

Proposal: Reduce DNE Cross-MDT Metadata RPC

Summary

Lustre Distributed Namespace Environment (DNE) distributes directories and metadata across multiple Metadata Targets (MDTs) to scale aggregate metadata throughput. However, cross-MDT operations require acquiring remote locks, fetching remote metadata, and executing distributed transactions.

This overhead significantly impacts metadata-heavy workloads—such as AI reinforcement learning sandboxes, code-agent evaluations, and multi-node compilation environments. These workloads frequently create small files, materialize Git worktrees, unpack virtual environments, and perform atomic renames. Redundant MDT-to-MDT round trips per request compound metadata load and tail latency across the cluster.

This proposal focuses on high-rate, small-file lifecycles and metadata preparation phases. Bulk I/O workloads like streaming reads and checkpointing remain outside its scope.

When processing a modifying operation in DNE, the coordinator MDT executes four distinct phases:

- **Discover and validate objects:** Fetch remote attributes, xattrs (LinkEA, ACLs, layouts), and look up directory entries for permission checks, lock ordering, and validation.
- **Acquire distributed locks:** Enqueue remote LDLM inode-bit and PDO locks. Granting UPDATE or XATTR locks invalidates the local OSP cache.
- **Execute distributed transaction:** Commit local OSD updates and pack remote updates into a single compound `OUT_UPDATE` RPC per participating MDT, logging update records for recovery.
- **Retire recovery records asynchronously:** The commit thread cancels update-log records in the background after subtransactions commit. This background cleanup does not contribute to foreground client latency.

While Phase 3 batches updates and Phase 4 runs asynchronously, Phase 1 currently issues multiple synchronous `OUT_UPDATE` RPCs for attributes, xattrs, and lookups. This serialized fetch phase creates a major latency bottleneck.

The table below summarizes foreground and background `OUT_UPDATE` RPCs for representative operations on an unmodified tree during the reint handler window:

Operation	Cross-MD T Sync Reads	Foregroun d Tx RPCs	Delayed Log Cleanup	OUT_UPDA TE Total	LDLM Enqueues	Handler Duration
<code>mkdir</code> with a remote parent	8	1	1	10	2	9.5 ms
<code>rmdir</code> with a remote parent	2	1	1	4	3	6.1 ms
cross-MDT directory rename	3	5	1	9	1	20.2 ms
cross-MDT file rename	9	1	1	11	3	8.5 ms

Lock enqueue counts depend on which MDT coordinates the operation: renames coordinated on MDT0000 take the Big Filesystem Lock (BFL) locally, whereas renames coordinated on other MDTs add a fourth remote enqueue RPC.

Categorizing preparation reads shows recurring, redundant fetch patterns across operations.

`mkdir` with a remote parent — the parent here is the filesystem root:

#	Operation	Caller
1	<code>attr_get</code>	<code>mdt_check_resource_ids()</code> , before the parent lock
2	<code>lookup</code>	lockless pre-lock lookup (LU-10235)
3	<code>attr_get</code>	<code>mdd_lookup()</code> permission check, after the lock
4	<code>lookup</code>	authoritative lookup under the lock

#	Operation	Caller
5	lookup	mdd_create_sanity_check()
6	xattr_get	mdd_acl_init() — default ACL
7	xattr_get	mdd_pin_init() — pin
8	xattr_get	lod_get_default_lov_stripping() via lod_ah_init() — default LOV

Note that the target name is looked up three separate times, and reads 3 through 8 occur sequentially after acquiring parent locks.

rmdir with remote parent: Executes two reads after lock acquisition: an `attr_get` for `mdd_lookup()` permission validation and the lookup itself. Both sit immediately after the parent's name-hash enqueue and both belong to a single `mdd_lookup()`, which is the same shape the create and rename parent intents already handle. A separate trace of a file `unlink` shows an identical pattern, so one intent covers both. See *Implementation sequence* item 10.

