Lustre: some protocol basics

HPC Workshop, Germany, September 2009

Johann Lombardi
Lustre Group
Sun Microsystems

Topics

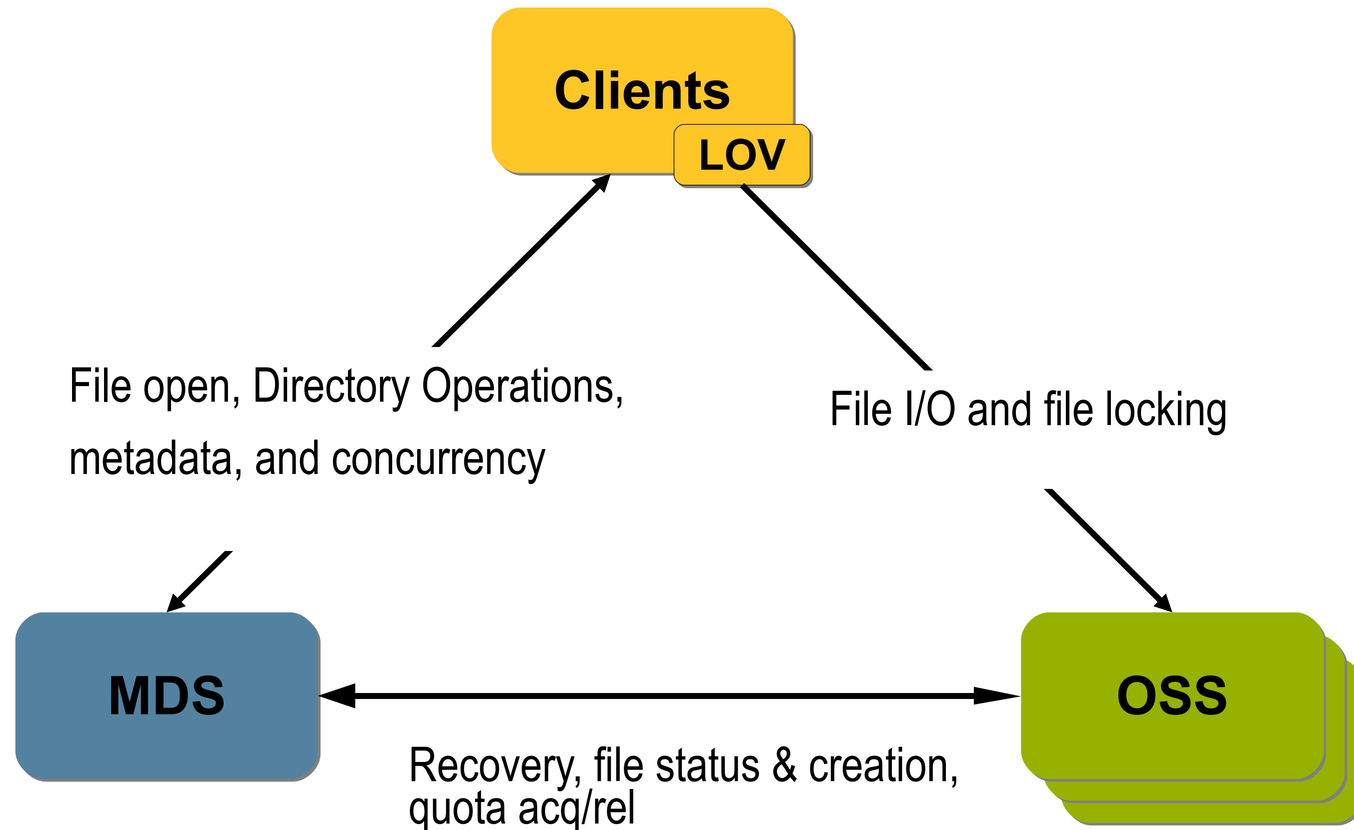
- Lustre striping
- Request lifecycle
- llog
- I/O in the OST
- ldlm

