Immediate Write Mirroring Design

1. Overview & Design Summary

1.1. FLR Today

In the current implementation of FLR (introduced in Lustre 2.11), the system uses a "delayed write" approach for maintaining file mirrors. This means:

Writing to Mirrored Files: When a client writes to a mirrored file, only one primary (preferred) mirror is updated directly during the write operation. The other mirrors are simply marked as "stale" to indicate they're out of sync with the primary mirror.

Today, we have manual synchronization: after a write, the lfs mirror resync command must be run to synchronize the stale mirrors with the primary mirror. This command copies data from the synced mirror to the stale mirrors and removes the stale flag from successfully copied mirrors.

Layout state and staleness is managed through a careful series of layout state changes which are described File-level replication state machine in CoreDesignConcept in this document.

This delayed write approach was implemented in the first phase of FLR to avoid the complexity of maintaining consistency across multiple mirrors during concurrent writes. By updating only one mirror during writes and marking others as stale, the system maintains a consistent view of the file data, at the cost of requiring explicit synchronization after writes complete.

The current implementation does not provide immediate redundancy, since only on a single mirror is up to date until an explicit resync operation is performed. This approach enables things like hot pools synchronization, but the lack of immediate write redundancy severely limits the use cases.

1.2. Immediate Mirroring

We have a requirement to do immediate mirroring on Lustre files, where all writes (and related ops) are replicated to multiple mirrors immediately. The infrastructure created for this will also be used for immediate erasure coding.

Whether a mirror participates in immediate write mirroring is controlled by a per-mirror `IMMEDIATE` flag in the composite layout (`lcme_flags`), alongside existing flags like `STALE` and `PREFER`. The flag is set at layout creation time (e.g., `lfs mirror create` or `lfs setstripe`) or added later via a layout-modifying operation. Mirrors without the flag are ordinary FLR mirrors — written lazily via resync — and do not participate in write duplication or AW lock epochs. This allows immediate and non-immediate mirrors to coexist on the same file (see §4.2).

The goal is to have redundancy immediately, but during writes only a single mirror (the primary) is available for reads — see Primary Mirror Selection and Read Visibility for the rationale.

The core idea of the design is this:

To write an IWM file, the client first acquires the layout lock (sending a write intent to transition the layout from RDONLY to WRITE_PENDING if needed), then takes an Active Writer (AW) lock — a CW lock on a separate IBITS bit (ACTIVE_WRITERS) on the same per-file resource. The layout lock and AW lock are independent: layout changes (component instantiation, SEL extension) revoke LAYOUT but leave ACTIVE_WRITERS untouched, so the write epoch spans layout changes without interruption (see AW Lock vs Layout Lock Interaction).

The client sends all writes to all online mirrors in parallel. A single primary mirror will be selected for reads during this time, since we cannot guarantee all mirrors are identical during writes. This mirror is the "write leader", also used for write ordering (locks are taken on this mirror first). All other mirrors are marked INFLIGHT during writes — see Mirror States for the full flag model.

Clients hold the AW lock until all data is committed to OST storage, then release it. Per-mirror errors are reported to the MDS via the cancellation LVB. On error (or for administrative operations), the MDS takes the AW lock in EX mode, forcing all writers to flush and release, then transitions the layout — clearing INFLIGHT from clean mirrors, setting STALE on failed ones. See Write Operation Flow and Active Writer Lock for the full protocol.

Write ordering is enforced by requiring client-side LDLM extent locks on the primary mirror. This means DIO to IWM files must use client-side locks where today it operates locklessly — a performance regression for shared-file DIO workloads. Restoring lockless DIO is a high priority for future work; see Write Ordering Alternatives.

Client eviction, MDS failover, write leader failure, and other recovery scenarios are covered in FailureHandling.

1.3. Relationship to Erasure Coding

Immediate mirroring is separate from FLR Erasure Coding (read-only EC), and can be done partly in parallel. IWM will be the foundation of immediate erasure coding — the write duplication infrastructure, Active Writer lock, epoch management, page consistency mechanisms, and error reporting all carry over to EC. Immediate EC will be covered in a separate design document.

2. Detailed Design

The write flow, AW lock mechanics, client-side IO duplication, and failure handling are each covered in the subsections below. Layout handling proceeds similarly to existing FLR, except that secondary mirrors are marked INFLIGHT rather than STALE during writes — see Mirror States for the full flag model.

CFLN_07f52859_BEGIN_drawio [Diagram: FLR Mirroring] CFLN_07f52859_END

FLR Immediate Mirroring State Machine

CFLN_a71a93b2_BEGIN_drawio [Diagram: FLR Immediate Replication] CFLN_a71a93b2_END

2.1. Write Operation Flow

Before starting a write (or other OST-modifying operation — see §2.3.2), the client acquires the layout lock and then the AW lock (see Lock Ordering). While holding the AW lock, the client sends writes to all mirrors in parallel. The primary mirror remains readable; secondaries are INFLIGHT (see Mirror States).

The client holds the AW lock until all data is committed to OST storage (see Lock Lifetime), then releases it with per-mirror error reports via the cancellation LVB (see Error Reporting). The exact point at which a write returns success to the caller is a policy choice — see Operation Completion Consistency Model in Open Problems.

In the normal (no-error) case, once all clients release their AW locks, the MDS closes the epoch and transitions the layout back to RDONLY. If any client reports errors, the MDS forces epoch closure — see Epoch Close for details.

Example Scenario
With one client, if there are three mirrors and three writes:

If write 1 errored to mirror 0
If write 2 errored to mirror 1

The client will report errors on mirror 0 and mirror 1 to the MDS as part of AW lock cancellation (see Active Writer Lock for the LVB structure). The MDS would then clear INFLIGHT from mirror 2 (it is now clean) and mark mirrors 0 and 1 as STALE (clearing INFLIGHT, adding STALE). Userspace can try to resync to these mirrors, and if this fails, will need to add new mirrors.

If all mirrors fail writes during an IO, the file is degraded to its pre-IWM state — all mirrors are marked STALE on epoch close and the file requires resync. This is the same outcome as a total write failure in delayed-write FLR today.

2.1.1. Primary Mirror Selection and Read Visibility

Only one mirror is readable during writes: without MVCC (which Lustre's storage backends lack), writes arriving at different mirrors at different times could expose inconsistent data to concurrent readers. The MDS enforces this by marking secondary mirrors INFLIGHT at epoch open, making them inaccessible to all clients. The primary mirror remains unflagged and readable. This also covers OST-authoritative metadata (notably file size) — since secondaries are INFLIGHT, size queries only reach the primary, so transient size divergence during writes is invisible. See Mirror States for the full INFLIGHT/STALE flag model.
However, this is not possible for DIO, which does not use LDLM locks on the client, but instead only uses them locally on each separate server.

2.2. Active Writer Lock

This section covers the mechanics of the Active Writer (AW) lock: how it is used, what it guarantees, and how the MDS uses it to manage write epochs.

2.2.1. Lock Semantics

The AW lock is a CW (concurrent write) lock on a special MDS IBITs bit called ACTIVE_WRITERS. Multiple clients can hold it simultaneously. The MDS uses lock presence to determine whether any writers are active on a file. When the MDS needs exclusive access (epoch close, mirror replacement, resync), it requests the AW lock in EX mode, which forces all clients to flush and release.

2.2.2. Lock Lifetime and Commit Requirements

The client holds the AW lock from the start of a write operation until all data is committed to OST storage — not just sent, but confirmed durable on disk. This is the fundamental guarantee: the AW lock is only released once the data is known safe.

The durability requirement for IWM is stronger than for normal Lustre writes. Without IWM, a buffered write flushed to OST page cache but not yet journal-committed is at risk if the OST crashes — but this is just data loss. With IWM, if one mirror's OST commits while another does not, the mirrors silently diverge with no signal to the MDS. Mirror inconsistency is qualitatively worse than data loss: it persists silently and can corrupt reads after recovery.