Cross-MDT file rename: Involves synchronous preparation reads across both the source parent directory and the source object:

#	Object	Operation	Caller
1	source parent	xattr_get	mdd_parent_fid() via mdd_is_subdir() — linkEA
2	source parent	xattr_get	mdt_attr_get_pfid() — the same linkEA, read again
3	source parent	xattr_get	mdt_stripe_get() — LMV

#	Object	Operation	Caller
4	source parent	attr_get	mdd_lookup() permission check, after the lock
5	source parent	xattr_get	lod_stripping_load() — LMV
6	source parent	lookup	lod_index_try()
7	source	attr_get	lod_object_init() instantiation
8	source	attr_get	mdd_rename()
9	source	xattr_get	mdd_linkEA_prepare() — linkEA

Read 2 duplicates Read 1 because `mdd_parent_fid()` invalidates the object cache; Reads 4–6 are serialized inside a single `mdd_lookup()`; and Read 8 duplicates Read 7 because acquiring the source lock flushes the OSP cache.

Cross-MDT directory rename: Requires only three preparation reads (attributes, LinkEA, and LMV), but issues five foreground transaction RPCs due to update-log object allocation, making it the slowest operation (20.2 ms).

MDTest results below illustrate DNE overhead, 10 iterations each. Setting `max-inherit-rr=2` keeps most directory operations local to a single MDT. Setting `max-inherit-rr=10` forces most directory operations across MDTs.

Operation	Mean rate, single MDT (ops/s)	Mean rate, cross-MDT (ops/s)	Mean latency, RR=2 (ms/op)	Mean latency, RR=10 (ms/op)	Latency ratio, RR=10 / RR=2
Directory creation	30,154	4,066	0.269	2.000	7.4x slower
Directory rename	36,876	5,151	0.217	1.589	7.3x slower

Operation	Mean rate, single MDT (ops/s)	Mean rate, cross-MDT (ops/s)	Mean latency, RR=2 (ms/op)	Mean latency, RR=10 (ms/op)	Latency ratio, RR=10 / RR=2
Directory removal	21,913	3,444	0.366	2.390	6.5x slower
Directory stat	86,893	16,491	0.093	0.488	5.2x slower
File stat	37,868	29,708	0.212	0.270	1.3x
Tree removal	78	75	0.013	0.014	1.1x
File removal	21,038	20,277	0.381	0.395	1.0x
File read	20,233	19,748	0.396	0.406	1.0x
File rename	36,267	36,143	0.222	0.222	1.0x
File creation	19,023	19,488	0.428	0.411	1.0x
File cross rename	506	535	31.654	29.953	0.9x

- Every directory operation costs 5x to 7x more latency once it crosses MDTs. Creation and rename are worst at 7.4x and 7.3x, removal 6.5x, and even stat - which performs no modification - 5.2x.
- File operations are essentially unaffected, all within 1.0x to 1.3x. Spreading files over two MDTs neither helps nor hurts them measurably, which is what makes the directory figures above attributable to cross-MDT preparation rather than to the benchmark or the hardware.
- Tree creation is excluded from the comparison: its run-to-run variance exceeds 100% (see the appendix), so no ratio drawn from it would be meaningful.

One result does not fit the pattern and is worth stating separately. **File cross rename costs about 30 ms/op in both configurations** - 140x more than an ordinary file rename in the same run - and crossing MDTs makes no difference to it. Its cost is therefore not cross-MDT preparation at all but the filesystem-wide rename lock, which a cross-directory file rename takes whether or not the two directories live on the same MDT. It is addressed separately, under *Measured RPC reduction and performance*.

In summary, cross-MDT layouts scale file operations well, but directory operations incur substantial inter-MDT RPC overhead.

To eliminate this overhead, we introduce semantic MDT-to-MDT lock intents for operation preparation. A cross-MDT operation must enqueue a lock on the remote target regardless; the intent piggybacks the operation's synchronous preparation reads onto that enqueue. The target MDT performs the lookup, permission checks, validation, and inheritance against its own local storage while granting the lock, returning the results in the lock reply. Reads that would otherwise require separate `OUT_UPDATE` RPCs are eliminated, removing serialized round trips. Existing locking, transaction, replay, and recovery semantics remain unchanged.

The same mechanism covers `create` and `mkdir`, `rename`, as well as `unlink` and `rmdir`. A few reads fall outside it—some occur before a lock exists to carry an intent, and others must be reissued after lock acquisition—and are addressed separately. The proposal below outlines the design, and the measured impact is presented under *Measured RPC reduction and performance*.

