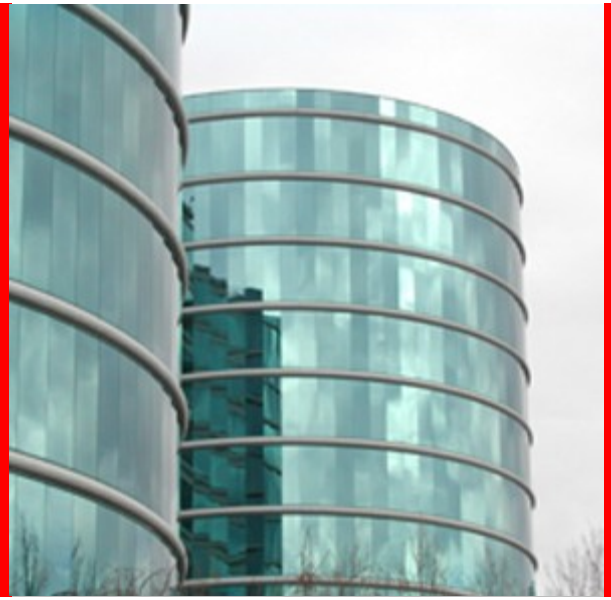

ORACLE®



ORACLE

Deep Dive into Lustre Recovery Mechanisms

Johann Lombardi
Lustre engineering

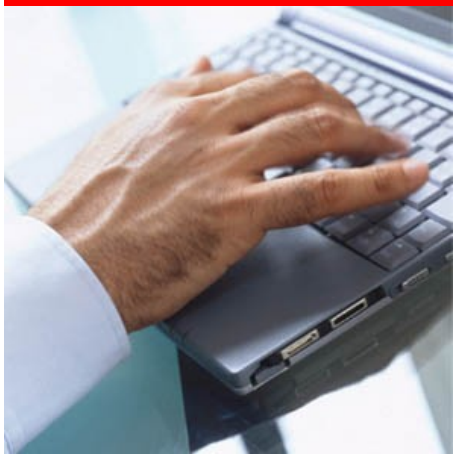
Possible failures

- Network failures
  - Packet loss
  - Can be either a lustre request or reply
- A lustre client or server is down
  - HW problems, SW crash, reboot, power outage, ...
- Distributed state inconsistencies
  - State of multiple nodes out of sync

Aspects of Recovery

- Connection recovery
  - Pings, evictions, lost requests or replies
- Persistent state recovery
  - Server failure, journal replay & lustre replay
- Replicated state recovery
  - State of multiple nodes out of sync

Agenda



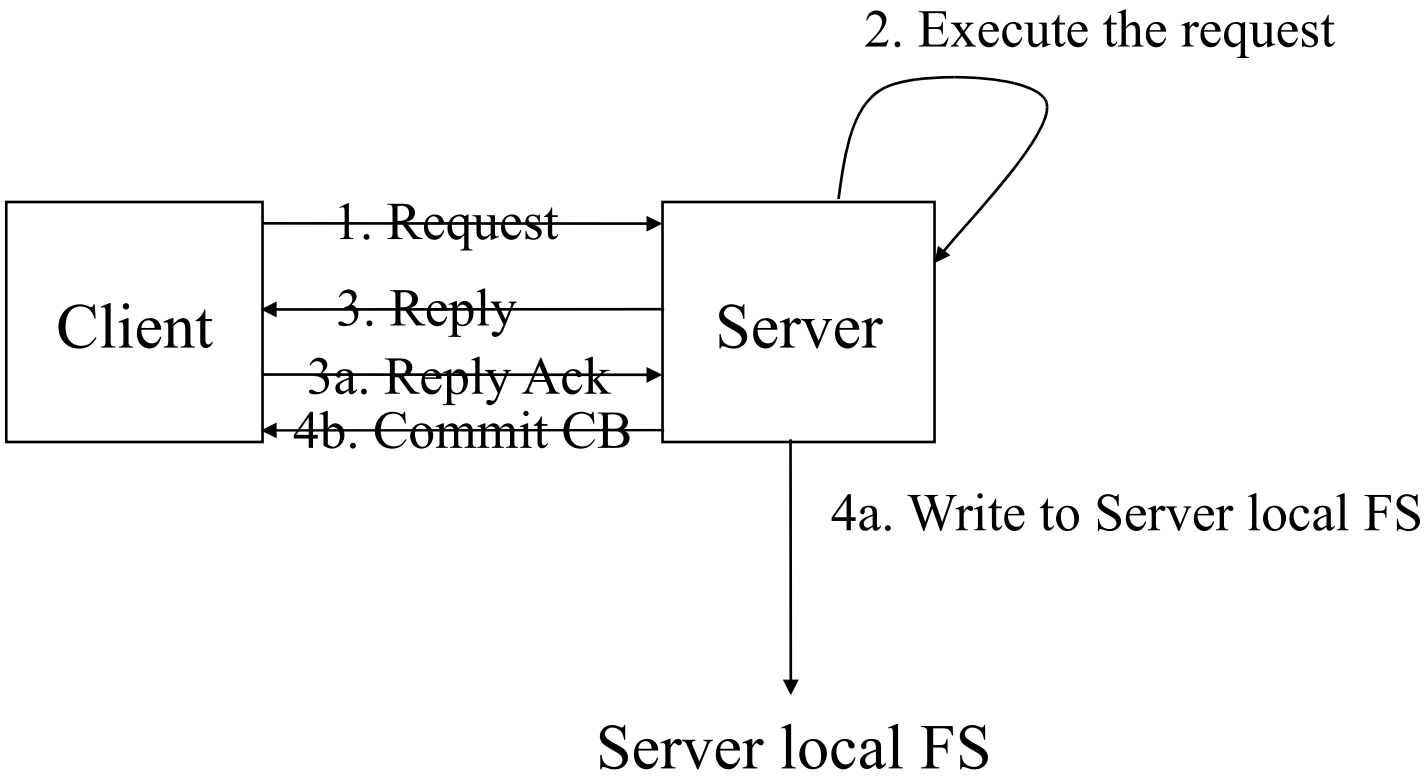
- Request Execution Flow
- Request Resending
- Request Replaying
- Version Based Recovery
- File I/O Recovery
- Distributed state recovery