To close this window, the client confirms journal commit on all mirror OSTs before releasing the AW lock. Each OST import tracks `last_committed` (highest durably committed transno). At AW lock release time:

1. For each OST the client wrote to during this epoch, check whether the last write's transno `last_committed` for that import.
2. If all writes are already committed (the common case — journal commit interval is ~5s, and the AW lock idle timeout is 1–5s), no extra work is needed. Release immediately.
3. If any OST has uncommitted writes, issue `OST_SYNC` to those OSTs and wait for acknowledgment. Only then release the AW lock.

Common-case cost is negligible (one comparison per OST). The expensive sync path only triggers when epoch close races with uncommitted writes — a narrow window. This is a tunable (`iwm_sync_on_epoch_close`, default: enabled). Disabling it accepts the mirror inconsistency risk in the client+OST crash window.

Interaction with `sync_on_lock_cancel`: OSTs default to `sync_lock_cancel=always` (set by `ofd_slc_set()` when `sync_journal=0`), which forces journal commit when an extent lock is cancelled — protecting lock handoff between clients. The IWM sync mechanism is complementary: SLC protects other clients from seeing uncommitted data, while the AW release sync protects mirror consistency across crashes.

Failure cases: If a client dies before releasing its AW lock (and therefore before syncing), the MDS evicts it and conservatively stales all non-primary mirrors — the same path as any client eviction. The sync mechanism only affects the clean-release path. If a client's sync to an OST times out or fails, the client reports it as a write error on that mirror via the cancellation LVB, and the MDS marks the mirror STALE — the existing error path.

For DIO, journal commit is part of the write RPC reply — DIO writes use `OBD_BRW_SYNC`, so the sync requirement is already satisfied. For buffered IO, the client must track page writeback through to RPC completion and then verify journal commit as described above, which requires associating pages with something that lives after the `write()` syscall returns.

The AW lock is acquired via standard LDLM enqueue to the MDS and cached for reuse by subsequent writes to the same file — the lock is held as long as writes are active, then released after a brief idle timeout (1-5 seconds). This avoids per-write MDS round-trips while keeping the cache window short, since holding AW locks prevents the MDS from considering secondary mirrors in sync, reducing data reliability (and prolonged holding risks client eviction).

2.2.3. Error Reporting via Cancellation LVB

When the client releases its AW lock, it communicates write results to the MDS via a lock value block (LVB) in the LDLM_CANCEL reply. The v1 LVB contains a bitmask of mirrors that experienced errors — one bit per mirror (FLR supports up to 16 mirrors, so a `uint16` suffices). The error semantics are simple: any error on a mirror sets the bit, and the MDS ORs the bitmasks from all clients' cancellation LVBs to determine which mirrors need to be marked STALE. Future versions could extend the LVB to per-object (per-stripe) error reporting — see Byte Range Error Reporting — though per-stripe bitmasks for 2K stripes × 16 mirrors would exceed typical LVB size limits and may require a different reporting mechanism. CLIO changes are needed to collect errors from all types of operations (write, setattr, etc.) that use the AW lock, since any modification failure on a mirror means that mirror will remain STALE after the epoch closes.

If an error occurs mid-write, the client completes in-flight writes and releases its AW lock, reporting errors via the cancellation LVB. This error report triggers the MDS to close the epoch (see Epoch Close). There is a brief window where new AW locks can be granted after one client reports an error but before the MDS's EX lock request arrives — this is acceptable because the EX lock request will flush these too.

2.2.4. Epoch Close: MDS Processing

The MDS closes a write epoch when it holds no outstanding AW locks for a file. In the normal case, all clients finish writing and release naturally. When a client reports errors via its cancellation LVB, the MDS actively closes the epoch by requesting the AW lock in EX mode, forcing all remaining writers to flush their writes, commit to OST storage, and release their locks. The same forced-close mechanism is used for administrative operations like mirror replacement or resync. Note that forced epoch close on a hot file with many concurrent writers will be expensive — every AW lock holder must flush synchronously before the EX lock can be granted. This is analogous to the cost of truncate on a widely-shared file today and is unavoidable given the consistency requirements.

Once the MDS holds exclusive access, it must handle any evicted clients before transitioning the layout. Evicted clients will not receive the AW lock cancellation and may have writes in-flight that land on OSTs after the epoch close — see Client Eviction from MDT and MDS Failover for how this is handled (OST extent lock flush to maximise primary mirror completeness).

After evicted clients are handled, the MDS examines the error reports from all clients' cancellation LVBs and transitions the layout:

- **No errors on a mirror:** clear INFLIGHT — mirror is now clean and in sync.
- **Errors on a mirror:** clear INFLIGHT, set STALE — mirror requires full resync. lamigo is notified via changelogs (the same mechanism used by Hot Pools today) and attempts recovery — see §2.4.5.
- **Client eviction (no error report available):** assume writes to all mirrors except the primary failed — clear INFLIGHT, set STALE on secondaries.

The layout transitions back to RDONLY. This transition bumps the layout generation (version). Note that the OST-side layout generation check (`ff_layout_version`) does not by itself fence late-arriving RPCs from evicted clients — the OST only rejects writes older than the minimum generation it has already seen, and it learns new generations from incoming writes rather than from the MDS. The actual fencing of evicted-client writes is provided by the OST extent lock flush described above.

2.2.5. Recovery After MDS Failover

For recovery to succeed after MDS failover or crash, the MDS must determine which files had active IWM write epochs at the time of failure, so it can close those epochs safely. AW lock state (including which clients hold AW locks) is held in memory and lost on restart, so the MDS cannot simply reconstruct the epoch state from its in-memory lock tables.

If any client fails to reconnect during recovery (is evicted), the MDS closes active epochs for that client's files with errors — see Client Eviction from MDT for the eviction behavior. The primary mirror is always identifiable after a crash: INFLIGHT is durably recorded in the layout xattr (`lcme_flags`), and the primary is the one without INFLIGHT set (set as part of the durable layout transition to WRITE_PENDING at epoch open). If all clients reconnect successfully, recovery proceeds normally: clients replay their AW locks, the MDS reconstructs the in-memory epoch state, and epochs close as they would in the non-crash case.

The central question is how the MDS identifies which files had active epochs. There are several possible approaches:

1. **Scan all inodes for INFLIGHT mirrors.** Correct but prohibitively expensive — a large filesystem could have billions of inodes and only a handful of active IWM files.
2. **Replay from client lock state.** Clients replay their AW locks during recovery, which tells the MDS which files have active epochs. But this only works if all clients reconnect — if a client is evicted, its lock state is lost and the MDS has no way to know which files it was writing. This is exactly the case where recovery matters most.
3. **Durable epoch FID set.** The MDS maintains a persistent set of FIDs with active write epochs, updated at epoch open/close time. On recovery, the MDS reads this set directly instead of scanning or depending on client replay.

Chosen approach: durable epoch FID set. When a file's first AW lock is granted (transitioning the layout to WRITE_PENDING and setting INFLIGHT on secondaries), the FID is added to the set. When the epoch closes, the FID is removed. This is one durable write per file entering a write epoch, not per client or per lock grant. The set must support fast insertion/removal and could be very large (potentially millions of concurrent IWM files). The storage and indexing design is deferred to the implementation phase.

During recovery, the MDS must first check the active-epoch FID set against current file state: files that are no longer IWM files (e.g., layout was changed before the crash) should be removed from the set. Then, if any client is evicted, the MDS iterates the remaining set and closes all active epochs with errors. This is conservative: the evicted client may not have held AW locks on all of these files, so some mirrors may be needlessly staled. But it is always safe and correct — mirrors that were genuinely consistent will simply be resynced unnecessarily.

Recovery iteration over the FID set should be parallelisable — each file's epoch close is independent, and the set could be large. The storage structure chosen for the FID set should not force sequential processing; multiple recovery threads should be able to claim and process entries concurrently.

The key trade-off: v1 does not durably record *which clients* participate in *which files'* epochs. Recording per-file client participation would narrow the blast radius on recovery — staling only files where the evicted client actually held AW locks. But this requires a durable write on every AW lock grant (not just the first per file), and the storage must scale to potentially thousands of clients per file. Possible approaches include per-file xattrs storing client NIDs (limited by xattr size), NID range compression (ineffective for IPv6), and separate per-file extent trees (heavy). The cost-benefit balance is unclear — broad staling is always correct and costs nothing at runtime. Per-file tracking adds per-lock-grant IO cost and on-disk complexity. It is worth revisiting if broad staling proves problematic in practice.

