http://goo.gl/Mdcal

High level design of Lustre End-to-end data integrity

By Andrew Perepechko <andrew_perepechko@xyratex.com>

Date: 2012/04/01
Revision: 1.1

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document. The rest is a fictional design document, used as a running example.]

This document presents a high level design (HLD) of Lustre End-to-end data integrity (EEDI). The main purposes of this document are: (i) to be inspected by Lustre architects and peer designers to ascertain that high level design is aligned with Lustre architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of Lustre architects, designers and developers.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

File system reliability may suffer from random hardware failures. Data corruption sometimes happens when data are stored on disk, transferred over network or even kept in RAM.

Lustre file system has very limited capabilities of detecting and fixing data corruption. Network data transfers are protected with bulk checksums. Software RAID can be used to recover from reliable disk read errors.

This design document describes an approach how to implement end-to-end data integrity for Lustre, so that data is protected everywhere between read and write calls issued by applications (including RAM, network and persistent storage). T10 Data Integrity Field technology is used to provide data integrity between a host controller and a disk. T10 Data Integrity Extension technology is used to pass protection information from Lustre to a host controller (or vice versa).

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible.]

Lustre end-to-end data integrity (EEDI)	a software component that protects data transferred between applications and persistent storage
T10 Data Integrity Field (DIF)	a protocol that provides data integrity between an I/O Controller (IOC) and a disk Type 0 - no protection Type 1 - GRD and REF Type 1 tags are generated and verified Type 2 - GRD and REF Type 2 tags are generated and verified Type 3 - only GRD tag is generated and verified
T10 Data Integrity Extension (DIX)	a protocol of passing protection information from software layer to I/O Controller (HBA), which enables to provide data integrity between an application and the IOC
T10 protection information	a guard tag, a reference tag and an application tag for a sector
guard tag (GRD)	16-bit value, T10-DIF CRC (or, optionally, an IP checksum)
reference tag (REF)	Type 1: 32-bit value, lower 32 bits of the sector number (REF[LBA]=LBA&0xFFFF) Type 2: 32-bit value, lower 32 bits of the seed value in the SCSI command + offset from beginning of I/O (REF[FIRST+i]=SCB.INITIAL+i)
application tag (APP)	16-bit value, not defined by the corresponding T10 standard, can be stored/retrieved by software if ATO (App Tag Own) bit is set in Control Mode page, otherwise it is used by hardware. Defined by this document for the purpose of Lustre end-to-end data integrity.
lost write	a data corruption pattern in RAID systems, either a data block or a parity block (but not both) is not updated on write, stripe data becomes inconsistent
torn write	a data corruption pattern, not all data sectors comprising a user data page/block reach disk on write, the page is updated partially; only meaningful when the hardware sector size is less than 4096 bytes (which equals the effective Lustre page and block size)
major page fault (major fault)	a valid page fault caused by a program's memory access which requires I/O to complete because the requested data are not in RAM

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]

[R.T10.DETECT_DATA_CORRUPTION] detect corruption in each 512-byte data chunk between applications and disks as guaranteed by the T10-CRC algorithm

[R.T10.DETECT_MISDIRECTED_WRITES] detect that data from a sector was supposed to be written in another sector as guaranteed by T10 reference tags

[R.T10.LOST_WRITES] detect and, if possible, fix lost writes to a single block (data or parity, but not both)

[R.T10.TORN_WRITES] detect and, if possible, fix partial writes of a 4 kb file system block

[R.T10.PREVENT_PARITY_POLLUTION] prevent parity block corruption by controlling CRCs of sectors used in parity calculation

[R.T10.HARDWARE_CHECKSUMS] use hardware checksum calculation on OSS when possible

[R.T10.HBA_DIX] HBA drivers should be DIX-capable

[R.T10.BIGENDIAN_CRC] Lustre client should receive and send CRC data in the big-endian format

[R.T10.LARGE_SECTORS] Disks with 4096-byte sectors must be supported

[R.T10.PREFORMAT] Drives should be preformatted for T10

3. Design highlights

[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

The main idea is to integrate T10 DIF/DIX support to provide end-to-end integrity for Lustre.[1]

Protection information for a disk sector defined by T10 consists of a guard tag, reference tag and application tag. T10 defines how guard tags and reference tags are handled by HBAs and disk drives. Application tags can be used by software such as file systems, Lustre stores versions in application tags. The three tags are stored on disk so that 512-byte sectors can be viewed as 520-byte sectors with 8 bytes of metadata (or 4104-byte sectors with 8 bytes of metadata), although protection information need not be stored continuously with data.

Guard tags contain data T10-CRC[10], they can be attached to data and can be used for data verification anywhere. We shall attach guard tags to client cache pages, OST cache pages, BRW RPCs, BIOs. Finally, guard tags will be saved on persistent storage. A possible extension would be to provide applications with means of generation and verification of guard tags, including passing and receiving them to/from Lustre, this will be occasionally mentioned, but not designed by this document.[7]

Reference tags store sector numbers of the corresponding data sectors. We can use reference tags where data disk mappings are defined. This is limited to MDRAID-HBA driver-HBA-Disk drive, reference tags are not exposed to Lustre client. After reading a sector from disk, its reference tag is compared with the requested sector number. This helps catch misdirected writes.

