LOV-OSC on MDS

Alex Zhuravlev
2009-11-09

Disclaimer

- it's a prototype in many aspects
- the most important aspect is to learn whether we can use OSD API for other than original purposes
  - > can this simplify things
  - > can this improve quality
- if it works for LOV/OSC, try to find more features/components which could benefit from OSD API

Two aspects of LOV-OSC

- used on MDS to
  - create OST objects
  - destroy OST objects
  - change uid/gid on OST
  - change OST object size in some cases
- used on a client to
  - access data stored on OST
- the functionalities don't overlap
  - except very few common cases

Problems

- prone to be source of many bugs
  - > especially related to recovery
- hard to read and maintain
- functionality is very scattered
  - > especially related to recovery
- double-failure cases aren't handled
  - > MDS+client failure leads to file w/o objs
- likely not CPU-efficient:
  - > interaction overhead
  - > 4 + 5*N allocations on regular file create

Not important details are exported

- with no real need
- MDT-MDD
  - is aware of OST connections
  - maintains per-OST states (last used id)
  - tracks max LOVEA size
- LOV exports LOVEA
  - just to use that data internally then
  - MDD calls obd_unpack()
  - then MDD sends LSM back to LOV

The proposal

- OSD API
  - simple
  - object based
  - support transactions
  - stackable

MDD

- MDD should be able to
  - create objects
  - destroy objects
  - change object attributes
  - ... and make any of them part of big tx

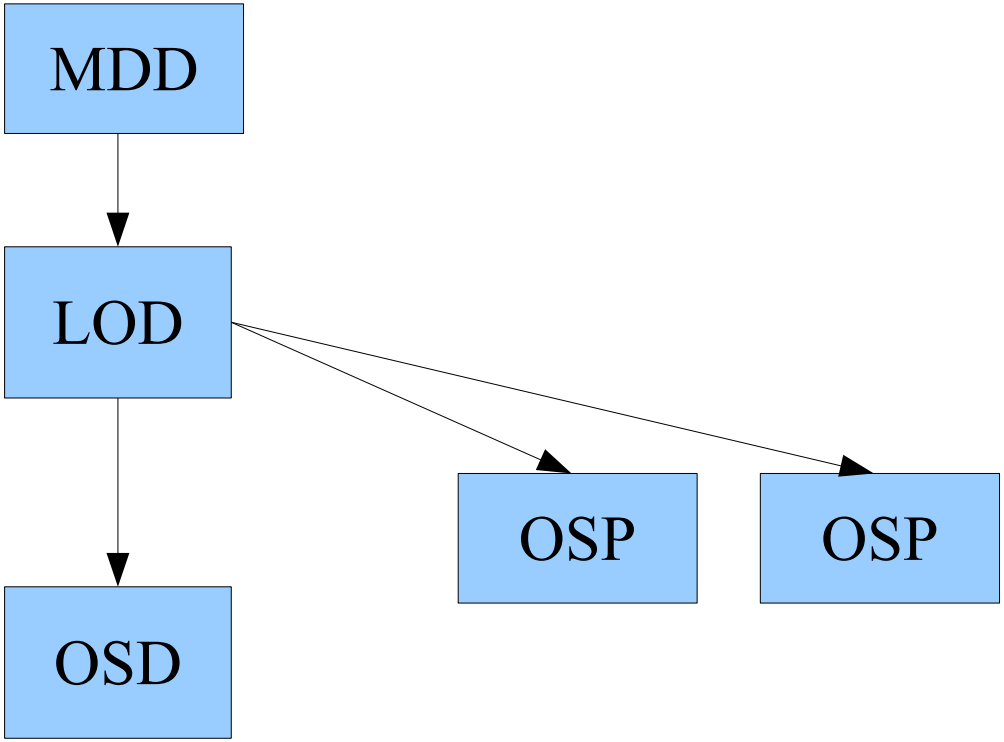
LOD: Logical Object Device

- replacement for LOV on a server
- takes care of striping
  - maintains striping on a disk
  - manipulates underlying objects
- exported via OSD API
- talks to others via OSD API
- don't initiate transactions, but can be part of transaction