2.2.6. Flush Semantics on Lock Release

The AW lock has two release paths, but both have the same end state: all data committed to OST storage, lock released with error report via cancellation LVB. The AW lock is not returned until all dirty pages are synced.

Voluntary release (client-initiated, no conflicting lock request): The client is done writing and lets the AW lock go (either explicitly or via LRU expiry). Before releasing, the client syncs all dirty pages to OST storage and confirms journal commit on all mirror OSTs (see §2.2.2 for the `last_committed` / `OST_SYNC` mechanism). The client reports its per-mirror error state via the cancellation LVB and releases the lock.

Forced revocation (MDS requests EX lock, blocking AST fires): The MDS is closing the epoch — due to error or administrative action. The client must immediately flush all dirty data to all mirrors, wait for in-flight writes to complete, and confirm journal commit on all mirror OSTs before releasing the lock with its error report. This is a synchronous operation: the blocking AST must not return until the data is durable on all mirrors.

Both paths produce the same result — all data committed, lock released with error report. The only difference is who initiates the flush timing: the client (voluntary) or the MDS (forced).

The AW lock flush syncs dirty data but does **not** clear the page cache — an epoch close should not destroy cached data. The pages remain cached and valid; they simply no longer need writeback.

Both release paths require the client to know which dirty pages and in-flight RPCs belong to the current epoch. The implementation must track this association — ensuring that pages dirtied under this AW lock are flushed before the lock is released. The specific mechanism (per-page epoch stamps, RPC reference counts, or reuse of `lo_active_ios`) is a development-time decision.

For evicted clients that cannot respond to lock callbacks, OST extent locks (`LCK_PW` on the full file extent) can be used to force-flush the evicted client's dirty data on the OSTs. This mechanism is used during both live eviction and MDS recovery eviction — see MDS Failover for details.

This is a new semantic for Lustre locks. Today, layout lock cancellation does NOT flush dirty pages — it only invalidates the layout and unmaps mmapped pages. The existing FLR resync flush works differently: the resync client explicitly triggers `LL_DV_WR_FLUSH` (which has each OST take `LCK_PW` on the full extent to evict all client caches), and this happens before the layout lock is revoked. The AW lock needs to incorporate flush behavior that the layout lock path has never needed. Whether the AW lock flush can reuse the existing `LL_DV_WR_FLUSH` machinery or needs a new mechanism remains an implementation question.

2.2.7. AW Lock vs Layout Lock Interaction

The AW lock and the layout lock track distinct concerns. The layout lock tracks *what the layout is* — stripe configuration, component structure, mirror membership. The AW lock tracks *the write epoch* — who is writing, what phase they are in, and when the epoch closes. These must not be merged — they are separate concerns — but they interact closely.

Lock ordering: The layout lock is always acquired first, the AW lock second. This ordering is strict and applies to both clients and the MDS:

- **Client write path:** The client takes the layout lock first (learning the current layout). If the layout has immediate mirrors, the client then takes the AW lock and proceeds with write duplication. If the layout does not require immediate mirroring (e.g., delayed-write FLR, or no mirrors), no AW lock is taken.
- **IO restart on layout change:** If the layout changes during IO (layout lock revoked), the client's IO restarts from the layout lock. The AW lock is **not** revoked by the layout change — it is a separate IBITS bit and survives independently (see Why Separate Locks). On restart, the client re-evaluates the new layout — if immediate mirroring was removed (e.g., by an admin operation), the client releases its AW lock and the write proceeds under normal delayed-write FLR semantics.
- **Epoch close (non-structural, MDS-initiated):** The MDS takes EX on the AW lock first, forcing all writers to flush dirty pages, report errors via cancellation LVB, and release. Once the epoch is closed, the MDS updates the layout flags (INFLIGHT/STALE) via a layout lock cycle — layout EX second. Because this is a flag-only change, the `STATE_ONLY_CHANGE` optimization applies: clients re-acquire layout CR without a page cache nuke.
- **Administrative operations** (resync, mirror replacement, mirror add, old-client write intent): Layout lock EX first (prevents new AW acquisitions, since clients must hold layout CR before taking AW CW), then AW lock EX to quiesce existing writers. The AW lock flush completes before any layout transition occurs. These are structural changes that require full page cache teardown on re-acquisition.

The AW lock is conditional on the layout. A client only holds an AW lock when the layout requires immediate write duplication. This means the AW lock population naturally tracks the set of clients doing IWM — if the layout changes to drop immediate mirroring, clients release their AW locks as part of IO restart and do not re-acquire them.

2.2.7.1. Why Separate Locks

The AW lock lives on a separate IBITS bit from the layout lock (see Lock Semantics), not a mode or flag on the layout lock itself. This separation is deliberate — combining them would create fundamental problems:

The layout lock is light; the AW lock is heavy. The layout lock is designed to be held "lightly" — the client reads the layout, then the lock can be revoked at any time without flushing IO. The client simply restarts the IO under the new layout. This is essential for operations that change the layout frequently (component instantiation in PFL files, self-extending layouts). The AW lock has the opposite semantic: its blocking AST forces a full flush and waits for OST commit before returning. If both semantics lived on the same lock, every layout revocation would require flushing all in-flight IO — turning a cheap IO restart into an expensive synchronous flush. For a many-component PFL file written sequentially, this would create an unnecessary epoch close at every component boundary.

The epoch must span layout changes. Component instantiation, SEL extension, and other layout changes revoke the LAYOUT bits but should not close the write epoch. The mirrors are the same, the write session is ongoing, and nothing about epoch consistency semantics changed. Because ACTIVE_WRITERS is a separate bit, taking EX on LAYOUT does not revoke it — the AW lock survives layout changes naturally. The epoch only closes when the MDS explicitly takes EX on ACTIVE_WRITERS (error handling, resync, admin) or all clients voluntarily release their AW locks.

No MDT/OST co-locking conflict. The layout lock was designed to avoid holding MDT locks during OST IO, preventing circular dependencies (client holds MDT lock does OST IO OST IO needs MDT deadlock). The AW lock reintroduces holding an MDT lock during OST IO, but this is safe because the dependency graph is acyclic: AW is CW (no conflicts between writers), the only EX request is MDS-initiated, and the flush path (client OST) never calls back to the MDS. Since AW epoch close is always an administrative or error-driven action — never triggered by another client's IO path — no circular dependency exists.

Alternatives considered: Combining the bits onto one lock with conditional flush semantics (flush only when in an IWM epoch) was considered, but this effectively reinvents the AW lock as a flag on the layout lock while adding mode-dependent complexity to every layout lock consumer. LDLM lock conversion (acquiring both bits, then converting to drop LAYOUT while keeping ACTIVE_WRITERS) could save a round trip but couples the implementation to a lightly-exercised LDLM feature with subtle recovery and race implications. Both are potential v2 optimizations; v1 uses separate locks for simplicity and debuggability.

2.2.7.2. Behavior During Layout Changes

This section describes the path for non-destructive layout changes (component instantiation, FLR flag transitions) where only the layout lock is involved and the AW lock is left untouched — the `STATE_ONLY_CHANGE` path described in §2.2.7.3. For destructive changes or any operation involving a non-IWM-aware client, the default path applies: layout EX first, then AW EX — see §2.4.6.

When a non-destructive layout change occurs while a client holds both the layout lock and the AW lock (e.g., another client's write intent triggers component instantiation):

1. The server takes EX on LAYOUT, revoking all clients' layout locks via blocking AST.
2. The layout lock blocking AST fires on the client and returns immediately — no flush, no awareness of AW state. This is unchanged from today's behavior.
3. In-flight OST writes either complete normally or are rejected by OSTs (stale layout generation), triggering IO restart.
4. The client's IO restart path detects the layout change and restarts the IO from the top.
5. The AW lock remains held throughout — the epoch is still open.
6. On restart, the client re-acquires the layout lock (via write intent if needed), gets the new layout, and continues writing under the same AW epoch.