Proposal

Design Overview

For each modifying request, the client selects one MDT as the coordinator based on the operation and target object FIDs. The request flows through the MDT, MDD, and LOD layers. LOD routes operations on local objects to the local OSD, while operations on remote objects go through an OSP proxy. Foreground MDT-to-MDT work consists of three main phases:

- **Remote lock acquisition:** Sending LDLM enqueue RPCs for objects managed by remote MDTs.
- **Remote preparation reads:** Issuing synchronous `OUT_UPDATE` RPCs to fetch attributes, xattrs, and directory lookups. These operations are strictly serialized because each result determines whether and how subsequent requests are issued.
- **Distributed transaction execution:** Committing local changes via the local OSD and packing remote updates into a single compound foreground `OUT_UPDATE` RPC per participating MDT.

After subtransactions commit, recovery-record cancellation may send an additional asynchronous `OUT_UPDATE` RPC. Although this step consumes network and server CPU resources, it runs in the background outside the client critical path.

Operations involving a remote parent directory require two sequential locks: a whole-directory lock (typically `LCK_CW`) to prevent splitting and restriping, followed by a name-hash lock (typically `LCK_PW`) to serialize access to the specific child entry. Rather than merging these lock enqueues, this proposal piggybacks operation preparation onto the second lock enqueue RPC.

```

None
Coordinator MDT                                Parent MDT
-----
enqueue whole-directory CW lock  -----> grant lock
                                <----- reply

enqueue name-hash PW lock
+ DNE create-prepare intent  -----> grant lock
                                run parent-side checks
                                derive child state
                                <----- lock + prepare result

```

The lock intent is operation-aware rather than a generic prefetch request:

```

C/C++
IT_DNE_CREATE_PREPARE = 0x00010000,
IT_DNE_RENAME_PREPARE = 0x00020000,
IT_DNE_UNLINK_PREPARE = 0x00040000,

```

Instead of sending raw parent attributes and xattrs back to the coordinator, the parent MDT performs the necessary checks locally while granting the lock. For creation operations, the parent MDT verifies parent directory validity, performs the authoritative child lookup, checks permissions (`MAY_WRITE` | `MAY_EXEC`) using the caller's credentials, validates layout versions and creation constraints, applies SGID and project inheritance, processes default ACLs, and evaluates pin and striping policies (LMV/LOV). Intermediate checks stay local to the parent MDT—only the overall status and essential initialization data for the child are returned.

Rename operations use the same intent structure with a smaller payload. When locking the source parent, the intent passes the target entry name, and the reply returns the resolved FID alongside the parent attributes and LMV layout—replacing the three synchronous reads inside `mdd_lookup()`. When locking the source object itself, no name is passed (`MCPR_NO_LOOKUP`), and the reply returns only the object's attributes required by `mdd_rename()`. Because rename requires a subset of create data, it shares the same request/reply wire structures.

Unlink and `rmdir` operations use an even simpler payload. The intent attached to the parent lock enqueue passes the entry name and receives the resolved FID and parent attributes. The coordinator then evaluates deletion permissions locally using those returned attributes. Like rename, unlink reuses the common request/reply structures without defining dedicated types.

Certain parent metadata parameters must be returned to initialize the child object created on the coordinator MDT:

Result	Why the coordinator needs it
Effective child mode and GID	SGID and ACL processing may change them
Effective project ID and flags	The child may inherit PROJINHERIT
Existing child FID	Required for EEXIST , replay, or partial-create handling
Access ACL	Written onto the new child when a default ACL exists
Default ACL	Inherited when the new child is a directory
Pin value	Inherited by a new file or directory
Effective LMV/LOV policy	Used to initialize the child's layout or placement

For standard files and directories without default ACLs, pins, or custom striping policies, the returned payload remains lightweight.

How the MDT changes

To prevent logic duplication and code drift between local and remote execution paths, parent-side validation logic is refactored from the MDD create path into a single helper function. This helper accepts the parent object, child name, and proposed child attributes, returning the prepared result. For local operations, the coordinator invokes this helper directly after acquiring local locks. For remote operations, the remote MDT executes the same helper during lock processing and returns the result in the lock reply. Namespace insertion, parent timestamp updates, child object creation, and transaction execution remain unchanged.

Modifications for each operation are isolated to the point where parent locks are acquired:

For each operation the change is confined to the point where the parent lock is taken.

