High Level Design of Remote ACL

Peter Braam, Eric Mei

Feb 22, 2005

1 Requirements

- Support ACL-based permission check on remote client.
- Support setacl/getacl on remote client.

The main issue need to be addressed here is that UID/GID on remote client no longer match which on MDS's, while usually ACL entries as well as permission checks are uid/gid based.

There's an important premise. On MDS we'll have locally mounted lustre filesystem, which is full functional lustre client with the exactly same view as other normal client. This will greatly simplify setacl/getacl from remote clients. This is another project, and in thie HLD we suppose it's already in place. So this also means the remote ACL project is depend on that project.

2 Functional Specification

2.1 Cached permission on remote client

The basic idea is: The actual ACL-based permission for each remote user is performed by MDS, and we cache the result on client.

On each inode on remote client, we'll have a flag to indicate whether the master inode on the MDS contains ACL. If the flag is on, we'll maintain a list of cached permission for each user, which is actually another form of ACL. Each entry in the list describes that which user have what permission on this inode.

Permission check on cached inode on client will be actually checking the cached permission. If there's no cached permission for affected user, an special rpc will be issued to fetch the permission from MDS. For those inode which has no ACL we'll use uid/gid to check the permission.

2.2 Permission lookup RPC

As above described, it's not possible to retrive permission for all users in one place. Instead we must fetch it from MDS one by one during run time, which is accomplished by a newly invented RPC 'MDS_ACCESS_CHECK'.

On a remote client, a user access an inode will result to permission check for this user against affected inode. We firstly check whether this inode contains ACL. If yes, we check whether we have cached permission for this user. If failed to find one, an RPC 'MDS_ACCESS_CHECK' will be issued to fetch the missing entries from a MDS. The reply from MDS will contains the read, write, and execute permission bits for this user against this inode. This permission will be cached at client, for further reference. From then on all permission checking will based on the cached entries.

We can optimize the permission retrive a little bit. When a client begin to populate an inode, an RPC which result to pass inode attributes to client, e.g. getattr, open, chdir, etc., will be fired to MDS. MDS will pack permission of current user into reply, if applicable. So if the same user access this inode some time later, we don't need further 'MDS_ACCESS_CHECK' RPCs.

2.3 Get/Set ACLs at remote client

We'll use the exactly same user interface as standard getfacl/setfacl tools. And actually we could modify based on source code of getfacl/setfacl. Please refer to manual pages of getfacl/setfacl for details.

There is one note: All the user/group name, printed out by getfacl or supplied to setfacl, is actually refers to name/group on MDS's user database, not user database of the remote client. This require the user of getfacl/setfacl have knowledge of MDS's user database, to pervent insane ACL manipulation.

Although the user interface is the same, the modified getfacl/serfacl internally do things very differently. It will not perform any ACL internal data structure mangling, or name/id translation. Instead it simply pass the command line and the destinate file name to kernel, and kernel will return the human readable text result, which is finally output to user.

The client kernel also did very simple things: pass the command to MDS without further translation. The reply from MDS should be human readable text result, directly deliver it back to user space.

2.4 MDS handle getfacl/setfacl request

MDS will invoke a user space helper, say "lacl_upcall", to perform most of the job. For both getfacl/setfacl cases, MDS kernel will start the lacl_upcall via upcall, feed with target file name, and the command line passed in from remote client. Then lacl_upcall will act for the correct user, issue the same getfacl/setfacl command against the file in the MDS's locally mounted lustre filesystem. When the lcal_upcall end, the actual getacl/setacl should have been finished. The output, which is human readable text messages, will be pass down to MDS kernel, and in turn be returned to client.

Actually it could be treated as a local lustre client perform the exactly same ACL command for a remote lustre client.

3 Use Cases

3.1 Permission check case 1

1. Alice on remote client access file1 which has no ACL at the first time.
2. Lustre client issue GETATTR rpc to MDS.
3. MDS return inode attributes.
4. Bob on same client try to access file1.
5. Lustre check permission only based on Bob's uid and gid, along with file1's owner and group.