Topics

Sun microsystems

- Lustre striping
- Request lifecycle
- llog
- I/O in the OST
- ldlm

Lustre Components



Sun microsystems

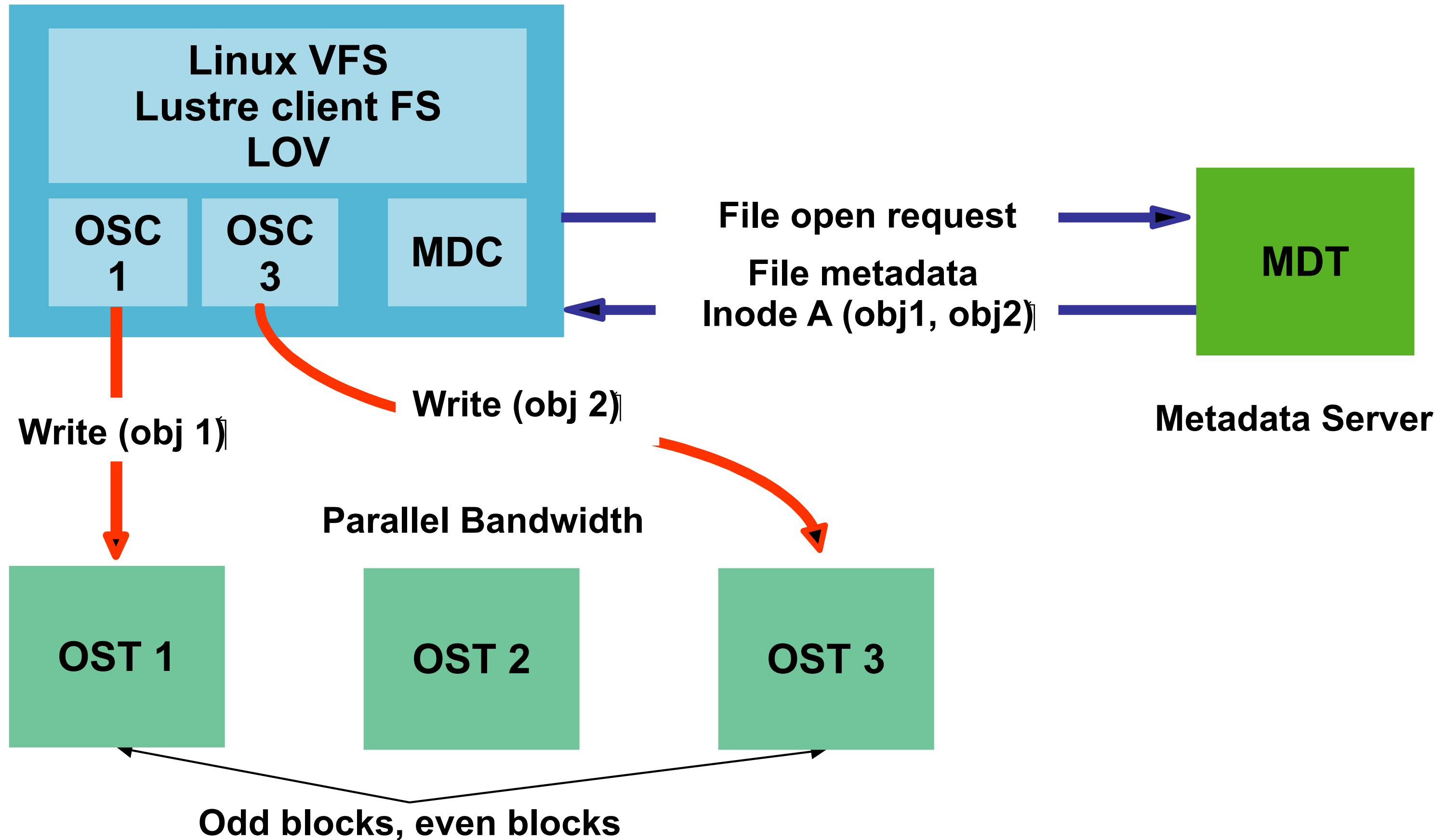
What is striping?

- Striping stores file data *evenly* in multiple places
- Lustre stripes file objects
  - RAID stripes extents over multiple block devices
  - Lustre currently only has network RAID0 striping
  - In the future there will be more Lustre network striping (mirroring)

Why stripes?

