File-Level Replication High Level Design

1. Introduction

The Lustre* file system was initially designed and implemented for HPC use. It has been working well on high-end storage that has internal redundancy and fault-tolerance. However, despite the expense and complexity of these storage systems, storage failures still occur and Lustre cannot currently be more reliable than the individual storage and servers components on which it is based. The Lustre file system has no mechanism to mitigate storage hardware failures and files become inaccessible if a server is inaccessible or out of service. This deficiency of Lustre file system in the area of fault tolerance becomes more acute in both large systems with a large number of individual components, and also in the era of cloud computing environments built with commodity hardware lacking redundancy and where failure rates are high.

In this document, a solution to the problem of data accessibility in the face of failing storage will be designed. The chosen solution is to use configurable file-level replication within the file system and is described in the Solution Architecture. With file-level replication, any Lustre file can store the same data on multiple OSTs in order for the system to be robust in the event of storage failures. In addition, IO availability can be improved because given a choice of multiple replicas, the best suited can be chosen to satisfy an individual request. Furthermore, for files that are concurrently read by many clients (e.g. input decks or executables) the aggregate parallel read performance of a single file can be improved by creating multiple replicas of the file's data.

The considerable functional improvements that replication promises also pose significant challenges in design and implementation. In this document we present a design for an initial implementation of replication with limited write support – writing to a replicated file will degrade it to a non-replicated file, and only one replica will be updated directly during the write while other replicas will be simply marked as stale. The file can subsequently return to replicated state again by synchronizing among replicas with command line tools (run by the user or administrator directly or via automated monitoring tools). There will typically be some delay before returning the other replicas are re-synced following a write, which is why the first phase implementation is called *delayed write*.

While delayed write replication will not cover all types of failures, it will still be useful for a wide range of uses. Since the replication of file data can be chosen on a per-file basis, it is practical to replicate one of every N checkpoint files (e.g. once a day), so that a long-running application can be restarted even in the face of concurrent compute node and OST failures. This also avoids the need to double or triple the storage capacity of the whole filesystem as is required for device-level replication.

In this document, we will present a solution for delayed write for Lustre. Within this document, the terms RAID-1 or RAID-0+1, *replicated*, and *mirr ored* are all used to reference the files with multiple copies of the data. More specifically, the term RAID-1 is used in the context of the file layout and *replicated* is used in the context of data itself.

2. Changes to the Solution Architecture

In the solution architecture, we presented a replicated file read with dynamic pages, i.e., the sub page of a cl_page can be replaced dynamically. Therefore, when a pre-selected replica fails to serve a read, we can reselect another replica and change the sub pages correspondingly, and then send the request to the new replica without interactions with upper layers.

After further analysis, a major drawback to this scheme has been identified: the retrying request ideally can be served by only one OSC otherwise the request has to be split. Spiting the request add significant complexity, which then puts a limit on the stripe size choosing - the stripe size has to be the same for replicas to make sure that the request can be redirected to only one OSC.

In this design, if the pre-selected replica becomes unavailable, the IO will be restarted all the way up in the LLITE layer. In addition, if the read is for readahead, then the failed read request will simply be ignored. This will leave some pages in memory without uptodate, or actively destroy those pages. This design makes it unnecessary to put a limit on stripe size.

3. RAID-1 Layout

Based on layout enhancement project, replication layout format and a set of layout operations will be defined. At present, there are no dedicated interfaces to interact with layouts, and using setxattr to modify the whole layout atomically is the current work-around. However, setxattr is insufficient to support the complex interactions needed with layout that RAID-1 requires. A set of layout operations will be designed to support the needs of replication. This work builds on the Layout Enhancement project.

3.1 Layout format

Layout format of replication is based on the layout enhancement project where composite layout format is defined. Replication layout will be

defined as follows:

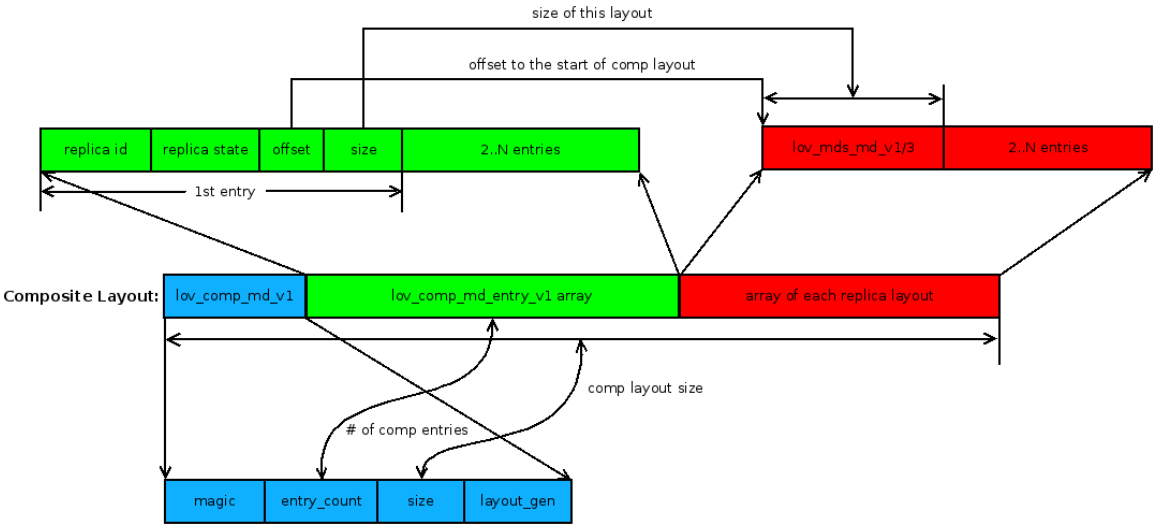
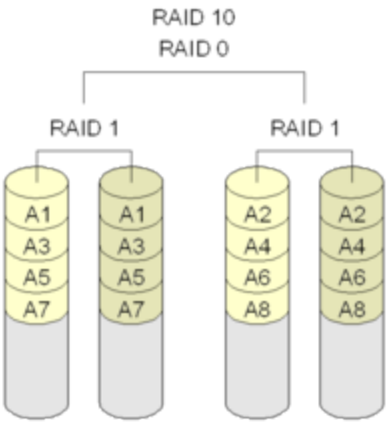


Diagram of composite layout for replication By Jinshan Xiong

The diagram above illustrates that each replica has a unique ID. The state of each replica is maintained separately. The replica state can be modified via layout operation interfaces.