3.2 Permission check case 2

1. Alice on remote client access file1 which has ACL at the first time.
2. Lustre client issue GETATTR rpc to MDS.
3. MDS return inode attributes, and indicate this inode contains ACL, and the permission of Alice on this file.
4. Lustre client cached Alice's permission in inode, and grant/deny Alice's access based on it.
5. Alice's access right on file1 will be determined by the cached permission.
6. Bob on same client try to access file1.
7. Lustre client failed to find the cached permission for Bob, then issue 'MDS_ACCESS_CHECK' rpc to MDS.
8. MDS return Bob's permission on file1.
9. Lustre client cached Bob's permission in inode, and grant/deny Bob's access based on it.
10. Alice on this client try to access file1 again.
11. Lustre client will grant/deny Alice's access based on cached permission.
12. Somebody else changed permission file1; Or somebody login/logout lustre on this client;
13. DLM lock canceling which initiated by above chmod will result in client to release all cached permissions.
14. Alice try to access file1 again. Inode attributes and permissions will be re-populated as above procedure again.

3.3 Listing a directory

1. Alice on remote client listing a directory by 'ls -l'.
2. Lustre client will issue GETATTR for each file within the directory. And Alice's permission for all those files will also be sent back by MDS's reply, and cached.
3. Alice do 'ls -l' again, no rpc will be issued.
4. Bob do 'ls -l', again no rpc will be issued.
5. Bob try to access one of those files which contains ACL, e.g. do open().
6. Lustre client will issue MDS_ACCESS_CHECK rpc to MDS.

3.4 getfacl

1. Alice on remote client run 'getfacl file1'.
2. Lustre client send request to MDS.
3. MDS kernel invoke "lacl_upcall getfacl file1", via upcall mechanism.
4. lacl_upcall will do standard "getfacl file1", pass the text output "u:alice:rwx,u:bob:r-x" back to kernel, and exit.
5. MDS kernel got the output, and simply return back to client.
6. Lustre client got the reply, return back to user space.
7. Alice's getfacl will output result "u:alice:rwx,u:bob:r-x" and exit.

3.5 setfacl

1. Alice on remote client run "setfacl -m u:alice:rwx,u:bob:rw- file1".
2. Lustre client send request to MDS.
3. MDS kernel invoke "lacl_upcall setfacl file1 -m u:alice:rwx,u:bob:rw- file1", via upcall mechanism.
4. lacl_upcall will do standard "setfacl file1 -m u:alice:rwx,u:bob:rw- file1", pass the success or fail result back to kernel, and exit.
5. MDS kernel got the output, and simply return back to client.
6. Lustre client got the reply, return back to user space.
7. Alice's setfacl will output result (if any), and exit with code indicated by kernel.

4 Logic Specification

4.1 Cached permissions

Each client inode, if has ACL entries on master inode, could have list of cached permission. Each permission entry would roughly looks like:

```
struct remote_perm_setxid {
    struct list_head list; /* permission list */
    uid_t          uid;
    gid_t          gid;
    uint16_t       perm;
};
struct lustre_remote_perm {
    pag_t          pag;
    uid_t          auth_uid; /* authenticated uid */
    gid_t          auth_gid; /* authenticated gid */
    uint16_t       perm;     /* permission bits */
    uint16_t       allow_setuid:1,
                   allow_setgid:1;
    struct remote_perm_setxid *perm_setxid;
};
```

This permission cache should be careful to conformed to which on MDS. The LSD on MDS might selectively allow some users setuid/setgid, while block this permission for others. So we record the uid/gid which used at authentication time. If setuid/setgid ever be executed, we need check whether allow setuid/setgid, and find in cached setxid permission list. The whole procedure will roughly like:

```
if (has_no_acl(inode))
  check based on uid/gid;
  return result;
gss_ctxt = pag_to_ctx(current->pag);
if (!is_valid(gss_ctxt ))
  issue_rpc(MDS_ACCESS_CHECK);
  return result;
for_each_in_cached_permission(pag)
  if (pag_matched)
    if (perm->auth_uid == current->fsuid &&
        perm->auth_gid == current->fsgid)
      return perm->perm;
    if (!allow_setuid && setuid ||
        !allow_setgid && setgid)
      return DENY;
    if (found_in_setxid_perms)
      return result;
```

```
else
  issue_rpc(MDS_ACCESS_CHECK);
  return result;
```

Note in all cases, no supplementary groups will be involved into permission check.