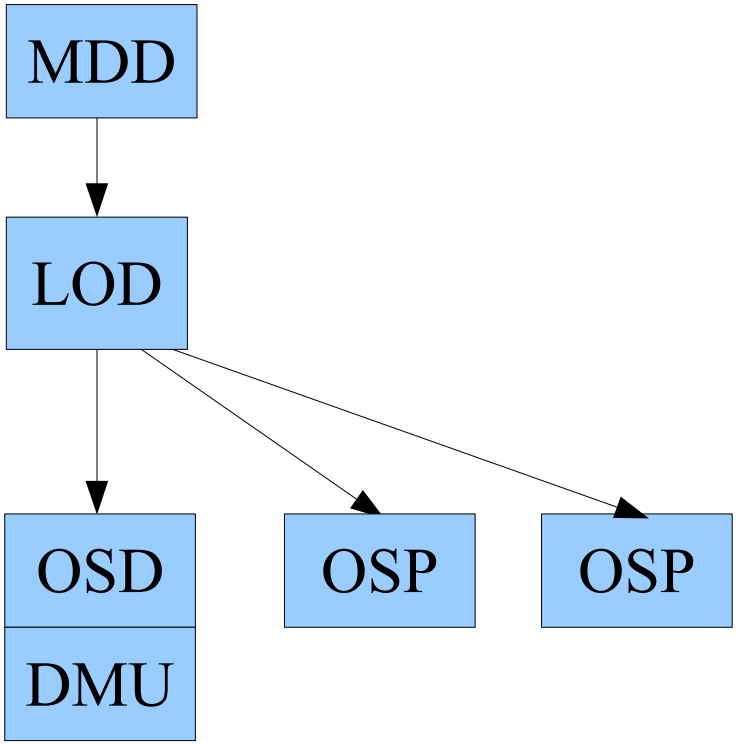
OSP: OST Proxy

- replacement for OSC on a server
- hides all precreation related logic
  - layers above just create single object
- hides everything about orphan recovery
  - MDD doesn't need to even know about last-used-id, connections, etc

The stack



Use Case: file creation



- MDD creates object with LOD
- LOD creates local object
- LOD creates stripes
- all (except MDD) are called with ->do_create() method
- details are hidden

Use Case: file creation in MDD

- asks LOD to fill an allocation hint
  - e.g., LOD can decide about details of striping from parent object
- create, declare and start transaction
- and create object with ->do_create()

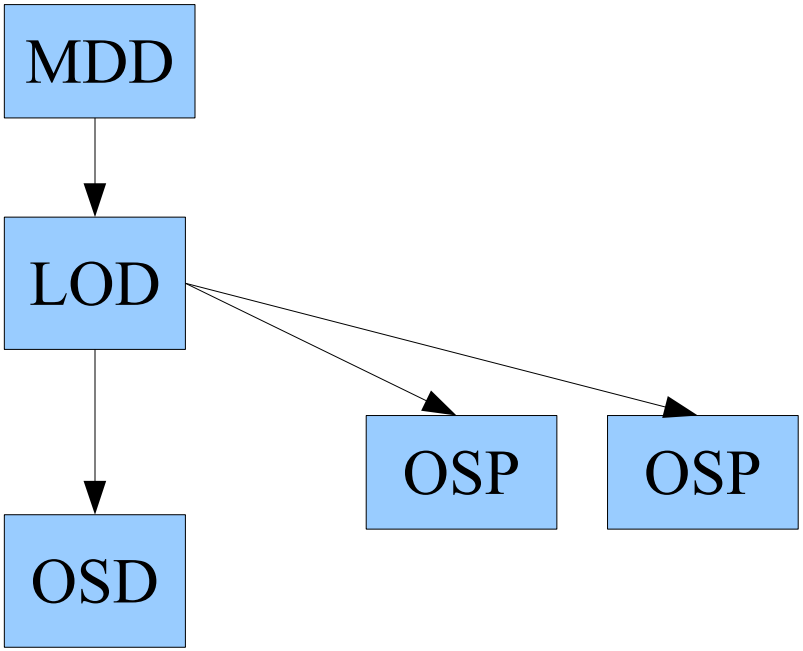
Use Case: file creation in LOD

- decides whether object will be striped
- finds OSTs for stripes
- calls OSPs to create objects on corresponded OSTs
- generates LOV EA
- store LOV EA in the local object
- MDD doesn't need to care about any of these, even about details like journal credits for LOV EA update

