control how that specific inode interacts with the ACL data structures. While this design is slightly more compact than the feature-based approach, it comes with a major cost in complexity. The feature-based approach still maintains the most important characteristic that deployments that do not use ACLs will not pay a cost. The current plan is to proceed with the feature-based implementation, but the alternative is described below in case it becomes worthwhile as a future change.

The **INode ACL Map** can associate any inode to one of the shared ACL instances. This data structure is sparse. It does not contain an entry for every inode in the file system. This data structure is only required in clusters that use a large number of distinct ACLs. (See below for further discussion.) The values of the INode ACL Map entries are references to the shared ACL instances maintained by the Global ACL Set. (This is an application of the flyweight pattern, allowing different INode instances to share the same underlying ACL instance.)

The value stored in the inode's permission bits controls its usage of either permissions or an ACL during permission checks. The permission bits currently are stored in 16 bits per inode (a Java 16-bit short). There are 3 cases to consider for the values in this field:

`0x0000 - 0x03FF`
: Values in this range indicate that the inode uses the existing permissions model, not an ACL. This range can be represented using just 10 bits: the 3 base user classes (owner, group, others) * 3 permissions (read, write, execute) + 1 for sticky bit. The reason this works is that any combination of permission bits in this range can be viewed as a degenerate case of an ACL or "minimal ACL". It is an ACL that has no entries for named users, named groups or defaults.

`0x4000 - 0xFFFE`
: Values in this range indicate that the inode uses an ACL that can be referenced by an indexed array lookup from the Global ACL Set. The encoded value is used to index an array of ACLs, after appropriate conversion from short (signed) to int (in an unsigned range). As stated above, we require 10 bits to represent the existing minimal ACLs. That leaves 2^16 - 2^10 - 1 = 65536 - 1024 - 1 = 64511 additional distinct ACLs that can be referenced by this range. Recall that multiple inodes can share the same ACL instance, so this limitation must not be interpreted as "64511 total inodes with ACLs." Instead, it should be interpreted as "64511 distinct combinations of ACL entries across all inodes."

`0xFFFF`
: This reserved value indicates that a separate lookup is required in the INode ACL Map to find this inode's ACL. This is only used in clusters that exceed the limit of 64511 distinct ACLs that we can represent in the indexed lookup technique described above.

In clusters that do not use ACLs, the memory footprint is limited to the 1024 entries required in the Global ACL Set to represent minimal ACLs. Clusters that use ACLs then use additional memory as the number of distinct ACLs increases. If a cluster exceeds the limit of 64511

distinct ACLs, then (and only then) the INode ACL Map is initialized lazily causing another increase in memory utilization. Thus, the design achieves the goal of minimizing memory footprint for clusters that do not use ACLs, and additionally seeks to optimize utilization for clusters that use a reasonably small number of distinct ACLs.

As an additional optimization, we can place the most popular ACLs in the 0x4000 - 0xFFFE range and place the least popular ACLs in the INode ACL Map. This is a runtime optimization that seeks to read more frequently by array indexing instead of the INode ACL Map. For example, we count the number of references of each ACL and sort them in descending order. The first 64511 ACLs are stored in the Global ACL array and the remaining ACLs are stored INode ACL map. This optimization is performed only at Namenode startup. Once the Global ACL array is initialized, it won't be updated when the reference counts are changed. It will be changed in the next Namenode startup.

In the implementation, both the Global ACL Set and the INode ACL Map will be encapsulated behind a single class, the AclManager, which is responsible for maintaining the invariants between the data structures. This also encourages separation of concerns, so that the data structures can be tuned later with minimal impact to the NameNode code that uses them. (For example, the AclManager could choose to delete the INode ACL Map if a file system reduces its number of distinct ACLs under the 64511 threshold.) The development plan intends to begin with a simplified AclManager that uses the INode ACL Map for all inodes with an ACL. This will enable writing functional tests. Afterwards, we can implement the space and time optimizations with the benefit of the test suite to catch regressions.

Disk Persistence

One new opcode is added to the edit log format: OP_SET_ACL. This opcode fully replaces the ACL of a single inode. Some file system operations partially mutate the ACL of an inode (e.g. modifying a specific ACL entry and leaving all other ACL entries unchanged). For these operations, the NameNode will merge the existing ACL with the requested modifications and encode the full result into an OP_SET_ACL. This prevents the need for adding multiple new opcodes to perform partial modifications. For simplicity, the full state of the ACL is serialized in every OP_SET_ACL. This is not expected to be a significant storage cost in proportion to the existing edit log format. Conceptually, each OP_SET_ACL is a tuple of (op code, path, ACL), as shown in the following examples. The actual implementation will encode this information in binary format, following established conventions for edit log serialization.

```
OP_SET_ACL /f1 user::rwx,user:bruce:rw-,group::r--,others::---
OP_SET_ACL /f2 user::rwx,user:diana:rw-,group::r--,group:sales:rw-,others::---
...
```

To support default ACLs, the existing OP_ADD and OP_MKDIR are enhanced so that they can optionally encode an ACL at time of creation of a new inode.