The key property: **layout changes are not epoch boundaries.** This is safe because ongoing IO will simply be restarted at the end, as it is for any layout change. In-flight writes that land on OSTs with a stale layout generation are rejected, and the client retries them under the new layout. The AW cancellation LVB accumulates errors across layout changes, and the epoch close (whenever it eventually happens) handles them all.

2.2.7.3. Layout Lock and Page Cache Interaction

Layout lock re-acquisition has a hidden cost: the client's entire page cache for the file is destroyed. Understanding this mechanism is critical for IWM performance.

The mechanism: When the client re-acquires its CR layout lock after revocation, `ll_layout_lock_set()` installs the new layout via `lov_conf_set(OBJECT_CONF_SET)`. This calls `lov_layout_change()`, which tears down and rebuilds the `lov_object` — destroying all sub-objects (`osc_object` instances representing individual stripes). Before teardown, `cl_object_prune()` `vvp_prune()` calls `cl_sync_file_range()` to flush all dirty pages, then `ll_truncate_inode_pages_final()` to nuke the entire page cache. The nuke is necessary because each cached page (`cl_page` `osc_page`) is pinned to a specific sub-object; when those sub-objects are destroyed, the pages cannot survive.

Why this exists: Layout changes can alter the stripe structure — different OSTs, different components, different stripe counts. The sub-objects for the old layout are invalid after the change. Tearing down and rebuilding is the only safe approach for structural changes.

Why this matters for IWM: Delayed-write FLR only mirrors inactive files — there is no hot page cache during layout transitions, so the nuke has negligible cost. IWM is the opposite: it targets files under active sustained IO with hot page caches on multiple clients. If every epoch transition (setting INFLIGHT, clearing STALE) required a layout lock cycle with page cache nuke, IWM performance under buffered IO would be unacceptable.

Non-destructive layout changes. Not all layout changes alter the stripe structure. Two categories preserve existing sub-objects:

1. **FLR state transitions** (RDONLY WRITE_PENDING, epoch flag updates to INFLIGHT/STALE flags): same components, same stripes, same OSTs. Only per-component flags change.
2. **Component instantiation** (PFL extend, SEL): new sub-objects are added for the new component, but existing sub-objects are untouched. Pages cached against existing components remain valid.

For these changes, the page cache nuke is unnecessary — the existing sub-objects are still valid and pages referencing them are still correct.

The optimization: `STATE_ONLY_CHANGE` flag. To avoid the page cache nuke on non-destructive changes, the MDS sets a transient flag (`STATE_ONLY_CHANGE` or similar) in the layout when the most recent `layout_gen` bump was non-destructive. A connect flag (`OBD_CONNECT2` bit) gates this — the MDS only sets the flag for clients that advertise support.

On layout lock re-acquisition, the client checks:

- If the client's current `layout_gen` is exactly N-1 and the new layout carries `STATE_ONLY_CHANGE`: update flags (and add new sub-objects for component instantiation) in the existing `lov_stripe_md` in-place, **skip `cl_object_prune()`**. The page cache survives.
- Otherwise (flag absent, client is more than one version behind, or destructive change): full teardown as today. Fall back to old behavior.

The N-1 constraint is critical: the flag only describes the most recent transition. A client that missed multiple layout changes cannot trust it and must do the full rebuild.

Safety properties:

- **Old clients** never see the flag (connect flag gates it) and always do full teardown. Correct by default.
- **Stale clients** (more than one version behind) ignore the flag and do full teardown. Correct.
- **MDS reboot** loses the transient flag. Clients reconnect without it and do full teardown. Correct.
- **Flag absent for any reason** (bug, race, mixed destructive/non-destructive sequence): full teardown. The optimization is purely opportunistic — absence never causes incorrectness.

Prior art: CSDC compressibility changes. The CSDC (client-side data compression) feature introduced `LAYOUT_INTENT_CHANGE` as a layout intent opcode for flag-only layout updates (EX-8355, Gerrit 54248). This allows the MDS to change the `LCM_FL_INCOMPRESSIBLE` flag on a layout without altering the stripe structure. The patch defines the wire protocol (`LAYOUT_INTENT_CHANGE` opcode, `LAIF_INCOMPRESSIBLE` flag, `OBD_CONNECT2_UPDATE_LAYOUT` connect flag) and the MDS-side handler (`lod_declare_layout_minor_change()`). However, the current implementation still bumps `layout_gen` and still triggers the full page cache teardown on the client — it avoids the problem by deferring the change until no layout lock is held, rather than making the client-side re-acquisition cheap. The IWM optimization extends this by making the client-side path aware of non-destructive changes, so the layout lock can be re-acquired without tearing down the `lov_object` or nuking the page cache. The `LAYOUT_INTENT_CHANGE` wire definitions from CSDC could be reused or extended to carry the IWM state transitions.

Relationship to the default layout transition path: The default path for all layout changes (used by old clients, destructive changes, resync, migrate, and any case the MDS cannot prove is non-destructive) is: MDS takes EX layout lock MDS takes EX AW lock (epoch fold-up) layout change clients re-acquire with full teardown (see §2.4.6). The `STATE_ONLY_CHANGE` optimization applies only to non-destructive transitions initiated by IWM-aware clients, where the AW lock is not touched and the epoch continues across the layout change.

2.3. Client Implementation

2.3.1. Write Duplication via BRW Page Fan-Out

The client will need to duplicate all writes to all of the mirrors (non-write modifying operations are covered separately in §2.3.2). The key design choice is at what level duplication occurs.

IO-level duplication is not viable. For buffered I/O, IO-level duplication would require sending pages to secondary mirrors at IO creation time — effectively forcing direct I/O to the secondaries — because the pages cannot be left in the page cache to be aggregated independently per mirror. This destroys asynchronous write semantics and write aggregation for secondary mirrors, which is a non-starter from a performance perspective. (For DIO, IO-level duplication would work but is not sufficient on its own.)

The selected approach operates at the BRW/RPC layer, intercepting after the primary mirror's write has been fully prepared (pages assembled, bounce pages allocated for compression/encryption). This handles both buffered and direct I/O uniformly: for buffered writes, the page cache performs normal write aggregation on the primary mirror and duplication fires at BRW send time from the aggregated pages; for DIO, duplication fires from the same BRW page array built for the primary. In neither case does the duplication layer need to re-enter the IO start path.

BRW page fan-out (selected approach) re-derives per-mirror RPCs from the primary mirror's assembled BRW page array via a call-up from the OSC layer to the LOV layer. When the primary mirror's OSC has built its BRW RPC (pages assembled, bounce pages allocated), it calls up to the LOV layer, which fans out the page array to each secondary mirror's OSC. Each secondary mirror's RPCs are built through that mirror's own stripe calculation, so mirrors can have different layouts (different stripe count/size). Note: this call-up path must be careful about the PTLRPC no-sleep requirement — the callback occurs in RPC submission context. This is more complex than simple RPC replay but is necessary to support heterogeneous mirror layouts, which real deployments require (see Combining Immediate and Non-Immediate Mirrors). See Qian Yingjin's prototype (LU-13643, Gerrit 63244) for a proof-of-concept of this approach.

An alternative considered was **RPC duplication**, which sends the primary mirror's RPCs to all mirrors identically. This is simpler but requires all mirrors to have identical layouts (same stripe count and stripe size), since the same RPCs — already addressed to specific OSTs at specific offsets — would be reused verbatim. This restriction is too limiting for real deployments, where mirror layouts may differ.

Page Consistency During Write Duplication {#page-consistency-during-duplication nh-numbering="2.3.1.1."}

The primary's BRW pages must remain valid and unmodified until all secondary RPCs complete. There are two consistency threats, and the BIO and DIO paths handle them differently:

- **Source page modification:** For buffered IO, another write to the same pages could modify them while secondary RPCs are in flight, causing the secondaries to receive different data than the primary — silent inconsistency. For DIO, modifying the userspace source buffer during a write is already an application error, but the implementation is inherently robust against it because the BRW page array provides kernel-side copies.
- **Competing IO on the same client:** A second write to overlapping byte ranges must be serialized against in-flight duplication to prevent the secondary mirrors from seeing a different write order than the primary.

The following two sections describe how each IO path addresses these threats.