Application tags store version information that helps detect and fix non-atomic writes leading to data corruption (lost writes, torn writes and some other corruption patterns). Versions are generated and updated for each data sector. MDRAID level 5 and level 6 maintain one or two parity sectors for data sectors with the same offset from the beginning of the chunk. Similarly, we store reduction function values for versions of these sectors in application tags of parity blocks (version vectors). If a data sector version does not match its reduced part in the version vector, then a write was lost. Depending on which version is older, we can decide if a write was lost to the data sector or the parity sector, then recover. Application tags are defined at MDRAID-HBA driver-HBA-Disk drive and are not exposed to Lustre client.

Clients and OSSes have file data caches. In order to implement protected cache access, guard tag vectors will be attached to pages. Before satisfying read requests from pages marked up-to-date, CRC are to be checked. If CRCs are wrong, corresponding pages are invalidated and the full I/O path is followed. Write mmap-ed client data pages cannot be properly controlled, checks will be disabled. Write requests will update cache pages' guard tag vectors.

Protection information for metadata will be generated by version mirroring code. If version mirroring is disabled, protection information for metadata will not be passed to and received from hardware, metadata will have only DIF protection (guard and reference tags will be calculated and checked by HBAs and disk drives).

4. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

4.1 Client-OST connection

A T10-capable client sends a CONNECT RPC with the T10 capability flag to an OSS/OST. If the OSS/OST supports T10, it sends the same flag and the hardware sector size in reply.

4.2 Read case

When an application calls *read()* for a file on a Lustre file system, VFS attempts to satisfy the request from cache (except when used with direct I/O). Cached data that passes integrity checks are returned to the application. Data missing from the cache (or failing the integrity checks) are requested from the OST (or the OSTs). A BRW RPC is sent to the OST. OST looks for the requested data in its cache and performs its integrity checks. Any data missing from disk or failing the integrity checks are requested from disk. Disk drive reads the data, performs the integrity checks and passes them to the HBA. HBA performs the integrity checks and passes them to MDRAID. MDRAID performs its integrity checks and attempts to recover data. After that, data is sent to the client. The client, finally, checks the data and returns them to the application. If the checks have failed, data are re-requested from the OST. Protection data could be returned to the application through a special API which is not part of this design.

4.3 Major fault case

A major fault caused by an application accessing a memory mapped file is a case very similar to the read case (4.1) with the obvious exception that the protection data cannot be returned to the application.

4.4 Write case

When an application calls *write()* for a file on a Lustre file system, VFS usually only changes data in the local cache (except when direct I/O is used). The real write may be called either immediately (when using O_SYNC or direct I/O) or delayed until requested by Lustre (lock cancellation, reaching limit of dirty pages, etc) or VFS (memory pressure, writeback deadline, etc). The application could also provide the protection information for the data, but there is no POSIX API for that, it can be developed separately. As soon as data reaches the kernel, its protection data are calculated. When the data are sent, their protection data are packed in the write request. The data are transferred to the OST which holds the corresponding part of the file. On the OST, the data and the protection information are passed to MDRAID. MDRAID updates the protection information with its specific metadata (APP, REF) and starts the disk I/O. HBA verifies data integrity wrt the protection information and passes it to disk. Disk drive verifies data integrity with respect to the protection information and records the data and the protection information. If either HBA or a disk drive finds data inconsistency, write is restarted either from Lustre obdfilter or the client.

4.5 Truncate case

When an application calls *truncate()*, the client saves the new file size, truncates all page cache pages past the end of file. If the last page is in cache, it is zeroed from the end of file till the end of the page, guard tags are recalculated. A setattr RPC is sent to an OST. OST looks for the last page in its cache or reads it from disk, zeroes it from the new end of file to the end of page and is written to disk. If the OST page is outdated, it will be overwritten by the client later anyway.

5. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]

Lustre end-to-end data integrity consists of:

- Lustre data integrity
- MDRAID integrity
- HBA driver support for DIX
- DIF-enabled hard drives

These three components interact with each other by passing T10 protection information associated with data from the client (ideally the application level via DIX) to the drive hardware. The components generate protection information and verify data validity (according to the protection information) as defined by this document.

5.1. Lustre data integrity

Lustre data integrity is an extension to Lustre client, Lustre OST and ldiskfs services which handles protection and verification of data based on guard tags (T10-CRC) by piping the protection information along with the data it protects.

Lustre data integrity *detects* data corruption, *resends* data, and *re-requests* data from MDRAID when data are transferred:

- from the application on the client node to the MDRAID driver on the OSS (write path)
- from the MDRAID driver on the OSS to the application on the client node (read path)

5.2. MDRAID integrity

MDRAID integrity guarantees that torn and lost writes are properly handled.

When data are transferred from Lustre to disk (write case), MDRAID:

- passes (and optionally verifies) the guard data through to the HBA driver
- creates new or updated reference tags
- creates version metadata and puts it in application tags.

When data are transferred from disk to Lustre (read case), MDRAID:

- passes (and optionally verifies) guard data through to Lustre
- uses version metadata from application tags to verify data validity
- possibly, recovers using parity blocks.