In the alternative implementation, there would also be a requirement to persist the INode ACL Map as a new section of the fsimage, located after all current sections (snapshot manager state, inodes, secret manager state, and cache manager state). Here also, the full state of the ACL is serialized for every INode ACL Map entry. Conceptually, persistence of the INode ACL Map to fsimage can be thought of as serialization of a table, where the key is the inode ID and the value is an ACL. This is shown in the following examples. The actual implementation will encode this information in a binary format, following established conventions for fsimage serialization. The length of the table is encoded as the first field in this section of the fsimage. (As a reminder, the INode ACL Map is not relevant to the current feature-based implementation, but is mentioned here in case the alternative becomes worthwhile to pursue in the future.)

```
1023 user::rwx,user:bruce:rw-,group::r--,others::---
11523 user::rwx,user:diana:rw-,group::r--,group:sales:rw-,others::---
...
```

Note that the Global ACL Set is not persisted to the edit log or fsimage. This is soft state that the NameNode reconstructs in memory when it loads fsimage. The fsimage contains a map of inode to ACL. The values of that map are full copies of the ACL. Loading the fsimage into memory aggregates distinct ACLs to form the Global ACL Set. Applying edits and subsequent file system operations cause incremental mutations in this data structure. In theory, persisting the Global ACL Set would enable storage optimizations, such that edits and fsimage would not need to store the full ACL state for multiple operations or inodes. However, this is likely to be premature optimization, and it could be done later as an enhancement if realistic usage patterns justify it.

Change History

2013-12-02:

- Initial revision.

2013-12-27:

- Updated text in UC4 describing Inherited-Default-ACL.
- Requirements section updated to describe who can set and get the ACL of a file.
- Persistence section updated to describe use of a new inode feature for in-memory representation of ACLs. Information on the prior design is still here, described as an alternative that was considered and rejected.
- All APIs (FileSystem, libHDFS and WebHDFS) updated as per discussion on issue HADOOP-10186.

2014-02-07:

- Add Sanjay Radia as co-author.
- Update UC4 with the conclusion that we'll follow the POSIX model for default ACLs.
- Add graphs describing memory utilization and discuss potential optimizations.
- Add statement that OP_ADD and OP_MKDIR are enhanced to encode an optional ACL

to support creation of new files and directories that are receiving a copy of a default ACL.

References

Apache Software Foundation (2013). Implementation of ACLs in HDFS. https://issues.apache.org/jira/browse/HDFS-4685

Barkley, J. (1994). POSIX Security Interfaces - Discretionary Access Control - Access Control Lists. http://ftp.sunet.se/pub/security/docs/nistpubs/800-7/node27.html#SECTION05142000000000000000

Barkley, J. (1994). POSIX Security Interfaces - Protection and Control Utilities - Access Control Lists. http://ftp.sunet.se/pub/security/docs/nistpubs/800-7/node42.html#SECTION05181000000000000000

Beame, C., Callaghan, B., Eisler, M., Noveck, D., Robinson, D., & Thurlow, R. (2003). Network File System (NFS) version 4 Protocol. http://www.ietf.org/rfc/rfc3530.txt

Eriksen, M., & Fields, J. (2005). Mapping Between NFSv4 and Posix Draft ACLs. http://tools.ietf.org/html/draft-ietf-nfsv4-acl-mapping-03

Ferraiolo, D., Hu, V., Kuhn, R., Miller, R., Sandlin, K., Scarfone, K., & Schnitzer, A. (2013). Guide to Attribute Based Access Control (ABAC) Definition and Considerations. http://csrc.nist.gov/publications/drafts/800-162/sp800_162_draft.pdf

Gruenbacher, A. getfacl(1) - Linux man page. http://linux.die.net/man/1/getfacl

Gruenbacher, A. (2005). NFSACL protocol extension for NFSv3. http://lwn.net/Articles/120338/

Gruenbacher, A. (2003). POSIX Access Control Lists on Linux. http://users.suse.com/~agruen/acl/linux-acls/online/

Gruenbacher, A. setfacl(1) - Linux man page. http://linux.die.net/man/1/setfacl

Peters, M. (2004). POSIX ACLs in Linux. https://www.cs.unc.edu/cms/help/help-articles/posix-acls-in-linux

Satyanarayanan, M. (1989). Integrating Security in a Large Distributed System. http://www.cs.cmu.edu/~./coda/docdir/sec89.pdf

Wikipedia contributors (2013). File system permissions - Traditional Unix permissions http://en.wikipedia.org/wiki/File_system_permissions#Traditional_Unix_permissions