Request Execution Flow

Server Execution

- Server 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

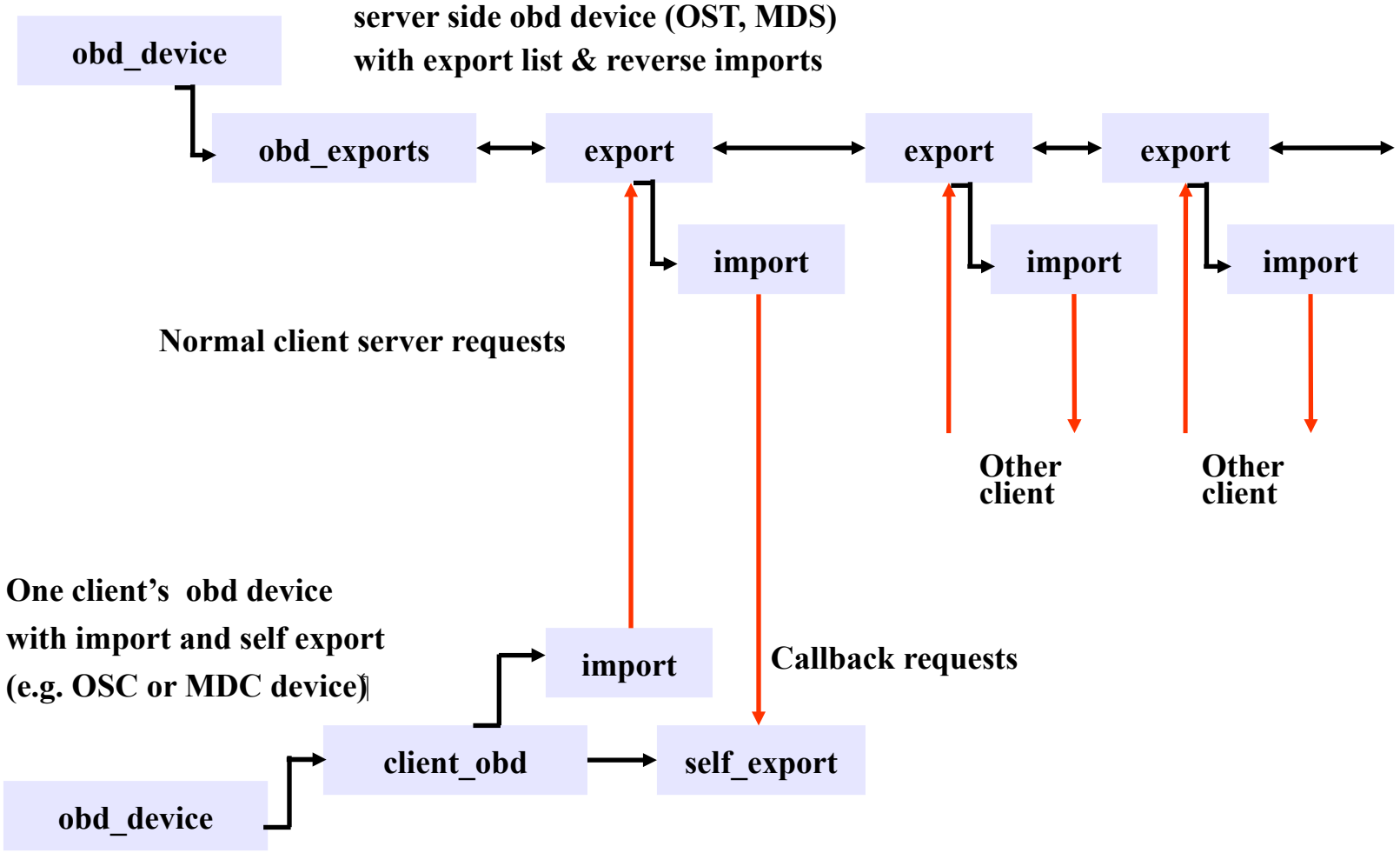
Request Lifecycle



Client/server interaction

- Send request
- Request is allocated a transaction number (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
- Each server has disk data per client
  - Last executed request, last reply information