Since versioning is fully performed at the MDRAID layer, data and metadata can be safely mixed in the same stripe. MDRAID will produce protection information for metada, which is not provided from upper layers.

5.3. HBA driver support for DIX

HBA driver support for DIX will be designed in a separate document.

DIX allows passing protection information from software (e.g. MDRAID) to HBA, as well as receiving protection information from HBA by software. If DIX is not supported, but DIF is supported, protection

information (guard tags and reference tags) is generated by the HBA itself, only HBA-disk-HBA path is protected by DIF.

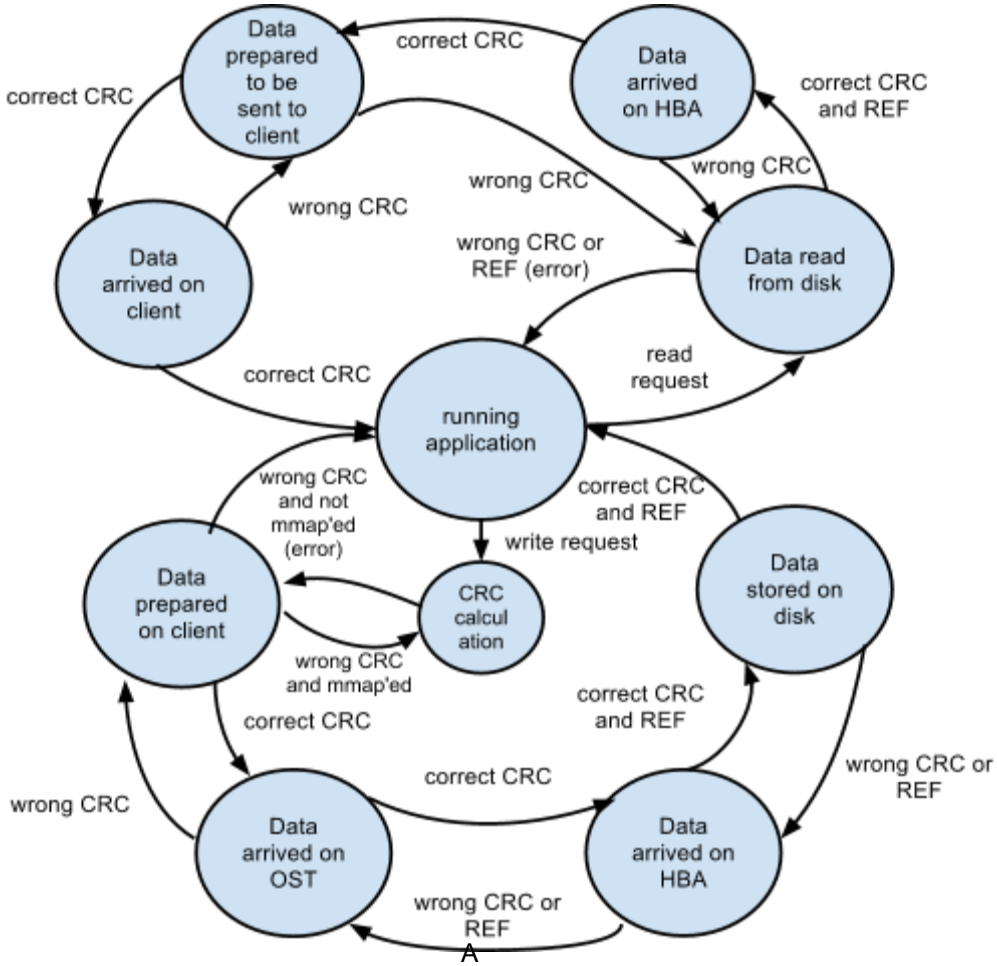
Lustre end-to-end data integrity cannot be effectively implemented on top of DIF alone.

6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

6.1. States, events, transitions

[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]



simplified verification-centristic state diagram of Lustre end-to-end data integrity

Notes to the diagram:

- APP tag and RAID handling are not shown
- writes are assumed to be synchronous
- some state transitions may not be implemented in hardware (read/write restart from HBA)
- some state transitions are optional (verification of arrived/prepared data)

6.2. State invariants

[This sub-section describes relations between parts of the state invariant through the state modifications.]

6.3. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, *etc*.) are used to maintain consistency.]

Protection information is updated synchronously with data. The only exception is with write-allowed memory mappings. Page cache pages from such memory mappings can be modified from userspace without Lustre control. When accessing data from client page cache, these pages will be considered valid even if guard tags do not match data. When an OSS finds that data do not match guard tags, the write request is re-sent from the client.

7. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

Connection phase

Lustre clients need to negotiate guard tag usage with OSTs. Client sets two special connection flags, which indicates support for the two following T10 PI network protocols (see section 8.5.1 for rationale):

- through a BRW RPC encapsulated T10-CRC array (protocol A), and
- through a sparse T10-CRC array in bulk tail (protocol B).

OST acknowledges by setting the same flags on its side and passing the hardware sector size. Either protocol can be disabled by administrator.

If the client (in software) has no guard tag support and the OST (in software or in hardware) has no T10 DIF/DIX Type 1 or 2 support, the tags are not calculated and not verified between this client and this OST. If OST uses an MDRAID device with different hardware sector sizes, then DIF/DIX-capability is not reported, since otherwise guard tag handling becomes too complicated.