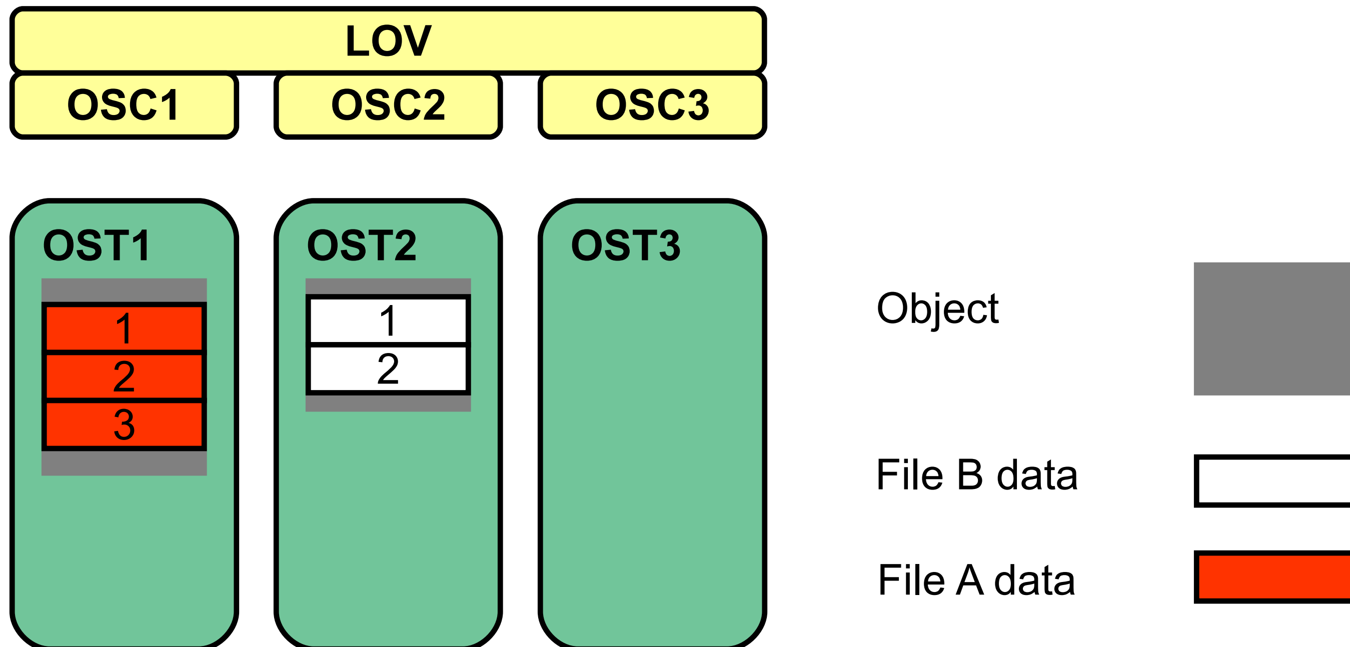
- Required aggregate bandwidth from one file exceeds OST bandwidth
- A single client may need more bandwidth than one OSS can offer
  - Somewhat rare, but it happens

File open & write



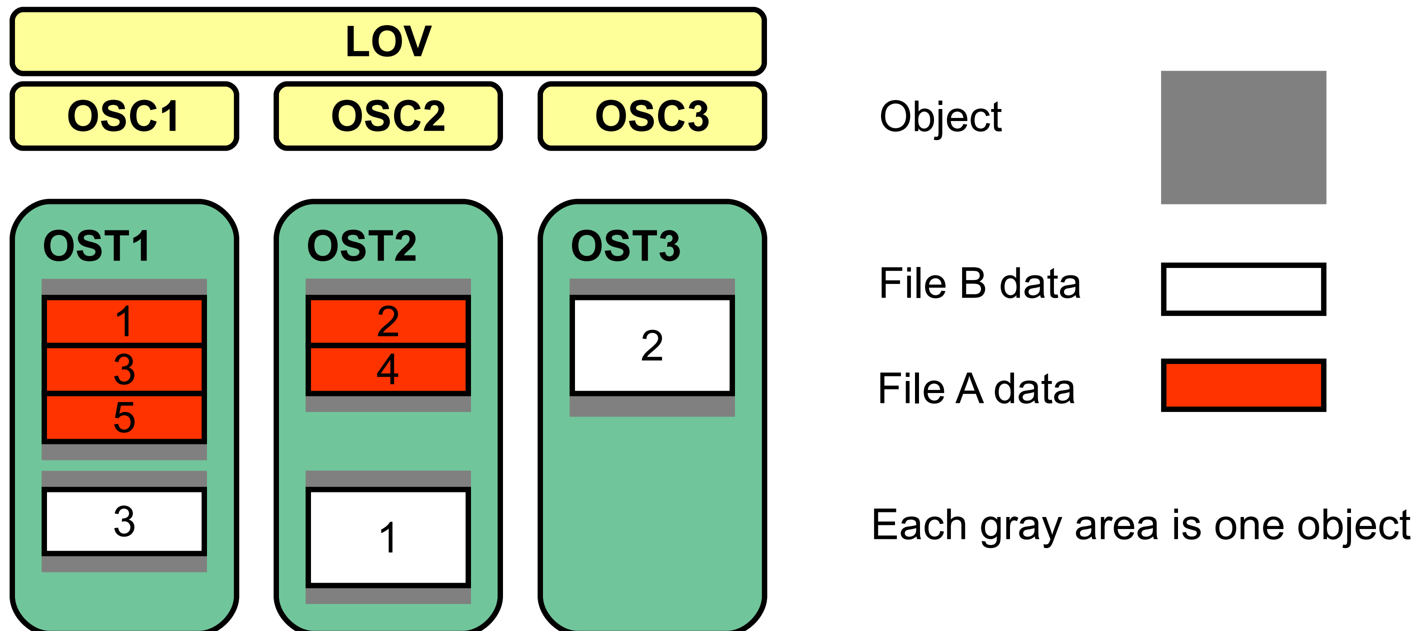
Single Stripe File

- All data for a file is in one OST
- File data is simply data in the (single) object