Create and mkdir: The lock acquisition step includes the preparation intent request. For local parents, the lock is acquired and the helper executes inline. For remote parents, the intent is attached to the name-hash lock enqueue. Upon success, the caller receives the granted lock alongside the resolved FID and child attributes, skipping redundant local lookups and attribute derivations.

Rename: Two intents are attached to separate lock enqueues: one on the source parent's name-hash lock (returning the source FID, parent attributes, and striping) and one on the source

object lock (returning object attributes). The first intent result directly satisfies the lookup step, while the second populates the OSP object cache transparently.

Unlink and rmdir: A single intent attached to the parent name-hash lock returns the resolved FID and parent attributes. The coordinator checks removal permissions against these cached attributes without additional remote RPCs.

Across all operations, lock granting is decoupled from intent outcome: the requested lock is granted regardless of whether preparation succeeds.

This separation is critical to prevent lock leaks. Current locking APIs return non-zero errors only when lock acquisition fails, triggering cleanup routines that assume no locks are held. If a lock function returned an error due to a preparation failure while holding the granted lock, that lock would leak.

If preparation fails or produces incomplete data, the lock response reports success with an empty result payload. The coordinator then falls back smoothly to standard lock-protected lookups and RPC reads. Furthermore, because the coordinator maintains full client group credentials while intent requests carry limited supplementary groups, this fallback ensures security checks remain exact.

For creation operations, lockless pre-lock lookups (used to detect existing entries without taking write locks) are bypassed when targeting remote parents. Because remote lockless lookups require synchronous RPCs and preparation performs an authoritative lookup under lock anyway, skipping the pre-lock check saves one synchronous round trip for all successful creations.

OSP changes

The OSP remote-lock layer normally issues standard LDLM enqueue RPCs. When a prepare descriptor is attached and the remote target advertises support, OSP updates its handling to:

1. Allocate the new intent request format.
2. Pack the intent opcode, target name, proposed child attributes, caller credentials, and reply buffer bounds.
3. Set `LDLM_FL_HAS_INTENT`.
4. Call the existing synchronous `ldlm_cli_enqueue()`.
5. Unpack the per-operation status and prepared child state before releasing the request.

For create and unlink operations, the intent is attached exclusively to the second (name-hash) UPDATE lock enqueue, leaving the initial whole-directory lock unchanged. For rename operations, intents attach to both the source parent name-hash lock and the source object lock.

Returned state and the OSP object cache

Intent replies return two result types: lookup outcomes (consumed directly by the coordinator) and object metadata (parent attributes, ACLs, pins, and layouts), which is populated directly into OSP's object cache.

Populating the OSP cache keeps subsystem changes minimal. Higher-level callers (such as MDD reading ACLs or LOD reading striping) access data through standard read routines without requiring intent-aware interfaces.

Caching intent-returned metadata maintains strict consistency. Acquiring an **UPDATE** or **XATTR** lock normally invalidates the local OSP cache to drop stale pre-lock data. However, metadata returned in a lock intent reply is fetched by the remote target under the exact lock being granted, guaranteeing freshness equivalent to a post-lock read.

To maintain correctness, standard post-lock cache invalidation is suppressed when an intent reply arrives. OSP invalidates stale cache entries immediately after these entries are accessed.

Internal lock interface

The optional prepare descriptor passes from MDT through MDD and LOD down to OSP. It uses a dedicated intent field in **struct `ldlm_enqueue_info`** to avoid conflicting with **`ei_cbdata`** (used for callbacks and striped directory state):

C/C++

```
/* optional variable-length results an intent reply may carry */
enum md_lock_intent_buf {
    MLI_BUF_PATTR = 0,
    MLI_BUF_DEF_ACL    = 1,
    MLI_BUF_PIN    = 2,
    MLI_BUF_LMV    = 3,
    MLI_BUF_LOV    = 4,
    MLI_BUF_DEF_LMV    = 5,
    MLI_BUF_MAX
};

struct md_lock_intent {
    __u64        mli_opc;    /* enum ldlm_intent_flags */
    const void    *mli_request; /* intent-specific request body */
    size_t        mli_request_size;
    void          *mli_reply; /* intent-specific reply body */
    size_t        mli_reply_size;
    const char    *mli_name;  /* entry name the intent operates on */
    size_t        mli_namelen;
```

```

    /* destinations for the variable-length results, provided by the
     * caller: lb_len is the inline limit on the way out and the length
     * actually returned on the way back
     */
    struct lu_buf mli_reply_buf[MLI_BUF_MAX];
    unsigned int  mli_replied:1;    /* set when the reply was filled in */
};

struct ldlm_enqueue_info {
    ...
    struct md_lock_intent *ei_intent;
};

```

Callers without intents set `ei_intent` to `NULL`. MDD and LOD pass the enqueue info transparently, while OSP serializes the intent context for remote requests.

