



> Version 1.5 Q1 2008

LECTURE 4.1

Lustre Architecture

Contents

- LNET
- Ext3
- DLM
- I/O protocol
- Metadata protocol
- Configuration protocol
- Recovery
- OST / MDT disk layout

LNET

More information

- The history of LNET is Sandia Portals
- There is a good manual for the portals API
 - > <http://sourceforge.net/sandiaportals>
- LNET is a message passing API

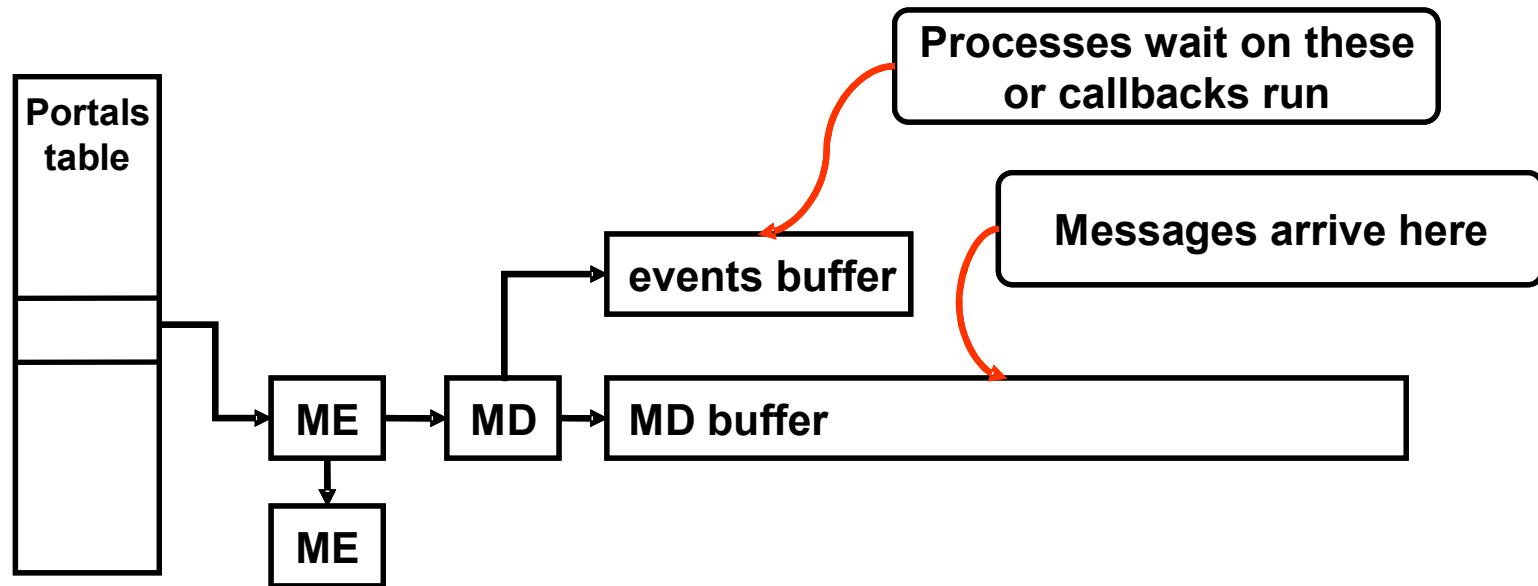
Fundamental data structures

- Destination of msg: `lnet_process_id_t` is a pair (nid, pid)
- Events: `lnet_handle_eq_t` points to buffer with `lnet_events`
- Memory Descriptor `lnet_md_t`: points to buffer, length, EQ
- Delivery indication: specify portal & matchbits when sending

- Delivery analysis: LNetMEAttach – attach match entry to MD
 - > Portal identifier: labels a collection of match entries
 - > ME's are traversed upon receipt
 - > ME's point to a receiving MD

- Delivery notification: write event in event queue buffer
 - > MD can point to event queue (EQ)
 - > EQ points to a buffer
 - > Buffer will receive `lnet_event_t` structures & callback runs
 - > LNetEQWait waits on an event arriving in a queue

Receiving an LNET message



Lustre connections, MDs and MEs

- Connection is a structure:
 - > peer_nid
 - > Lustre specific data
 - > xid is a per connection integer labeling requests
- MDs
 - > Request buffers – a pool that grows
 - > Reply buffers – individual buffers
 - > All buffers are allocated before use
 - Interesting hackery to handle request buffer overflow
- Requests & reply carry the xid as matchbits
 - > Request buffers ME: not sensitive to what xid is sent
 - > Reply buffers ME: match exactly on xid
 - > Bulk transports gets MD on the fly

Lustre event queues

- A thread can wait on an event queue
 - > Similar to select system call
 - > Waits for events
- Events that are used
 - > Events for incoming buffers
 - Waiting for Lustre requests
 - Waiting for Lustre replies
 - Waiting for bulk data
 - > Events for buffers that have “left” the node
 - Sometimes they can be freed
 - > Events for ACK messages
 - Replies are sometimes ACK’ed for recovery reasons

Error handling

- LNET detects some errors
 - > When it does it lazily frees the buffers
 - Buffers may be used by DMA engines
 - > Lazily means through events
- Lustre detects all errors
 - > Some are timeouts
 - > Others are reported by LNET

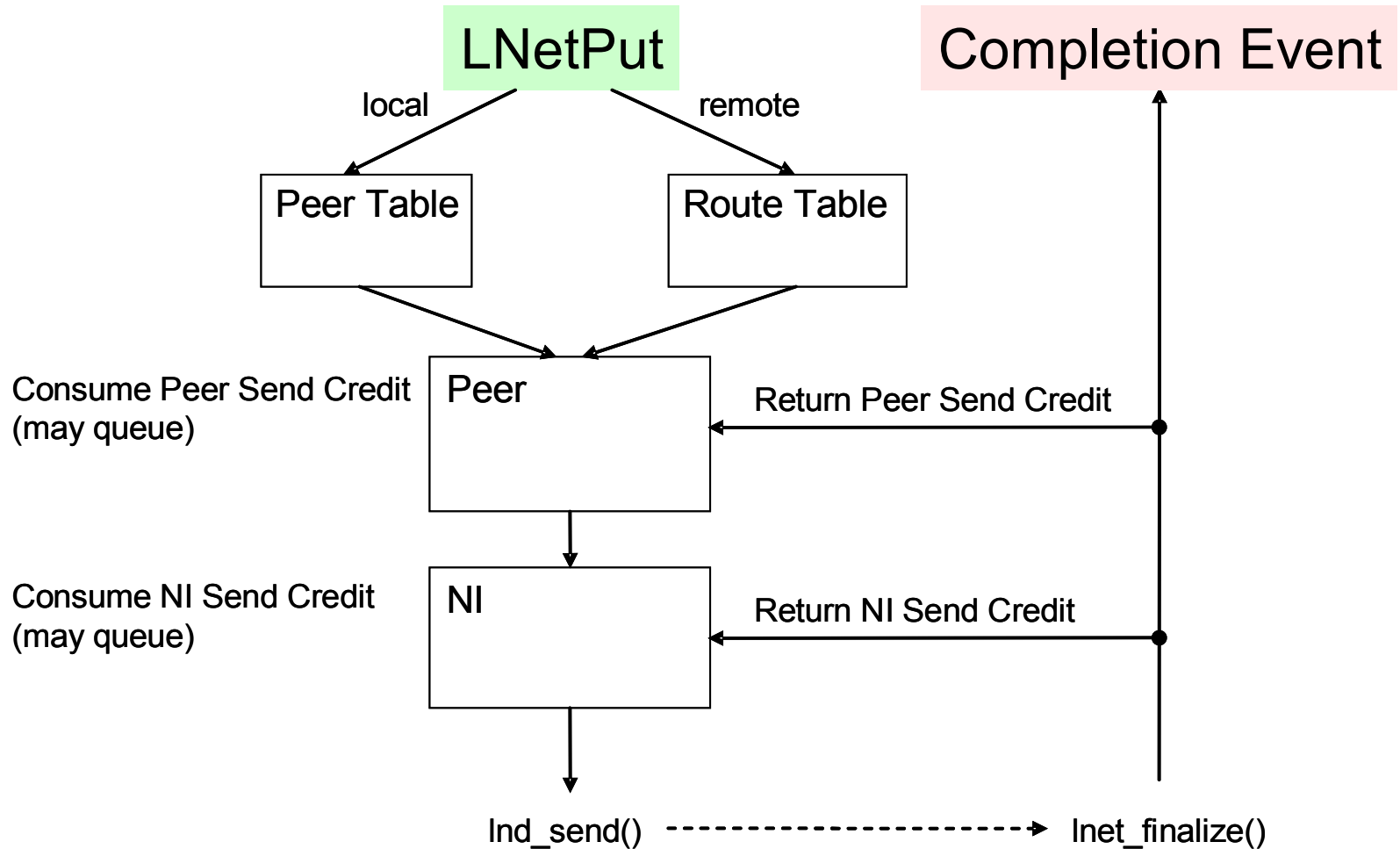
LND – Lustre network drivers

- Lustre Network Drivers
 - > NAL's in Portals speak
- Lustre sees, i.e. LNET exports, only one “interface”
- Initialization is based on `/etc/modules.conf`
- There are ~ 10 LNDs
 - > 4x IB flavor, Elan, Myrinet, TCP, 2 Cray LNDs
- So what does an LND do?

LND internals

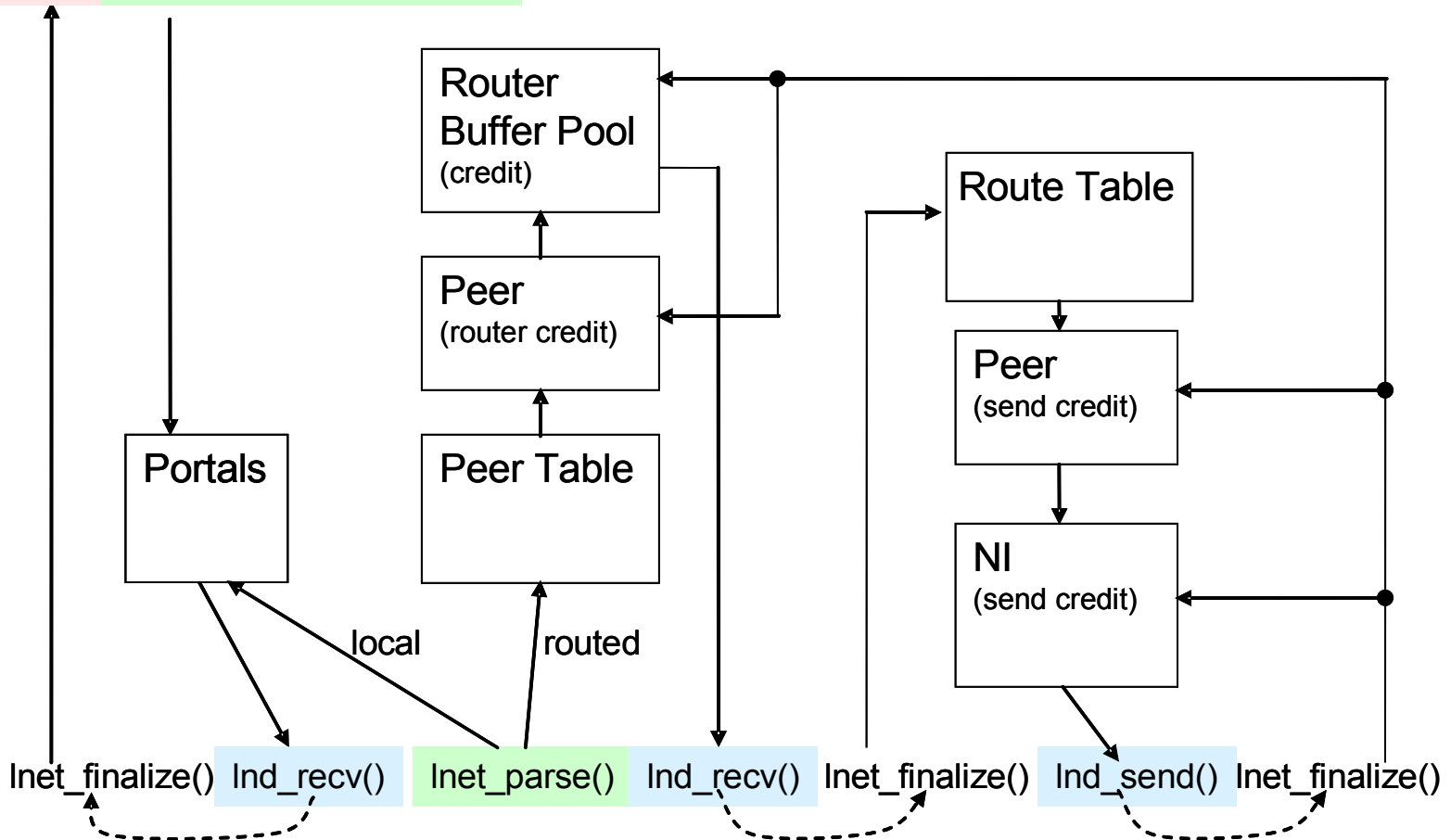
- Implements a method table, called by LNET
 - > Ind_*
 - > Equally important reverse API, Inet_* called by the LND
- E.g., Ind_send – put a message on the wire
 - > Can involve a DMA handshake
 - > Or managing TCP transmit buffer space
- Most LNDs maintain “connection” information
 - > E.g., for TCP this points to the sockets we use
 - > For DMA capable NALs send/rcv credits