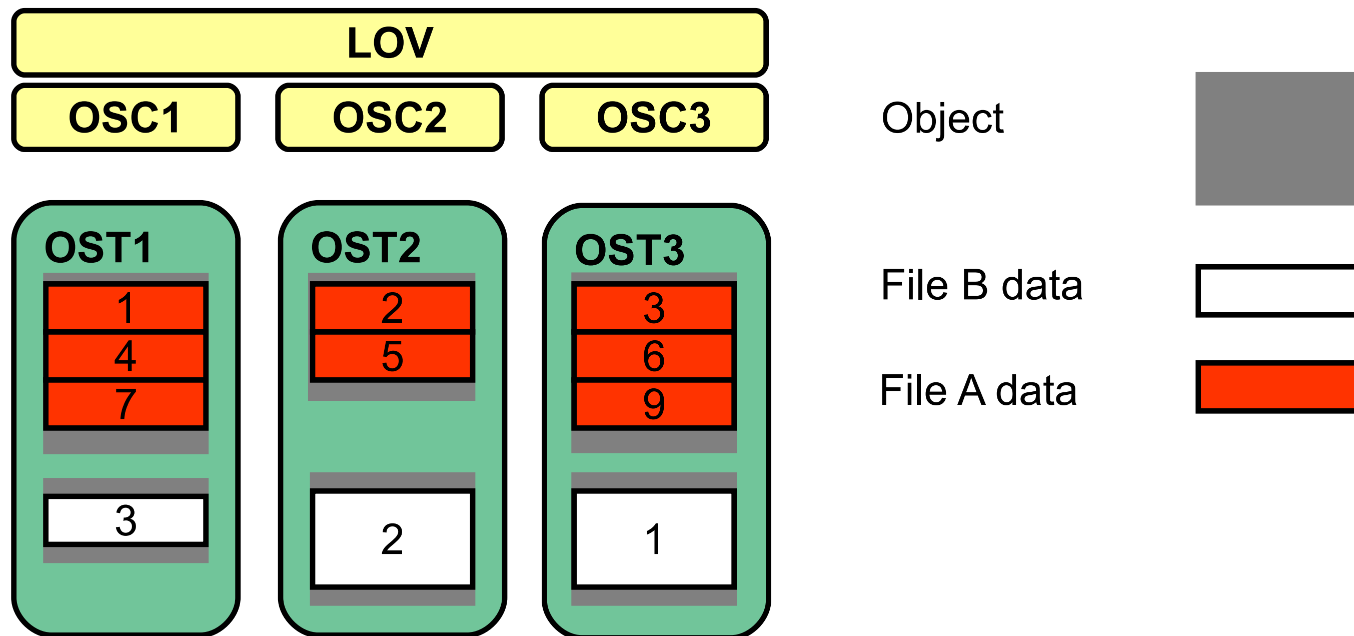
Files with multiple stripes

- Stripe count for file A is two, for file B it is three
- Stripe size is typically 4MB, stripe size of file B is twice that of file A
- Stripe object sizes add up to total file size



One stripe on each OST

- These are fully striped files
  - Note that file A appears to have a hole (sparse extent)
  - The white file is unusually striped (but this is possible)
- Typically this achieves maximum bandwidth to one file



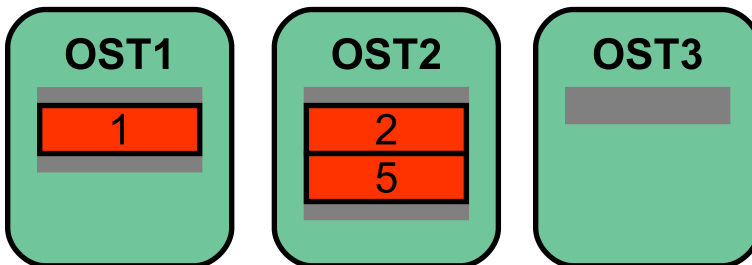
Sun microsystems

Sparse sections in files

- You can create sparse sections (hole) in 2 ways
  - > Seek + write
  - > Truncate a file to increase its size
- Filesystems and holes
  - > Good filesystems don't allocate blocks until a real write happens
  - > Reading in a hole returns 0
- With objects
  - > Holes can be in any of the objects
  - > This happens quite naturally in N:1 application
    - – All N clients running to one file

Example

- One movie file with 6 frames
- A 6 client cluster, each rendering one frame
- Not all rendering takes equally long

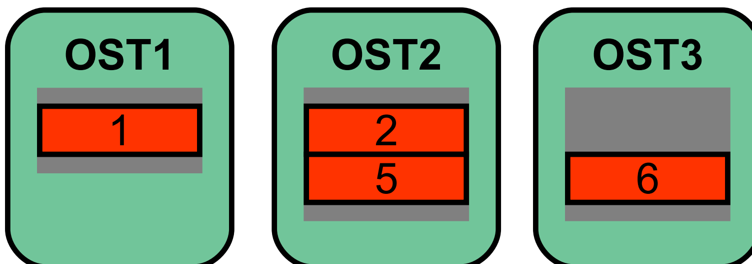


Clients 1,2,5 done

File size 5

One hole in the file (3,4)