Import/Export for RPC's

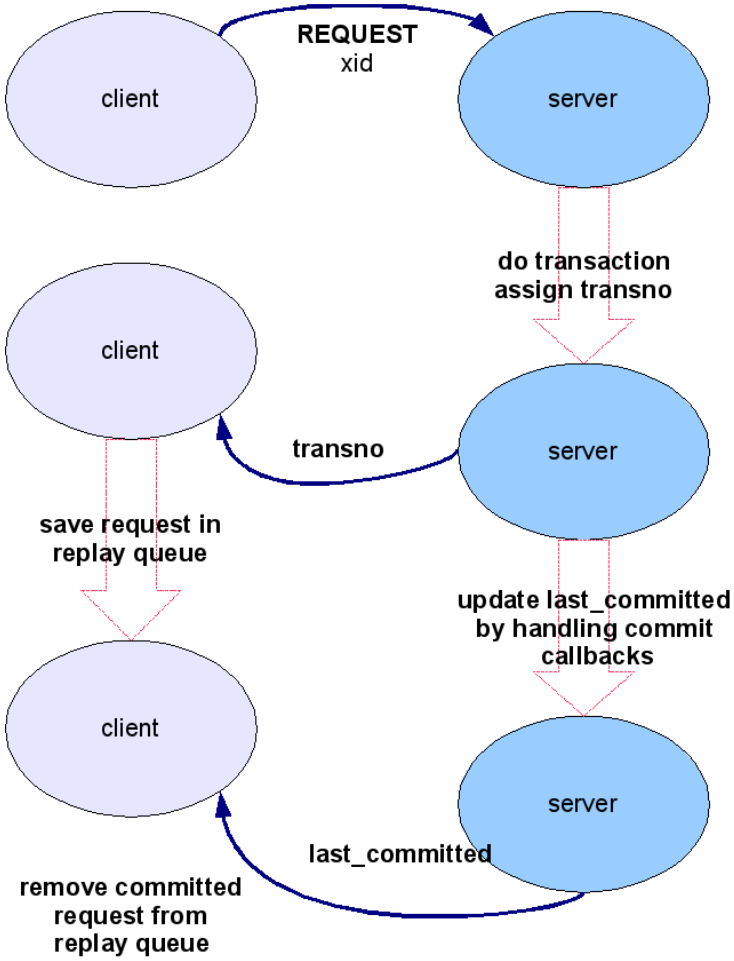


Xid & transno

- Xid
  - Each ptlrpc request is tagged with a specified xid
  - Xid is unique for an import/export pair
- Transaction number (transno)
  - Server side transaction number
  - unique per target
  - Servers record on disk last committed transno for each client

Normal Operation

- Client sends request
  - Unique xid
- Server replies
  - Transno allocated
  - last_commited transno
- Client gets reply
  - Put request in replay list
  - Remove from replay list all reqs with transno <= last committed



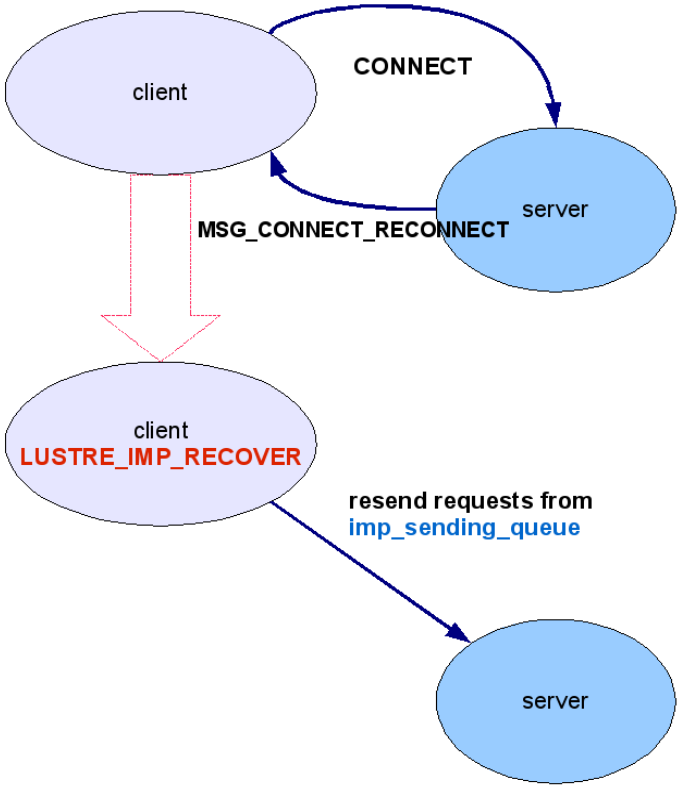
Request Resending

Resending

- Client has not received any reply
- What happened?
  - request OR reply was lost
- Request may have executed or not
- Server determines whether it has already processed this request
  - Original request was lost and never reached the server
    - Server executes the request
  - Request was processed already but reply was lost
    - Server reconstructs the reply
    - Only done on MDT
      - Request processing idempotent on OST

Resending (cont'd)

- Client sends connect and gets reply
- LUSTRE_IMP_RECOVER
- Resend requests from sending list



How does MDT know if a request has been executed already?

- MDT stores last executed xid
  - Works because each client has <= one RPC in flight
  - MDT processes request in xid order
  - Stored on disk in the last_rcvd file
- if req xid == last processed xid
  - Request has already been processed
  - Reconstruct the reply

Reply reconstruction

- Only done on MDT
  - No reply reconstruction on OST
  - Request processing is idempotent on OST
- Use reply data stored in last_rcvd file
  - store Information about each client
  - lsd_client_data structure
    - 1 separate slot for MDS_CLOSE request
    - 1 slot for all the other request types
    - Store results of RPC processing