Outgoing Message



Incoming Message

Completion LNetMDAttach



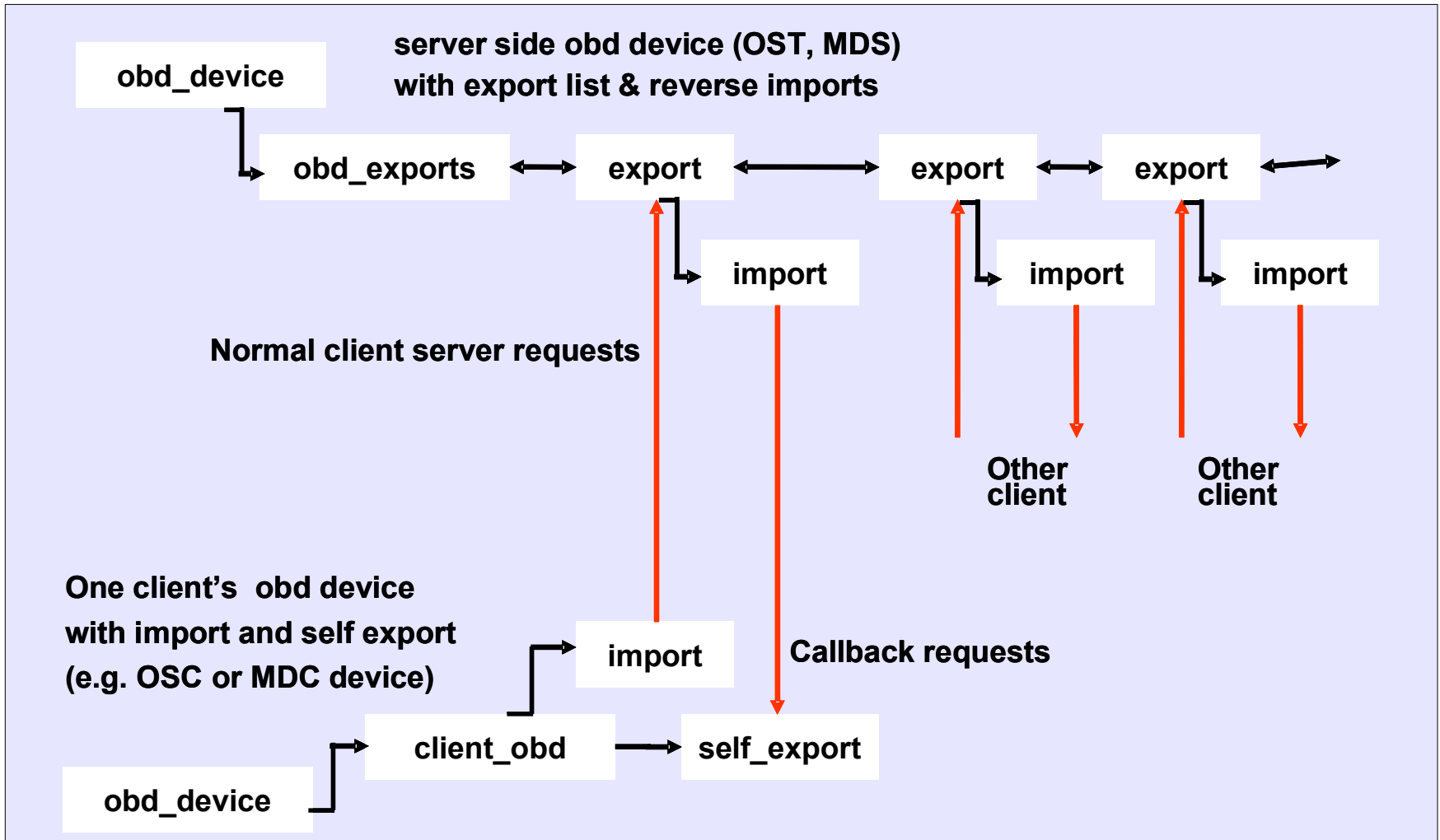
Irpc – Lustre RPC

- Used to be called ptlrpc
- Basic mechanisms to:
 - > Send requests from import, receive replies
 - > Receive and process requests on export, send replies
 - > Including bulk transport between request and reply
- Recovery
 - > Establish connections – establish import / export pair
 - > Irpc will re-send requests that need recovery
 - > Recovery state is managed on an import – export pair
- This involves:
 - > Buffer handling – particularly request buffer management
 - > Error handling – manage the state of an import/export

Generic Irpc function library

- Generic connect / disconnect
 - > client_connect_import(...)
 - > target_handle_connect(...)
 - > target_handle_reconnect(...)
- A request set handler
 - > Handle multiple requests asynchronously
- A ping infrastructure
 - > For health / liveness checks
- A bulk llog callback handlers
 - > Collect pages of cookies and send across

Exports & Imports for RPCs



Request processing

- Client – allocate request and reply buffer
- Fill request buffer
- Call `Irpc_queue_wait`

Recovery

- Much of recovery takes place in Irpc!
- Resending requests
- Reconnecting
- Pinging
- Server waiting for clients to join for recovery
- Processing a queue of replay / resend requests

Ext4 & Idiskfs

Ext3 & Idiskfs

- Ext4 is a fork of Ext3 - too many changes (from CFS)
- Idiskfs is the Lustre version of ext4
- Sometimes equal to ext4, usually ahead of ext4
- In this section we will describe
 - > Recent ext4 enhancements originating in Lustre
 - > Special use of ext4 by Lustre

Allocation with extents

- Filesystem extent
 - > Contiguous on-disk & in-file area
 - > Triple (StartBlockInFile, StartBlockOnDisk, BlockNumbers)
 - > Benefits
 - Make allocation MD (“mapping”) much smaller
 - Speedup truncate and metadata overhead
- Extent library used in other places
 - > Snapshots
 - > Hierarchical stripe descriptors

Extents

- Critical data structures
 - > struct ext3_extents_tree
 - Extents mappings of a whole file
 - > struct ext3_extent
 - Mapping of a piece of contiguous file area
 - > struct ext3_extent_idx
 - Used inside the htree
- Extent allocation policy
 - > Always try to allocate N contiguous blocks

mballoc - the block allocator

- Old allocation style
 - > Per block, not contiguous extents of free blocks
 - > Needed lock_super()
 - > Too much linear scanning on bitmaps
- Mballocc design
 - > Buddy bitmap:
 - Buddy block describe 2^N contiguous blocks, where $N \geq 0$
 - In addition, there is the existing block bitmap
 - > Allocation of extents of size 2^N
 - Is efficient, and aligned
 - > Refined locking - avoid the super block lock, per cpu locks
 - > Keep files that are expected to be small close together

Allocation policy

- Three mechanisms
- Goal
 - > Write behind the previous data if possible
- Large I/O policy
 - > Write the data in a contiguous block anywhere
 - I/O is large enough to eliminate seek overhead
 - > Physical and logical alignment (mod X Mb)
- Small I/O policy
 - > Keep small chunks close together, close to inode
 - Like ext3
- Writing a large file with small synchronous files hurts
 - > Fixed with an OSS cache

Transactions

- Lustre frequently nests transactions
- E.g., in `mds_reint_unlink`
 - > Start a transaction (`fsfilt_start_{log}`)
 - > Unlink the inode (`vfs_unlink`)
 - > If it is open, put it in the orphan directory (`mds_orphan_add_link`)
 - > Add an llog entry (`mds_log_op_unlink`)
 - > Set the parent mtime/ctime (`fsfilt_setattr`)
 - > Finish the transaction (`fsfilt_finish_transno`)

Commit callbacks

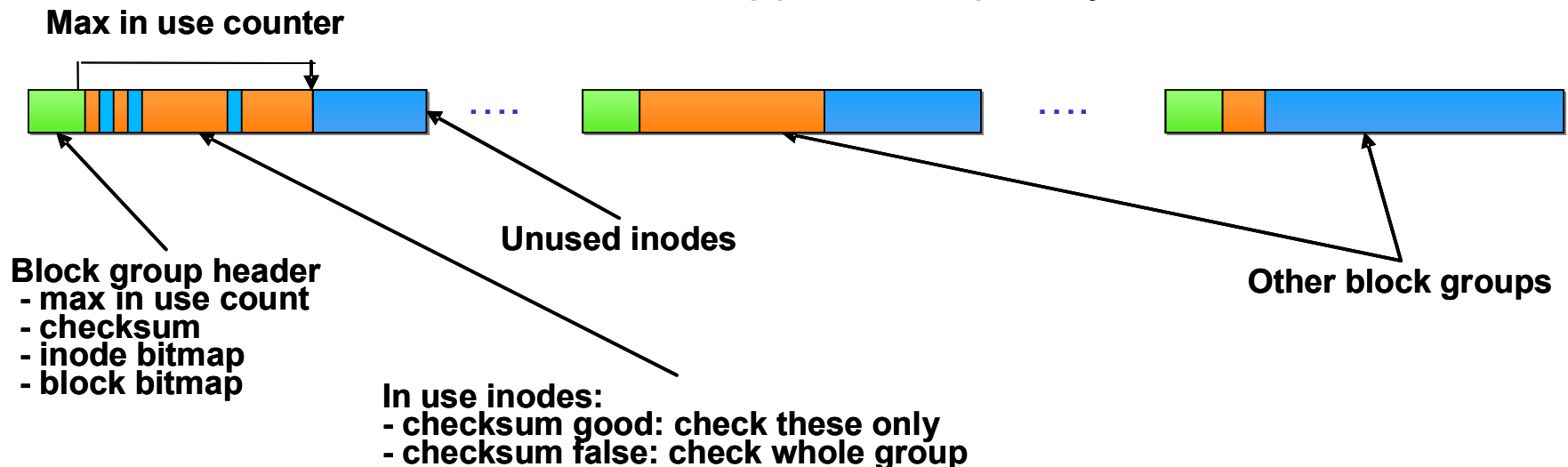
- When disk data commits, run something else
- E.g., OST commits objects destroy
 - > Then it's time to cancel the MDS llog records
 - > Add the cookies to the llog cancel page
 - > ... truncate the object
 - > Start a transaction (fsfilt_start_{log})
 - > Remove the object (filter_destroy_internal)
 - > Add the commit callback (fsfilt_add_journal_cb)
 - CB is filter_cancel_cookies_cb
 - > Finish the transaction (fsfilt_finish_transno)