3.2 RAID-0+1 and RAID-1+0 layout

A RAID-0+1 layout is mirroring across a collection of RAID-0 layouts. Conversely, RAID-1+0 is striping over a collection of RAID-1 mirrors. RAID-0+1 is simpler to implement within in the current Lustre IO framework and file layout infrastructure since it is possible to take any existing RAID-0 file and add one or more replicas to create a RAID-0+1 file. RAID-1+0 is complex to implement in the initial phase. RAID-1 (mirror of individual objects) will be supported as a degenerate case of RAID-0+1.



3.3 Recursive RAID-1 layout

RAID-1 of RAID-1 layout is not supported.

4. Command line tools

4.1 Create RAID-1 file by parameters similar to lfs setstripe

To create RAID-1 file, we have to make sure that different replicas must be on different OSTs, even OSSes and racks. An understanding of cluster topology is necessary to achieve this aim. In the initial implementation the use of existing OST pools mechanism will allow separating OSTs by any arbitrary criteria: i.e. fault domain.

In practice, we can take advantage of OST pools by grouping OSTs by topological information. Therefore, when creating a replicated file, the users can indicate which OST pool can be used by replicas.

```
lfs layout --pattern mirror [--component idx] [--preferred] [other setstripe options]] <file_name>
```

The above command will create an empty RAID-1 file. The `--component` option indicates the replica component number that is being specified, and can be repeated multiple times so that multiple replicas can be created at one time. The `--prefer` option marks that replica as being preferred for writes (e.g. stored on fast/local OSTs). As mentioned above, it is recommended to use the `--pool` option (one of the `lfs setstripe` options) with OST pools configured with independent fault domains to ensure different replicas will be placed on different OSTs and/or servers and/or racks, therefore availability and performance can be improved. If the setstripe options are not specified, it is possible to create replicas with objects on the same OST(s) in the Phase 1 implementation, which would remove most of the benefit of using replication in that case.

4.2 Extend an existing file to RAID-1 format

```
lfs layout --pattern mirror --extend [--read-only|-r] [--prefer] [other setstripe options] <raid1_file>
```

This command will append a replica by stripe options into an existing file `raid1_file`. The existing file can already be a replicated file, or just a normal file.

This command will create new volatile file with any optional setstripe options that are specified, or using the defaults inherited from the parent directory or filesystem. The file contents will be copied from the existing file to volatile file and then the layout will be modified to add the volatile file as a new replica. Lease will be held on the existing file to prevent it from being modified during the copy. If the existing file is modified during this operation, `-EBUSY` will be returned to the replication tool. Administrators can repeat this command later.

The `--read-only|-r` option can be specified if the file is known to be accessed by older clients that do not directly support File Level Replication, which will prevent them from marking the file stale if the file is opened `O_RDWR` (in particular by Fortran programs), but will prevent the file from being modified. This is similar to marking the file immutable, but it can still be deleted or have attributes such as timestamps and ownership modified..

4.3 Split a specific replica index from a RAID-1 file

```
lfs layout --split --component idx <raid1_file> [target_file]
```

This command will split the specified replica by its component index `idx`. `raid1_file` must be a RAID-1 file otherwise the command will be returned with -ENOTSUPP. If target_file is specified, the file will be created with the layout of the split replica; otherwise the layout of the specified replica will be destroyed.

4.4 Replica resynchronize

```
lfs layout --resync <raid1_file>
```

This command is used to resynchronize an out-of-sync RAID-1 file. If there are no any stale replicas for the replicated file, this command does nothing. Otherwise, this command will first hold the exclusive lease of raid1_file, and then the file will be resynchronized.

In terms of resynchronization, the file will be read with normal read, because read must only use uptodate replicas; and then the read contents will be written to stale replicas with a dedicated replica write interface. We're going to use direct IO for the read and write in this case. After the replicas are synchronized, this command will change the layout to mark the replicas as uptodate.

4.5 Verify replicated files

`lfs layout --verify <raid1_file>`

This command is used to verify that each replica of a replicated file contains the exactly same data. This command will pull data from each replica, and then do comparison or calculate checksum, so that we can get some help from hardware, to make sure they're the same.

5. Read from replicated file

In the design of replication, an effort is made to mandate writers to write all replicas and readers can get valid data by reading from any replica. In the Phase 1 implementation, only a single replica will be written and there will be a delay before the other replicas are updated from the uptodate replica. However, if there is any hidden error in disk, two readers may be returned with inconsistent result if the readers happen to use different replicas. The solution of this problem is out of the scope for Phase 1 and we make the assumption that all uptodate replicas include the same data.

5.1 Policy to select read replica

When serving a read, a policy will be run in the IO framework to select a replica. The initial replica would ideally be chosen by the following information:

- Status of replica - avoid stale replicas
- Type of replica - replica on faster disks should be chosen firstly
- Network topology - Nearer the replica, faster the IO can be finished
- Load balance - to avoid busy servers

However, running this kind of policy needs information that is not currently available at the client, so the Phase 1 initial replica selection is only by replica status as seen by the client.

The pseudo code of replica selection policy is as follows:

```
for each replica of the file; do
               if (replica is stale)
                       continue;
               if (replica including inactive OST)
                       continue;
               if (replica was previously tried and failed)
                       continue;

               add this replica to list of available replicas;
               break;
        done
        select replica from available replicas
```

Replicas may be prioritized using the past usage statistic. If multiple valid replicas exist, clients should deterministically select one of them (e.g. c`lient_nid % replica_count`) so that the load is distributed across replicas. This will mean that multiple clients reading from the same file should utilize the bandwidth from all the servers.

Replica selection occurs in the IO initialization phase; lock request relies on the result of selection. If a read from a replica fails, then the replica is marked stale and the replica selection process is run again. On successful reads, the same replica is re-used for the later reads.

