Lustre Capability DLD

Lai Siyao

7th Jun 2005

OSS Capability

1 Functional specification

OSS capabilities are generated by MDS, sent to client when client opens/truncate a file, and is then included in each request from client to OSS to authorize an action.

In case that the client might modify the capability obtained from MDS, the capability is signed with HMAC.

1.1 new data types

1.1.1 struct lustre_capa

```
struct lustre_capa {
        __u32   lc_uid;    /* uid, mapped uid */
        __u32   lc_op;     /* operations allowed */
        __u32   lc_mdsid;  /* mds# */
        __u32   lc_igen;   /* inode generation */
        __u64   lc_ino;    /* inode# */
        __u32   lc_ruid;   /* remote uid on client */
        __u32   lc_flags;  /* security features for capability */
        __u64   lc_expiry; /* expiry time (sec): servers have clocks */        __u3
        __u8    lc_hmac[CAPA_DIGEST_SIZE]; /* HMAC */
} __attribute__((packed));
```

lc_ruid remote uid on client, this will only be used on client, both MDS and OSS will use lc_uid.

lc_op CAPA_READ/CAPA_WRITE/CAPA_TRUNC.

lc_expiry capability expiry. And similar to kerberos, when capability is enabled all nodes on the system are supposed to have the almost synchronized time.

lc_flags only one flag now: CAPA_FL_SHORT_EXPIRY, which is set when capability timeout value is less than CAPA_EXPIRY(1024 sec). And this flag will be used in client capability renewal.

lc_hmac is the HMAC for all fields above lc_hmac.

1.1.2 struct lustre_capa_key

```
struct lustre_capa_key {
        __u32   lk_mdsid;      /* mds# */
        __u32   lk_keyid;      /* key# */
        __u32   lk_expiry;     /* expiry */
        __u32   lk_key[CAPA_KEY_LEN];  /* key */
};
```

1.1.3 struct obd_capa

```
struct obd_capa {
        struct list_head        c_list;
        struct lustre_capa      c_capa;
        int                     c_type;
        atomic_t                c_refc;   /* ref count */
        unsigned long           c_expiry; /* jiffies */
        union {
                struct client_capa cli;
                struct target_capa tgt;
        } u;
};
struct client_capa {
        struct inode          *inode;
        struct lustre_handle  file_handle; /* mds_file_data handle */
        struct list_head      lli_list;    /* link to inode */
        atomic_t              open_count;  /* capa open count */
};
struct target_capa {
        struct hlist_node     hash;
}
```

struct obd_capa is used to manage capabilities cache on different OBDs: client, MDS and OSS.

Capabilities in MDS and OSS are cached to avoid signing penalty. On MDS and OSS the capability cache size is fixed: 3000. Via ***u.hash*** capabilities are hashed, it is used for lookup. And ***c_list*** links capability on MDS and OSS to LRU lists.

Client caches capabilities because of page cache: client doesn't know when pages in client page cache will be obtained from/flushed to OSS, so all capabilities are cached until they expire. Compared to MDS and OSS capabilities on client are not hashed, but they will be linked on ***ll_inode_info.lli_capas*** via ***lli_list*** for lookup, and increase ***open_count*** once open, decrease when close. The ***handle*** field in struct client_capa contains the handle to struct mds_file_data on MDS, which will be used to verify access permission while renewing capability.

OSS might have two available capability keys at a certain time, the latest one is called red key, and the old one black key. And clients might use either of them.

1.1.4 struct mds_capa_key

```
struct mds_capa_key {
        struct list_head         k_list;

        struct lustre_capa_key   k_key;
        struct obd_device       *k_obd;
        unsigned long            k_expiry;       /* jiffies */
};
```

This struct is used to update capability key periodically on MDS, and the update key will be propogated to all OSS's.

2 Use Case

2.1 read file

client ll_lookup_it(), which calls mdc_enqueue() with IT_OPEN.

MDS mds_open() will handle this intent request, it will pack the signed capability in reply message.

client capability is unpacked and updated locally.

client OST_READ finally will call osc_build_req(). In osc_brw_prep_request() packed the proper capability in request to OSS.