1 empty & 1 short object



Clients 1,2,5, 6 done

File size 6

One hole in the file (3,4)

1 sparse & 1 short object

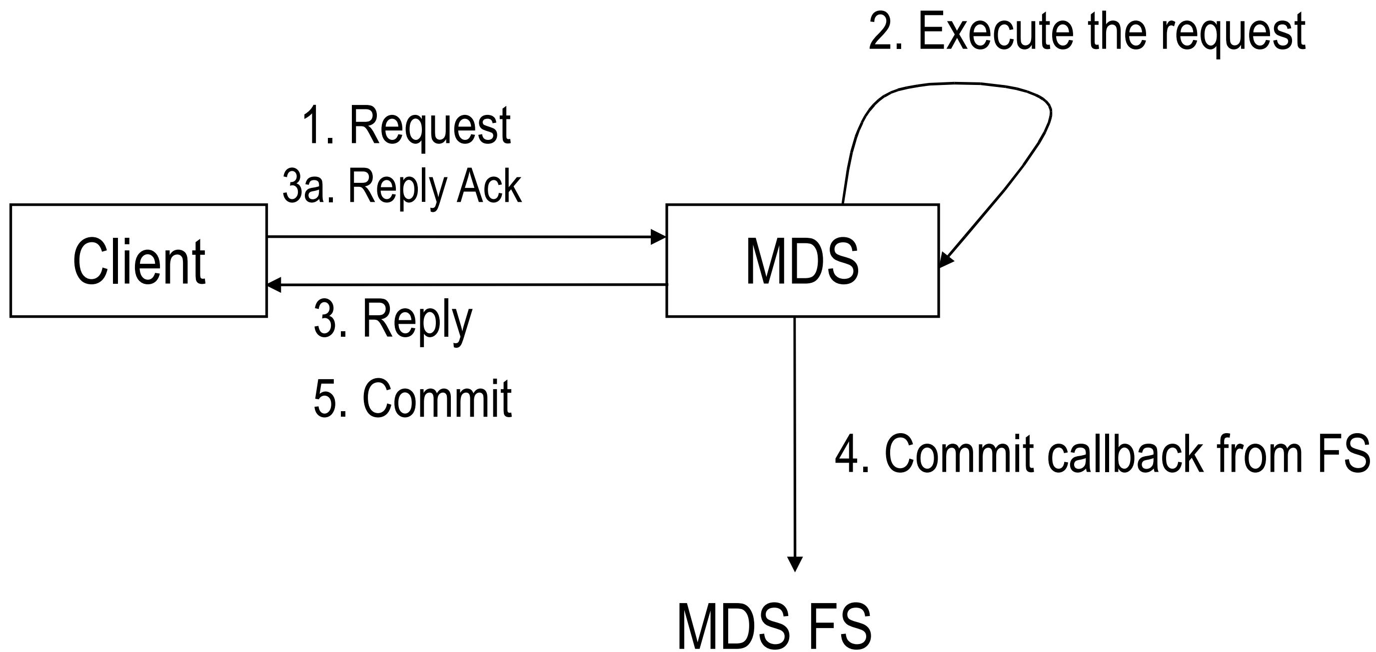
Topics

> Lustre striping
> Request lifecycle
> llog
> I/O in the OST
> ldlm

MDS execution

- MDS executes transactions
  - > In parallel by multiple threads
- Two stage commit:
  - > Commit in memory – after this results are visible
  - > Commit on disk – in same order but later
  - > This batches the transactions
- Key recovery issue
  - > Lustre MDS can lose some transactions
  - > Clients need to replay in precisely same order

Request lifecycle



Client MDS interaction

- Send request
- Request is allocated a transno
- Send reply which includes transno
- Clients acknowledge reply
  - Purpose: MDS knows clients has transno
- Clients keep request & reply
  - Until MDS confirms a disk commit
  - That's where we need commit callback
  - Purpose: client can compensate for lost trans
- MDS has disk data per client
  - Last executed request, last reply information

Commit callbacks

- Run a callback, when disk data commits
- Ability to register & run callbacks has been removed from JBD in 2.6.10
  - > Added back by the jbd-jcberr* patches
- Similar mechanism needed for DMU support

OST Request Handling

- In 1.6
  - OST waits for journal commit after each write
  - Once the write rpc is acknowledged, data & metadata are safely written to disk
  - No need to repaly bulk write requests
- Async journal commit introduced in 1.8
  - No longer wait for journal after each write
  - Implement bulk write replay
    - Roughly same scheme as for MDS requests now
  - Disabled by default
    - Due to some recovery issue under investigation
    - lctl set_param obdfilter.*.sync_journa=0 to enable it