Remote MDT changes

The remote MDT intent dispatcher routes incoming requests to dedicated handlers sharing a unified wire format. Handlers grant the requested lock, reconstruct user credentials, execute the shared validation helper against local storage, and return status along with derived child attributes. Replayed and resent lock enqueues are processed consistently with existing intent handlers.

Locks are granted regardless of preparation status. If validation fails or optional attributes overflow response buffers, the reply flags the omission and the coordinator falls back to standard reads while retaining lock ownership.

Intent handlers are strictly read-only and create no transactions or durable updates. All namespace mutations and recovery logging remain managed by the subsequent distributed transaction.

Two key operational rules govern handler execution:

Lock parameters: Handlers grant the exact lock resources and policy modes requested by the coordinator rather than recalculating them locally, preventing mismatches caused by differing local node configurations.

Credential handling: Handlers adopt user identities directly from incoming RPC contexts without re-evaluating identity mappings, avoiding authorization discrepancies between MDTs. Handlers accept intent requests only over secure MDT-to-MDT connections.

Rename intent handlers support two modes: entry mode (resolves target entry names, checks lookup permissions, and returns FIDs, attributes, and striping) and object mode (operates directly on target objects and returns object attributes).

Wire protocol and interoperability

Wire protocol additions consist of one intent opcode per operation type alongside unified request and reply payloads. A bitmask flags which optional fields are present in each message.

Replies contain fixed-size fields for parent attributes and dynamic fields for returned xattrs. Output size is strictly bounded by requester-specified buffer limits.

Rename and unlink operations reuse create wire structures, setting only relevant fields and flags.

Intent support is negotiated as an MDT-to-MDT connection feature flag. If a peer lacks intent support (e.g., during rolling upgrades), the coordinator seamlessly falls back to standard lock enqueues and separate RPC reads.

Credentials and locking correctness

Because authorization checks execute on the parent MDT, intent requests must transmit full caller context—including effective UID/GID, group memberships, capabilities, and umask. Remote target nodes execute checks strictly under the caller's identity.

Prepared results remain protected by locks held across the preparation and transaction execution phases, covering child entry existence, parent attributes, permissions, ACLs, pins, and layout policies.

Locking ordering follows standard DNE rules to prevent deadlocks. All local reads performed by target intent handlers access local storage directly, avoiding recursive OSP RPCs.

Replay, errors, and fallback

Intent handlers are read-only. On request resends or replays, the target MDT re-evaluates the intent under existing lock protection.

Intent replies indicate three possible processing outcomes:

- **Incomplete preparation:** Reported as success with an empty result payload. The coordinator proceeds with standard lookups and reads under acquired locks.
- **Advisory results:** Processed similarly using fallback lookup paths.

- **Missing or truncated xattrs:** Indicated by clear bitmask flags. The coordinator uses returned attributes and fetches only the missing item individually.

Transaction replays bypass intents entirely. Replayed creations require full lockless lookup and version checks, acquiring locks without intents. Replayed renames and unlinks execute explicit name resolution to preserve version bookkeeping.

If intents are unsupported, unpackable, or incomplete, the system defaults unconditionally to standard DNE workflows without altering operation semantics.

Measured RPC reduction and performance

A prototype has been built to verify the idea, and the figures below are taken from it rather than estimated. It covers **mkdir** and **rename** only for now. Everything reported here should therefore be read as evidence that the approach works and is worth completing, not as the result of the finished design.

RPC counts

Measured from single-operation debug traces, one before-and-after pair per operation:

Operation	OUT_UPDATE before	after	Sync before	reads after
mkdir with a remote parent	7	3	5	1
cross-MDT file rename, target does not exist	12	7	10	5
cross-MDT file rename, target exists	16	9	14	7

The LDLM enqueue count is unchanged in every case - two for **mkdir**, four for rename - because the intents ride on enqueues that were already being issued.