I/O in the OST

- The page cache made things too slow in Linux 2.4
- Reserved memory registered for DMA can help
- OSS does non-cached direct IO
 - > Nothing ends up in the OSS page cache
 - > We expect to resurrect an OSS page cache
 - Certainly for reading small files back
 - Maybe for dirty data in conjunction with liblustre

Fast FSCK & Format

- FSCK has changed
 - > Previously fsck scanned all inodes
 - > Now only inodes that are possibly in use
- The most interesting part of this is a checksum
 - > The checksum indicates if the metadata that follows is consistent
 - > If it is the counter can be used to check up to the maximum inode
- Speedups of 4x to 10x
 - > Good, but fsck needs to disappear completely, it doesn't scale



Lustre DLM

Terminology

- A lock protects a resource
- 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

- Typically a lock protects something a client caches

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 cancelled 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

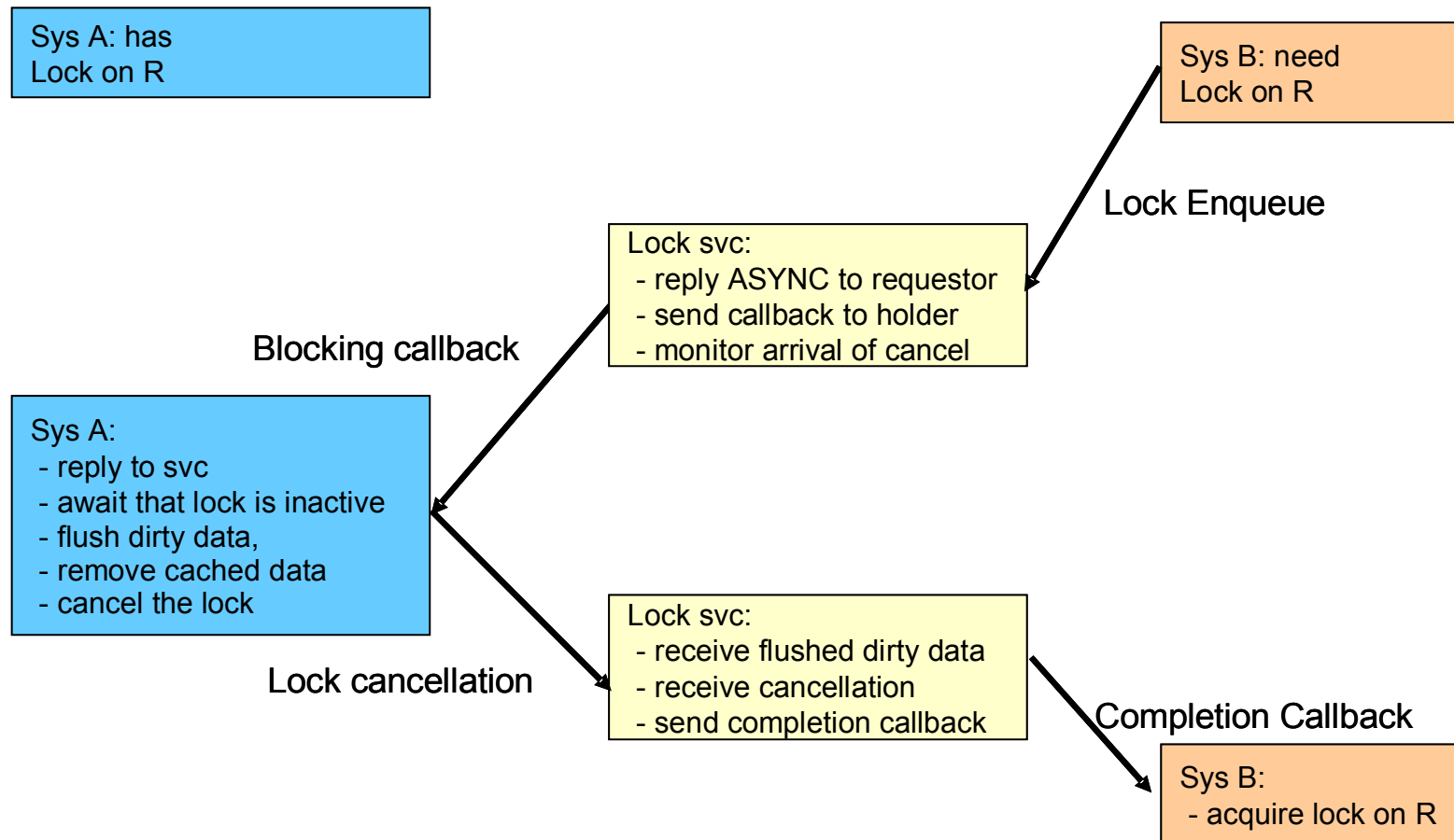
Lock modes

- A lock on a resource can be taken out in six modes.
 - > EX exclusive
 - > PW protected write
 - > PR protected read
 - > CW concurrent write
 - > CR concurrent read
 - > NL null

Compatibility

	NL	CR	CW	PR	PW	EX
NL	1	1	1	1	1	1
CR	1	1	1	1	1	0
CW	1	1	1	0	0	0
PR	1	1	0	1	0	0
PW	1	1	0	0	0	0
EX	1	0	0	0	0	0

Typical simple lock sequence



Locking file extents for POSIX

- Lustre extent lock
 - > Lock covers an area inside file during R/W lustre file.
 - > Acquired by Lustre filesystem clients for
 - R/W
 - File size computation
 - Truncation
 - > Release reference on it after R/W, keep cached for quick re-use
 - > Handle blocking callbacks for revocations
 - > File locks broken down by LOV into lock per object
- Use
 - > For data consistency R/W, and truncate
 - > File size management
 - > Important client constant is maintained:
 - Known Minimum Size (KMS) of each stripe
- No user API
 - > Locks are referenced only within a single syscall

Extent optimization algorithm

- Motivation
 - > Typical use is one client doing I/O to a file
 - > Reducing lock requests for contiguous areas
 - From the same client
 - > One user: one lock
- Optimization scenarios
 - > Client: search the cached locks for compatible locks
 - Use cached locks – can use write lock for reading!
 - > Server runs `ldlm_extent_policy ()`
 - Scan server granted/waiting queues
 - Calculate nearest extents in incompatible locks found
 - Grant biggest possible extent

IO and 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 recovery events on many servers
 - 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

User controllable locks

- `fcntl` / `flock`
 - > Fairly standard interface
 - > Backed by DLM locks
 - > Implementation feels preliminary
 - > Some of our partners activate this by default
- A special group lock for MPI use
 - > Allows a group of processes to exclude others
 - > Can be mandatory or advisory
 - > Typical use: keep visualization clients away from busy files
- Special use of locks by clustered exports of NFSv4
 - > See CITI documentation

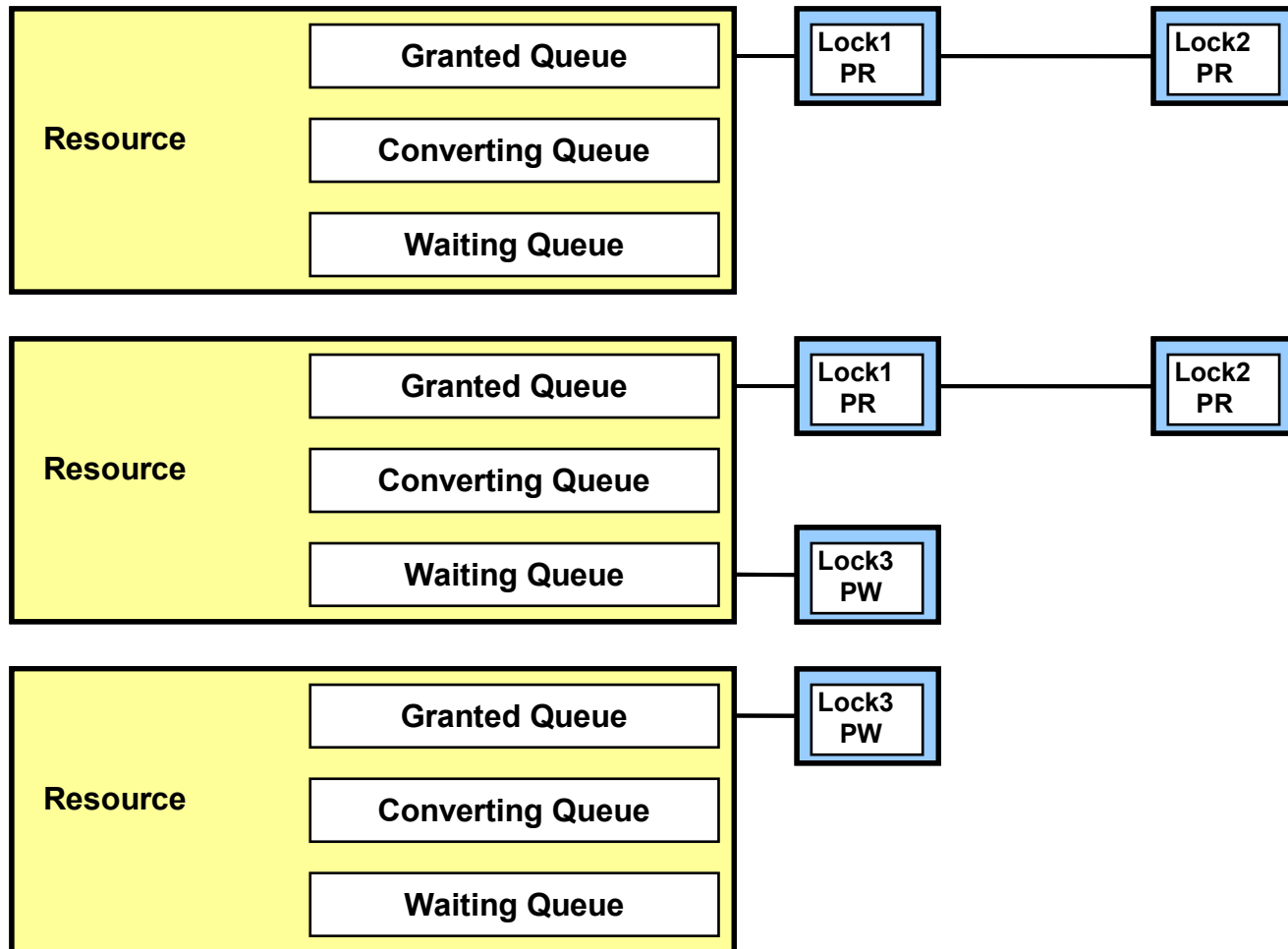
Conversion from read to write

- Motivation
 - > Things would start with a read – lock
 - > Cancellation of read locks would lead to cache flush
 - > Conversion to a write lock allows retention of cached data
 - But it can deadlock
- Lock conversion is not used yet in Lustre
 - > A version on the inodes could allow retention of cache

Client Lock callback handling

- **Callback function is bound to lock**
 - > Upon client side lock enqueue
 - > RPC's made to the client Idlm service by servers
 - > Handed by client lock callback thread : Idlm_cbd
- **Completion callback**
 - > When lock is granted or request failed AST notification
- **Blocking callback**
 - > Called when servers try to cancel locks in clients
 - > Causes cache flush
- **Glimpse callback – to get file size**
 - > If lock cannot be granted, just give size
 - > Don't flush cache
 - > Cancel lock if it is not busy