If the client has no integrity support, but the OST has it, integrity data are generated on write and checked on read on the OSS, but not returned to the client.

If the client and the OST support Lustre data integrity, legacy bulk checksumming is not needed and is disabled between this client and the OST.

Also, at this point we can decide if we are going to do versioning. If we are using MDRAID and there are more than 8 chunks per stripe, then there is not enough protection storage for the version vector, and application tags will neither be generated nor verified. Versioning is also to be disabled if the ATO bit is unset and application tags are reserved for use by HBA (see [3] for more information on ATO).

Note on sector sizes (R.T10.LARGE_SECTORS):
The T10 standard is very unclear as to how T10 PI should be organized for large sectors. Some documents indicate 8 bytes of PI appended to 4096 byte sectors for 4104-byte sectors. Some indicate 64 bits of PI appended, for 4160-byte sectors. There's even a proposed Type 4 protection with 4096+16. (Apparently today, RHEL6 code assumes 4104: bio_integrity_hw_sectors() divides number of 512 bytes sectors by 8 if it finds that the underlying device has 4096-byte sectors, so that 8 512-byte linux sectors will be converted into 1 4096-byte sector and 1 integrity tuple.)
There also seems to be a proposed change to the T10 standard dubbed "Protection Information Interval", which is designed to interleave 8-byte PI at arbitrary intervals (e.g. 512 bytes) within a sector. This obviously would be the easiest scenario to adapt to, since there would remain a logical 8:512 ratio of PI to data that could be assumed everywhere. For any system other than this, it becomes very difficult / impossible to track the different sector size requirements all the way back to the client application. Unfortunately, none of the HW vendors seem to be supporting PII at the moment. Also, I don't think the Linux T10-DIX code can handle PII currently.

I think our plan at the moment (5/11/12) should be:
1. Do NOT support T10 at all in MDRAIDs with mixed-sized sectors / protection types. The MDRAID device will report back as not T10 capable.

2. DO report the PI schema (size, type) of each OST during the connection phase (and re-connection phase; may change after failover).
3. If and only if all of the OST's in the FS have the same PI schema, then T10-DIX to the application can be enabled. Otherwise, the kernel must compute the PI on a per-OSC basis.

Write path

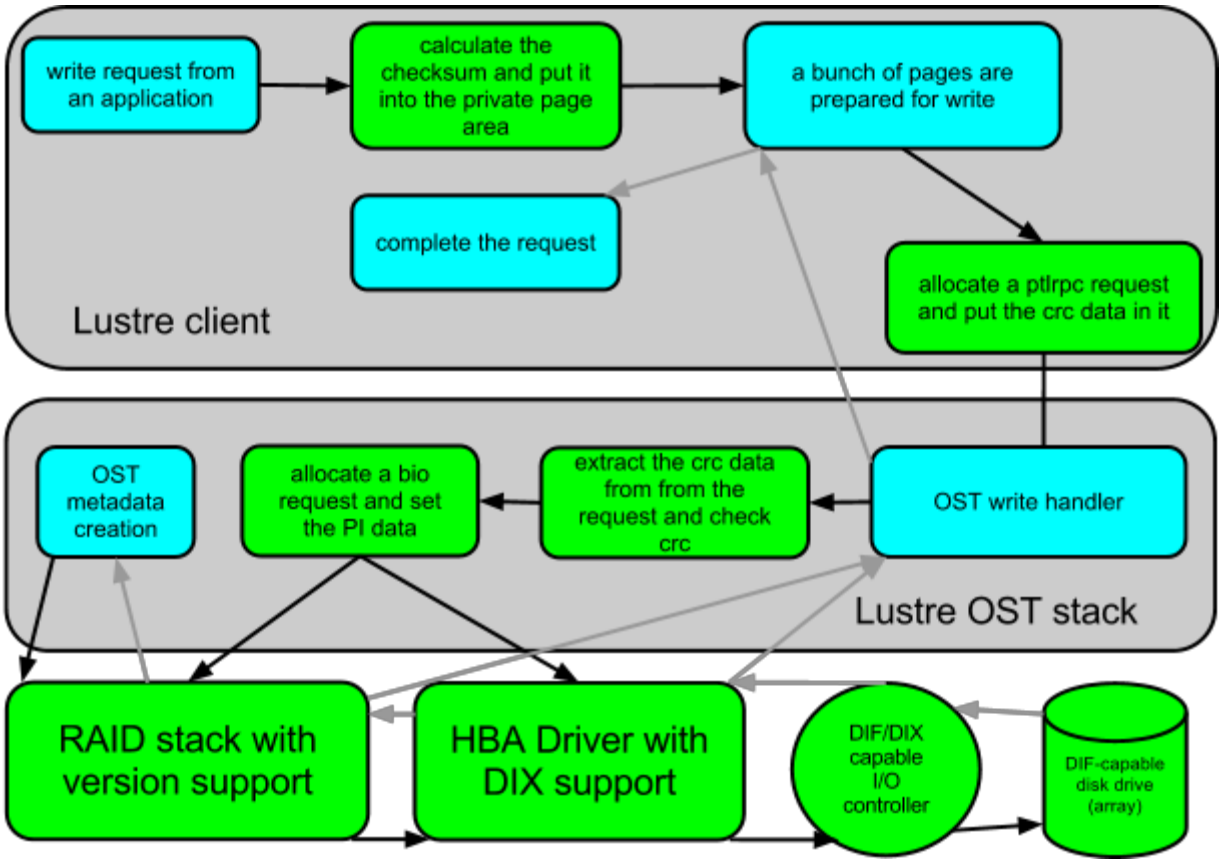


Figure 1. Flow of control on the write path.
New or significantly changed SW/HW components are marked with green.