Topics

- Lustre striping
- Request lifecycle
- llog
- I/O in the OST
- ldlm

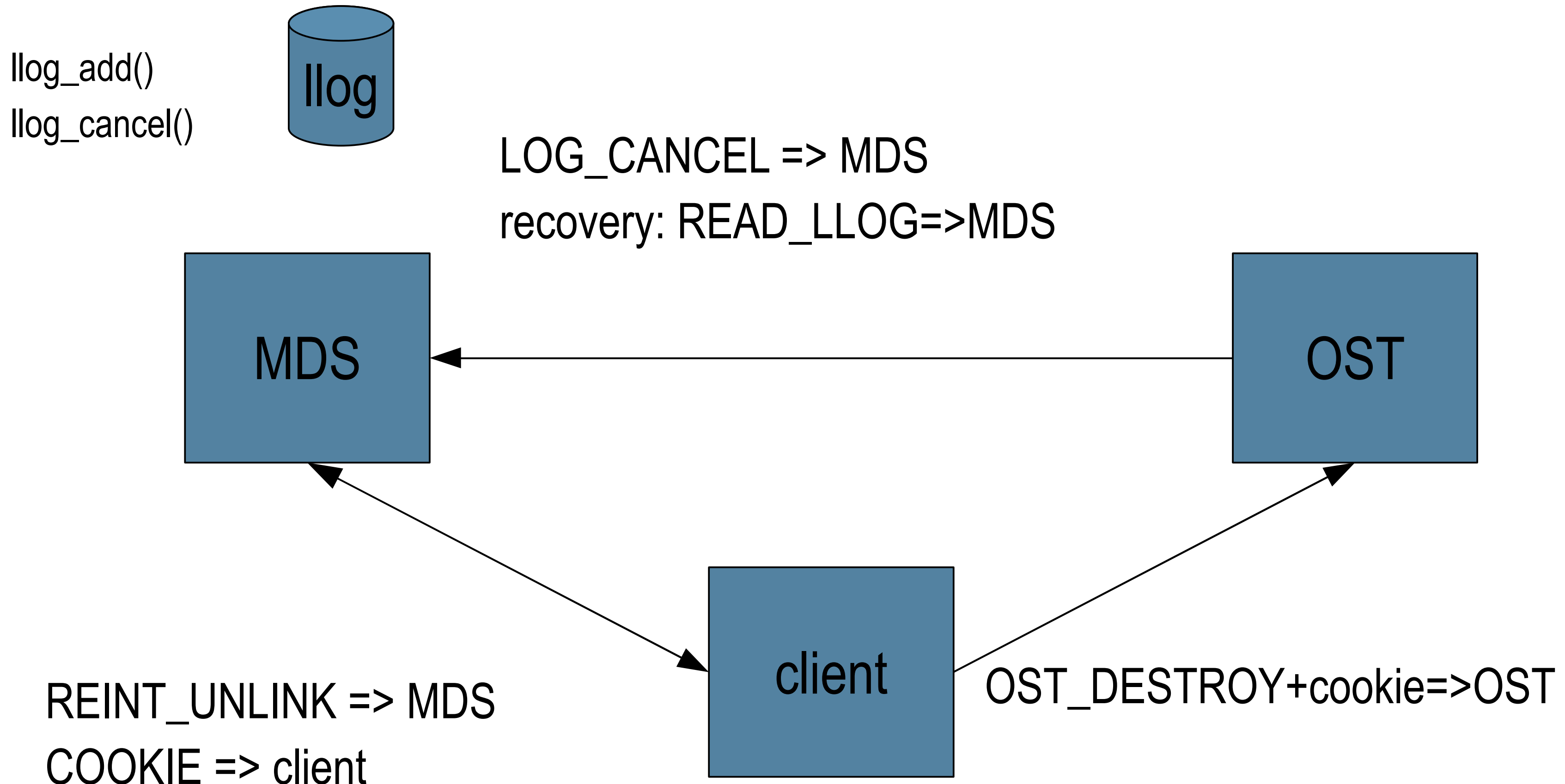
Problem Statement

- Lustre is distributed filesystem
- some POSIX calls change on-disk state on few nodes
- Examples:
  - unlink removes MDS and OST inodes
  - setuid changes owner on MDS and OST
- need to maintain consistent state after failure

Maintaining Consistency: llog

- For distributed transaction commits
- Terminology
  - Initiator – where the transaction is started
  - Replicators – other nodes participating
- Normal operation
  - Write a replay record for each replicator on the initiator
  - Cancel that record after the replicators commit, in bulk
    - Commit callback needed here
- Recovery
  - Process the log entries on the initiator

Use case: unlink



Use case: unlink (cont'd)