Ldlm queues – processed on server

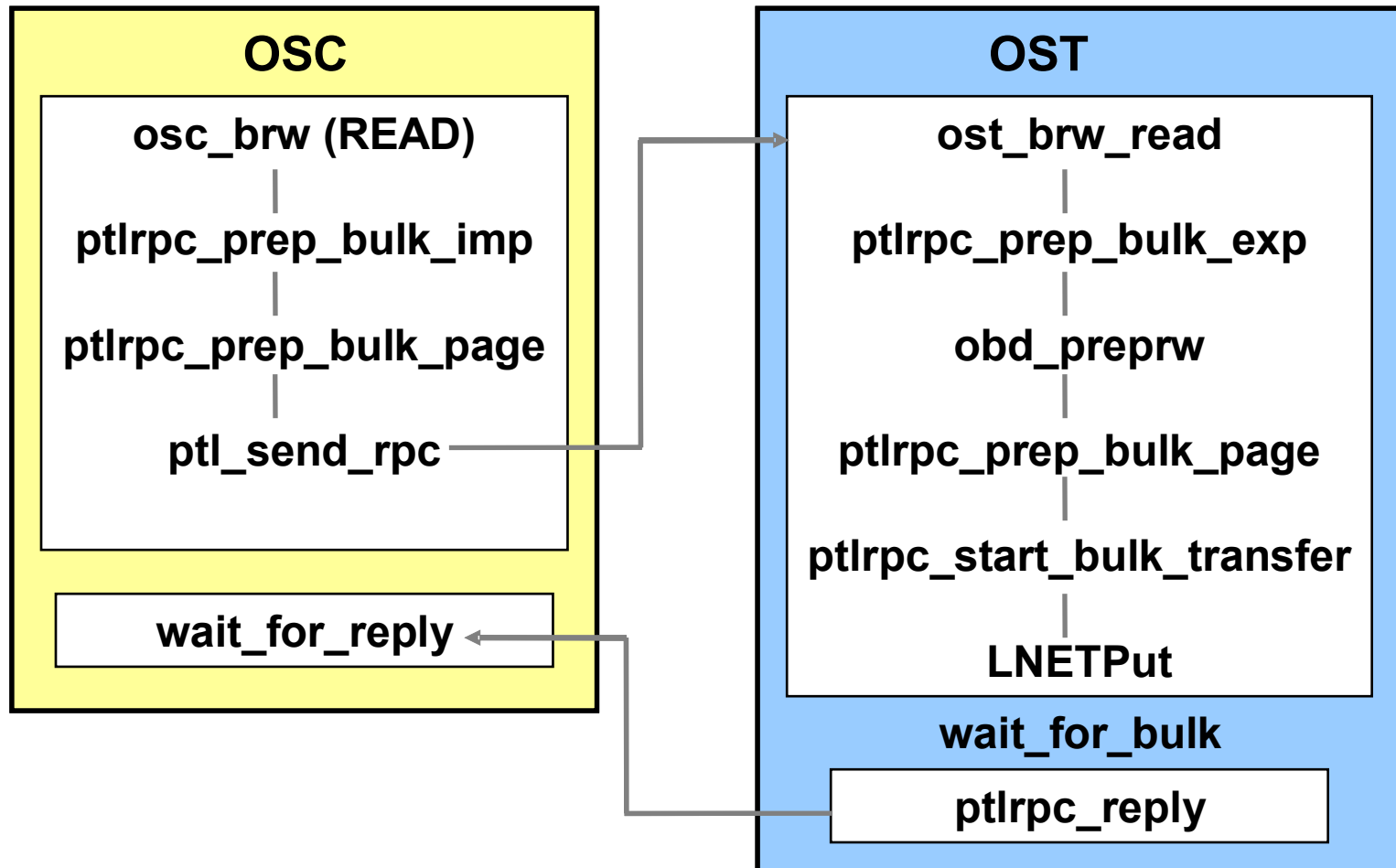


I/O protocol

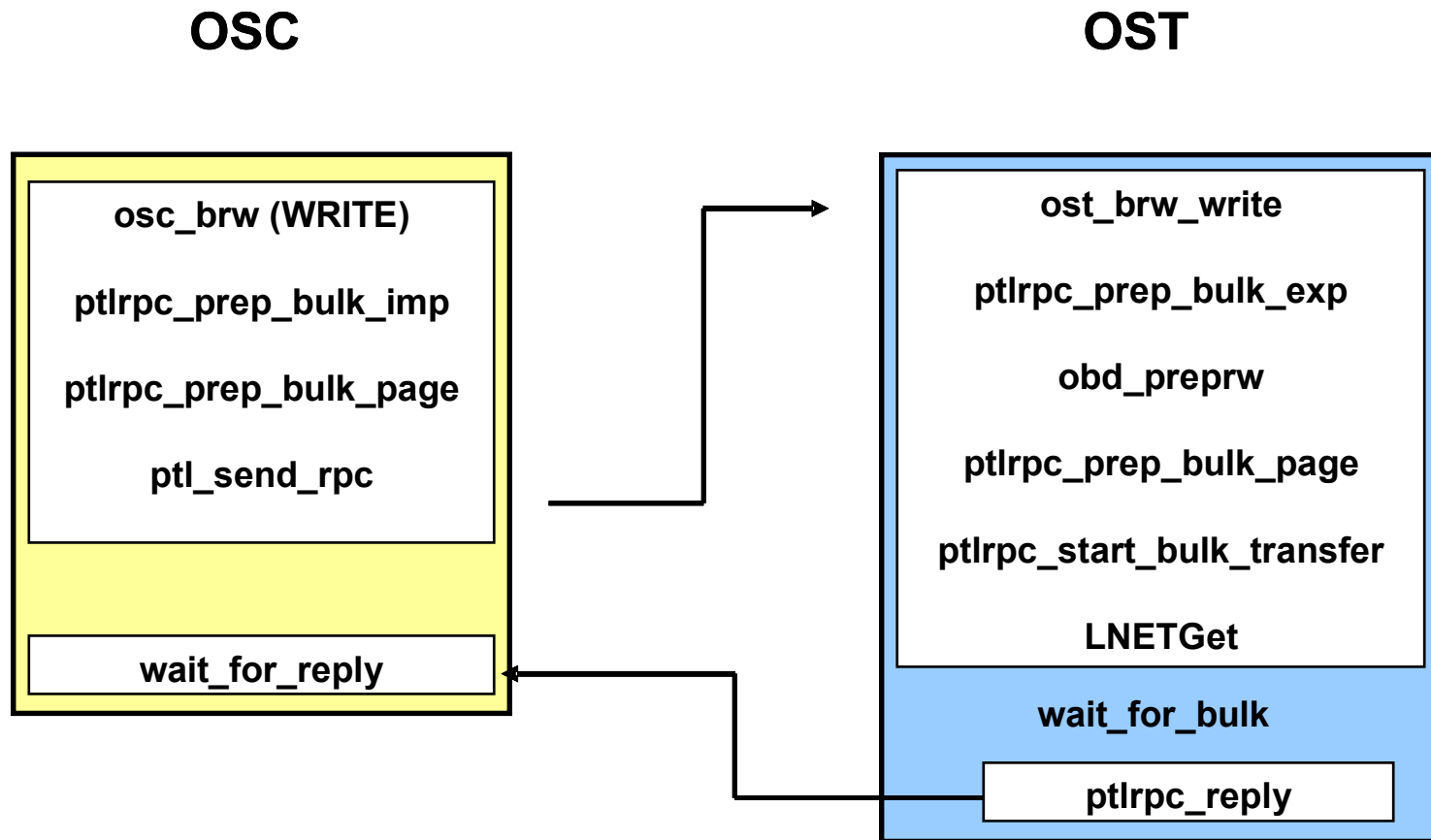
I/O from 30,000 feet

- Lustre lite
 - > Dirties pages
 - > Asks for pages to be flushed (fsync, etc)
- LOV directs pages to correct OSC
- OSC
 - > Assembles IOV of dirty pages
 - > Keeps flight groups to OST on the wire
- OST / OSD
 - > Receive or send data (when possible with RDMA)
 - > Perform block allocation in objects
 - > Do direct I/O

osc-ost READ diagram



Osc-ost WRITE diagram



Event callbacks for bulk

- **Server side callback**
 - > Registered in bulk preparation phase
 - > `server_bulk_callback()` wakes up sleeping server
 - On bulk transfer completion event
 - > Then server sends `lrpc` reply message to client
- **Client side callback**
 - > Registered in bulk preparation phase
 - > `client_bulk_callback()` wakes up client
 - Which is sleeping on `lrpc` reply from server

lrpc sets

- Infrastructure
 - > ptlrpcd() kernel thread
 - Check for lrpc requests in set_requests list
 - Move new request in set_new_requests list to pc_set
 - Handle requests in pc_set with ptlrpc_check_set()
 - > There are two sets: one for recovery, one for normal use
 - > APIs
 - Add set requests by ptlrpcd_add_req()
 - Wait on completion of the set with ptlrpc_set_wait()
- Usage in osc_brw_async ()
 - > Uses ptlrpc_set_add_req() to add async lrpc request
 - > Use brw_interpret() async callback to wakeup process waiting for it

OSC page management

- **Warning**
 - > This is likely going to change somewhat
 - > Linux 2.4 forced us down the current path
- **At the moment Lustre**
 - > Tracks all dirty pages
 - > When enough are there, the OSC pushes them out
- **Linux disk file systems**
 - > Flush pages from the page cache
 - > Only then they are sent to storage
 - > Lustre will move in this direction in the 1.8 timeframe

Page Queueing in the OSC

- Asynchronous OBD API
 - > Benefits: batching RPC to maximize efficiency
 - > Callers: `commit_write()` & `writepage()`
- Page Queueing APIs
 - > Registering Page in OSC
 - API: `osc_prep_async_page()`
 - binding page to a new created OSC async page and set RPC callback pointers for post processes
 - > Queueing Page in OSC
 - API: `osc_async_page_io()`
 - Using the cookie returned from Page Registering function to find the OSC async page

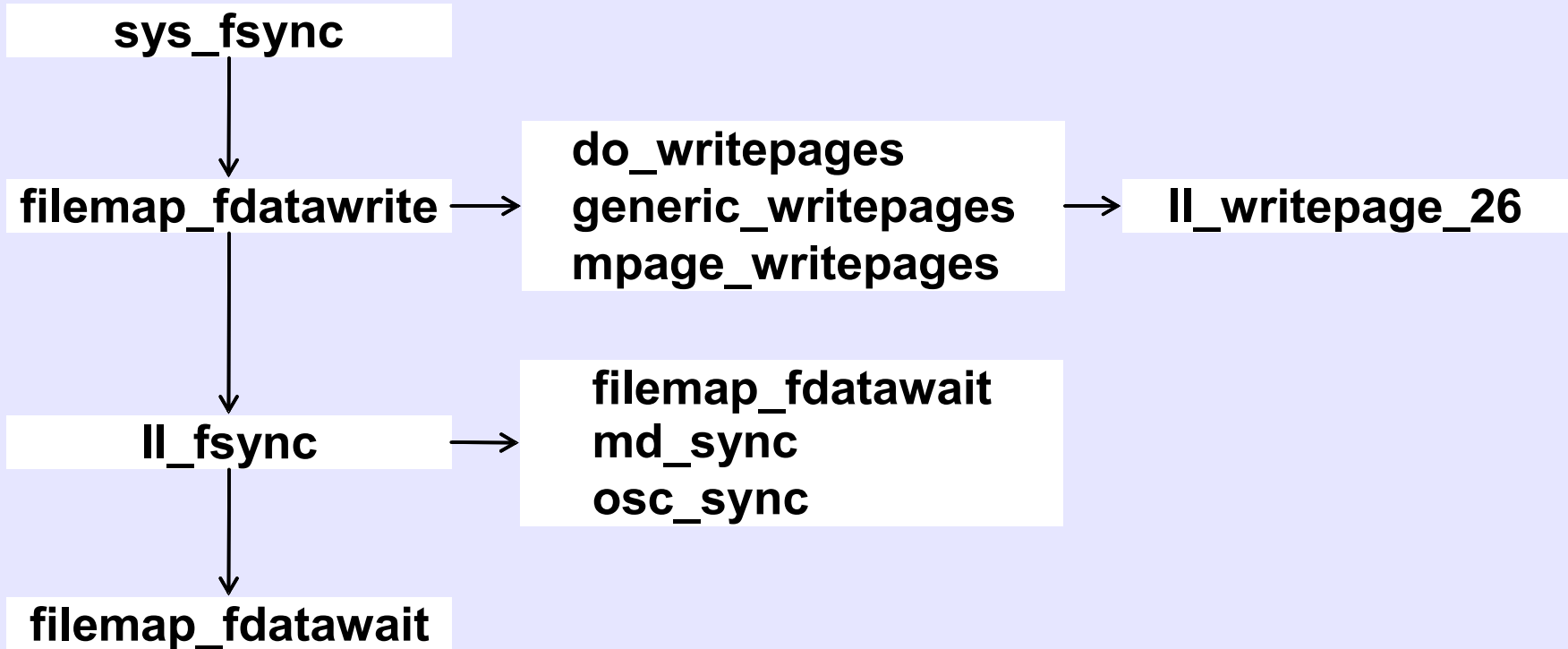
Trigger RPC when reach iov size

- APIs
 - > `osc_check_rpcs()`
 - > `osc_send_oap_rpc()`
- Scenarios
 - > Inside `osc_send_oap_rpc()`
 - When constructing async RPC request
 - If OSC async page count reach `cl_max_pages_per_rpc`
 - Then create an async RPC request for it
 - > If `osc_send_oap_rpc()` races with `commit_write()`
 - Retry 10 times before exit this RPC creation