1. [May be implemented in the future] Client application computes guard tags for each sector in userspace.[7] Applications will work with sector-size aligned data (otherwise guard tags cannot be calculated in userspace). Since hardware sector sizes can be different among OSTs, library API should also be striping-aware (and migrator-aware) and use striping versioning.
2. kernel write call handler (re)computes the guard tags for each sector after kernel copy and saves them in the private page area. This is done as soon as the data is accessed and prepared for write by Lustre to avoid later corruption on the client. On the error path, if the pages are write-mmap'ed, the guard tags should be updated. On the error path, if the pages are not write-mmap'ed and guard tags are different after recalculation, the data have been corrupted in an unrecoverable way, an error will be reported to the application if the write is synchronous.
3. OSC packs the guard tags array for the requested pages in the big-endian format in the brw rpc request (with protocol A) or allocates integrity pages and copies guard tags into them in a padded manner and adds these pages to the bulk (with protocol B) (see 8.5.1, 8.5.2 for rationale).
4. OSS pulls the BRW rpc request from the client over the network, picks the request from queue and performs the bulk transfer.
5. **[Optional]** OSS recomputes the guard tags for each sector, on error restart from step 2 (could be due to a mmap race).

6. Integrity-capable OSS generates guard tags itself if the client is not integrity-capable.
7. OST maps the pages to disk blocks and sets the guard tags and the reference tags for the pages (this includes the BIO and the private page area). With protocol A this includes allocating integrity pages and copying guard tags into them, while with protocol B the page from bulk is used.
8. **[MDRAID case]** set up versions and the version vector if versioning is enabled, use the passed GRD tags for data, generate GRD tags for the parity blocks, update REF tags after remap for both (we cannot let the block layer generate GRD/REF for the parity blocks because the BIO integrity API expects passing either all three tags or none and we need to pass APP), do the usual RAID stuff (see 8.5.3 for rationale). The APP tag value:
    - **[Full stripe rewrite]** Generate a random 14-bit number for each chunk rand(i), use rand(i)00 as the version for chunk i, set the APP tags to rand(i)00 for each chunk, set the APP tags for the version vector to (00)(00)(00)(00)(00)(00)(00) = 0.
    - **[Partial stripe rewrite]** Read the version vector ver, generate a random 14-bit number rand(i) for each chunk to be written, use rand(i)(ver(i)+1) as the version for each chunk, set the APP tags for each chunk accordingly, update the version vector to (ver(0)+1)(ver(1)+1)..., set the APP tags for the version vector. For any read requests, follow the strategy from step 13 of the read path.
9. **[Non-MDRAID case, may be implemented in the future]** Application tags can be used to store page version information so as to detect (but not fix) torn writes just like in the MDRAID case.
10. HBA verifies the guard tags. On a T10 error (Linux error code EILSEQ), step 5 (if implemented) or step 2 is restarted.
11. Disk verifies the guard and reference tags, possibly retrying from HBA, restart from step 10 (if supported by hardware) or step 5 (if implemented) or step 2 on error.