- OST commits objects destroy
  - Then it’s time to cancel the MDS llog records
  - Add the cookies to the llog cancel page
  - … truncate the object
  - Start a transaction (fsfilt_start_{log})
  - Remove the object (filter_destroy_internal)
  - Add the commit callback (fsfilt_add_journal_cb)
    - CB is filter_cancel_cookies_cb
  - Finish the transaction (fsfilt_finish_transno)

Topics

- Lustre striping
- Request lifecycle
- llog
- I/O in the OST
- ldlm

Sun microsystems

OSS Read Cache

- The page cache made things too slow in Linux 2.4
- Reserved memory registered for DMA can help
- In 1.6, OSS does non-cached direct IO
  - > Nothing ends up in the OSS page cache
- OSS page cache has been resurrected in 1.8
  - > For now, only for read
  - > Huge performance increase when reading small files back
- Two new parameters
  - > /proc/fs/lustre/obdfilter/*/read_cache_enable
  - > /proc/fs/lustre/obdfilter/*/writethrough_cache_enable

Topics

- Lustre striping
- Request lifecycle
- llog
- I/O in the OST
- ldlm

Sun microsystems

Lustre Distributed Lock Manager

- A lock protects a resource
  - Typically, a lock protects something a client caches
- A client enqueues a lock to get it
- An enqueued lock has a client and server copy
- Servers send blocking callbacks to revoke locks
- Servers send completion callbacks to grant locks
- Processes reference granted client locks for use
- Processes de-reference client locks after use
- Clients cancel locks upon callbacks or LRU overflow

- Callbacks were called AST's in VAX-VMS lingo
- Cancel was de-queue in VAX-VMS lingo

Sun microsystems

LDLM history

- Basic ideas are similar to VAX DLM
  - > You get locks on resources in a namespace
  - > All lock calls are asynchronous and get completions
  - > There are 6 lock modes with compatibility
  - > There are server to client callbacks for notification
  - > There are master locks on the “server” and client locks
- Differences
  - > We don’t migrate server lock data, except during failover
    - – LDLM is more like a collection of lock servers
  - > There are extensions to:
    - – Handle intents – interpret what the caller wants
    - – Handle extents – protect ranges of files
    - – Handle lock bits – lock parts of metadata attributes

Client Lock Usage

- DLM locks are acquired over the network
  - The locks are owned by clients of the DLM
    - MGC, OSC & MDC are examples
- Use of locks
  - Locks are given to a particular lock client
  - Processes reference the locks
  - Locks can be canceled only when idle
- Differences
  - Locks are not owned by processes (VAX)
- Servers can take locks also

Lustre Lock Namespaces

- OST: namespace to protect object extents.
  - Resources are object ids
  - Extents in the object are “policy data”
- MDS: namespace to protect inodes and names
  - FIDs are the resources
  - Lock bits are policy data
  - Intents bundle a VFS operation with its lock requests
- MGS: namespace for configuration locks
  - Presently only one resource
  - Protects the entire configuration data