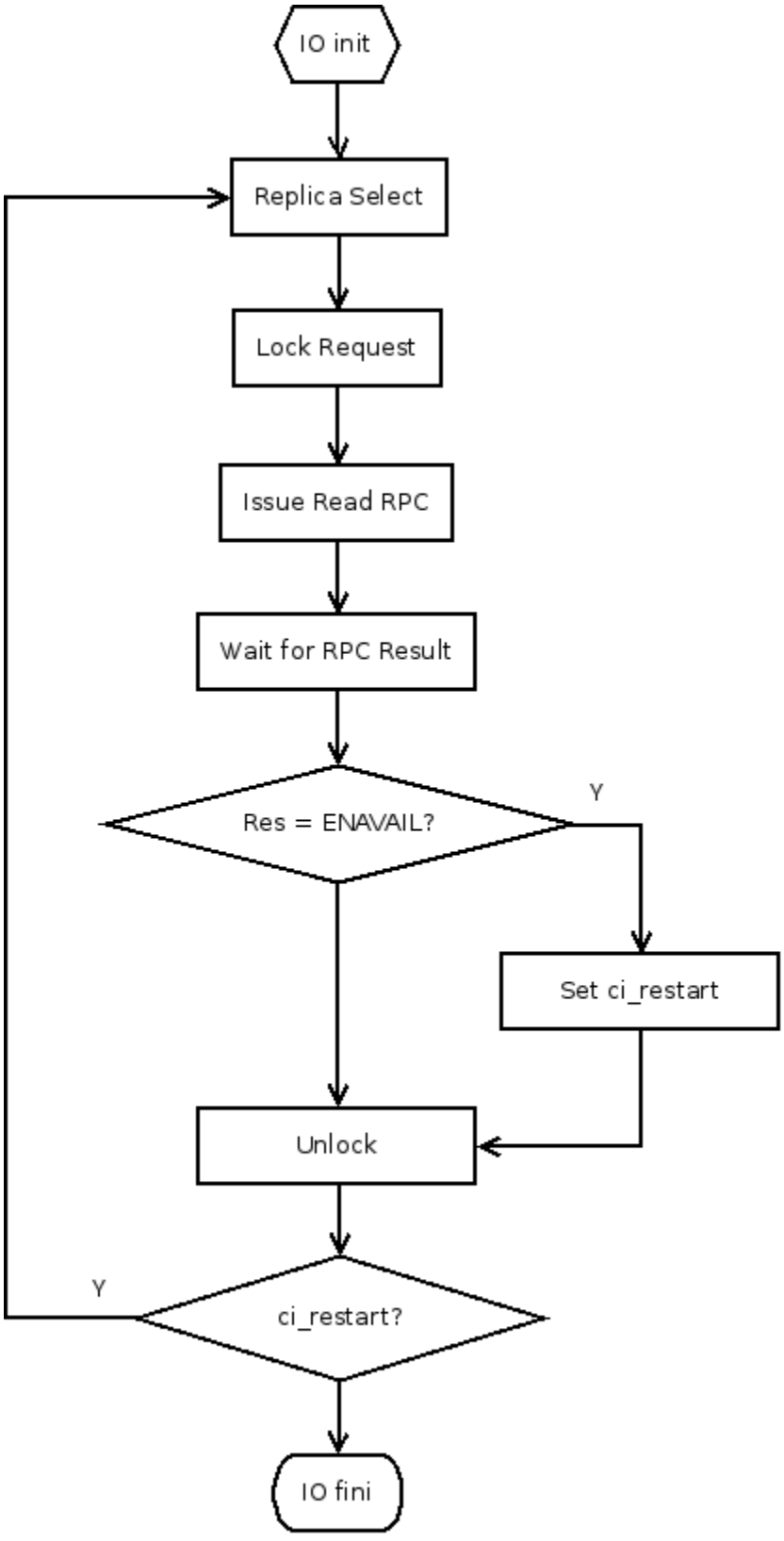
5.2 IO framework for read

IO framework is on the client side. Clients must recognize the RAID-1 layout and choose a replica to serve read requests by policies.

To improve read availability, the IO framework for read must have a retry mechanism, which means in case a replica becomes unavailable to serve a read, the IO framework must retry that IO with another replica, and so on. Applications must not see any errors if the data can be read from one of the replicas finally.

When reading from replica, the IO RPC should be issued with `rq_no_delay` bit set in `ptlrpc_request` so that if the object becomes unreachable, the RPC won't be stuck in the resending list at the RPC layer, but instead it will return with the error code `-ENAVAIL`. The IO

framework should reissue the IO and invoke the replica selection policy to try the next replica.



Read RPC flow chart

The diagram above depicts a typical process of sync read. For sync read, the reading process has to wait for the RPC result, therefore if it sees the error `-ENAVAIL` that indicates the replica is unavailable, it will set `cl_io::ci_restart` so that the IO can be restarted.

5.3 Read ahead

Read ahead RPCs are asynchronous, the process that issues the RPC won't wait for the result of RPC, nor will the pages be needed right away. In this case, it's not necessary to take immediate actions if the replica becomes unavailable. The pages will remain in cache without uptodate until they are needed by process, and then sync RPC will be sent with valid replicas being selected.

6. Write to replicated file

For simplicity of implementation, write is implemented by two phases:

- Phase 1: Delayed write – the MDS will choose one replica, named primary, to update and mark the others STALE, based on a hint from the client. An external tool will be used to synchronize the other (STALE) replicas after write is done.
- Phase 2: Immediate write – the major problem with delayed write is that even if all replicas are accessible when the file is being written, it will still only update one replica. This is inefficient since the file will need to be read and updated again at a later time. Immediate write also picks the primary replica for update and marks the others STALE, but it also writes to the secondary replica(s) at the same time. When the write completes, after all dirty data has been written, the writing client(s) can set the secondary replicas' state to SYNC again. Immediate write will use fanout page and it's out of the scope in Phase 1. The design and implementation of Immediate write is out of scope for this project.