Read path

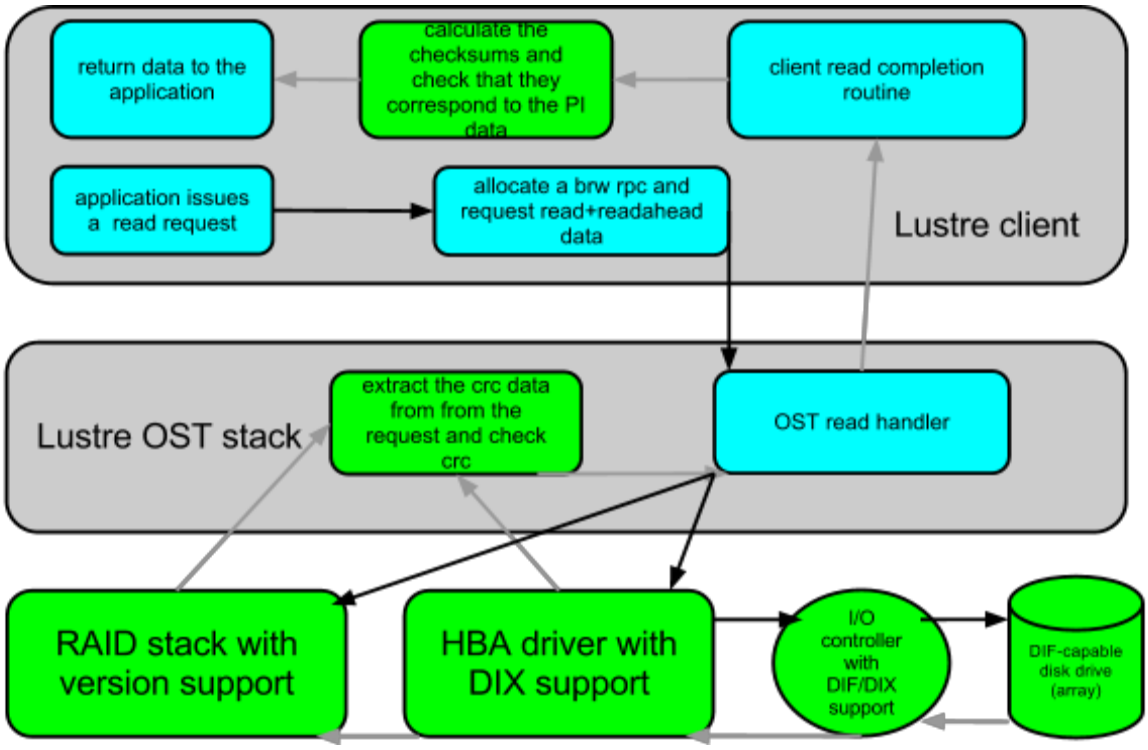


Figure 2. Flow of control on the read path.
New or significantly changed SW/HW components are marked with green.

1. Client tries to satisfy the request from its cache, non-write-mmap'ed pages should also pass guard tag CRC check, otherwise they are invalidated and the full I/O is performed.
2. Client sends a bulk rw request for the pages not present in its cache (or invalidated)
3. OSS receives the request
4. OSS tries to satisfy the request from its cache, pages should also pass guard tag CRC check (guard tag vector is saved in the page private area), otherwise they are invalidated
5. OSS maps the requested pages to blocks
6. OSS requests reading or writing the blocks from the linux block layer
7. Linux block layer routine allocates the integrity structures for the bio (for DIF-capable devices).
8. Linux, depending on the Lustre configuration, passes the request to MDRAID or the corresponding HBA driver.
9. **[MDRAID case]** MDRAID remaps the read request to the underlying devices.
10. The HBA driver programs the I/O controller to read the blocks
11. The disk drive finally performs reading of the data. Also, the application tag, the guard tag and the reference tag (for DIF-capable devices).
12. The disk drive checks that the guard tag corresponds to the read data, it also checks that the reference tag corresponds to the sector number (case of DIF-capable devices). The application tag is not processed at this stage.
13. **[MDRAID case]** check the version vector, fail inconsistent sectors, try to recover (see 8.5.3 for rationale):

- Read the requested chunks, check that the higher 14 bits of APP tags for each chunk are the same
- If not, find the highest version from the lower 2 bits from the APP tags of the failed chunk
- Try to recover from P or PQ blocks with the same 2-bit version
- Verify that APP tags for all blocks within a chunk match (to detect torn writes)

14. OSS packs the array of guard tags into the private page area. With protocol A OSS packs the guard data into the reply buffer, with protocol B OSS add the integrity page to bulk. (see 8.5.1, 8.5.2)
15. Client receives the reply and the bulk
16. Client copies the received guard tags in private page area of the read pages
17. Client checks that the data is consistent with their guard tags, re-request read (step 1) on error
18. Client returns the data to the application
19. **[May be implemented in the future]** Client returns the guard data to the application

Tag storage and page cache

Additional storage is required to attach guard tag arrays to page cache pages: 2 (for 4096-byte hardware sectors) to 16 (for 512-byte hardware sectors) bytes per page.

Lustre client implements private page storage, so the corresponding structures just need to be expanded accordingly.

Lustre OSS does not implement private page storage. It will be allocated when pages are prepared for BRW RPC, storage will be released through a special page release/invalidate callback at the ldiskfs layer.

7.1. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

- **R.T10.DETECT_DATA_CORRUPTION**

Data corruption is detected by calculating T10-CRC as early as possible, storing it on disk and verifying at critical points during reads and writes.

- **R.T10.DETECT_MISDIRECTED_WRITES**

Misdirected writes are detected by passing reference tags together with the data and verifying it after reads.

- **R.T10.LOST_WRITES**
- **R.T10.TORN_WRITES**

This is achieved by storing and comparing versions in application tags of data blocks and parity blocks.

- **R.T10.PREVENT_PARITY_POLLUTION**

Guard data detects read errors and prevents from being propagated into parity blocks.

- **R.T10.HARDWARE_CHECKSUMS**

Guard data is verified by the T10-DIF capable disk and I/O controller on reads.

- **R.T10.BIGENDIAN_CRC**

T-10 CRC is sent in the big-endian format from the client and received in the big-endian format by the client. BIO API receives and returns data in BE on the server side, no conversion is required on the OSS side.

7.2. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc*.) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

7.2.1. Hardware dependencies

Only a few DIF/DIX (or DIF-only)-capable I/O controllers are supported by RHEL6[5]:

- qla2xxx.ko (QLogic 25XX, 81XX: DIF Type 1, 2, 3 DIX Type 1, 2, 3, T10-CRC)
- mpt2sas.ko (LSI MPT Fusion: DIF Type 1, 2, 3, T10-CRC)
- lpfc.ko (Emulex Lightpulse FC: DIF Type 1, T10-CRC and IP)

Linux BIO integrity API mandates that protection information be passed in the big-endian format (R.T10.BIGENDIAN_CRC).