Request Replaying

Persistent State Recovery

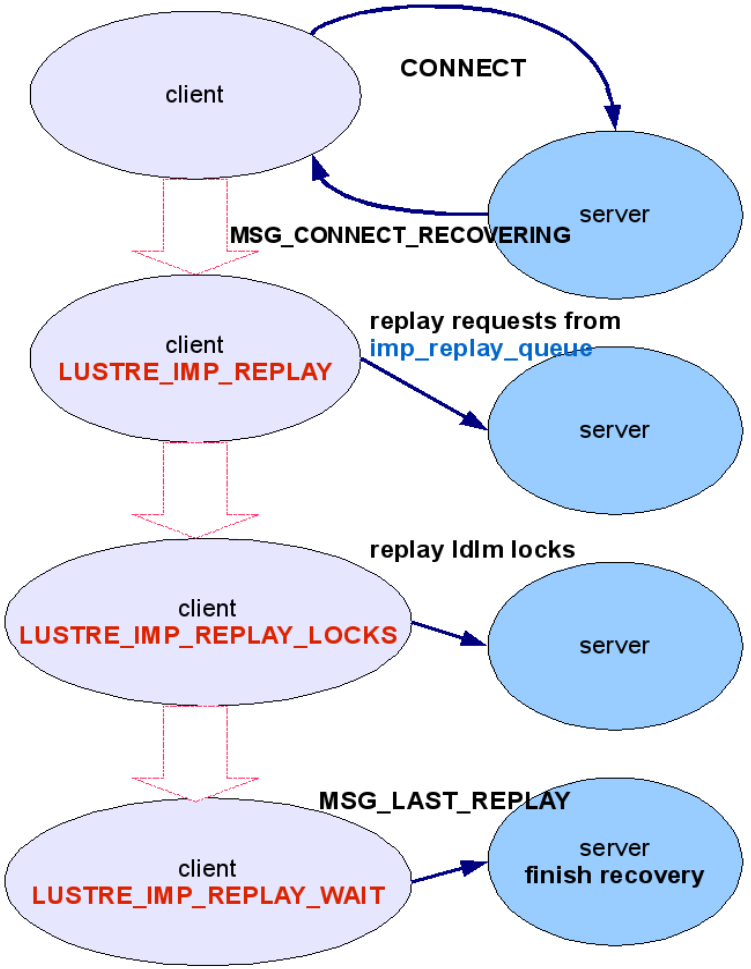
- Server restarts after uncleaned shutdown
  - SW crash, HW failure, power outage ....
- Disk filesystems need recovery
  - ldiskfs journalling
- Servers roll back when they crash
  - server can thus lose some transactions
  - Rely on client to rebuild server state before crash
- Clients keep request & reply
  - Until server confirms a disk commit
  - Purpose: client can compensate for lost transactions

What happens during client recovery?

- Clients reconnect to the server
- Server reports the last transno it committed
- Request Replay – resending requests that have replies
  - Clients resend requests including transno's
  - Server merges & sorts requests to get correct sequence
  - Server re-executes requests in transno order
- Lock Replay
- After replay, only a few requests remain
  - Requests for which client has not seen a reply
  - Resend phase

Replay

- Client sends connect and gets reply
- MSG_CONNECT_RECOVERING is set
  - Get last committed from reply which is starting point for replay
  - LUSTRE_IMP_REPLAY
    - Clients replay requests
  - LUSTRE_IMP_REPLAY_LOCKS
    - Clients replay locks
  - LUSTRE_IMP_REPLAY_WAIT
    - Clients send MSG_LAST_REPLAY
  - Continue with resend



Server side

- List of clients from last_rcvd file
  - Calculate last_committed transno
  - Set next_replay_transno to last_commit + 1
- Server orders replays by transno
  - Check if replay transno == next_replay_transno and execute it, otherwise put replay request to waiting queue

Gap in transaction sequence

- Clients offer transaction sequence to servers during replay
- There can be gaps in the sequence
  - Some clients are missing and failed to offer transaction
- Correct replay requires all clients to join
  - During restart, server waits for clients to join
  - No new clients are allowed to connect during recovery window

Recovery Window

- Clients have to reconnect within dedicated recovery window
  - No new clients allowed to connect
- Server starts recovery window when first client reconnects
  - Adjusted as the clients reconnect
    - Clients report request service time
- After recovery times out
  - if not all clients have reconnected
    - clients are evicted (1.6)
    - clients with version mismatch are evicted (1.8)

Example: Open-unlinked files

- On a local filesystem: both application & fs crash
  - Inode put in the orphan list and is destroyed during filesystem recovery
- On a cluster filesystem, if server crashes
  - Applications on client nodes are still alive
  - Client nodes want to reopen open-unlinked files
  - Open unlinked files must be retained in a separate directory (PENDING dir on MDT)