The **mkdir** pair was taken on a different filesystem state from the survey in the Summary, which is why its absolute counts differ from the 8 reads recorded there; each pair is internally consistent, before against after on the same tree.

For `mkdir` the post-lock parent reads go to zero: the lookup, the permission check, the default ACL, the pin and the default layout all arrive in the lock reply. For the target-exists rename, every removed read is accounted for as well: the duplicate linkEA read; the parent's attributes, striping and lookup, all three of which belong to a single lookup call; the source's attributes; and two parent attribute reads that the old cache invalidation had stranded. Confirmation in the traces is direct - there is no lookup RPC anywhere in the patched log, and each removed read appears instead as a cache hit.

Not every read can be merged into an enqueue, which is why the counts do not fall to the transaction alone. Some happen before any lock exists to attach an intent to: an object's attributes are fetched when it is first instantiated, well before the operation reaches its locking phase. Others have to be reissued after a lock is granted, precisely because the grant is what made the earlier answer unusable - the re-check taken under the rename lock is the clearest case, since obtaining a fresh answer is the entire point of it. Those remain ordinary synchronous reads.

IOPS

mdtest with two MDTs. The baseline is the unmodified tree; the prototype carries the `mkdir` and `rename` intents. Two directory layouts were measured, because they stress the design differently: all tasks working in one shared directory, and one directory per task.

Shared directory

5 iterations each.

Operation	baseline (ops/s)	prototype (ops/s)	change	
Directory creation	1,015 ± 50	2,058 ± 92	+103%	significant
File cross rename	296 ± 4	399 ± 30	+35%	significant
Directory rename	1,740 ± 53	1,969 ± 70	+13%	significant
Directory removal	1,143 ± 33	1,211 ± 42	+6%	within noise
File read	16,056 ± 837	16,295 ± 1,027	+1%	within noise
File rename	5,420 ± 177	5,456 ± 287	+1%	within noise

Operation	baseline (ops/s)	prototype (ops/s)	change	
File creation	4,826 ± 241	4,831 ± 385	0%	within noise
File stat	41,081 ± 1,724	38,484 ± 3,262	-6%	within noise
File removal	3,147 ± 123	2,847 ± 533	-10%	within noise
Directory stat	20,050 ± 1,397	15,475 ± 846	-23%	significant

Tree creation and removal are omitted; their variance makes any comparison drawn from them meaningless.

The controls make this run readable. File creation, file read and file rename are all within 1.5%, and none of them is touched by an intent, so there is no drift between the two runs to discount. Whatever else moved, moved because of the change.

Directory creation doubles, which is the result the design predicts most directly: a shared parent is remote for half the tasks, and every such create drops from 7 RPCs to 3.

File cross rename gains 35%, and **directory rename 13%**. Cross rename remains slow in absolute terms because its cost is dominated by the filesystem-wide rename lock rather than by preparation reads, which the next section takes up.

Directory removal is unchanged, and it is the one operation here whose intent has not been built. That is the control the rest of the table should be judged against.

Directory stat loses 23%, and this is a genuine regression. Though directory stats seems varies a lot, since the stats mostly are memory operation in this test, and I will verify this further in a more scale test environment.

One directory per task

10 iterations each.

Operation	baseline (ops/s)	prototype (ops/s)	change
Directory creation	4,066 ± 566	5,657 ± 1,534	+39%
File cross rename	535 ± 20	664 ± 25	+24%
File rename	36,143 ± 1,777	41,859 ± 2,155	+16%

Operation	baseline (ops/s)	prototype (ops/s)	change
File removal	20,277 ± 580	23,242 ± 857	+15%
Directory rename	5,151 ± 848	5,809 ± 1,543	+13%
File creation	19,488 ± 566	21,923 ± 833	+12%
File read	19,748 ± 798	22,063 ± 1,421	+12%
Directory stat	16,491 ± 1,423	16,721 ± 1,891	+1%
Directory removal	3,444 ± 663	3,465 ± 692	+1%