The major purpose of picking the primary replica is to simplify the recovery and coordinating concurrent read/write access to a replicated file. The primary replica selection process is similar to that for read, with the main difference that once a primary replica has been selected all other clients reading or writing the file must also use that same replica as the primary. Once the other replicas are marked stale, the replicated file degenerates to a normal file so read/write locking will ensure data coherency and recovery.

It will also be possible to persistently mark a replica in the layout as preferred for selection as the primary replica for writes, if it is stored on faster OSTs compared to others (e.g. flash vs. SATA). Since the client is able to specify a hint for selecting the primary replica, this will allow future extension for the client to be able to select the replica based on LNET locality or observed OST performance.

6.1 Delayed write

With delayed write, one replica will be chosen as primary replica to dirty and the others will be marked as STALE. The MDT can either do this at o pen(O_WRONLY), or will be notified to change the layout before write really occurs for open(O_RDWR).

After the write is finished, an external tool will acquire read mode lease of the file and use replica write to copy data from the primary replica to staled ones. Marking a replica stale will be recorded in the Lustre ChangeLog so that the resync tool or external policy engine can efficiently detect and resync stale replicas.

The benefit of delayed write is that it's recovery-free, because only one replica is chosen to write so we don't need to worry about the differences among replicas at write time. This also defers the IO overhead of writing to multiple replicas until after the primary replica has finished writing, allowing the full write bandwidth to be used for the primary replica. However, the drawback is that the resynchronization will consume of IO bandwidth at later time. Delayed write is just a temporary solution before immediate write is introduced.

6.1.1 Layout Change for Write

Before writing a replicated file, MDT has to be notified in advance so that it can choose one replica as the primary and mark others as STALE. Then, the layout generation will be increased to notify the other clients to clean up their cache. The process of updating layout must be a synchronous write operation on the MDT, though this is not expected to be a significant overhead since updating files after initial creation is a relatively rare event, and the MDT normally is using high-IOPS storage and can commit the update relatively quickly.

If the file is opened with O_WRONLY, the MDT will immediately pick one replica out as primary and mark the others as STALE at file open time.

6.1.2 Versioned Write Request and Replica

OST write requests will carry the layout generation of the file when it was opened for write. OST will check if this version number matches the generation number of replica, see section 6.1.3, or the write request will be denied because it is from a client with an incorrect understanding of the current file layout.

The version of write request will be filled in cl_req_attr_set(), and the source of version is fetched from struct obd_client_handle, where

a new field is added to remember the layout generation when the file is opened for write. Please notice that the file's layout generation may be unavailable due to DLM false sharing, so we can't rely on comparing current file version to decide if to send the write request.

Versioned replica applies on OST objects of replicas; each OST objects will store a 32-bit version number on disk to identify which layout generation of the file the object is attached to.

Before the objects can be written, the client has to update the layout generation of replica. On the OST side, the write can only succeed if it matches the layout generation of objects.

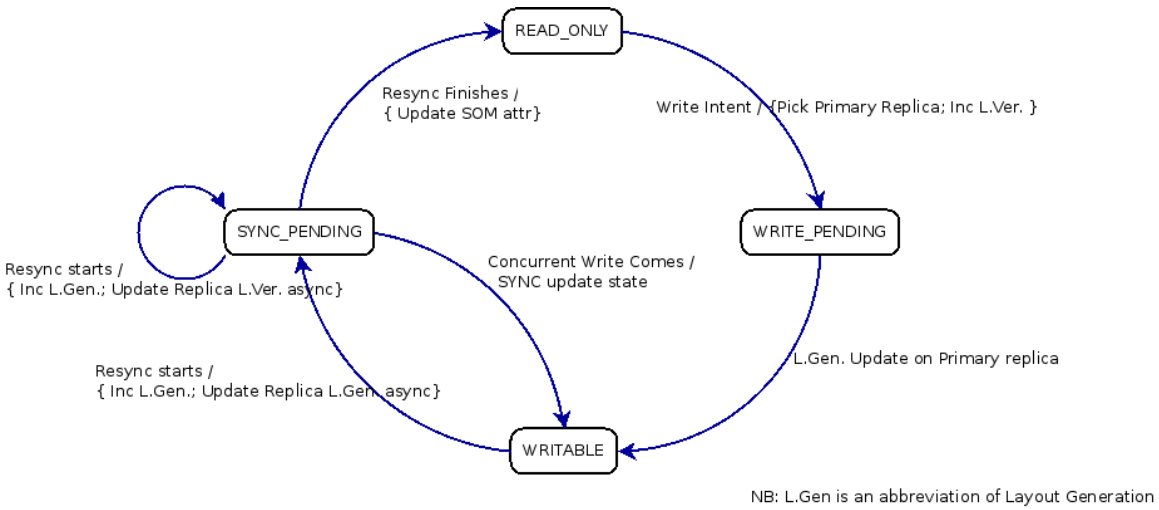
The major purpose of versioned replica is to prevent evicted clients from modifying the replicated file. It also enforces data protection because an arbitrary writing request, for example requests from older clients, will fail to modify the data.

6.1.3 Replicated File State

Once write is supported, replicated file can be one of the following states:

- `READ_ONLY` - the replicated file is on read only state;
- `WRITE_PENDING` - the file is going to be written, layout change has been finished. This is a middle state between `WRITEABLE` and `READ_ONLY`;
- `WRITABLE` - the file is writable. No extra operation is needed to write the file.
- `SYNC_PENDING` - the file is picked to resynchronize, layout generation has increased so that evicted client can't write to the replicas.

Replicated files transit their state when they are being written and resynchronized. The diagram below shows the FSM to transit the states.



State Machine of Replicate File

When the first write comes, and the replicated file is in `READ_ONLY` state, it will increase the layout generation and pick the primary replica, then change the file state to `WRITE_PENDING` and mark non-primary replicas `STALE`. The writing client will then uptodate primary replica's layout generation, and then notify the MDT to set the replicated file state to `WRITABLE`. After the replicated file is resynchronized, the file state will go back to `READ_ONLY` again.

Whenever a writing client sees `WRITE_PENDING` state replicated files, it will update the replica generation and then notify the MDT to set the state to `WRITABLE`. If clients see a `WRITABLE` state replicated files, it can write the file without any extra operations. Therefore, the first write is expensive because it needs two transactions to talk with the MDT; but for the following writes, it won't have extra overhead at all.