OSC flight group management

- Client Side in flight RPCs
 - > `cl_brw_in_flight`: in-flight RPC count
 - > `cl_max_rpcs_in_flight`: upper limit of in flight RPCs
- Scenarios
 - > When '`cl_brw_in_flight`' reaches upper limit
 - Stop constructing async RPC requests
 - > Without grant, one RPC only:
 - Put process to sleep when grant is below lower limit
 - Wakeup it when `cl_brw_in_flight` is 0
 - > In-flight count
 - Increase when constructing async RPC requests
 - Decrease it when receiving async RPC reply

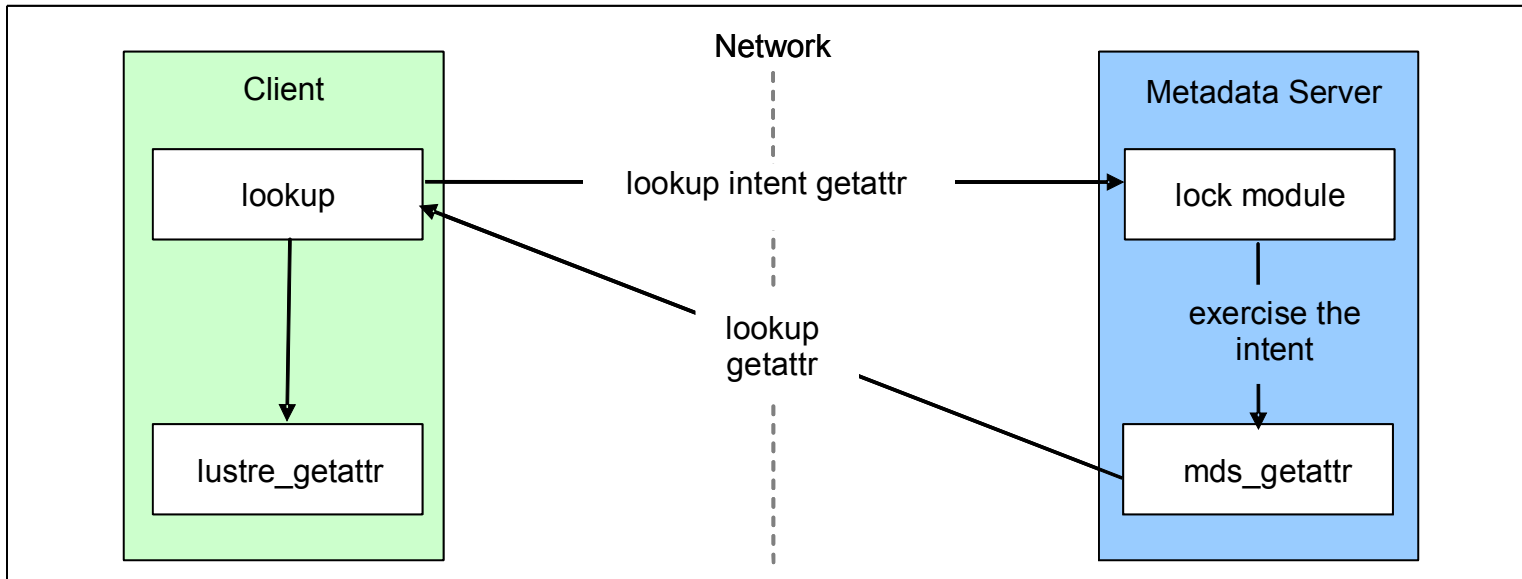
example of use fsync()



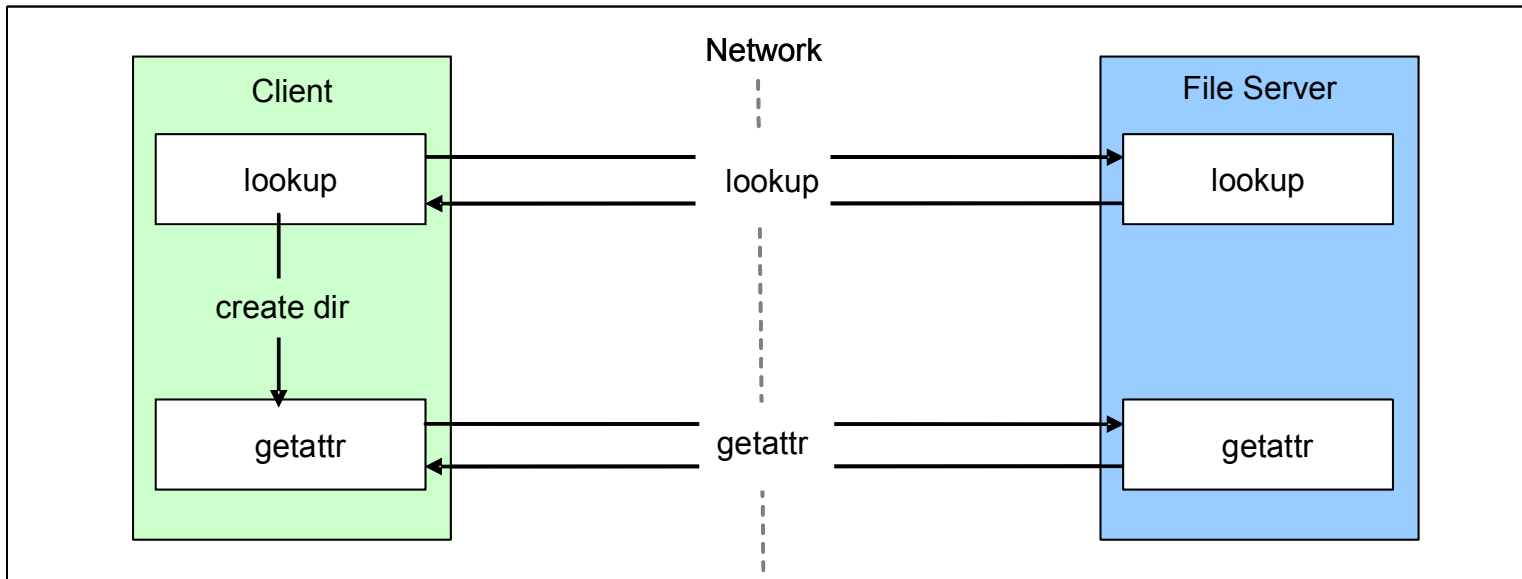
Metadata protocol

Metadata Intents

- mds_enqueue
- Client tries to acquire a lock
 - > Lock manager has a policy function which looks at intent
 - > Lock request contains a lot of information
 - lookup, getattr: returns inode & dentry with a lock
 - open: return inode, dentry, file handle and status
 - > Interpreting the returned results on the client:
 - Key question – at what phase did an error occur?
 - Disposition – client knows what executed on server
 - Was a file created remotely? Was open executed?
 - Status – error code of last operation



Lustrer `getattr`



Conventional `getattr`

VFS Changes: Intent Lookups

VFS

FS

sys_getattr
namei

intent getattr

Test if OK
no:

d_intent_release

vfs_getattr

d_intent_release



Inode lookup operation /or/
Dentry revalidate operation
FS arranges for 'getattr' locks

Release lock

Inode getattr operation (**use intent**)

Release lock

Raw metadata operations

- **mds_reint - raw operations**
 - > mkdir, unlink, rename, symlink, link, setattr
 - > Arguments: parent fid(s), name(s), attrs
 - > Execute operation on the server
 - > Only return status
 - > Do not return new namespace elements
- **Much simpler to make things stable**
 - > More server enforced security
 - > Considerably faster
- **Current patchless client has no raw operations**

Protocol - ctd

- mds_getattr
 - > Get attributes by fid
- For revalidation of attributes of open files

Locks & Handles

- Locks used during one system call
 - > During this time a lock is pinned with a refcount
 - > No long living pinned locks!
 - > Cached locks live long
- Open File Handles are long lived
 - > Handle implemented as MDS pinned dentry
- Open files and directories
 - > Have server based handles in c-s operation
 - > Will have client handles during w-b caching
 - > Open & Files with `i_link > 1` always use server handles
- Current working directories & mtpts
 - > Pinned with server handles, mtpts cannot be unlinked
 - > Still not done

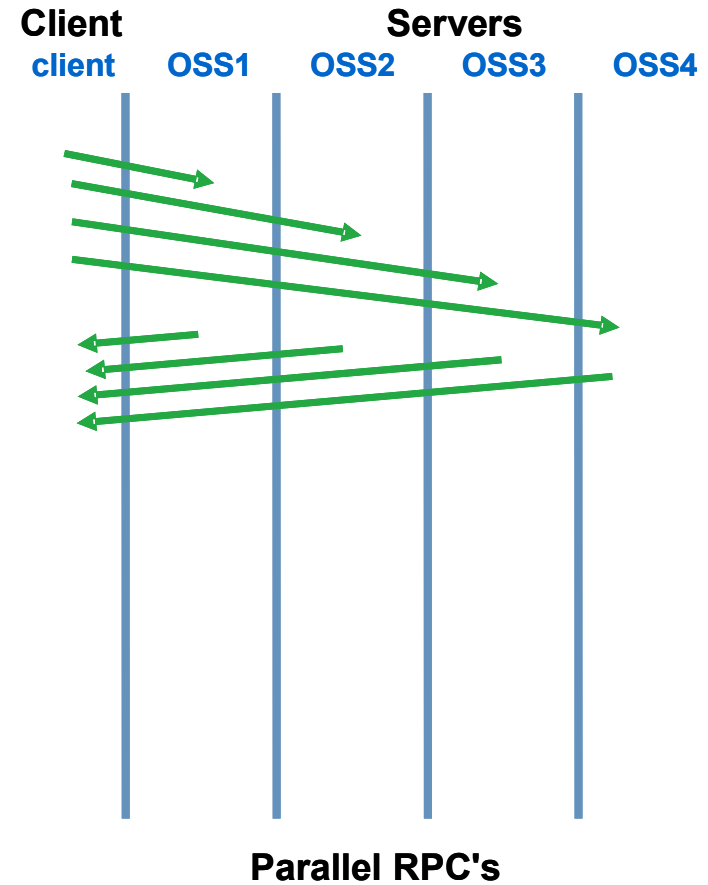
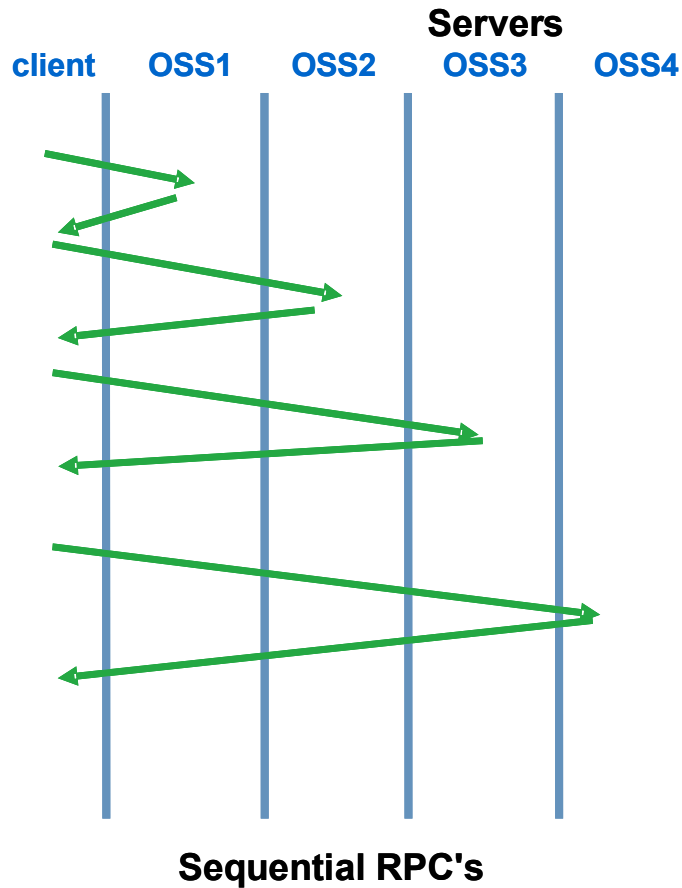
Lock bits

- Example: create many files in /lustre/dir
- Each time the client needs to lookup “dir”
 - > But “dir” is changing, because files are added to it
 - > A single lock would lead to lock bounces
- Separate the locks on dir inode for
 - > Lookup – mode, owner, ACL
 - > Updates – mtime, size, data
 - > The lock request indicates what bits it wants to acquire
- Another lock bit controls opening of files
 - > Lustre 1.4.7 has an open file cache.