A third requirement follows from write duplication: a page is not complete for epoch purposes until all mirror RPCs for that page have been acknowledged. The implementation must track per-page (or per-RPC) completion across all mirrors so that the AW lock is not released — and the epoch not closed — while any mirror still has unacknowledged writes outstanding.

2.3.1.1. Buffered IO Completion Semantics

Pages are already held locked throughout RPC sending today — this is existing behavior that provides the foundation. IWM extends the hold: pages must remain locked until all mirror RPCs complete, not just the primary. This addresses both threats from Page Consistency — a locked page cannot be modified by another write, and competing IOs on the same byte range are serialized by the page lock. A write is not finished until every mirror's RPC has completed (or failed), so the page cannot be unlocked and made available for the next write until the full fan-out completes. This requires changes to page completion semantics and page states — the current page state machine assumes a write completes when the single (primary) RPC completes.

Memory-mapped (mmap) writes require no special handling — mmap dirties pages in the page cache, and writeback proceeds through the same buffered IO path described here. The AW lock is acquired at writeback time as part of normal BRW submission, not at page fault time. IWM's page completion extensions (holding pages locked until all mirror RPCs complete) apply to mmap-dirtied pages identically.

2.3.1.2. Direct IO Completion Semantics

DIO does not use the page cache, so page locks are not available to provide serialization. Instead, v1 waits for all mirror RPCs to complete (or fail) before returning — this is required for serialization, since without the page cache there is no other mechanism to prevent competing IOs from interleaving. The return code reflects only the primary mirror's result; secondary failures are recorded in the AW lock cancellation LVB, not surfaced to the application (see Fast-fail and Secondary Mirror Failure Policy). This is the simplest correct approach: by not returning until all mirrors are done, there is no window for competing IOs to interleave. Making secondary DIO writes asynchronous is a possible future enhancement — it would require copying the BRW page array (the kernel-side buffer), and the copies must be protected against competing IOs — the tree lock and LDLM extent lock must be held throughout to prevent a concurrent write from landing on the primary before the async secondary writes complete.

2.3.1.3. Implications for Erasure Coding

The BRW page fan-out and page consistency mechanisms developed for IWM are intended to carry over to immediate EC writes, though significant design work remains. EC requires absolute consistency between source data pages and computed parity pages — primary locking provides ordering, and the tree lock (rounded to raidset-aligned extents) would ensure source pages are stable during parity computation. Tentatively, EC parity pages would be dispatched to their target OSTs via the same LOV-level RPC callback used for IWM secondary mirror fan-out, reusing the completion barrier infrastructure. However, the details of parity computation timing, partial-stripe writes, and the interaction between EC raidset geometry and the fan-out path remain to be resolved.

2.3.1.4. Compression and Encryption Across Mirrors

Ideally we would not have to compress or encrypt data more than once when sending to multiple servers Each secondary mirror's OSC compresses independently from the original (uncompressed) source pages. Since compression is per-component in the Lustre layout (`lcme_compr_type`, `lcme_compr_chunk_log_bits` in `lov_comp_md_entry_v1`), different mirrors can independently specify compression settings — different algorithms, chunk sizes, or one compressed and one not — and the BRW page fan-out path handles this naturally. The compression work is performed once per mirror, but this is an acceptable cost for IWM's typical 2-3 mirrors. Different chunk sizes across mirrors pose no correctness issue: if a write is not aligned to the secondary mirror's chunk boundary, the secondary's OST handles the read-modify-write internally, as it does for any unaligned compressed write.

A possible future enhancement is bounce page reuse: when all immediate mirrors share identical compression settings, bounce pages built for the primary mirror could be reused for secondaries, avoiding redundant compression/encryption work. This is not planned for v1.

2.3.2. Other Modifying Operations

The preceding section covers write IO duplication via BRW page fan-out — a mechanism specific to data writes, where the page cache and write aggregation require fan-out at the RPC assembly layer. Other modifying operations — truncate, hole punch, fallocate (space preallocation), and certain setattr variants — also modify OST object state and must be applied to all immediate mirrors. These operations do not transfer data pages, so the BRW fan-out path does not apply. Instead, they use a different fan-out mechanism at the LOV sub-io level.

2.3.2.1. Operation Classification

Modifying operations fall into three categories based on their mirror interaction:

OST object operations (must be duplicated to all mirrors):

- **Truncate** — reduces file size via `OST_PUNCH` over [new_size, EOF). The LOV maps this to the affected stripes, as with any ranged operation.
- **Fallocate** — covers both space preallocation (mode 0) and hole punch (`FALLOC_FL_PUNCH_HOLE`). Preallocation changes the physical allocation map without writing data; punch deallocates a byte range, replacing data with a hole. Both use `OST_FALLOCATE` with the appropriate mode flags.
- **Time setattr** (`ATTR_MTIME`, `ATTR_ATIME`, `ATTR_CTIME`) — sets timestamps on OST objects via `osc_setattr_async`. Lightweight, no range calculation.

MDS-only operations (no mirror interaction needed):

- **chmod, chown** — pure MDS inode metadata. OSTs are not contacted. No IWM concern.
- **setstripe** — layout changes via MDS ioctl. Not a modifying operation in the IWM sense; layout changes interact with IWM through the layout lock, not through operation duplication.

Mixed operations:

- **Truncate** is technically mixed: `ll_setattr_raw` always sends an MDS RPC first (the MDS must approve the size change and check `ETXTBUSY`), then dispatches `cl_setattr_ost` for the OST-side truncation. The MDS phase is not mirror-specific; only the OST phase requires duplication.

2.3.2.2. LOV Sub-IO Fan-Out

In current FLR, truncate, punch, and fallocate go through the `cl_io` `CIT_SETATTR` path. The LOV layer selects a single mirror (`lov_io_mirror_init` sets `lis_mirror_index`), creates sub-ios only for that mirror's stripes, and sends the write intent RPC to stale other mirrors on the MDS. The operation executes on one mirror; resync later copies data to bring stale mirrors up to date.

For IWM, these operations must execute on all mirrors in real time — there is no deferred resync step. The fan-out point is `lov_io_iter_init`: instead of iterating only the primary mirror's stripes, it creates sub-ios for every non-stale mirror's stripes that overlap the operation's byte range. Each sub-io calls the appropriate per-stripe RPC (`osc_punch_send` for truncate, `osc_fallocate_base` for punch/fallocate, `osc_setattr_async` for time setattr). This is analogous to BRW page fan-out for writes — the LOV is the mirror-aware layer that manages per-mirror dispatch — but at the sub-io level rather than the RPC assembly level, because these operations have no page data to fan out.

The existing `lov_foreach_io_layout_mirror` macro already accepts a mirror index parameter. For IWM, the iteration expands to cover all eligible mirrors. Per-stripe size and offset translation (`lov_size_to_stripe`) uses each mirror's own component layout, so heterogeneous mirror layouts (different stripe count/size) work correctly — the same file-level truncation offset maps to different OST object offsets depending on the mirror's geometry.

The BRW fan-out for writes uses an OSC-to-LOV callback — a concession to the fact that write data must pass through the page cache and BRW assembly before the LOV can replicate it. These operations have no such constraint. The operation parameters (byte range, mode flags) are known at `cl_io` init time, so the LOV can create all per-mirror sub-ios up front during `iter_init`. This is a cleaner fan-out pattern: no callbacks, no PTLRPC no-sleep concerns, just parallel sub-io dispatch.

2.3.2.3. AW Lock and Epoch Interaction

These operations participate in AW lock epochs identically to writes. The client acquires the AW CW lock before the operation (or reuses a cached one), the MDS transitions the layout to WRITE_PENDING on first acquisition, and the operation executes under the epoch. Per-mirror errors are collected and reported via the AW lock cancellation LVB on release — if truncate succeeds on mirror 0 but fails on mirror 1, mirror 1 has the wrong (larger) size and is marked STALE on epoch close. Resync later copies the correct state from the primary.