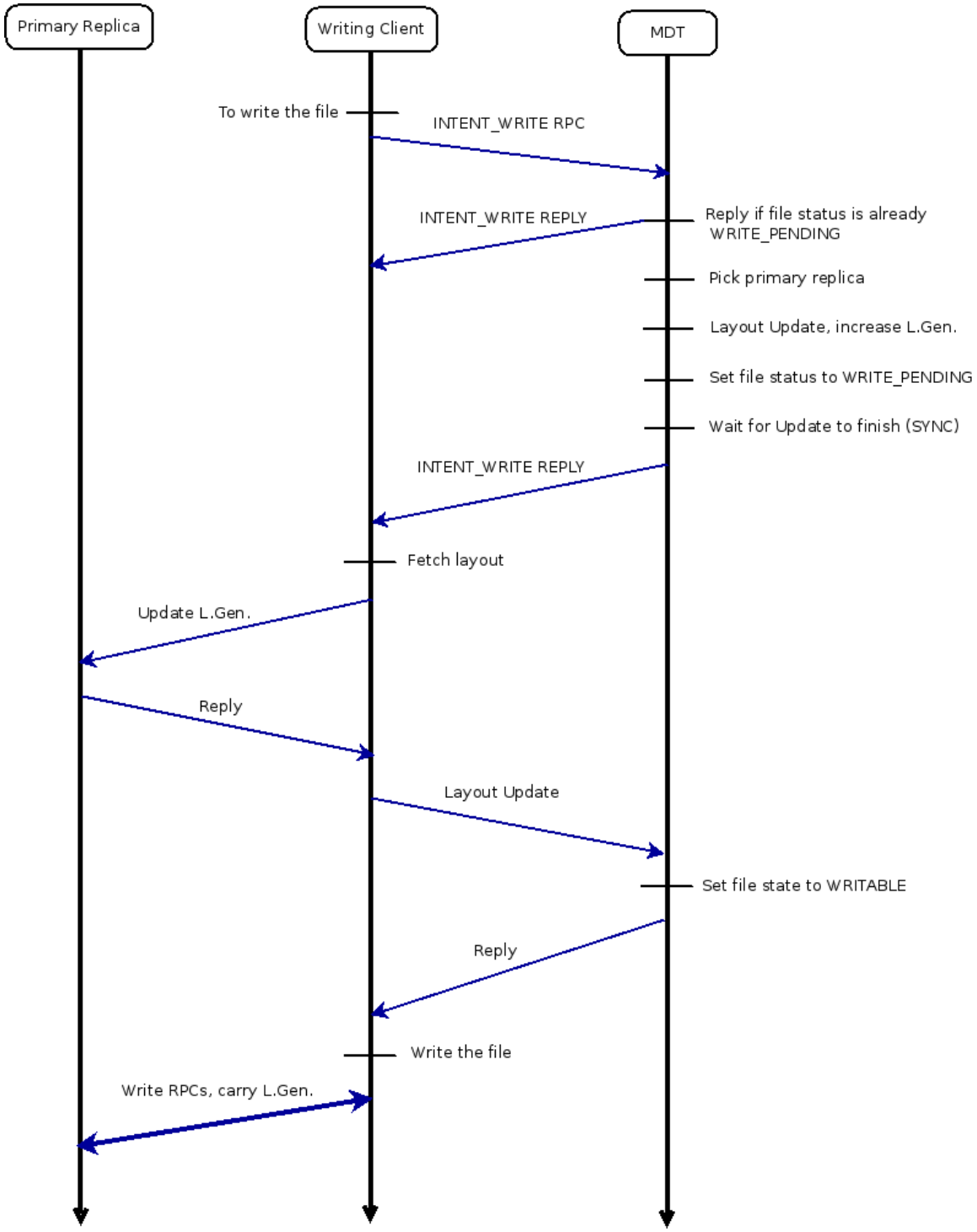
Besides, each replica can be one of the following states:

- `SYNC` – the replica includes the uptodate data;
- `STALE` – the replica may contain out-of-date data. The replica can be written (in Phase 2) but it cannot be used to serve read if it's in this state.
- `OFFLINE` - The replica failed to be resynchronized and is not being updated. A full resynchronization is required to bring the file back to sync.

When the replicated file is in READ_ONLY state, all but one of the replicas must be in SYNC state. If it is in WRITE_PENDING or WRITABLE state, only primary replica is in SYNC state, other replicas must be in STALE or OFFLINE state.

6.1.4 Wrap it up

The diagram below shows the process of how a writing clients interacts the MDT before writing a replicated file.



Write a replicated file

The client sends an RPC called MDT_INTENT_WRITE to the MDT before it writes replicated files. When the MDT receives the MDT_INTENT_WRITE RPC request, and if it turns out that layout has to be changed, it will update the layout synchronously. The reason to update the layout

synchronously is that in case the MDT crashes before the update is committed, and if the writing client failed to replay the RPC, it may render corruption to the replicated files.

The format of `MDT_INTENT_WRITE` RPC is as follows:

```
struct req_msg_field *mds_intent_write_server = {
     &RMF_PTLRPC_BODY,
     &RMF_MDT_EPOCH
};

struct mdt_write_intent {
     struct lustre_handle mwi_handle; /* open handle */
     struct lu_fid        mwi_fid;
     __u64                mwi_start;  /* write start bytes */
     __u64                mwi_end;
};

struct req_msg_field RMF_WRITE_INTENT =
DEFINE_MSGF("write_intent", 0,
   sizeof(struct mdt_write_intent),
   lustre_swab_mdt_write_intent,
   NULL);

struct req_msg_field *mds_intent_write_client = {
     &RMF_PTLRPC_BODY,
     &RMF_WRITE_INTENT
};

struct req_format RQF_LDLM_INTENT_WRITE =
        DEFINE_REQ_FMT0("LDLM_INTENT_WRITE",
                        mdt_intent_write_client,
                        mdt_intent_write_server);
```

`MDT_WRITE_INTENT` RPC will be sent in `ll_file_write()` before writing any data to the file.