T10-capable disk drives must be T10 capable.

Disk drives must be preformatted to be used in end-to-end integrity capable configurations (R.T10.PREFORMAT). The IOC and the drives must be configurable to use T10-DIF only in this case, which may require IOC driver changes.

7.3. Security model

[The security model, if any, is described here.]

Lustre End-to-end data integrity is implemented as an extension to the usual BRW I/O. No security additions are implemented.

7.4. Refinement

[This sub-section enumerates design level requirements introduced by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

8. Analysis

8.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc*. Configuration and work-load parameters affecting component behavior must be specified here.]

A client will need to calculate the guard data for each written/read page. It will not take additional time on the full I/O path since the legacy checksumming is replaced, the amount of data for CRC calculation is the same and the CRC algorithm is almost the same. It will take additional CPU time if reading from its local cache just to verify data correctness. The client will also need some additional memory to keep the guard data for each page being read or written (2 bytes for each 512-byte sector, or 0.4% overhead; 2 bytes for each 4096-byte sector, or 0.05% overhead).

The same applies to an OSS, except it may choose not to recalculate guard data on software level.

On the MDRAID layer, versioning and reference control support will take a few additional CPU cycles. In many cases it will also have to perform an additional read of the version vector from disk. Since guard data is checked in hardware, some CPU resources on the OSS will be freed as compared to the legacy Lustre checksumming.

Protection data sent through network consist of guard tags, the additional overhead is nearly the same as the additional memory taken on the client and the OSS (0.4% for 512-byte sectors; 0.05% for 4096-byte sector) for method A. For method B we transfer 4 times as much as for method A.

8.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

Most failures the component deals with are data corruption failures. Lustre will re-send and/or re-read data when it sees data corruption. MDRAID capabilities are used to recover data from a version failure or a failed read. <network, disk>

In the cases where Lustre (on write) or MDRAID (on read) cannot recover the data, the failure should be reported to the client.

In the T10-DIF only (R.T10.PREFORMAT) case, driver changes may be required to prevent the drive from being remounted read-only on T10 errors. Failures should result in EIO returned to clients.

8.3. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]
[UML use case diagram can be used to describe a use case.]

Scenario	usecase.t10.connect
Relevant quality attributes	operability
Stimulus	connect operation with integrity flag
Stimulus source	Lustre client with integrity capabilities
Environment	OSS/OST with integrity capabilities
Artifact	integrity support
Response	OSS replies with integrity flag and sector size
Response measure	
Questions and issues	

Scenario	usecase.t10.read.client.network.crc.failure
Relevant quality attributes	fault tolerance
Stimulus	brw read request completion
Stimulus source	read system call or major fault
Environment	Lustre client
Artifact	corrupted object data
Response	brw read request restarted
Response measure	the number of requests must be limited
Questions and issues	how many retries? there's no obvious answer, we can use the logic for the existing checksums (sysctl-controlled)

Scenario	usecase.t10.write.client.network.crc.failure
Relevant quality attributes	fault tolerance
Stimulus	brw write request completion
Stimulus source	synchronous write system call or pdflush-originated write
Environment	Lustre client
Artifact	operation completion code signalling CRC error

Response	brw write request restarted
Response measure	the number of requests must be limited
Questions and issues	how many retries? there's no obvious answer, we can use the logic for the existing checksums (sysctl-controlled)

Scenario	usecase.t10.read.client.local.crc.failure
Relevant quality attributes	fault tolerance
Stimulus	an up-to-date page found in the client's cache
Stimulus source	read system call or major fault
Environment	Lustre client
Artifact	non-write-mmaped page data not matching PI
Response	the page is invalidated and a brw rpc is sent
Response measure	
Questions and issues	

Scenario	usecase.t10.read.ost.local.crc.failure
Relevant quality attributes	fault tolerance
Stimulus	an up-to-date page found in the cache
Stimulus source	brw rpc request
Environment	OSS
Artifact	page data not matching PI
Response	the page is invalidated and a direct I/O request is sent to the block layer
Response measure	
Questions and issues	

8.4. Recovery

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed

state consistency, concurrency control issues.]

The component uses the recovery framework of the components it is integrated into. Protection information is re-sent as part of BRW RPCs during replays.

8.5. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

8.5.1 Possible network protocols

8.5.1.1 Send CRC data array in BRW RPCs

Pros:

- easy to implement
- all 4 MB are available for data bulk
- no additional network load

Cons:

- occupies OSS's memory when rpcs are in queue (2 KB for a 4 MB bulk with 4 KB sectors, 16 KB for a 4 MB bulk with 512-byte sectors)
- need to copy into bio integrity pages (2-16 KB per 4 MB bulk)

8.5.1.2. Send CRC data in a sparse array (2 bytes of CRC, 6 bytes undefined) in BRW RPCs

Pros:

- easy to implement
- all 4 MB are available for data bulk
- no need to copy into bio integrity pages on OSS since layout is compatible with PI array

Cons:

- occupies OSS's memory when rpcs are in queue (8-64 KB per 4 MB RPC)
- occupies additional client's memory when rpcs are being processed (6-56 KB per 4 MB RPC)
- additional network load, including additional NIC-to-RAM DMA transfers (6-56 KB per 4 MB RPC)

8.5.1.3 Send CRC data array in bulks

Pros:

- does not occupy OSS's memory when rpcs are in queue
- no additional network load

Cons:

- 4 MB of data will not fit into the same bulk
- need to copy into bio integrity pages on the OSS (2-16 KB per 4 MB bulk)

8.5.1.4 Send CRC data in bulks in a sparse array (2 bytes of CRC, 6 bytes undefined)

Pros:

- no need to copy into bio integrity pages on OSS since layout is compatible with PI array

Cons:

- 4 MB data will not fit into the same bulk
- occupies additional memory on clients

- additional network load, including additional NIC-to-RAM DMA transfers (6-56 KB per 4 MB RPC)
- intermediate integrity pointers are needed for obdfilter code to handle partial reads (when some pages are in cache in up-to-date state and do not require reading)

Options 1 (protocol A) and 4 (protocol B) will be implemented. Based on testing with real hardware, the worse may be discontinued.

8.5.2 Big-endian T10-CRC swabbing

T10-CRC is calculated in the CPU format (likely, low-endian), but should be passed to Linux BIO integrity API in the big-endian format (see *struct sd_dif_tuple*, functions *sd_dif_type1_generate()* and *crc_t10dif()* for the reference). Conversion can be done:

- on the client
- on the server

It seems more efficient to perform conversion on the client, because one OSS serves many clients.

8.5.3 Version Numbering Methods

There are a few different choices for what constitutes a version number, all with tradeoffs.

- **Monotonically increasing version number per chunk**. Start from 1 and increase by 1 with every rewrite of this chunk. Version vector stored in the parity APP field must represent the versions of each chunk; for an 8-stripe RAID, that only gives 2 bits per version in the version vector. While not great, it seems likely sufficient to detect torn writes (would need to miss 4 torn writes in a row to overrun). Also, the version vector or the chunk version must be read at each full or partial stripe write in order to know the value to increment. (Note the “reconstruct-write” approach used by RAID 6 requires re-reading all the chunks anyhow to recompute parity, so getting the old versions might be a freebie for the partial write case.)
- **Random version number per RAID stripe**. All blocks in a stripe have the same random number, including the parity blocks. If any block differs, it represents a torn write. For a full stripe write, no reading of existing version is required; a new random number is generated. For partial stripe write, however, all versions must be updated, turning all partial-stripe writes into full-stripe writes.
- **Per-chunk random version number**. Avoids the requirement to obtain the old version number from 1), but with only 2 bits per random number, there’s a 1 in 4 chance of missing a torn write. We could potentially use the APP field in both the P+Q blocks to get 4 bits per chunk, at the price of reading both parity blocks on every read.
- **Combined random + incremental**. For example, use 14 high bits for a per-stripe random number and 2 low bits for a per-chunk incremental. The 2 low bits are mirrored in the version vector. The version vector must still be read for partial-stripe writes (to learn the old version, free with RAID6 reconstruct-writes), but for a full stripe rewrite a new random number is chosen for the high bits, eliminating the need to read anything for a full-stripe write.

It seems fairly clear that option 4 makes the most sense.

8.6. Other

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed state consistency, concurrency control issues.]

9. Deployment

9.1. Compatibility

[Backward and forward compatibility issues are discussed here. Changes in system invariants (event ordering, failure modes, *etc*.)]

9.1.1. Network

Network protocols are changed so as to allow transferring guard data through network. A client and an OSS have to negotiate protection data usage. If either does not support the protocol or if the OST is not using T10-capable hardware, Lustre end-to-end data integrity is disabled.

9.1.2. Persistent storage

Lustre end-to-end data integrity requires specific T10-DIF capable persistent storage. On-disk data layout as seen from the Linux block layer is not changed compared to earlier Lustre versions, additional PI storage is used.

9.1.3. Core

[Interface changes. Changes to shared in-core data structures.]

9.2. Installation

[How the component is delivered and installed.]

Lustre end-to-end data integrity is implemented and delivered as part of Lustre kernel modules.

10. References

[References to all external documents (specifications, architecture and requirements documents, *etc*.) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]

1. "Lustre End-to-End Data Integrity with T10 and Version Mirroring" by Nathan Rutman
2. "T10 Data Integrity Feature (Logical Block Guarding)" by Martin K. Petersen
3. "DIF/DIX Aware Linux SCSI HBA Interface" by Martin K. Petersen
4. I/O Controller Requirements for Data Integrity Aware Operating Systems by Martin K. Petersen
5. RHEL 6.0 DIF/DIX support
6. Parity Lost by Andrew Krioukov
7. Extending Data Integrity Support in Linux by Derrick Wong
8. LSI SAS2308 HBA spec
9. Hitachi 7k3000 SAS drive spec
10. MRP-511
11. Kernel documentation
12. Prototype code development for Lustre data integrity

11. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]

11.1. Logt

	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
Alexander Zarochentsev		HLDINSP					
Vitaly Fertman		HLDINSP					

11.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1					
2					
3					
4					
5					
6					
7					
8					
9					