OSS ost_brw_read will handle this request, it will unpack and verify the capability, if valid, IO will go on, else this request is rejected.

client will close the open file and update capability.

2.2 client renew capability

llite find the capability to renew, call mdc_getattr to renew the capability.

MDS will receive the request, then check the access mode of this capability based on mds_file_data, if it's ok, update the capability and send back.

llite update capability.

2.3 MDS update capability key

MDS the capability key is to expire, and the mds_capa_key_timer_callback is triggered. A new capability key is generated and the key id is increased, and the new key is propogated to all OSS' through obd_set_info.

OSS will receives the new capability key, and then updated the capability key list in memory.

MDS obd_set_info returns successfully, then it will store the new key in its capability key file, from then on the new key will be used to sign capability.

3 Logical Specifications

3.1 client side

3.1.1 obtaining capability

Upon lookup finishes, the capability for the opened file is packed in the reply. In ***ll_update_inode***() this capability will be linked into ***ll_inode_info.lli_capas***:

```
struct ll_inode_info {
    ....
    struct list_head      lli_capas;
}
```

And in ***ll_file_open()***, the open count and file handle (mds opened file handle) of this capability are updated. Accordingly, the open count will be dereferenced in close. The open count and open client handle here are used by capability renewal (see below).

3.1.2 renewing capability

Client will renew capabilities whose open count are larger than 0 with MDS when they are close to expiring. The renewal request is prepared by a thread ll_capa_thread, and then handed to ptlrpcd to send asynchronously. A new timer ***ll_capa_timer*** is added to track renewal. In this request the file handle to ***struct mds_file_data*** along with the capability will be packed. And all client capabilities are in a sorted list to help find the capabilities to renew.

Obd function getattr will be used to renew capability:

```
    int md_getattr(struct obd_export *exp, struct lustre_id *id,
```

One issue here: how to ensure the capability doesn't expire before the page is flushed to OSS?

This is achieved by renewing capability much earlier than it expires, that is, the delta time pre-expire should be larger than dirty_expire_centisecs (The longest number of centiseconds for which data is allowed to remain dirty, the default value is 30 * 100, that is 30 sec). And the default pre-expire delta time for capability is CAPA_PRE_EXPIRY (300 sec).

3.2 MDS side

3.2.1 capability HMAC

The HMAC value of capability will be calculated by kernel function ***crypto_ hmac()***, and by default the crypto algorithm is SHA1. For MDS and OBDFILTER the crypto will be initialize during module setup.

NOTE: in ***crypto_ hmac()***, struct crypto_tfm is not thread-safe.

3.2.2 packing capability

```
    int mds_pack_capa(struct obd_device *obd, struct lustre_capa *capa,
```

mds_ pack_ capa will generate the capability for the specified user and operation on specified inode if it's not found in hash, otherwise it just packs the capability found in the reply message. It will be called in three places:

1. upon mds_open, a capability is sent back.
2. client renew capability with md_getattr.
3. upon truncate client will setattr on MDS, if ATTR_CAPA is set a CAPA_TRUNC capability is packed in the reply.

3.2.3 capability hash

Capabilities are hashed on MDS' and OSS', the size of hash is fixed (3000). The capability hash code should be put in obdclass. There should be reference count for capability, and a hash lock will protect the capability hash and list.

MDS' and OSS' capabilities are in a LRU list: the most unused capabilities will be released when generating new capability but the capability count has exceeded 3000.

3.2.4 permission check for capabilities to renew

When MDS receives capability renewal request, ***capa.lc_ op*** will be checked aginst ***mds_ file_ data*** the file handle pointing to, because the permission check is based on original open, not current access mode.

3.3 IO with capabilities

3.3.1 direct IO

This is in 2.4 only: ll_direct_IO_24. A capability with uid: current->fsuid and specified opc will be looked up in capability list of this inode, if found, this capability will be used for this direct IO.

3.3.2 synchronous read in partial write

This is in ll_preare_write(), and it will then call ll_brw. Just like above, a proper capability is looked up for it.

3.3.3 synchronous/asynchronous IO

Both synchronous and asynchronous IO requests are packed in ***osc_ build_ req***, however there are two issues here:

- all the pages in one request might not belong to one fsuid, but only one capability for one request.
- there isn't a clean way to obtain the fsuid for the mmaped pages.