Version Based Recovery

Motivation

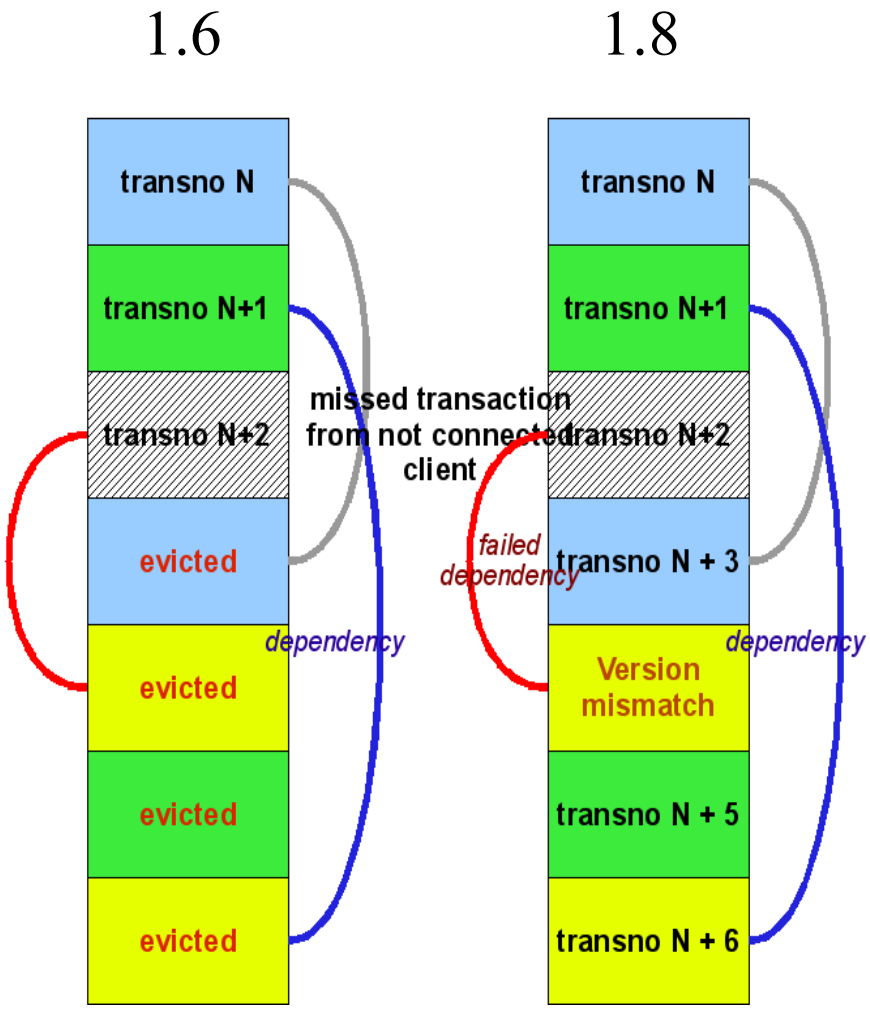
- Recovery requirements are too strict
  - All clients have to reconnect within specified window
  - No gaps in transaction sequence are allowed
    - Not aware of transaction dependencies
- Objectives
  - Relax requirements
    - Allowing gaps
    - Allowing clients who missed the recovery window to rejoin
      - support for disconnected operations in the future

Version Based Recovery Primer

- Use inode versioning
  - stored on disk
  - set to last transno which modified the inode
- Server replies include
  - operation transno (as before) which is the new inode version
  - BUT also the pre-op version of the inode
- Clients provide this pre-op version during replay
- Transaction is replayed only if the inode version matches provided pre-op version

Gap in transaction sequence

- Client can recover even if gap in transaction sequence
  - Only clients with version mismatch are evicted
- Client can recover later after recovery window is finished
  - Client reintegrates fully if versions match
  - Make disconnected operation possible

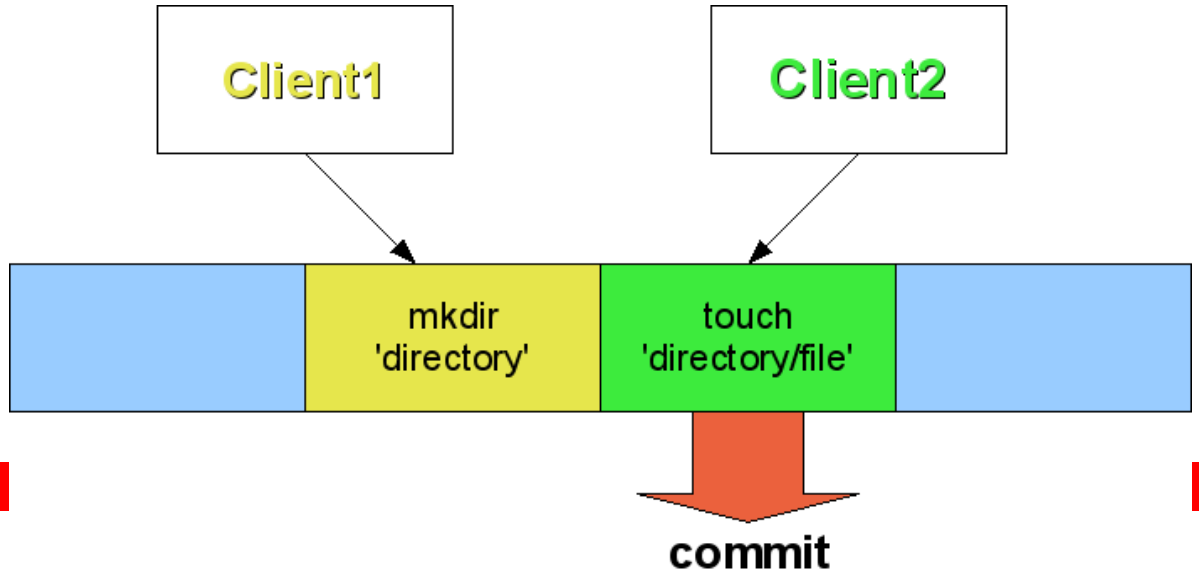


Delayed Recovery

- Clients who missed the recovery window allowed to replay
  - Provided that versions have not changed
- Delayed recovery is not working yet
  - Inode versioning key feature to support this
  - But still some problems to address
    - e.g. Orphan recovery