Lock bits

- Example: create many files in /lustre/dir
- Each time the client needs to lookup “dir”
 - > But “dir” is changing, because files are added to it
 - > A single lock would lead to lock bounces
- Separate the locks on dir inode for
 - > Lookup – mode, owner, ACL
 - > Updates – mtime, size, data
 - > The lock request indicates what bits it wants to acquire
- Another lock bit controls opening of files
 - > Lustre 1.4.7 has an open file cache.

Parallel operations



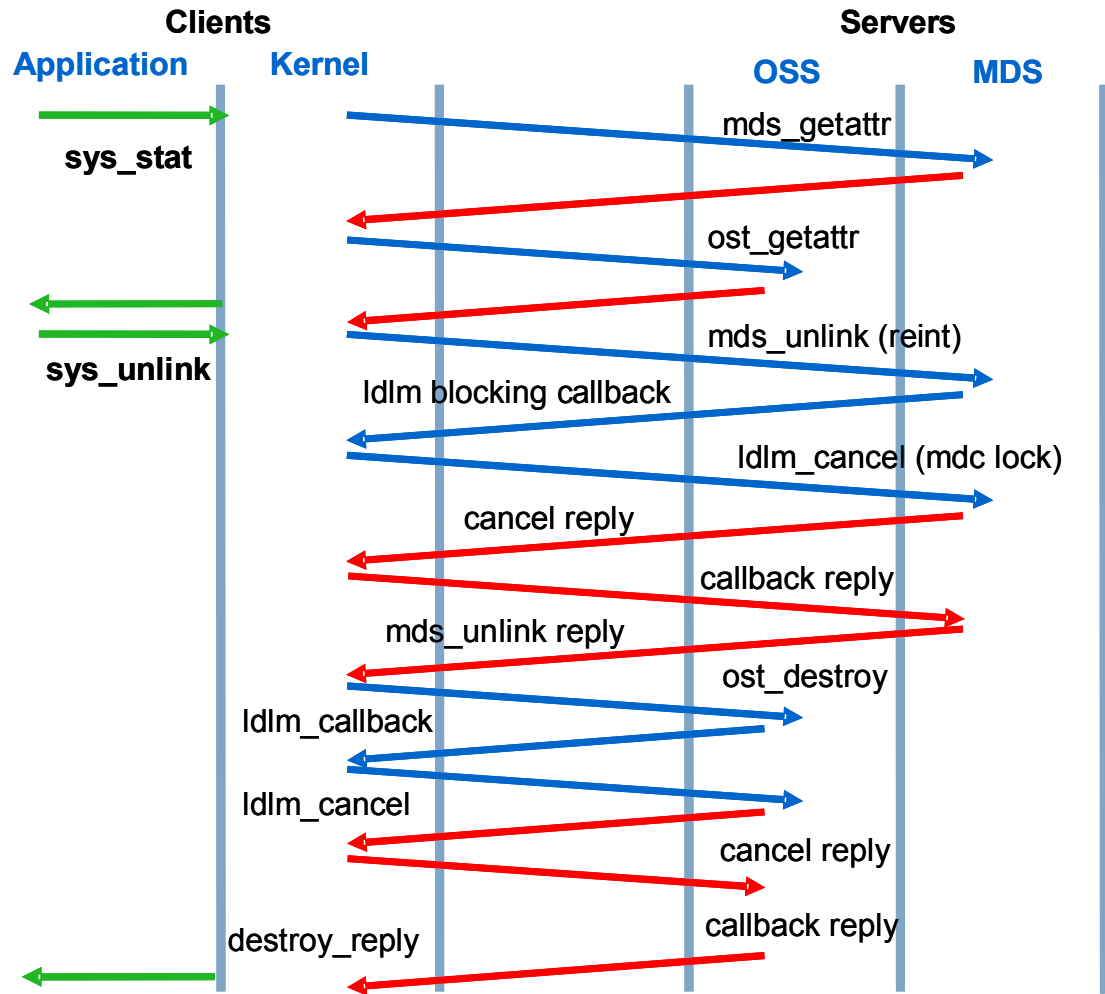
Early lock cancellation

- Clients obtain locks
 - > Sometimes it is known that they will see a cancellation callback
 - > Early cancellations eliminate the callback
- Performance improvement
 - > Serious RPC reduction
- Use cases
 - > removing files, directories
 - > rename
 - > link
 - > updating attributes
- Example GNU “rm” file utility
 - > Does a stat on the file
 - > Then makes an unlink system call

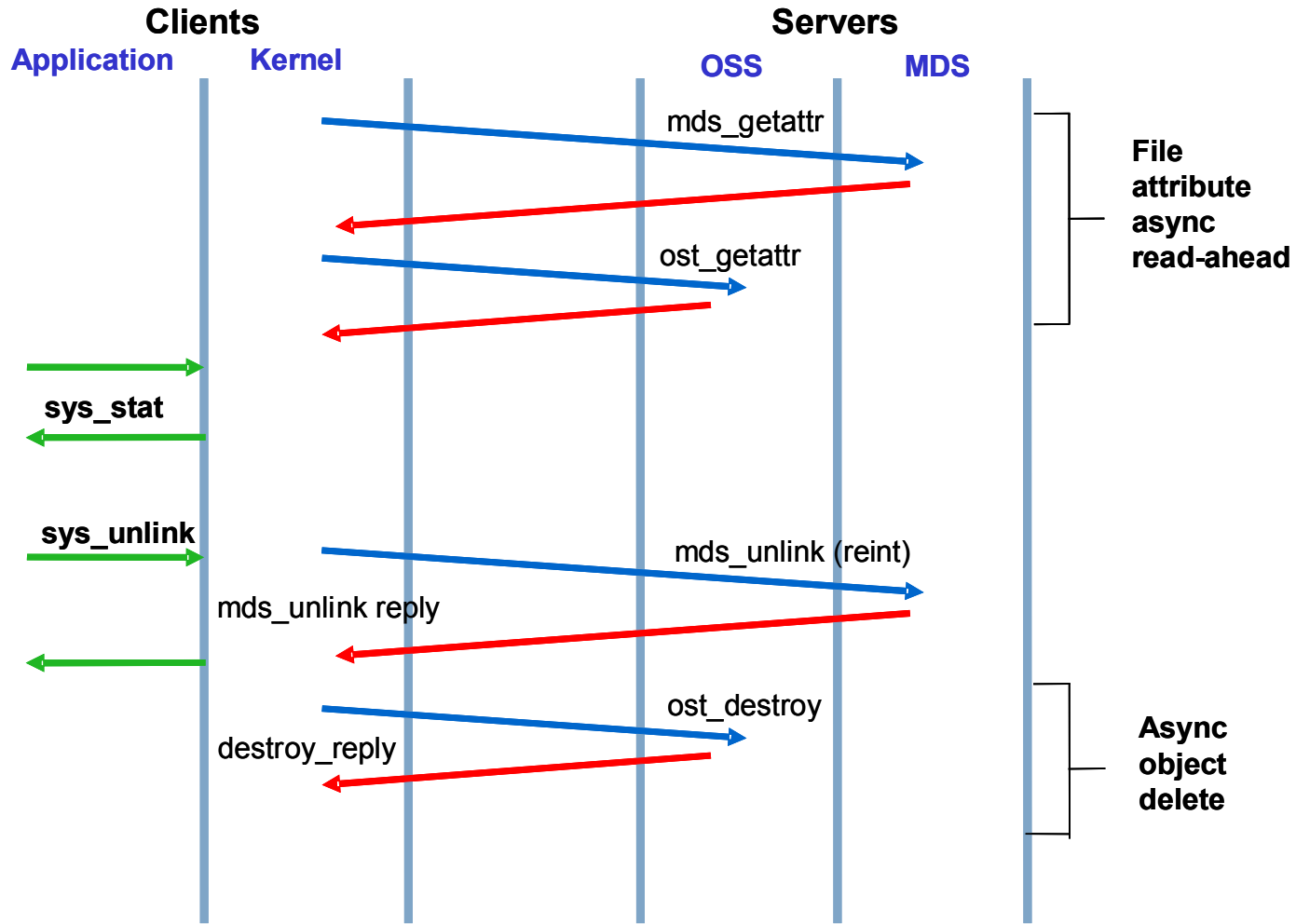
Directory & attribute read-ahead

- RPC intensive patterns
 - > Read directory pages
 - > Operate on all inodes in the page
- Attribute read-ahead:
 - > Fetch attributes from MDS and OSS in the background
- Common use cases
 - > List all files in a directory
 - readdir
 - get mds inode attributes
 - get oss object attributes
 - > Remove all files in a directory
 - readdir
 - get mds inode, oss object attributes
 - unlink inode, object

Common “rm file” – wait for 8 RPCs



Sequential RPC's



Early lock cancellation

Configuration protocol

obd devices

- Lustre consists of 3 types of modules
 - > Lustre (obd) devices – accept API calls
 - LOV, OSC, MDC, obdfilter
 - > Server devices – translate network to Lustre api
 - OST, MDT
 - > Applications – use Lustre API calls
 - lustre client and liblustre client
 - > Each device has configuration operations
 - > attach – a name is associated with the device
 - > setup – the device is made ready for use
 - > connect – the device begins to use other devices
 - Connect overloads network and local connections
 - > Of course, there is disconnect, cleanup, detach

Lustre API

- Implemented as method table for `obd_device`
 - > Reasonable amount of sharing
 - > Should split into multiple tables
- Several different kinds of methods:
 - > Object API – create objects, do IO, manage attributes
 - > MD API – metadata operations
 - > Locking – mostly embedded in object / MD api
 - > Management API – attach, setup, connect
 - > Recovery related API
- Some cleanup of this can be expected in due course

What does configuration do?