The solution is: the latest capability with the correct opc for this inode will be used in the request. (a security flaw?)

3.3.4 truncate

For truncate, client will send a setattr rpc to MDS with attr->ia_valid set with ATTR_MTIME | ATTR_CTIME | ATTR_CAPA, mds will pack a truncate capability in the reply in case of ATTR_CAPA.

After punch on all oss, this truncate capability will be cleared right after. (no concurrent truncate in vfs, so it's safe)

3.4 OSS side

After OSS receives the read/write/truncate request, it will first verify if the capability is valid, which is achieved by verifying HMAC associated with the capability. And also the capability content (opc and fid).

3.5 capability key

3.5.1 capability key update

MDS will renew its capablity key with all OSS' periodically. A timer ***mds_eck_timer*** is added to track this. And the ***obd_set_info*** will be used to propagate new capability key to OSS'. This rpc should be replayable.

3.5.2 capability key file on disk

Capability keys are stored in disk file on MDS only, and this key file will contain two keys: red and black key.

3.6 OST authorization revocation

This can't be supported by current implementation.

4 Recovery

* MDS' will propagate capability keys to OSS' during setup, and all obdfilters will setup capability keys list based on them.

* In case of connection failure between MDS and OSS, and the MDS is alive, it will be mds_notify() when the connection recovers, in mds_dt_synchronize() the capability keys should be sent to all OSS as above.

* Generate capability in open resend case. (in reconstruct_open).

5 Test

Since by default the capability and its key expiry might be fairly long, to help test, the capability and key timeout should be set through proc.

5.1 basic operations

read/write/truncate.

5.2 enable/disable capability

5.3 capability renewal and capability key update

5.4 two users access the same file concurrently

5.5 dbench

5.6 MDS recovery of capability

5.7 OSS recovery of capability

MDS Capability

6 FUNCTIONAL SPECIFICATION

MDS capability, in another words, could be called fid capability, which proves a client has access to a fid. The fid capability is obtained in lookup request, and each request to MDS which concerns fid operation will pack it the request.

The lustre_capa and obd_capa struct defined above will be reused here.

7 USE CASE

7.1 obtain fid capability with open

client find fid capability for the parent dir.

client if the fid capability has expired, renew it.

client pack parent fid capability in lookup (with intent IT_OPEN) request.

MDS verify fid capability for parent dir.

MDS open file, and pack the fid capability for this inode in the reply.

client store this fid capability for future use.

8 LOGIC SPECIFICATION

8.1 client fid capability

Each inode has a single fid capability, once it's obtained, it's stored in ***ll_inode_info.lli_fid_capa***. And the following requests will lookup fid capability from here.

There are two places where client fid capabilities get freed:

- ll_clear_inode(): where inode is cleared.
- ll_mdc_blocking_ast(): when MDS_INODELOCK_LOOKUP lock is canceld.

8.2 fid capability renewal

Similar to OSS capability, fid capability will expire after its timeout. To renew fid capability, current fid capability is sent to MDS via MDS_GETATTR request (the same as OSS capability renewal), and then MDS will check the validity of this fid capability, if it's ok, update the expiry of this capability and generate new HMAC, and then reply back to client.

But unlike OSS capability, fid capabilities are not renewed by a separate thread, instead they will be renewed synchronously before sending the request to MDS. By this way we could avoid the overhead of renewing unused fid capabilities periodically, which might be huge.

But the above policy will lead to a problem: the fid capability might become too aged that the capability HMAC key used to sign this capability is not in used in MDS any more, then if client want to renew it, but MDS will fail to validate it. To solve this, all unused fid capabilities will be renewed by a separate thread as before, but the interval will be much larger, it's the timeout of capability key. (by default, this timeout value is 1 day)

9 RECOVERY

During recovery the fid capabilities packed in resend and replay requests might have expired in some cases, but all these are handled in ptlrpc layer, and these capabilities won't have chance to get renewed. Therefore MDS only check the validity of the fid capability for resend and replay requests (HMAC is correct), but ignore the capability duration.

10 TEST

Focus on compatibility with old version.