6.2 Prevent stale data from being read

To avoid cascading problems, when a layout lock is being canceled at client side, the corresponding file will be marked to have a `STALE` layout but caching pages are not destroyed. This results in a problem that the reading process may read stale data when the replicated file is being written.

Let's take an example:

A replicated file has two replicas: replica #0 and #1. Client A reads the file and caches some pages from replica #0. Client B starts to write the file and replica #1 is chosen so replica #0 is marked as stale. Client A can continue to read the cached pages that are already stale.

We're going to solve this problem in this section.

6.2.1 Versioned cl_page

A field will be added to `cl_page` to remember the layout generation at page creation time. Pages' layout generation number can be changed under some circumstances.

To avoid stale page being read, clients have to make sure that there are no pages with previous layout generation existing by traversing page cache before IO starts. Stale pages will either be destroyed, or version is upgraded to latest version if the corresponding replica is still valid.

In implementation, a field will be added into `ll_inode_info()` to remember what the lowest layout generation for any `cl_page()` in page cache. If the layout lock was lost due to false sharing, it won't do page cache cleanup at all.

6.2.2 Mapped page handling

There exists one case that processes can access pages without initiating an IO: mapped pages. If the pages are already mapped into processes' memory space, they can be read directly via `memcpy()`. Versioned `cl_page` can't help in this case. This poses the risk that stale pages can still be read.

One solution to solve the problem is to tear down ALL mapped pages when the layout lock is being revoked. This is expensive due to false sharing but mmaped processes have to pay a price themselves.

6.3 Replica Resynchronization

After a replicated file is written, it will have only one valid replica and others will be marked as `STALE`. We need a mechanism to synchronize replicas, i.e., copy data from `SYNC` replica to `STALE` replicas, and change the layout to mark all successfully copied replicas as `SYNC` again.

Obviously we don't want to synchronize the replicated files that are still being written, nor the files may be modified again in the near future. There will be a configurable time, and files that haven't been modified in that time can be picked for synchronization. This is called quiescent time.

6.3.1 Replica Read and Write

When reading from a replicated file, the data must be read from `SYNC` replicas. However, to synchronize replicas, we need to write the data to a specific one, this is called replica write.

Sometimes we want to periodically verify that the replicas are synchronized to avoid software BUGs or hardware malfunctions. In that case, replica read is really useful. We can read blocks from one replica and compare it to the others to make sure that each replica contains exactly the same data.

An `ioctl()` interface will be created to specify replica index for opened replicated files. Replica read and write can only apply to direct IO otherwise the client page cache may be tainted by stale pages. During implementation, independent open handlers will be provided for each replica, otherwise `ioctl()` has to be called to switch among replicas.

6.3.2 File lease

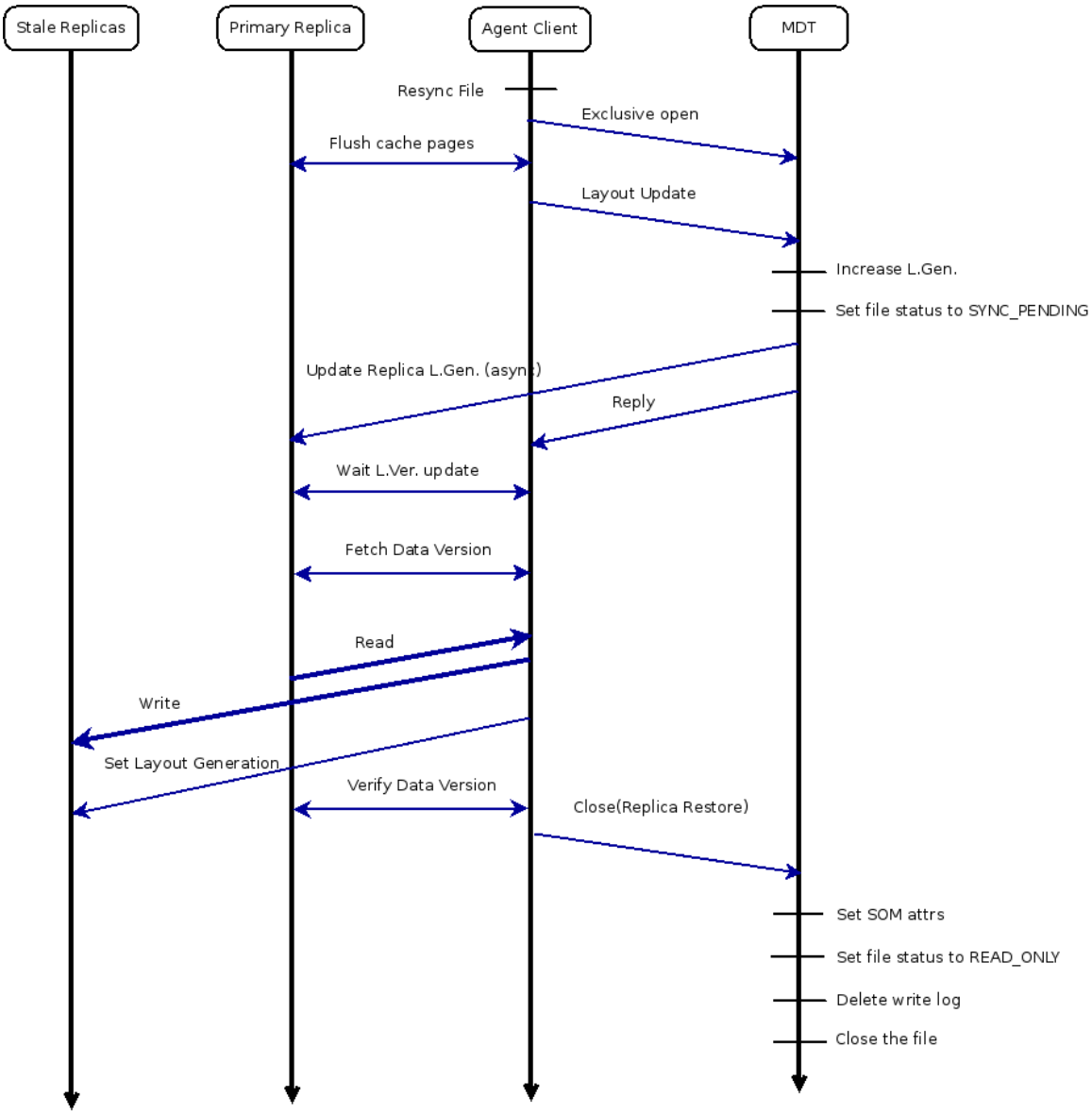
When the replicated file is being resynchronized, definitely we want to be notified if the file is to be modified. Exclusive lease is initially going to be held during the time when the file is being synchronized. Therefore, when the file is opened by other entities, the synchronizing process will stop and restart again after quiescent time elapses.