For IWM-aware clients, the write intent RPC that FLR uses today (`LAYOUT_INTENT_TRUNC`, `LAYOUT_INTENT_WRITE` for fallocate) could potentially be subsumed by the AW lock epoch mechanism in a future optimization — the AW lock acquisition already triggers the RDONLY WRITE_PENDING transition on the MDS, and IWM applies the operation to all mirrors directly rather than staling secondaries. However, the MDS must continue to handle write intents from non-IWM clients (see §2.4.6), so the write intent code path cannot be removed. For v1, both paths coexist.

Page cache flush ordering: Before truncate or punch, the client flushes dirty pages in the affected range. Under IWM, this flush triggers BRW fan-out to all mirrors (via the write duplication path in §2.3.1), ensuring all mirrors have consistent data up to the truncation point before the truncate sub-ios execute. The interaction between this flush and AW lock dirty tracking — specifically, how the client knows which dirty pages belong to the current epoch and must be flushed — is an open design question (see §4.1).

2.3.2.4. Implications for Erasure Coding

The LOV sub-io fan-out for truncate/punch/fallocate extends naturally to immediate EC. Where IWM creates sub-ios per-mirror-per-stripe, EC would create sub-ios per-parity-group. Truncate and punch are particularly relevant for EC because they can create partial-stripe boundaries that require parity recomputation — a stripe that was fully allocated may become partially punched, requiring the parity stripe to be updated. The sub-io fan-out provides the dispatch infrastructure; the parity computation logic layers on top.

2.3.3. Error Management

2.3.3.1. Fast-fail and Secondary Mirror Failure Policy

Today, if a write RPC fails, the client retries indefinitely — there is no alternative target. For FLR reads, the client uses non-delay RPCs (`ci_ndelay`): if the RPC fails quickly, the LOV rotates to the next mirror and restarts the IO, transparently retrying on a different copy. This is set at IO init time: `io->ci_ndelay = !(iot == CIT_WRITE)` — reads get fast-fail, writes do not.

With IWM, writes have multiple targets, so the same fast-fail principle applies to secondary mirrors. If a write to a secondary mirror fails, the client should not retry indefinitely — the data is safe on the primary, and the whole point of redundancy is to absorb failures without blocking the application. The primary (write leader) is different: it is the ordering anchor and has no alternative, so primary writes must still use blocking RPCs and retry normally. Primary failure triggers IO restart and epoch close (see Primary Mirror Write Failure).

Secondary retry persistence is a tunable tradeoff. Failing fast on secondaries maximizes availability — the application keeps running and the failed mirror degrades to delayed replication. But it means any transient failure triggers a full mirror resync, which may be expensive. Sites with fast resync infrastructure (lamigo always running, good network) want fail-fast. Sites where resync is slow or costly may prefer to retry the secondary longer, accepting reduced responsiveness for a chance to avoid resync. This should be configurable — a secondary mirror RPC timeout or retry count, separate from the normal `obd_timeout`, defaulting to fail-fast behavior. The exact tunable mechanism (per-filesystem `lctl` parameter, per-file policy, or similar) is a detailed design question.

Application visibility: Secondary mirror failures are not surfaced to the application. The write returns success as long as the primary succeeds. Per-mirror errors are reported asynchronously to the MDS via the AW lock cancellation LVB, and the MDS marks failed mirrors STALE on epoch close. This parallels read mirror behavior, where mirror failures are invisible to the application.

Degradation to delayed replication: When a secondary mirror is marked STALE, it effectively falls back to existing delayed-write FLR semantics — it requires a full resync (via `lfs mirror resync` or lamigo) to bring it back in sync. This is the same recovery path that non-immediate mirrors use today. A failed immediate mirror does not break the file; it degrades the redundancy level until resync completes. This is a key design property: **IWM does not introduce new failure modes.** A failed immediate mirror becomes an ordinary stale mirror, handled by existing tools and infrastructure. The file remains accessible (the primary is always readable), redundancy is reduced until resync completes, and the admin is notified via changelog.

How hard to try before giving up is a design question with a spectrum of options:

1. **Hard fail** — secondary RPC fails once with `ci_ndelay`, immediately record the error in the AW lock cancellation LVB. The mirror will be marked STALE on epoch close and require a full resync. Simplest approach, but a transient network blip triggers a full mirror resync.
2. **Background retry** — on secondary failure, retry in the background and return success to the application immediately. Only record an error in the cancellation LVB if retries are exhausted. This primarily applies to DIO (which is synchronous) — for buffered IO, writes are already asynchronous via the page cache and fire at BRW send time, so "background retry" is the default behavior. For DIO, background retry requires copying data to a kernel-side bounce buffer (the unaligned DIO infrastructure already provides this). The key consistency constraint: **the source data must not change between the original write and the retry** — otherwise the secondary gets different data than the primary received, creating silent inconsistency. For buffered IO, page cache pages can be overwritten by subsequent writes, so the client must hold the tree lock during retry (blocking overlapping writes) or use copy-on-write shadow pages. For DIO, the bounce buffer is a point-in-time snapshot so consistency is inherently preserved, but the tree lock must still be held to prevent a concurrent buffered write from modifying the same range on the primary.
3. **Queued delayed replication** — don't even attempt the secondary synchronously. Queue writes and replicate in the background, similar to existing FLR but with the data hot (no need to re-read from the primary OST). Note that buffered IO is already asynchronous, so this option is really about DIO. This blurs the line between immediate and delayed mirroring but could be useful as a degraded-mode fallback.

v1 should start with option 1 (hard fail) for simplicity. Options 2 and 3 are optimizations that reduce unnecessary resyncs and could be added later. The key invariant across all options: the application never sees secondary mirror failures, and the AW lock cancellation LVB always reports the final per-mirror error state to the MDS.

Whether the retry complexity (option 2) is worthwhile is questionable. Lustre RPCs already retry aggressively — the default `obd_timeout` is 100 seconds, and the client retries multiple times before declaring failure. By the time a secondary write actually fails, the synchronous path has already been trying for minutes. A background retry mechanism would only help failures longer than this window but shorter than a full resync — a narrow band. Combined with the consistency hazards and locking complexity, background retry may not be worth the engineering cost. Hard fail plus full resync is likely acceptable for v1 and possibly beyond.

2.4. Failure Handling

2.4.1. Mirror States: STALE and INFLIGHT Flags

During an IWM write epoch, secondary mirrors are marked INFLIGHT. The primary (write leader) is not flagged — it remains readable. The AW lock tracks that a write epoch is active.

INFLIGHT — indicates an IWM write epoch is actively in progress on this mirror. This is a new flag. A mirror marked INFLIGHT is being updated by immediate writes and is expected to become consistent when the write epoch closes. Old clients that do not recognize the INFLIGHT flag will treat it as an unrecognized flag and refuse to read the mirror — this is the correct behavior, as it prevents non-IWM clients from reading partially-written data during an active epoch.

STALE — the mirror is an unknown amount of out of sync with the primary. Its contents should not be used and the issue can only be resolved with a full resync. This is the same flag used by delayed write mirroring (FLR today). STALE is only applied on error or eviction — it is NOT set during normal writing. Old clients understand STALE and can trigger resyncs.

Backward compatibility: INFLIGHT alone (without STALE) is sufficient to prevent old clients from reading secondary mirrors during a write epoch. Old clients treat any unrecognized flag as a reason to refuse access, so INFLIGHT serves the same protective role that STALE would — but without conflating "actively being written" with "damaged and needs resync." This is why STALE is not set during normal writing: it preserves a crisp semantic distinction where STALE exclusively means "this mirror requires explicit intervention to resolve."

The MDS transitions these flags at epoch close — see Epoch Close for the full rules. Mirrors left STALE after a write failure are visible to lamigo via changelogs, triggering recovery (see §2.4.5).

2.4.2. Write/Update Error from Client

Clients will collect all write (or non-write update, eg, setattr) errors, associating them with the Active Writer lock. Errors matter to the MDS when they would create inconsistency between mirrors — i.e., at commit time. An error during the initial write attempt is retried normally by the RPC layer; it only becomes an epoch-level error if it persists through to OST commit failure, meaning data may not have landed on that mirror. After a commit-time error, the client completes in-flight writes and releases its AW lock immediately (no caching), reporting errors via the cancellation LVB. This triggers the MDS to force epoch closure (see Epoch Close).