Use case: file creation in OSP

- OSP grabs object id from a pool
  - pool is maintained by OSP
- OSP updates last used id on a storage
  - within same transaction, automatically
- the pool is maintained asynchronously by OSP
- ->do_declare_create() reserves id
- MDD/LOD are not aware of anything this
  - they just create object(s)

Use Case: file removal



- MDD destroys object with LOD
- LOD destroys local object
- LOD destroys stripes
- all (except MDD) are called with ->do_destroy() method
- details are hidden

Use Case: file removal in MDD

- when nlink becomes 0
- and file isn't open
- MDD destoys object
- MDD doesn't care about llogs
  - nor maintaining, nor declaration
- MDD doesn't even know whether object is striped or no

Use Case: file removal in LOD

- LOD learns whether object is striped
  - loading and parsing LOV EA
- LOD calls OSP to destroy stripes
- LOD destroys local object
- doesn't care about llog/cookies, etc

Use Case: file removal in OSP

- appends llog record about object removal
- once record is committed, can send OST_DESTROY
  - would be good to support bulks
- once OST_DESTROY is confirmed by last_committed cancel llog record

Use Case: recovery in MDD

- essentially MDD does nothing about recovery
- doesn't even know about reconnection to OST, etc

Use Case: change uid/gid

- very similar to previous examples
- MDD changes attributes on the object
- LOD changes attributes on all the stripes, if needed
- OSP uses llog to track changes
- OSP sends bulks of changes to OST

Use Case: recovery in LOD

- doesn't need to know anything about recovery

Use Case: recovery in OSP

- this is where all recovery about MDS-OST is
  - loads last-used-id from disk
  - destroy orphans ...
  - ... or recreate missing objects
  - scans llogs and completes transactions

Problems: very easy on a paper only

- OSD API requires fid to be known before first access to an object
- LOV uses 3 methods in precreation:
  - obd_create(), obd_precreate(), obd_create_async()
  - can we implement the policy using less?
- many OSD API methods are sync
  - is it OK that OSP maintains statfs cache?
- create replay sends LOV EA
  - is it OK if MDT just sets it as xattr?
- more on demand ...

Interesting observation

- LOD can implement striped dirs
- CROW becomes simpler
  - in-core stripe representation can track status: deleted or not
  - OSP can order changes/RPCs
- MDT/MDD don't need to know max LOV EA size
- declaration method allow OSP to reserve object in pool, assign id later
  - see bz21186

This is not the final picture

- OSP becomes empty (not needed)
  - no need in special recovery mechanism
    - when distributed transactions are here
  - no need in precreation
    - generate fids on demand (fids on OST)
    - create objects on first access (CROW)

The outcome

- modules have very clear functionality and responsibility
- API is the same:
  - knowledge of API apply to many modules
- modules can be very simple/robust:
  - e.g. MDD
    - just translates operations into updates
    - no optimizations and tricks required
    - written once can be used for long w/o change as POSIX doesn't change often

Remember disclaimer?

- in general, OSD API can be useful if
  - there is storage (local or remote)
  - there are transactions
  - changes are single-object

What other features/components?

- remote OSD
  - let MDD talk to remote disk like it's local and get CMD + WBC + proxy
- distributed transaction manager
  - can track all updates going through OSD API to a disk (remote or local)
  - log undo-redo requiring no parsing
- quota
  - can track all quota related changes via OSD API, not ad-hoc in MDD/OFD
- mirroring
  - monitor all updates, send copy to mirror/backup