Sun microsystems

File I/O locks and lock callbacks

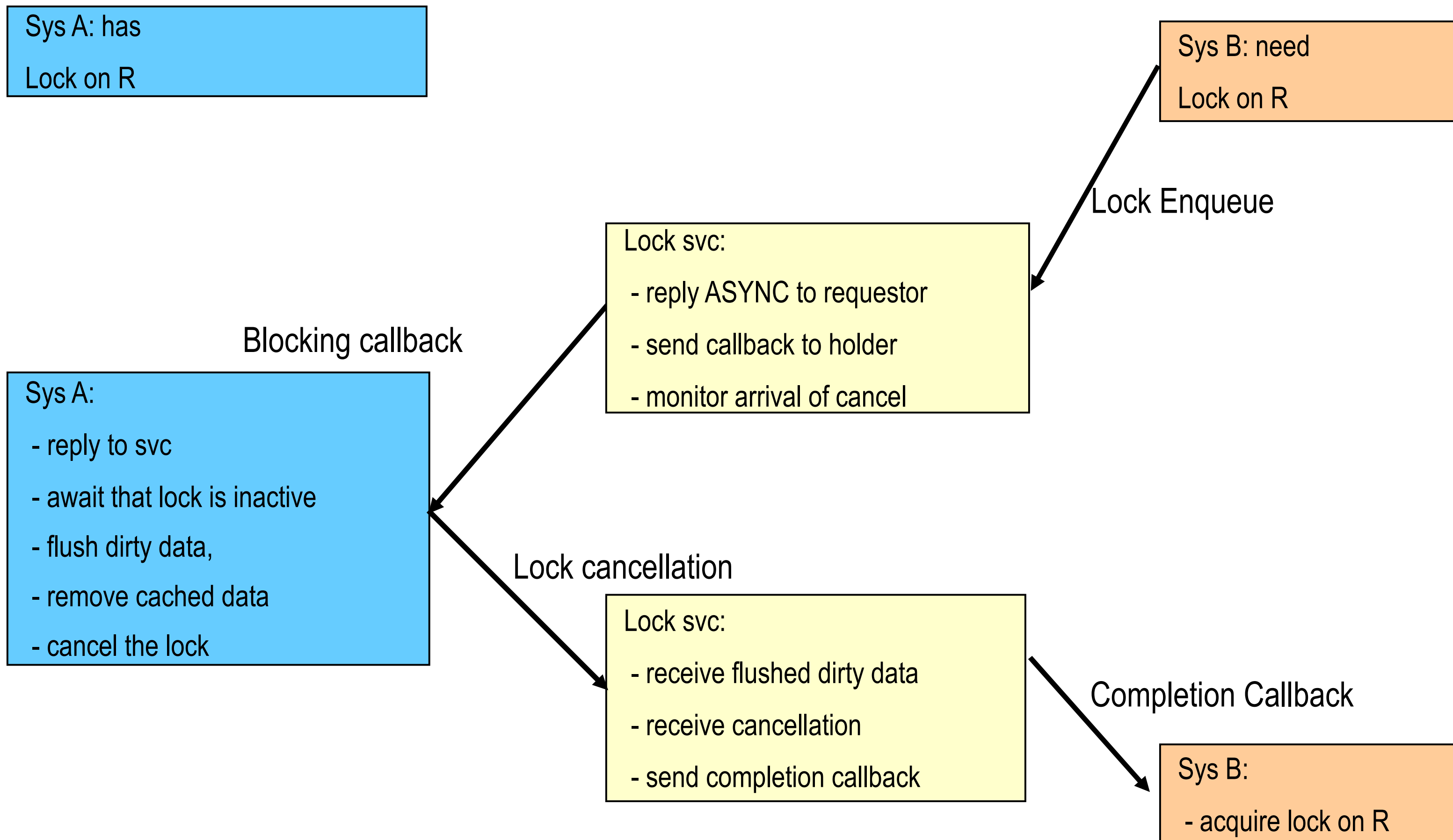
- Clients must acquire a read-lock to cache data for read
  - > Locks cover an optimistically large file extent
  - > Locks are cached on clients
- Before writing, a client obtains a write lock
- Upon concurrent access by another client
  - > Client locks see a callback when others want a conflicting lock
  - > After the revocation callback arrives, dirty data is flushed
  - > Cached data is removed
  - > Then the lock is dropped

Sun microsystems

Client Lock Callback Handling

- Callback function is bound to lock
  - > upon client side lock enqueue
  - > RPC's made to the client ldlm service by servers
  - > Handed by client lock callback thread : ldlm_cbd
- Completion callback
  - > When lock is granted
- Blocking callback
  - > Called when servers try to cancel locks in clients
  - > Causes cache flush

Typical Simple Lock Sequence



Sun microsystems

I/O & Locking

- Stripe locking
  - > Change from
    - Lock all stripe extents, do all IO in parallel, unlock all
  - > To
    - For all stripes in parallel: lock, do IO, unlock
  - > Holding locks from multiple servers
    - Can lead to cascading aborts
    - Is necessary for truncate and O_APPEND writes
- Disallow client locks under contention
  - > When an extent in a file sees concurrent access
    - Ask the client to write through to the server
  - > This eliminates callback traffic and cache flushes

Sun microsystems

File size and glimpses

- Normal case
  - Only one client does IO to a file, this client knows the size
- Size of file without active IO from any client
  - Currently file size derived from object sizes
  - Will be on the MDS in the future (SOM) - optimal for quiescent files
- Size of a file under active IO
  - Now any client with “far write lock” maybe growing the file
  - A full file write lock would protect the size, but flushes all caches!
    - Lustre does NOT DO THIS, unless the file is not busy
  - In Lustre the OSS’s ask the clients with furthest locks for the size
    - This is a glimpse callback - gives one view of file size
    - A glimpse callback causes clients to cancel locks if they are not using them
  - Glimpsing is the optimal method to get file size during active IO

Configuration Lock

- The central configuration server is the MGS
- When a client fetches a log it also gets a lock
  - The lock gets callbacks when the configuration changes
- Callback triggering events
  - Online addition of OST devices
  - Setting timeouts is global now
  - Many others usage (OST pools creation, quota setup, ...)
  - More robustness fixes

Sun microsystems

Timeouts and Eviction

- Client requests time out unless a reply is received
- Client-originated RPC timeouts will cause the client to:
  - Disconnect from the affected server
  - Ping, reconnect to server or failover and retry/complete operations
- Server callback RPC timeouts *evict* the affected client
  - Reconnects to server like an evicted NFS client (not a perfect solution, but OK)
  - The client will learn of eviction during its next request
  - Upon eviction the client must purge its cache
    - if data is dirty, this means a small amount of data loss!
  - In-flight network ops will return -EIO to application
  - Eviction prevents one bad client halting the whole cluster

Questions?

johann@sun.com