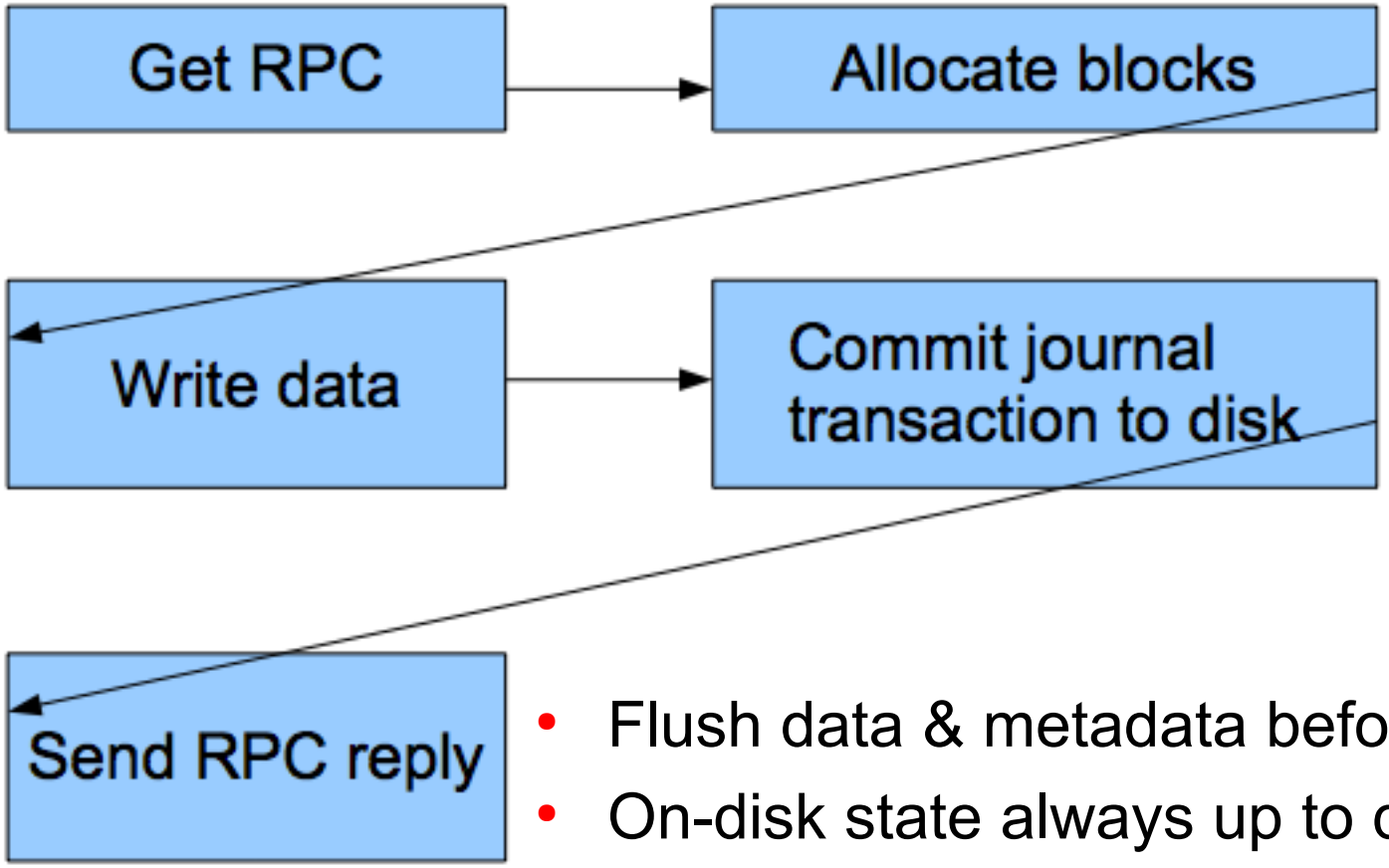
Commit On Share (COS)

- Version based recovery allows to recover as much as possible with reconnected clients
  - one of the original goals
- **but** can lead to incomplete recovery
- With Commit On Share we regain correctness
  - Eliminates dependencies between clients by committing immediately
  - Commit removes depended replays from clients
  - COS only available in 2.0, not in 1.8