In the later phase of replication, we're going to hold read lease to resynchronize files. In this way, read and resynchronization can coexist, but when the file is being opened for write, resynchronization process will stop.

6.3.3 Resynchronization

The client that is resynchronizing the replicated file is not required to be the same client that wrote that file. Actually, some dedicated clients, named agent clients, could be used to pick files that have elapsed at least quiescent time since the last write. A simple ChangeLog monitor will be provided a list of files (FIDs) that have been marked stale, and can monitor the open state on the MDT. It would also be possible to integrate this resync operation with a policy engine like RobinHood to prioritize if and when to resynchronize the file, or if the replica(s) should be moved other OSTs if the existing ones are unavailable for a long time. This potential integration with RobinHood is out of scope of Phase 1 design and implementation, but a simple ChangeLog monitoring tool will be provided.

Of course, the file can be resynchronized manually with command `lfs layout --resync`.



Replicas Resynchronization

The flow chart above depicts the process of resynchronization.

`CLOSE` RPC will be extended to update the layout and do exclusive close atomically. We should pack the list of replica IDs that have been resynchronized successfully, just in case some replicas were failed to write.

The format of `MDS_CLOSE` will be revised as follows:

```
struct close_data {
        struct lustre_handle    cd_handle;
        struct lu_fid           cd_fid;
        __u64                   cd_data_version;
        __u64                   cd_flags;
  __u64     cd_reserved[7];
        __u32                   cd_nr_replicas;
        __u32                   cd_replica_ids[];
};
struct req_msg_field *mdt_release_close_client[] = {
        &RMF_PTLRPC_BODY,
        &RMF_MDT_EPOCH,
        &RMF_REC_REINT,
        &RMF_CAPA1,
        &RMF_CLOSE_DATA
};
```

The replicas that have been successfully synchronized are recorded in `cd_nr_replicas and cd_replica_ids` field. MDT will clear STALE bit on those synchronized replicas.

6.4 Client Eviction

Versioned write request and versioned replicas are the key point to solve the eviction problem. Before resynchronization starts, the agent client will flush cached pages and increase layout generation number. If a client with writing open handler is evicted, the request's version won't be updated so that any write requests from this client will be denied by OSTs.

Writing handlers from evicted clients can be recovered under some circumstances. If the file is reopened for writing on the same client, then the writing from the original file descriptor becomes valid as well. However, if the file state has been changed, then it is impossible to recover the open handler, for example, if the file was opened for execute or it has been deleted during the eviction.

7. Other operations

7.1 File Attributes

As a by-product of the replication mechanism, Size-On-MDT (SOM) will be fully functionally for files with up-to-date replicas. If the replicated file is in `READ_ONLY` state, the file's atime, mtime and size will be static and can be safely stored on the MDT, and no OST RPCs are necessary for fetching the file's attributes. It is desirable that the blocks count reported for a file include all of the blocks from all of the replicas, so that users and administrators can use common tools like "`du`", "`ls -s`", or "`find`" to accurately identify the space and quota used by each file.

If the file is not in the `READ_ONLY` state, it's operated as normal file so no attributes will be cached on the MDT, and glimpse RPCs will be sent to the OSTs of the primary replica to fetch these attributes from its objects. However, the blocks count is the sum of every object the file is using, even those in stale replicas. This implies that `stat()` of the file needs to retrieve the attributes from all of the OST objects in all of the replicas in the file's layout, even for the stale replicas. However, the block count does not have to be accurate under certain conditions. We will glimpse the primary replica, and then multiply the returned block count by the replica count. This ensures the block count will be this number after resynchronization is complete. We call this eventual block count.

7.2 Truncate

Truncate will be handled by the same mechanisms as write – the primary replica will be truncated and other replicas marked `STALE`. The replicas can be returned to `READ_ONLY` state once they have also been truncated.

7.3 Mmap

For write map, it's handled the same as normal write, and the MDT will be notified in `ll_file_mmap()`.

7.4 Lockless IO

Lockless IO is handled in the same manner as regular writes. Since the secondary replicas are marked `STALE` at open time or before the first write, the presence or absence OST object locking is irrelevant.

7.5 Missing OSTs

We already have a tool ("`lfs find --ost {index}`") to scan which files are affected if an OST is removed permanently. The tool will be updated for replication to be able to report the index of replica on a failed OST so that it's easier to find and fix the affected replica.

7.5 Quota

Quota accounting for replicated files will account the full space usage of each replica against the file owner's quota limits. This is both the simplest mechanism to implement and understand, and is also a natural consequence of users being able to selectively specify the replication level of each file. In sites where quota limits are enforced for users, the users can make the decision which files need one or more replicas, and the additional space usage of the replicas will be counted for their UID accordingly. This also allows users to create multiple replicas when they are needed, but discourages the use of replicas when they are not necessarily required (e.g. for every short-lived application checkpoint or other temporary files). This accounting mechanism is also no different than the user creating two separate copies of a particular file in the filesystem namespace, which would also consume twice as much quota as having a single file.

7.6 Compatibility

Older clients accessing replicated files must support the layout lock that arrived in Lustre 2.4. If this requirement is not met the client will be returned with a LOV_BAD_MAGIC in the layout magic number, therefore the attempt to access layout will fail. Clients that support layout lock but do not understand the replication feature will cause the MDS to immediately select a primary replica and mark the other replicas as stale synchronously when they open the file `O_RDWR` or `O_WRONLY`.
At this point the old clients can modify the file and the replication tool will resynchronize it when it is run on the file.