- Configuration
 - > With lconf in 1.4
 - > With mkfs / mount through the MGS in 1.6
- The MGS writes a sequence of records in a llog file
 - > The records contain commands with arguments for:
 - New device, Attach, Setup, Connect etc.
 - Every log record contains a sequence number, date & version
- Servers read this log from the disk
- Clients fetch this log over the network
 - > Nodes can also re-fetch and process all updates

More on mountconf

- Formatting a server leaves some options behind
 - > In particular a flag “first mount” and a flag “needs index”
- During a first mount of a server
 - > The server offers its configuration to the MGS
 - > The MGS adds new records to the llog
 - > The MGS tells the new server its sequence number
 - The new server updates the disk label on the partition
 - > The MGS revokes a configuration lock
 - > All other nodes in the cluster learn about the new server
- More details about this will be discussed later in the course

Llog processing

- Llogs have method tables
 - > Quite abstract to use logs very generally
- There is a general iteration function
- On clients this iteration function
 - > Reads 8K pages of the llog from the server
 - Same function that reads directory data!
 - > Processes the records in the pages
 - > Continues (as long as the callback function exits with 0)

Configuration lock

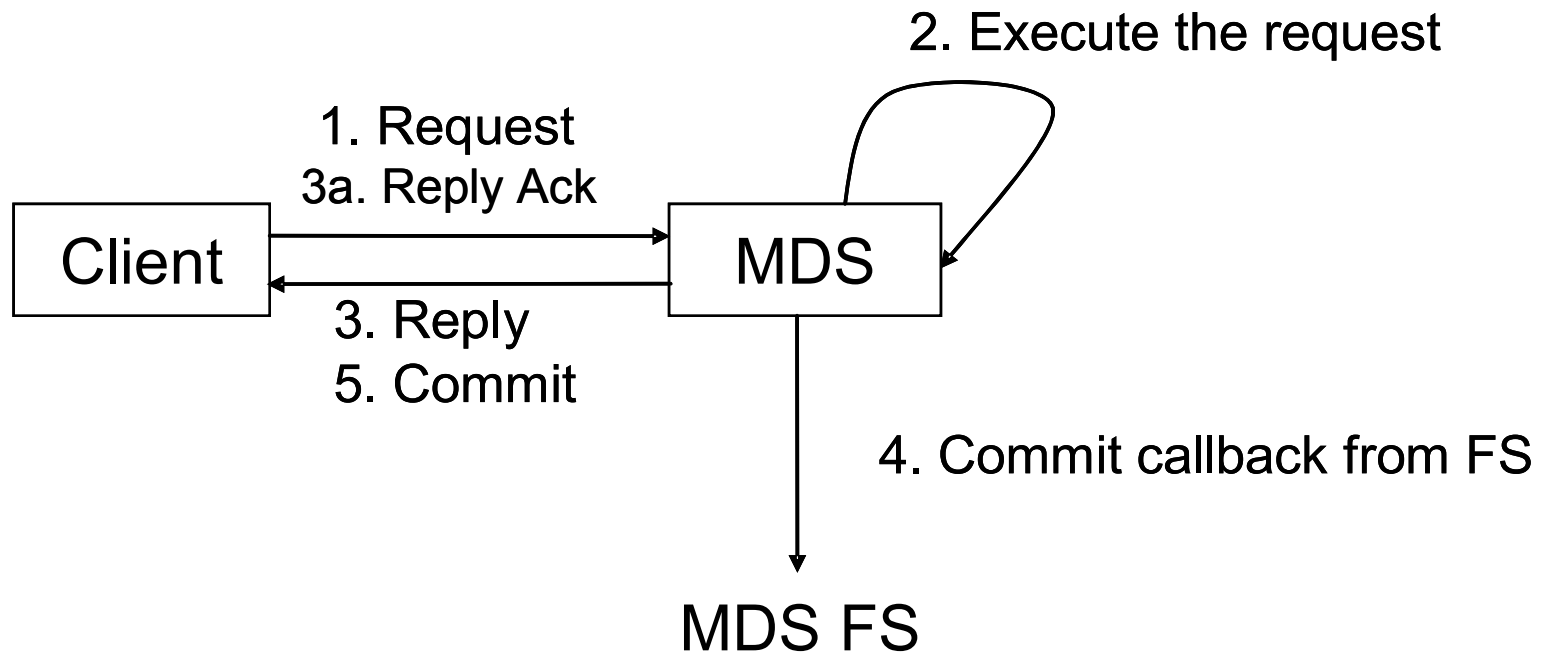
- The central configuration server is the MGS
- When a client fetches a log it also gets a lock
 - > The lock gets callbacks when the configuration changes
- Callback triggering events
 - > Online addition of OST devices
 - > Setting timeouts is global now
 - > Many others will follow (snapshots, policy dispatch)
 - > More robustness fixes

Lustre Recovery

Aspects of Recovery

- Disk file systems need recovery
 - > Not discussed here
- Routers may crash - must be avoided
 - > Not discussed here
- Servers roll back when they crash
 - > Clients will try to rebuild the server state
- Open (maybe unlinked) files need re-opening
- Inode - Object relationship can go bad
 - > Objects without file inodes need cleanup
- File size synchronization from OSSs to MDS
- Clients can crash - servers need to clean up state
- Recovery aims to restore globally consistent state

Request lifecycle



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
 - > Other CFSs commit synchronously
 - > Lustre MDS can lose some transactions
 - > Clients need to replay in precisely same order

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
 - > Purpose: client can compensate for lost trans
- MDS has disk data per client
 - > Last executed request, last reply information

What happens during recovery?

- Clients reconnect to the MDS
- MDS reports the last transno it committed
- Replay - resending requests that have replies
 - > Clients resend requests including transno's
 - > MDS merges & sorts them to get correct sequence
- After replay only a few requests remain
 - > Client has not seen a reply
 - Reply was lost - OR - request never executed
 - > Resend phase:
 - MDS reconstructs replies (using MDS disk data for client)
 - MDS re-executes requests that were lost

Delicate issues - replay

- Replay gaps
 - > Clients offer transaction sequence to servers during replay
 - > There can be gaps in the sequence, for two reasons
 - Clients are missing and fail to offer transactions - BAD
 - The gaps are cases where replies are missing – BENIGN
 - > 100% correct replay requires all clients to join
 - During restart the server waits for clients to join
- Why acknowledge replies?
 - > Client A creates dir D - MDS sends A fid(D)
 - > Client B creates file F in fid(D)
 - So client B can only replay if fid(D) will be re-created
 - > Lustre 1.8 will get past this problem
 - Clients will generate fids

Delicate issues - replay2

- Re-opening files
 - > Opening “foo” requires permissions
 - > After the open permissions can change
 - > During re-opening in recovery must “force” re-open to work
 - This is an open bug in Lustre
 - There is not even a test that shows that it fails
 - `open(“foo”); chmod(“foo”, 0)`
- Open, unlinked files
 - > On a normal FS both the application and the FS crash
 - Nothing to recover
 - > In a cluster FS if the server crashes
 - The client will want to re-open the open unlinked file
 - Open unlinked files must be retained in ORPHAN directory

Version recovery

- Clients want to revalidate their caches
 - > Always, even after disconnection
 - > Answer: introduce a version of files
 - Any update, except atime updates change the version
 - > Even after long disconnections clients can verify the cache
- Version based recovery
 - > Allow much later, out of order replay
 - > Provided the versions have not changed
 - > Key to re-integration with disconnections
 - > Under development at the moment
- Version based recovery can lead to incomplete replay
 - > But with “commit on share” we regain correctness

Recovery Refinements - commits

- **Nothing to replay**
 - > Any failure combination is possible (many clients & servers)
 - > Nothing to replay - commit everything
- **Commit everything**
 - > Is very slow
 - > Hurts single client performance
 - > With more WB caching in the client this may be OK
- **Commit on share**
 - > Before sharing data - commit it
 - > Now single client performance is good
 - > Short of unlikely races any failure can be handled

Delicate issues - resending

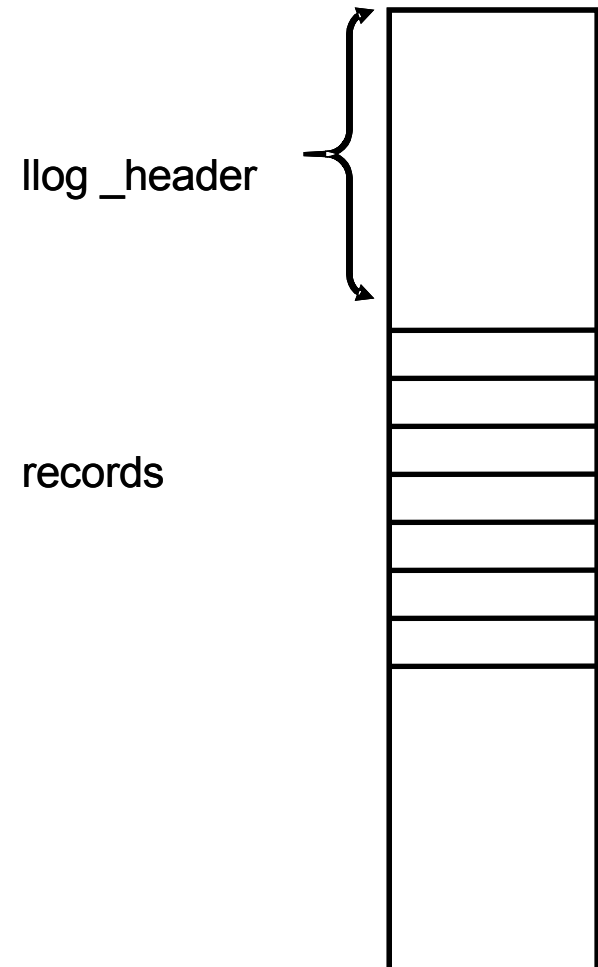
- Resending - this means no reply received
- Transaction may have executed – or not
 - > The xid is re-sent by the client, last executed xid is known
 - > Stored in per client export information in last_rcvd file
 - No server crash: The last executed xid
 - After server crash: The xid of the last committed request
- If the transaction executed
 - > MDS performs reply reconstruction
 - Currently this uses reply data stored on disk
 - > Re-doing the transaction doesn't cut it
 - E.g., how to recover previous failure or success for `open("foo" , O_CREAT | O_EXCL)`

Log recovery

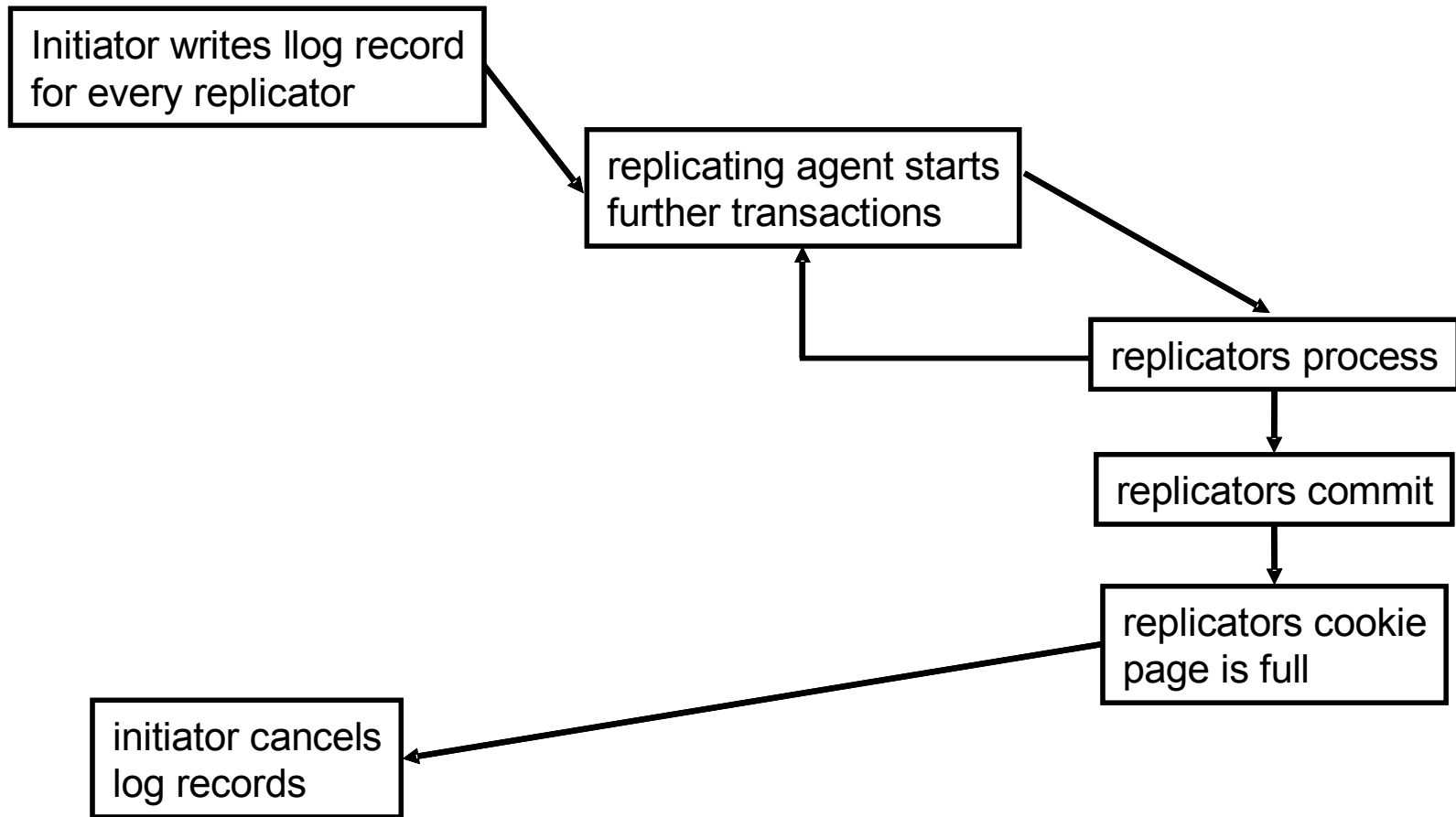
- For distributed transaction commits
 - > Unlinking a file and destroying its objects
 - > Updating the file size on the OSTs and on the MDT
- Terminology
 - > Initiator – where the transaction is started
 - > Replicators – other nodes participating
- Normal operation
 - > Write a replay record for each replicator on the initiator
 - > Cancel that record after the replicators commit, in bulk
- Recovery
 - > Process the log entries on the initiator