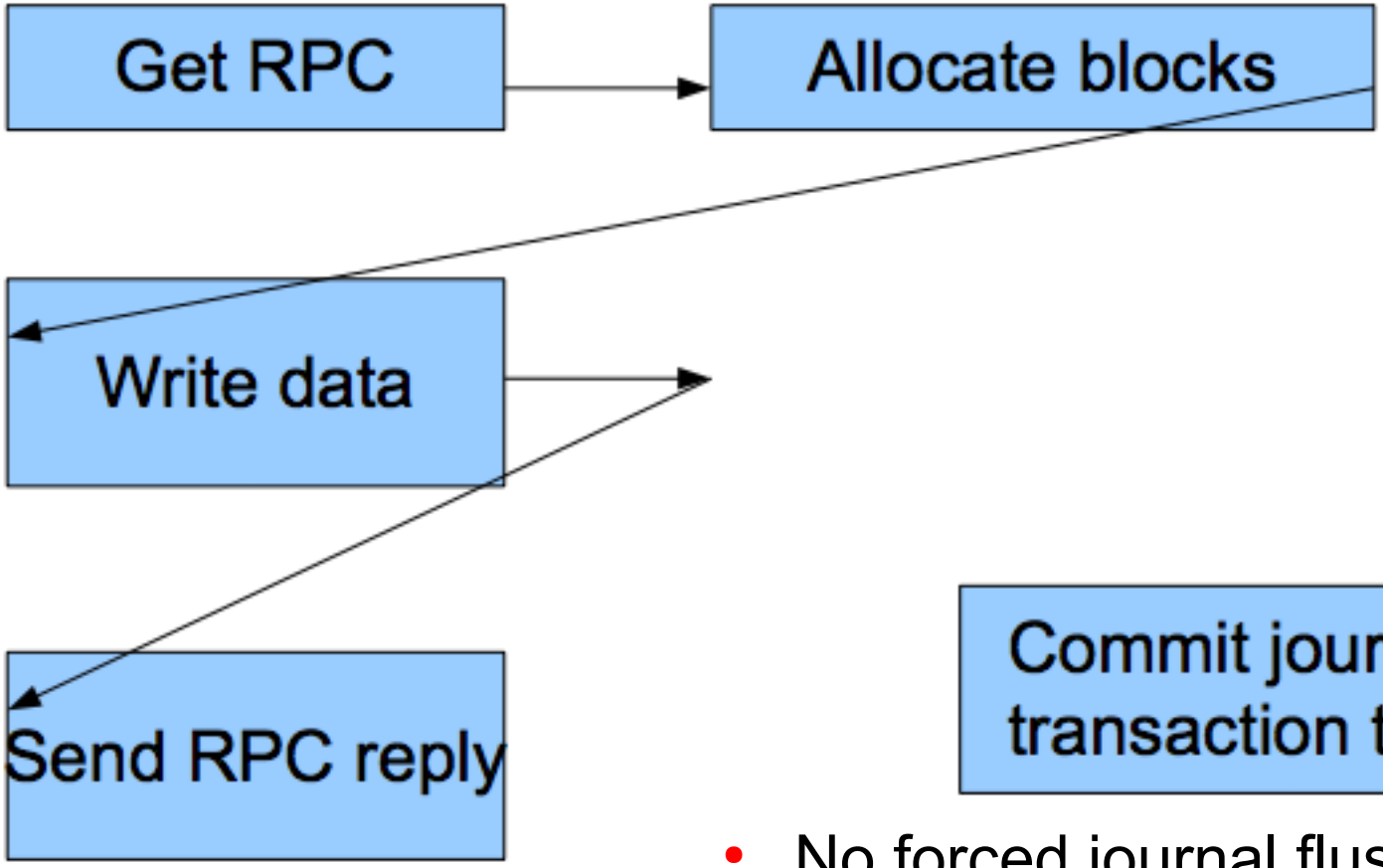
File I/O Recovery

OST Bulk Write Handling



- Flush data & metadata before replying
- On-disk state always up to date
- No need to replay write requests

OST Bulk Write Handling with async journal commit feature enabled



- No forced journal flush any more
- Less disk seeks

Recovery aspects

- Data still written synchronously but metadata updates are now asynchronous
- **Consequence**: bulk writes have to be replayed by clients now
- Clients have to keep copy of written data in page cache
  - Until OSS confirms metadata updates have hit the journal on disk
  - Only possible with cached I/O, not with direct I/O

Flush journal on lock cancel

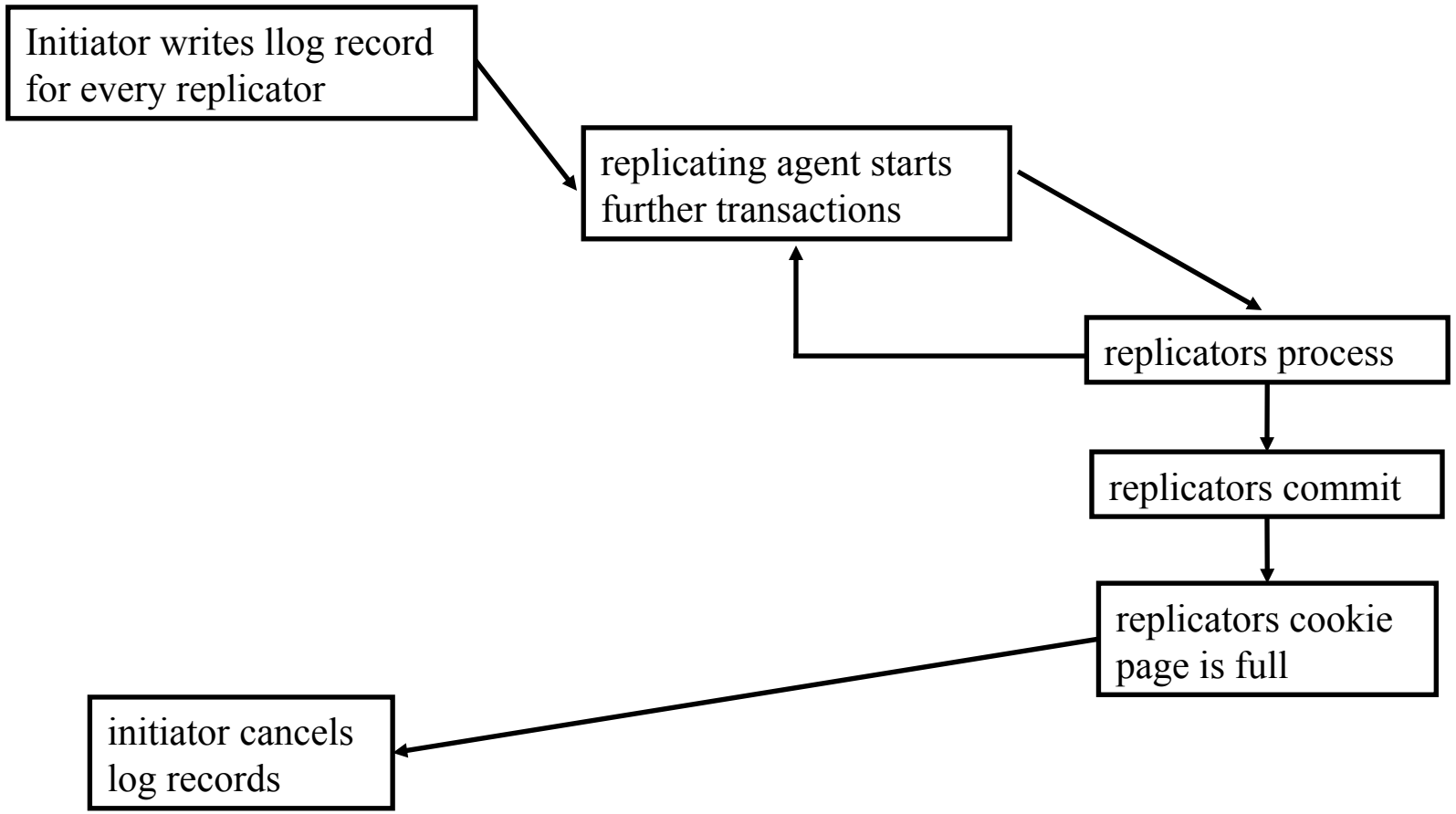
- Problem:
  - Client holds extent lock and receives blocking AST
  - Client flushes dirty data protected by extent lock
  - Client sends lock cancel to OSS
    - Client no more able to replay bulk writes
  - OSS crashes
    - Cannot recover state before crash
- Solution:
  - Flush journal on lock cancel
  - Problem with excessive stack consumption
    - fixed in 1.8.3
  - Procfs tunable sync_on_lock_cancel