Appendix

Master with current master

```
Shell
Single MDT operation. (with --max-inherit-rr=2)

SUMMARY rate (in ops/sec): (of 10 iterations)
Operation      Max      Min      Mean      Std Dev
-----
Directory creation 36545.220 25034.586 30154.402 3896.568
Directory stat    98784.804 77697.116 86893.120 7385.451
Directory rename  38471.338 35097.273 36876.413 1189.058
Directory removal 22711.213 19945.653 21912.575 871.136
File creation     20463.339 12756.831 19023.351 2255.510
File stat        40990.573 33739.459 37868.077 2211.085
File read        21496.298 18611.273 20233.337 811.341
File rename      39783.351 29319.802 36266.547 3020.317
File cross rename 528.074   491.083   505.670   10.929
File removal     22486.627 19972.187 21038.295 880.323
Tree creation    31.514    1.707     8.087     8.699
Tree removal     101.491   58.520    78.038    13.549

SUMMARY time (in ms/op): (of 10 iterations)
Operation      Max      Min      Mean      Std Dev
-----
Directory creation 0.320    0.219    0.269     0.034
Directory stat    0.103    0.081    0.093     0.008
Directory rename  0.228    0.208    0.217     0.007
Directory removal 0.401    0.352    0.366     0.015
File creation     0.627    0.391    0.428     0.071
File stat        0.237    0.195    0.212     0.013
File read        0.430    0.372    0.396     0.016
File rename      0.273    0.201    0.222     0.021
File cross rename 32.581   30.299   31.654     0.675
File removal     0.401    0.356    0.381     0.016
Tree creation    0.586    0.032    0.244     0.189
```

Tree removal	0.017	0.010	0.013	0.002
--------------	-------	-------	-------	-------

2 MDT operation. (with --max-inherit-rr=10)

```
mpirun --hostfile ./machine_file --mca routed direct --map-by node -np 16 ./src/mdtest -n 500
-i 5 -v --renameCrossDir --renameDestDir /mnt/lustre/rename_dst -P -u -d /mnt/lustre/mdtest
```

SUMMARY rate (in ops/sec): (of 10 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	5148.376	3382.648	4065.842	566.032
Directory stat	18719.181	14892.648	16491.074	1422.617
Directory rename	6556.759	4225.317	5150.864	848.020
Directory removal	4680.177	2757.504	3443.849	663.302
File creation	20141.997	18473.397	19488.327	565.500
File stat	31112.649	27544.882	29707.841	1151.295
File read	20757.651	18785.162	19748.132	797.785
File rename	40060.497	33879.232	36143.238	1777.142
File cross rename	555.181	489.168	534.888	20.116
File removal	20958.316	19236.684	20277.368	580.470
Tree creation	33.481	1.719	8.082	9.342
Tree removal	95.366	50.204	74.723	14.640

SUMMARY time (in ms/op): (of 10 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	2.365	1.554	2.000	0.257
Directory stat	0.537	0.427	0.488	0.042
Directory rename	1.893	1.220	1.589	0.244
Directory removal	2.901	1.709	2.390	0.388
File creation	0.433	0.397	0.411	0.012
File stat	0.290	0.257	0.270	0.011
File read	0.426	0.385	0.406	0.016
File rename	0.236	0.200	0.222	0.011
File cross rename	32.709	28.819	29.953	1.183
File removal	0.416	0.382	0.395	0.011
Tree creation	0.582	0.030	0.259	0.202
Tree removal	0.020	0.010	0.014	0.003

```
mpirun --hostfile ./machine_file --mca routed direct --map-by node -np 16 ./src/mdtest -n 500
-i 5 -v --renameCrossDir --renameDestDir /mnt/lustre/rename_dst -P -d /mnt/lustre/mdtest
```

SUMMARY rate (in ops/sec): (of 5 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	1068.986	966.217	1014.511	49.666
Directory stat	22026.015	18469.147	20050.129	1396.732
Directory rename	1810.878	1667.056	1739.928	52.968
Directory removal	1180.598	1092.735	1143.245	32.849
File creation	5093.213	4592.407	4825.595	241.412
File stat	42986.924	38724.233	41080.586	1723.832
File read	17318.122	15229.364	16056.340	837.172
File rename	5581.042	5176.476	5419.780	177.371
File cross rename	300.000	290.643	296.067	4.034
File removal	3303.411	3011.372	3146.996	122.785
Tree creation	111.881	81.733	89.341	12.704

Tree removal	280.106	139.239	235.866	57.592
--------------	---------	---------	---------	--------