One optimization could be done is: Embed one permission (e.g. the last accessed one) into ll_inode_info, which has the biggest chance to be referenced.

4.2 MDS_ACCESS_CHECK RPC

This rpc should be quite simple, just with one note. During permission checking, client will hold inodebits DLM of affected inode, thus keep holding it during MDS_ACCESS_CHECK rpc. So mds handler of MDS_ACCESS_CHECK should avoid obtaining any DLM lock to prevent deadlock. Just find the inode by fid, lock local inode, perform access test and reply the result.

On MDS side, the permission check should go through all the normal procedure: uid/gid translation, checking against LSD, etc. Just like any other normal request handling, except did nothing but return the permission result.

4.3 Client support getfacl/setfacl

We can take code from standard getfacl/setfacl, or patch against them, only keep the parameter parsing part, to prevent the wrong format, etc. After make sure the parameters are valid, getfacl/setfacl call standard syscall "getxattr()/setxattr()" against the directory of the target file. The extended attribute name could be a special one, like "system.lustre_remote_acl", kernel will recognize it; The parameter include the target file name, along with the parameters user supplied. For example, user invoke:

```
setxattr dir1/dir2/file1 -m u:alice:rw-
```

And setxattr will finally invoke syscall:

```
char *param = "file1 -m u:alice:rw-"
int   size  = strlen(param);
sys_setxattr("dir1/dir2",
             "system.lustre_remote_acl",
             param, size, flag);
```

Client kernel handler for getxattr/setxattr recognize the special attribute name, and send the FID of the parent dir, here is FID of "dir1/dir2", and the "param" to MDS. The reply from MDS will contains:

- Error code. 0 for success.
- Text output. For getacl it's the ACL list, for setacl it might be nothing. Or just error message.

Client kernel just pass the reply directly back to user space. The progarm getfacl/setfacl will output the text message return from syscall, and return the code as indicated by kernel.

4.4 MDS handle getfacl/setfacl

MDS kernel got the request, find the parent directory from the FID sent from client, and check whether current user execute permission on the directory. If not, just deny the request. After that, we'll set the directory as "current work directory", and invoke lacl_upcall to user space via upcall, with the parameters sent from client. So when lacl_upcall running, it's already in the right directory.

lacl_upcall could simply invoke standard getacl/setacl with the exactly same parameters, which will interact with locally mounted lustre filesystem on MDS. lacl_upcall will get the output and exit code, and pass to kernel via /proc or any other proper mechanism.

After got result from upcall, MDS kernel just send the output and error code back to client.

5 State Management

The cached permission will also be treated as a normal attribute of inode. Every request issued by md_intent_lock() from client which intend to obtain inode attributes, i.e. IT_LOOKUP, IT_GETATTR, and IT_CHDIR, will also be replied the permission for this user, and then cached at client inode. They are protected by inodebits DLM lock, just like other inode attributions. In this point the state management of cached permission is nothing special.

A new rpc MDS_ACCESS_CHECK is added, which is a simple, non-transactional rpc, so no special recovery consideration is needed. For the getfacl/setfacl, they use the existing interface, and no DLM lock will be involved at client side.

No disk format changes.

6 Alternatives

NFSv4 support two kinds of setacl/setacl interface:

1. POSIX ACL: uid/gid is passed across kernel boundary. Tools like setfacl will resolve user name into uid, passed to kernel, kernel then map uid to NFSv4 domain name, via idmap daemon, then send text name to server; Server again reverse map domain name back to uid via idmap daemon.

2. NFSv4 ACL: User tools will resolve name to NFSv4 domain name, directly pass text name into kernel.

7 Focus of Inspection

- Are the client side cache and logic reasonable?
- Any security hole there?