8. Open Issues

We expect no performance impact on the initial file write when using delayed replication, since the write mechanism is largely unchanged from the current implementation. We also expect improved performance on concurrent reads of replicas from multiple clients since there will be multiple OSTs holding the file data. However, producing a performance a detailed model for this design is beyond the scope of this work since the overhead of creating and resyncing file replicas is entirely driven by user-space policy and environment-specific characteristics of how often existing files are modified and the overall duty cycle of the filesystem.

9. Implementation Efforts and Tasks Partition

9.1 Feature List

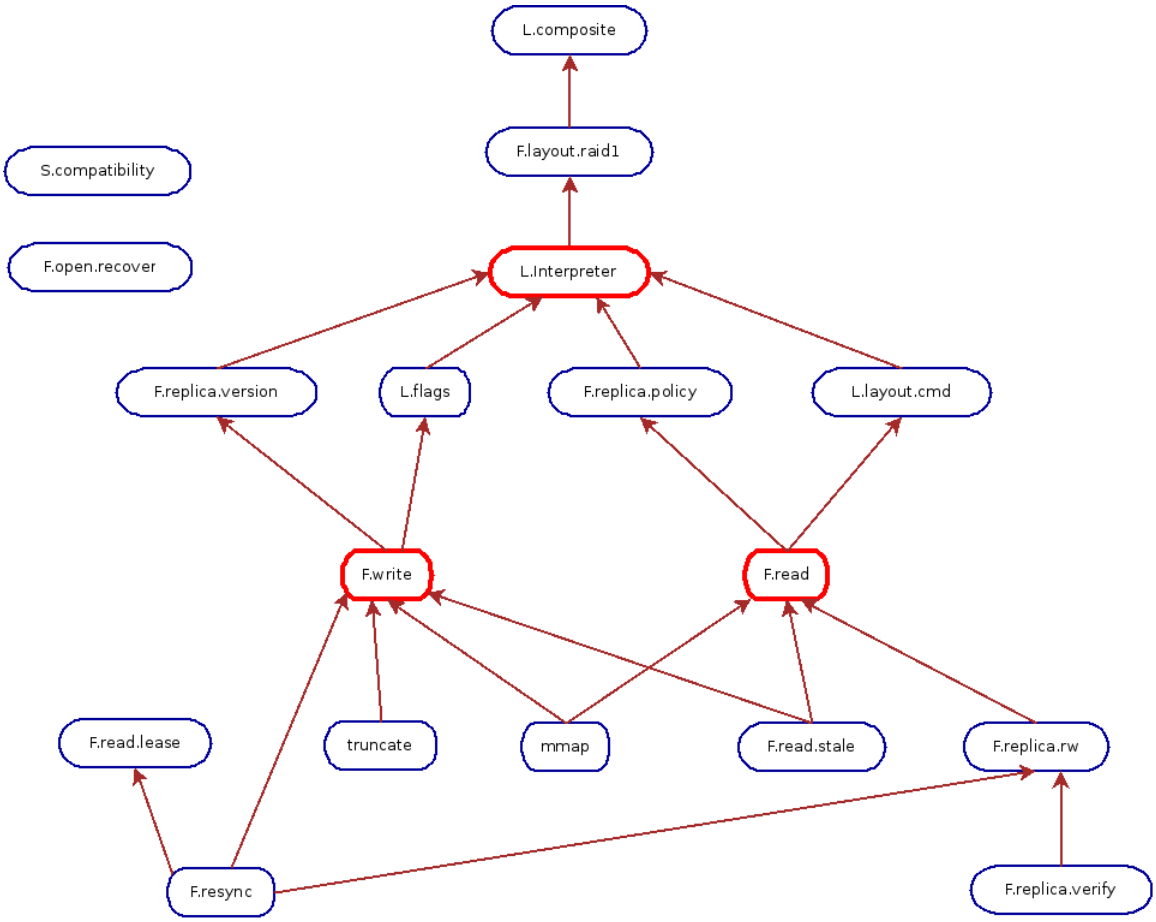
Each feature must be testable. Size is a value ranged from 1 to 10 to mark the relatively size of the feature.

No.	Name	Description	Index	Size
1	L.composite	Composite layout - by layout enhancement project	3.1	-

2	L.interpreter	Build up objects layout based on the composite layout at LOV and LOD layer; Show layout information by lfs getstripe; Any attempt to access replicated file will return -ENOTSUPP	3.1	7
3	L.flags	Define layout flags for replication use; Define replicated file state flags; Provide interfaces to change, set, and clear those flags	6.1.3	4
4	F.layout.raid1	Command line tool. Create an empty RAID-1 file	4.1	4
5	F.layout.cmd	Command line tools. Convert a normal file to composite file and add a replica into it; Detach a replica by index, the replica should be stored into a separate file; Delete a replica by index and destroy the replica	4.2 4.3	6
6	F.replica.policy	Define policy to choose replica; Glimpse should return correct file size	5.1	3
7	F.read	Read for replicated file; Return correct file content even if some replicas are inaccessible	5.2 5.3	5
8	F.replica.version	Versioned replica support. OST write requests carry layout generation; layout generation on OST objects; Set and get layout generation from OST objects	6.1.2	5

9	F.write	Write support for replicated file. Notify MDT to pick primary replica and marks other replicas stale; Writing client updates replica version; Write can succeed; Read returns correct data from primary replica;	6.1.6	9
10	F.read.stale	Make sure no stale data will be read after writes.	6.2	4
11	F.replica.rw	Replica read and write; Command line tool to resync	6.3.1 4.4	8
12	F.layout.verify	Command line tool to verify replicas	4.5	4
13	F.resync	Resynchronization	6.3.3	9
14	F.mmap	Mmap support	7.3	1
15	F.truncate	Truncate support	7.2	1
16	F.read.lease	Read lease support	6.3.2	3
17	F.open.recover	For evicted clients, sometimes the open handlers can be recovered	6.4	6
18	S.compatibility		7.5.1	-

9.2 Dependency Map



Dependency Map of Feature list for Replication