SUMMARY time (in ms/op): (of 5 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	8.280	7.484	7.901	0.382
Directory stat	0.433	0.363	0.401	0.028
Directory rename	4.799	4.418	4.601	0.141
Directory removal	7.321	6.776	7.002	0.204
File creation	1.742	1.571	1.661	0.082
File stat	0.207	0.186	0.195	0.008
File read	0.525	0.462	0.499	0.025
File rename	1.545	1.433	1.477	0.049
File cross rename	55.050	53.333	54.050	0.740
File removal	2.657	2.422	2.545	0.099
Tree creation	0.012	0.009	0.011	0.001
Tree removal	0.007	0.004	0.005	0.002

prototype with cross-MDT remote intent (directory per rank)

Shell

```
mpirun --hostfile ./machine_file --mca routed direct --map-by node -np 16 ./src/mdtest -n 500
-i 5 -v --renameCrossDir --renameDestDir /mnt/lustre/rename_dst -P -u -d /mnt/lustre/mdtest
```

SUMMARY rate (in ops/sec): (of 5 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	6903.193	5607.542	6170.828	624.057
Directory stat	19987.748	15444.822	17688.766	2034.637
Directory rename	8076.436	4496.870	5571.709	1464.479
Directory removal	3613.489	3052.994	3373.492	206.127
File creation	22617.265	20314.634	21415.285	961.808
File stat	35382.579	31824.847	33196.975	1358.646
File read	22647.491	18073.463	20501.750	1960.361
File rename	43154.328	39763.032	41087.617	1407.015
File cross rename	626.293	553.305	600.452	28.214
File removal	23695.846	18750.416	21411.747	1980.724
Tree creation	59.134	2.485	15.264	24.637
Tree removal	91.200	71.388	79.029	7.883

SUMMARY time (in ms/op): (of 5 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	1.427	1.159	1.307	0.128
Directory stat	0.518	0.400	0.457	0.053
Directory rename	1.779	0.991	1.502	0.319
Directory removal	2.620	2.214	2.379	0.151
File creation	0.394	0.354	0.374	0.017

File stat	0.251	0.226	0.241	0.010
File read	0.443	0.353	0.393	0.039
File rename	0.201	0.185	0.195	0.007
File cross rename	28.917	25.547	26.696	1.314
File removal	0.427	0.338	0.376	0.036
Tree creation	0.402	0.017	0.234	0.160
Tree removal	0.014	0.011	0.013	0.001

V-1: Entering PrintTimestamp...

```
mpirun --hostfile ./machine_file --mca routed direct --map-by node -np 16 ./src/mdtest -n 500
-i 5 -v --renameCrossDir --renameDestDir /mnt/lustre/rename_dst -P -u -d /mnt/lustre/mdtest
```

SUMMARY rate (in ops/sec): (of 5 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	2081.462	1868.925	1959.301	101.998
Directory stat	16827.007	13571.741	14500.075	1318.825
Directory rename	2072.866	1827.079	1922.569	97.052
Directory removal	1240.509	912.301	1072.732	124.817
File creation	4937.138	4657.256	4854.427	112.892
File stat	44902.208	38255.366	40419.230	2701.414
File read	16258.513	14440.411	15169.564	829.933
File rename	5472.177	5068.488	5251.367	171.430
File cross rename	433.315	378.367	401.872	23.175
File removal	3143.691	2779.128	2959.365	140.226
Tree creation	160.769	33.232	101.722	53.386
Tree removal	291.879	3.675	150.559	102.056

SUMMARY time (in ms/op): (of 5 iterations)

Operation	Max	Min	Mean	Std Dev
-----	---	---	----	-----
Directory creation	4.281	3.843	4.092	0.209
Directory stat	0.589	0.475	0.555	0.045
Directory rename	4.379	3.859	4.169	0.205
Directory removal	8.769	6.449	7.540	0.887
File creation	1.718	1.620	1.649	0.039
File stat	0.209	0.178	0.199	0.013
File read	0.554	0.492	0.529	0.028
File rename	1.578	1.462	1.525	0.050
File cross rename	42.287	36.925	39.918	2.255
File removal	2.879	2.545	2.708	0.129
Tree creation	0.030	0.006	0.013	0.010
Tree removal	0.272	0.003	0.059	0.119