2.4.2.1. Client Eviction from OST

A client eviction from an OST will cause a write error at commit time, which is reported to the MDT by the client when it cancels the Active Writer lock. An OST eviction may leave partially-committed data that creates inconsistency between mirrors, so it cannot be handled identically to a simple write failure — the affected mirror must be assumed inconsistent. The affected mirror will have INFLIGHT cleared and STALE set, to be resolved by a later RESYNC operation from lamigo or other userspace tool.

2.4.3. Client Eviction from MDT

If a client with an active AW lock is evicted from the MDT, we cannot know whether it completed writes successfully. The MDS force-flushes the evicted client's caches via OST extent locks (see Flush Semantics) to maximise the chance that in-flight writes complete on the primary mirror, then closes the epoch with all non-primary mirrors marked STALE (clearing INFLIGHT, setting STALE). The flush only benefits the primary's completeness — secondaries are staled regardless.

2.4.3.1. Client Loss

Client loss - where a client crashes or similar - is the same as MDT eviction, since such a client is evicted from the MDT and OSTs, and the MDT eviction takes priority over the OST evictions. All mirrors except the primary will have INFLIGHT cleared and STALE set.

2.4.3.2. MDS Failover

When the MDS crashes, AW lock state is lost. On recovery, the MDS uses its durable active-epoch FID set (see Recovery After MDS Failover) to identify files with active epochs. If all clients reconnect, AW locks are replayed and epochs resume normally. If any client is evicted, eviction proceeds as in Client Eviction from MDT — the only difference is how the MDS discovers which files have active epochs: in-memory lock state (live) vs the durable FID set (recovery).

2.4.4. Primary Mirror Write Failure

In case of a write failure to the primary mirror, the clients will attempt to complete all other writes and inform the MDS as usual. The MDS closes the epoch normally (see Epoch Close): the failed primary is marked STALE, and secondaries that completed without error have INFLIGHT cleared (they are now clean and in sync). The next write epoch naturally selects a new primary from the non-stale mirrors — the MDS selects it by examining the write statuses from active writers, finding a mirror with no errors. If all mirrors had write failures, all mirrors are marked STALE and the file degrades to its pre-IWM state, requiring a full resync — the same outcome as a total write failure in delayed-write FLR today. See Client Loss in this section for what to do if a client is lost entirely.

This should work transparently with minimal changes.

2.4.5. Replacing Mirrors/Permanent Failure

When a mirror is left STALE after a write failure, lamigo needs to be notified to attempt resync. The existing mechanism is changelogs — lamigo already watches changelogs for files with STALE mirrors as part of Hot Pools, so IWM does not require a new notification path. The changelog overhead is modest (5-10% for metadata-intensive workloads, less for IO-intensive). If resync fails repeatedly (e.g., due to permanent OST failure), lamigo could add a new replacement mirror; the failed mirror may be removed later.

These operations require quiescing active writers. The lock ordering follows the administrative path (see §2.2.7): the resync client takes layout EX (via `LL_LEASE_RESYNC`), then the MDS takes AW EX to close the epoch and force all writers to flush. lamigo does not need any special commands — it uses the normal mirror add (`lfs mirror extend`) and resync (`lfs mirror resync`) operations. After the operation completes, both locks are released and clients can resume writing.

This means writers will stall during mirror replacement or resync, but they will not fail — they simply block on AW lock acquisition until the operation completes. Since these are separate commands (first add the mirror, then resync it), there will be two quiesce windows. This is acceptable since mirror replacement is an infrequent recovery operation, not a steady-state path.

2.4.6. Conflicting Operations During Write Epochs (Resync, Old Clients)

Clients that understand FLR (delayed-write mirroring, 2.11+) but do not support IWM need explicit handling during active write epochs. These clients can open mirrored files (`exp_connect_flr()` passes) but lack `OBD_CONNECT2_FLR_IMMED_MIRROR` and cannot participate in write duplication. Pre-2.11 clients (pre-FLR) are blocked at file open and are not a concern.

The problem: If a non-IWM client issues a write intent during an active IWM epoch, it would write only to the primary mirror. The secondaries would be marked STALE through normal FLR handling, so there is no silent data corruption — but the epoch's invariant (all IWM writers duplicating to all mirrors) is broken. The epoch cannot close with all mirrors consistent.

Reads are safe: Non-IWM clients cannot read secondary mirrors during an epoch because the INFLIGHT flag is unrecognized — old clients refuse to access mirrors with unknown flags. They read only from the unflagged primary, which is correct.

Design: Epoch fold-up. When the MDS receives a layout write intent (or resync request) from a non-IWM client for a file in an active IWM epoch, it forces epoch closure before processing the request. This follows the default layout transition path:

1. The MDS checks `exp_connect_immed_mirror()` on the requesting client's export.
2. If the client is not IWM-aware and the file has INFLIGHT mirrors, the MDS takes EX on the layout lock (revoking all client CR layout locks), then takes EX on the AW lock (forcing epoch close — IWM clients flush dirty pages and report per-mirror errors via cancellation LVB).
3. Once the epoch is cleanly closed and all participants have flushed or been evicted, the MDS performs the layout change.
4. The non-IWM client's request is then processed normally: a write intent triggers the standard delayed-write FLR transition (RDONLY WRITE_PENDING with STALE secondaries), and a resync proceeds through the normal resync flow.

All clients re-acquire the layout lock via the full path — `lov_layout_change()` tears down and rebuilds the `lov_object`, and `vvp_prune()` flushes dirty pages and nukes the page cache. The non-IWM client blocks at the layout lock level while the fold-up proceeds; this is standard Lustre lock ordering behavior and is transparent to the client. There is no special "degraded mode" or sticky downgrade. After the fold-up, the file is in RDONLY with mirrors either clean or STALE. The next write from an IWM-capable client starts a fresh IWM epoch normally.

Note: for IWM-aware client operations that only change layout flags (epoch open/close, FLR state transitions) or instantiate new components, an optimized path avoids the page cache nuke — see § Layout Lock and Page Cache Interaction. The default path described here applies to all operations from non-IWM clients and to destructive layout changes regardless of client version.

Sustained mixed-client access: If a non-IWM client is actively writing alongside IWM clients, the pattern repeats — each non-IWM write intent triggers the full default path: EX layout lock, EX AW lock (epoch fold-up), page cache nuke for all clients, then the old client writes under delayed-write FLR semantics (staling secondaries). The next IWM write re-opens a fresh epoch. This is expensive (epoch thrashing, repeated page cache flushes and nukes, repeated resyncs) but not incorrect. In practice, mixed-version clusters are a transitional state during rolling upgrades, not a steady-state deployment. Fencing old clients from writing entirely would avoid the thrashing but is unnecessarily disruptive during upgrades — the degradation to delayed-write FLR is acceptable for a transitional period.

Resync requests during an active write epoch are handled identically — any resync closes the epoch first, whether the requesting client is IWM-aware or not. This is just the normal resync behavior described in Replacing Mirrors/Permanent Failure: the AW lock provides the serialization, the epoch closes, and the resync proceeds. The non-IWM write intent is the only new case here; resync already works this way.

3. Future Enhancements and Alternatives

3.1. Byte Range Error Reporting in Cancellation LVB

The v1 cancellation LVB reports errors at mirror granularity — a mirror either succeeded or failed. A future enhancement would be to report errors at per-stripe (per-OST-object) granularity, limiting resync scope to only the damaged stripes rather than the entire mirror. This is especially important for erasure coding, where single-stripe degradation must be tracked precisely (the DEGRADED flag is future work — see the immediate erasure coding design).

v1 LVB format (current):

A `uint16` mirror error bitmask — one bit per mirror, ORed across all clients' cancellation LVBs at epoch close. FLR supports up to 16 mirrors, so 2 bytes. Any set bit means the entire mirror is marked STALE. Although v1 only uses mirror-level granularity, the v2 format below is designed for forward compatibility — v1 should use the v2 self-describing format from the start, with the MDS free to ignore per-stripe detail until single-stripe degradation is implemented.

v2 LVB format (for per-stripe reporting):

The LVB becomes self-describing and variable-length. The format is:

- `uint16 lmve_mirror_errors` — which mirrors had errors (same as v1)
- For each set bit in `lmve_mirror_errors`, in order:
  - `uint16 lmve_stripe_count` — this mirror's stripe count (mirrors can have different stripe counts in heterogeneous layouts)
  - `uint16[ceil(stripe_count / 16)]` — stripe error bitmask, one bit per stripe, padded to 16-bit alignment

The MDS walks the set bits in `lmve_mirror_errors` to determine how many per-mirror entries follow. Each entry's `lmve_stripe_count` prefix tells the MDS how many bytes of stripe bitmask to consume before the next entry.