Distributed state recovery

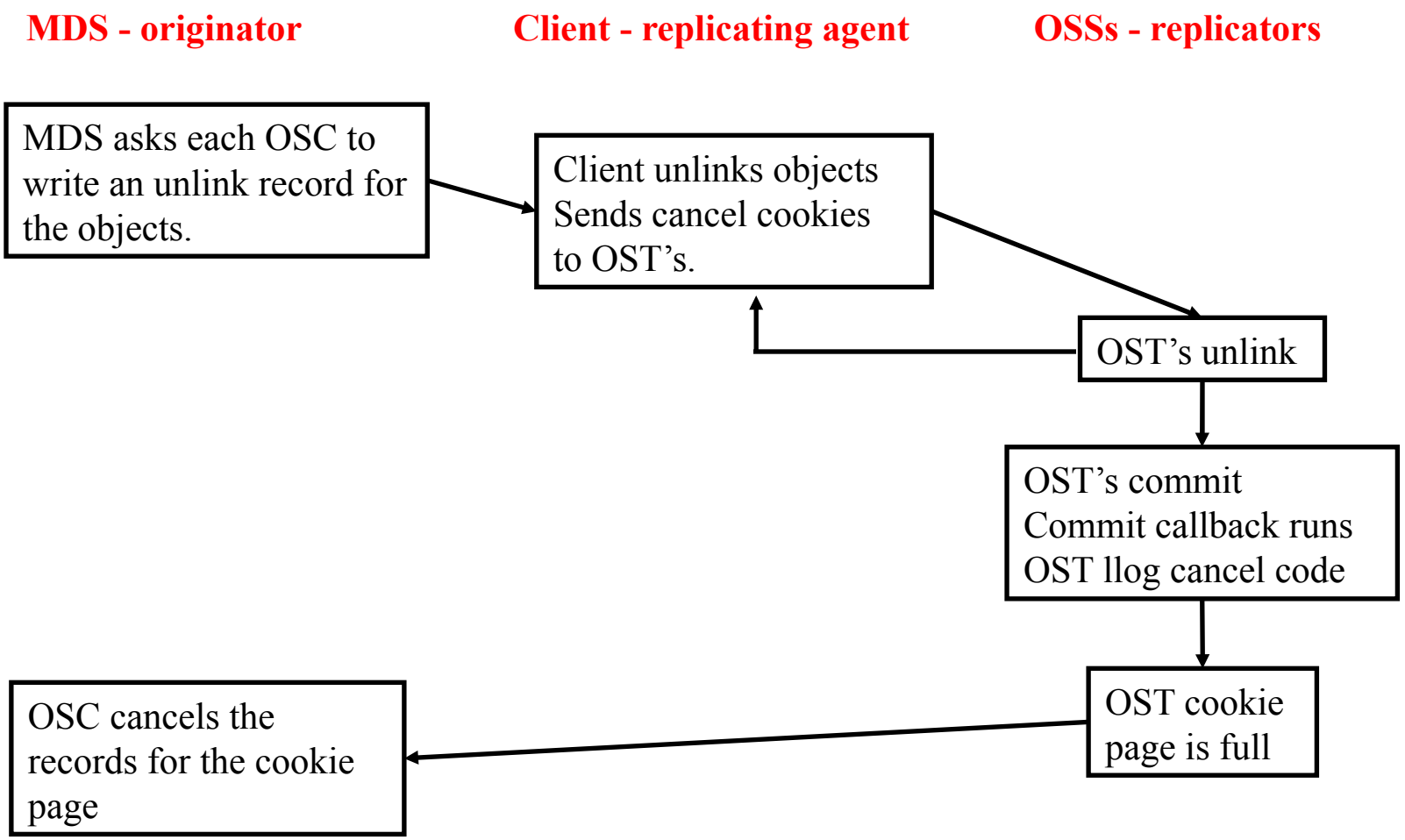
LLOG

- For distributed transaction commits
  - e.g. unlinking a file and destroying its objects
- 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

Replicated transaction execution



File & Object Removal - example



The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE IS THE INFORMATION COMPANY