llog disk structure

- llogs are append only logs
- 8K header, mostly bits
 - > 64K records, typical size: some MB's
- llog_add
 - > set bit, write record, within trans
 - > writes 2-3 blocks
 - > But usually coalesce in a transaction
- llog_cancel
 - > Clear the bit
- When all bits cleared
 - > Remove the llog file
 - > Many cancellations – one block



replicated transaction execution

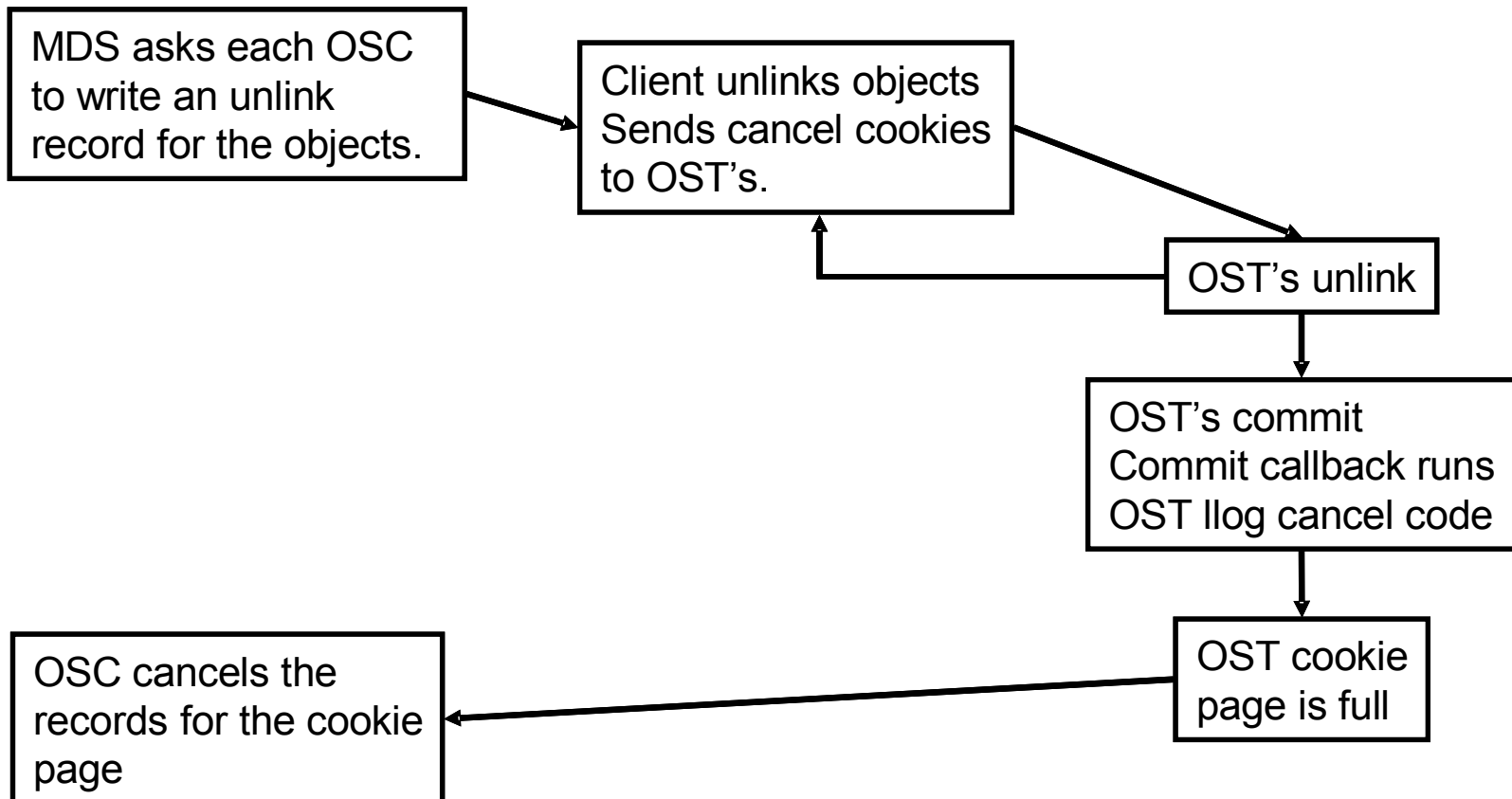


File and Object Removal - example

MDS - originator

Client - replicating agent

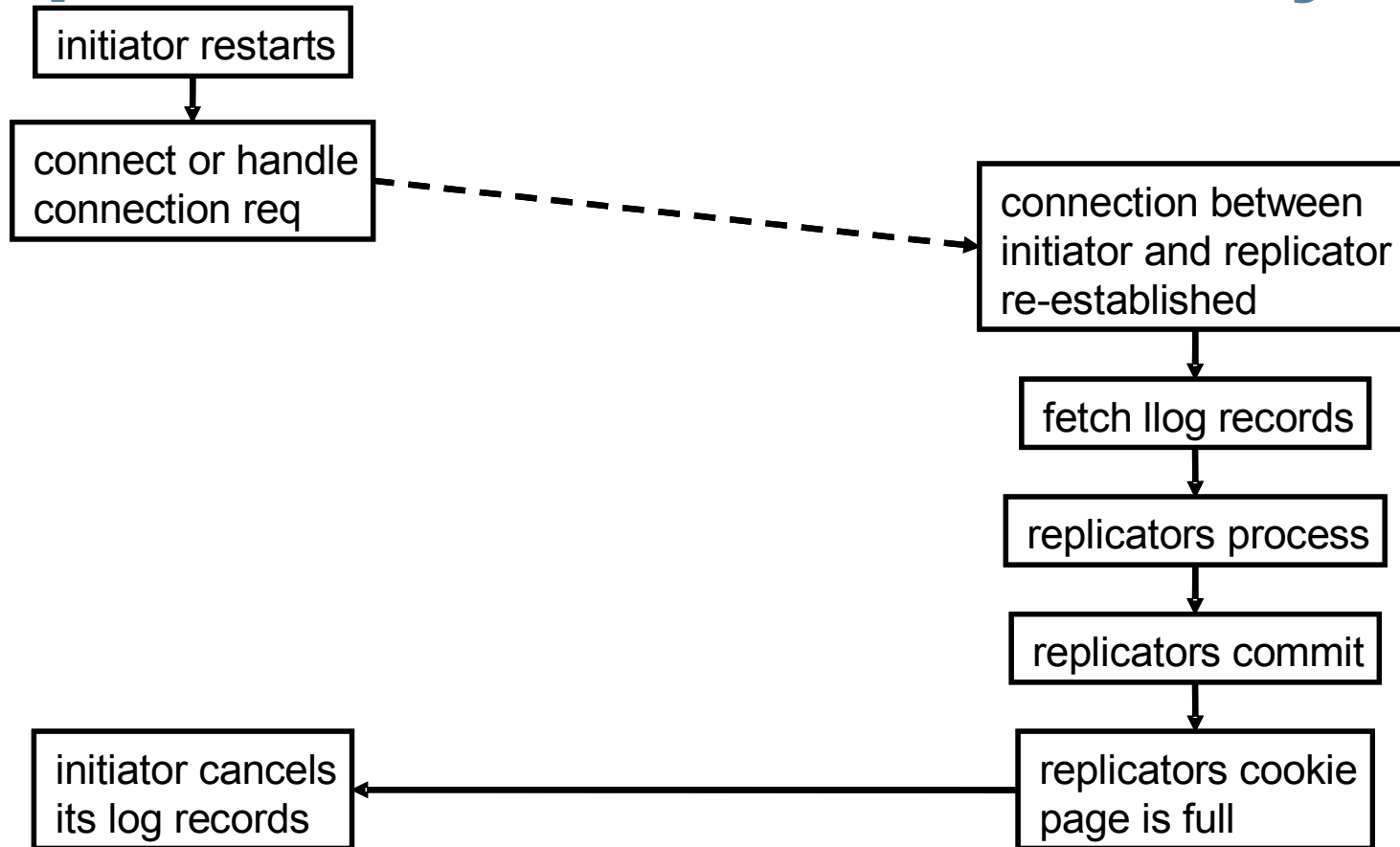
OSSs - replicators



llog – a complex API

- Have a context
- The context has methods, often overloaded:
 - > Originator cancel method cancels disk records
 - > Replicator cancel method sends cookies to the originator
 - This is done during a commit callback on the replicator
 - The originator cancels the records on disk upon receipt
- An initiator context has a storage context (disk)
- An initiator needs a connection with a replicator
 - > Unlink recovery- the OSC's on the MDT are the originator
 - Other arrangements would expose striping info to the MDS
 - > The OSC's storage context is on the MDT

Replicated transaction recovery



E.g. - initiator is MDS with unlink llog records for file inodes

E.g. – replicators are OSS's with objects that were not yet removed, but should have been

OST & MDT disk layout

MDT disk file system layout

- **ROOT/**
 - > This is the real namespace, with only Lustre files
- **PENDING/**
 - > Holds unlinked files which are still open somewhere in the cluster
- **LOGS/**
 - > Hold config and MDS/OSS recovery logs
- **OBJECTS/**
- **last_rcvd**
 - > Header contains the last transaction number for the entire MDS
 - > Also contains one entry per client, includes a UUID, last transaction number for this client, last XID, and reply-reconstruction data
- **lov_objids**
 - > For each OST, one 64-bit object number -- the last objid that was allocated to a created file

OST disk file system layout

- O/
 - > Contains objects laid out in object groups
 - > Each group has a 32-directory hash
 - d0/ through d31/
 - > Each group also contains a LAST_ID file (last objid given to MDS)
 - > b1_4 uses:
 - Group 0 for “real” objects
 - Group 1 for echo_server objs
 - Group 2 for recovery objects
 - > CMD and writeback MD cache also use groups
 - Each CMD server and each client with a WB cache gets a group
- Example real file system object #328:
 - > O/0/d8/328
- Example echo-server object #328:
 - > O/1/d8/328

OST disk file system layout (ctd)

- last_rcvd file
 - > Contains a “UUID” (which is not really a UUID but should be)
 - > Contains last transno for this OST
 - > Contains per-client UUID, transno, last XID
 - > Has no reply-reconstruction info on OST
 - Write is synchronous; read and destroy are idempotent
- CATALOGS/
 - > Contains llogs of llogs
 - > Each llog gets an entry in a catalog



THANK YOU