Sizing: The v2 format is compact in practice. `LOV_MAX_STRIPE_COUNT` (2000) is a per-file limit, but since stripes are partitioned across mirrors, a single mirror can in principle use nearly all of them. The per-mirror stripe bitmask is at most 250 bytes (2000/8, rounded up to 16-bit alignment). Typical sizes:

Scenario	Size
1 mirror errored, 200 stripes	2 + (2 + 26) = 30 bytes
2 mirrors errored, 500 stripes each	2 + 2 × (2 + 64) = 134 bytes
2 mirrors errored, 2000 stripes each	2 + 2 × (2 + 250) = 506 bytes
Worst case: 16 mirrors, 2000 stripes	2 + 16 × (2 + 250) = 4,034 bytes

The worst case (all 16 mirrors errored at max stripe count) is ~4KB. In practice, errors typically hit 1-2 mirrors, keeping the LVB well under 1KB. The LDLM RPC buffer format supports variable-sized LVBs via buffer length descriptors; the cancel path requires modification to pack and unpack the new LVB (v1 protocol work).

Per-extent reporting (deferred): A third level of granularity — byte ranges of failed writes within a stripe — is theoretically possible but has a significant drawback: the stale extent list grows unboundedly as the primary mirror continues to receive writes. Every new write to a good mirror extends the region that is "stale on the bad mirror," so for sustained failures the stale region converges to the entire object anyway. Per-extent tracking is most valuable for brief transient failures where the damaged region is small relative to the object, and the complexity is not justified for v1 or v2.

3.2. Write Ordering Alternatives

v1 uses the primary mirror as locking leader, requiring client-side LDLM locks for all IO including DIO — a performance regression for shared-file DIO workloads that normally operate locklessly. Possible future approaches to restore lockless DIO:

3.2.1. Chained RPC Checksums

One option is to have OSTs maintain a chained checksum of committed writes. In the simplest form, this requires identical RPCs to all mirrors, which requires mirrors to have identical layout geometry — a constraint the v1 BRW page fan-out approach was specifically designed to avoid (see IO Duplication). However, the chained checksum concept could be extended to work with heterogeneous layouts by checksumming at the logical file offset level rather than the RPC level — the OST would chain checksums of (offset, length, data_checksum) tuples as they commit, producing a layout-independent ordering fingerprint. This would require the MDS to compare per-stripe chains after mapping them back to file-level extents, adding complexity but removing the identical-layout restriction. When a write epoch opens, the OST would be informed the stripe object is part of an immediate mirror file. It would take the write checksum from each committed write and chain them together as they are committed (write commits can be ordered by their journal transaction number, even if they are occurring in parallel). The result is a single checksum value which encodes both the writes and their ordering. Non-write operations can be included by checksumming their arguments (e.g., the byte range and fallocate op type for fallocate).

The write primary mirror would be the correct ordering, to which the others are compared. If ordering on a secondary mirror disagrees with the primary, that mirror would be left STALE (not un-staled), requiring a full data resync.

Open questions for this approach include: whether to apply this only to overlapping operations (which requires tracking the extent of all data-modifying operations through a write epoch, but avoids false positives from non-overlapping writes), and recovery scenarios when an OST fails mid-epoch and the checksum chain is incomplete.

A related but heavier-weight idea is having OSTs maintain full write commit vectors (ordered lists of committed write extents) and comparing them across mirrors at epoch close. This subsumes chained checksums but requires significantly more per-OST state and MDS-side comparison logic.

4. Open Problems

4.1. Operation Completion Consistency Model

An open question is when a write (or other modifying operation) returns success to the caller. The options are:

1. **Return on primary completion** — return as soon as the write leader mirror completes, with secondaries finishing asynchronously. Lowest latency, but a crash after return could leave secondaries stale.
2. **Wait for all immediate mirrors** — strongest guarantee, but tail latency is bounded by the slowest mirror. "All" cannot mean "forever" — there must be a timeout or health threshold after which a slow or failed mirror is marked STALE and the operation completes.
3. **Tunable policy** — make this a per-mount or per-file policy choice.

This choice affects error reporting, AW lock lifetime, and the flush semantics described in Flush Semantics on Lock Release.

4.2. Combining Immediate and Non-Immediate Mirrors

Real deployments will combine immediate and non-immediate mirrors on the same file. For example, DDN's hot pools feature places latent mirrors on HDD that are only populated once a file ages out of a flash tier — these mirrors are not written at write time but filled in later by resync (lamigo). A file might have two flash mirrors (written immediately for HA) and two HDD mirrors (synced lazily for capacity/archival).

This means IWM cannot simply treat all mirrors as immediate. The per-mirror IMMEDIATE flag that IWM already requires naturally handles this: mirrors with IMMEDIATE set participate in write duplication and AW lock epoch tracking; mirrors without it are ordinary FLR mirrors — STALE, populated by resync (lamigo) whenever policy dictates, using the same resync path as any other stale mirror. No new "immediacy group" concept or layout state is needed — the file-level layout state (RDONLY WRITE_PENDING SYNC_PENDING) remains singular, and non-immediate mirrors are simply unbothered by it since they are already STALE.

The impact on the core IWM design is narrow: the duplication fan-out and AW lock scope are already determined by the IMMEDIATE flag. The AW lock, epoch close, flush semantics, and error reporting mechanisms are unchanged — they operate over immediate mirrors only.

The tiering lifecycle would work as follows (e.g., migrating from flash to HDD):

1. File starts on flash tier with immediate mirroring (two flash mirrors, IMMEDIATE set)
2. HDD mirrors are added without IMMEDIATE (not synced, not immediate — ordinary FLR mirrors)
3. When the file ages out and should migrate to HDD:
    a. Resync the HDD mirrors (bring them into sync with the flash mirrors)
    b. Set IMMEDIATE on the HDD mirrors
    c. Clear IMMEDIATE from the flash mirrors
    d. Delete data on the flash tier

This needs to be reconciled with how Lustre tiering works today. Steps 3.2-3.4 are not atomic — there will be a transient window where no mirrors are IMMEDIATE, and that's acceptable for v1. A future refinement could have the MDS validate the transition atomically via `lod_declare_layout_change` (refuse to clear IMMEDIATE on the last immediate mirror unless another is being set in the same layout transaction), but this is not required initially.

Interaction with resync: Resync of a non-immediate mirror during an active write epoch closes the epoch. The MDS takes the AW lock in EX mode (the same forced-close mechanism used for error handling and administrative operations — see Epoch Close), which forces all writers to flush and release. The resync client also takes the layout lock EX (via `LL_LEASE_RESYNC`) as part of its normal resync flow, but it is the AW lock EX — not the layout lock — that actually closes the epoch and quiesces writers. After resync completes, writers re-acquire AW locks and open a new epoch. This is expensive but correct — concurrent resync and active writing would be a consistency nightmare. This is the same behavior as the existing FLR resync-vs-write interaction.

Old client compatibility: Old clients that do not understand the IMMEDIATE flag are already handled — see §2.4.6. Clients lacking `OBD_CONNECT2_FLR_IMMED_MIRROR` are fenced from writing to files with IMMEDIATE mirrors.

Remaining design question:

- **Tiering transitions**: The promotion/demotion sequence (resync set IMMEDIATE clear IMMEDIATE delete) needs to be worked out with the tiering infrastructure.

4.3. Heterogeneous Compression Across Mirrors (Resolved)

Resolved — heterogeneous compression across immediate mirrors does not require special handling. Each mirror's OSC compresses independently from the original source pages; the redundant compression cost is negligible for 2-3 mirrors. See Bounce Page Reuse for the full analysis.