# End to End Data Integrity Design

| Author | Date | Description of Document Change | Client Approval By | Client Approval Date |
|---|---|---|---|---|
| Rahul Deshmukh | 03/20/09 | Initial draft of design outline | | |
| Andreas Dilger | 06/06/09 | Updated content and diagrams | | |
| John Dawson | 06/13/09 | Formatting and editing | | |
| | | | | |

# 1 Introduction

## 1.1 Objective

In order to meet the HPCS program objectives Lustre needs to have end-to-end resiliency equivalent to T10 DIF or better.

To achieve this, Lustre will implement efficient end-to-end data integrity verification by combining the Lustre data checksumming with the ZFS on-disk data checksumming using a scalable hash method known as a hash tree or Merkle hash[1][2].

# 2 Requirements

In order to achieve this objective the following changes to Lustre are required:

1. Lustre will support ZFS as the backing filesystem.

2. ZFS, in particular the ZFS DMU, needs to provide an API for integration of Lustre network checksumming with ZFS disk checksumming. e.g. an interface for the caller to extract checksum value from DMU buffers etc.

# 3 Functional Specification

This section describes the necessary changes needed to support end-to-end checksumming.

## 3.1 Lustre:

Currently in Lustre, network data integrity is enabled by default on the Lustre client when the data is sent between the client and server, and optionally while the data is kept into the client memory cache. This allows Lustre to detect data corruption introduced by the network between the client node and the server, or in the server memory at write time.  However, the backing ext3 filesystem on the OSTs today does not persistently store any checksums for the file data.  In contrast, the Sun ZFS filesystem implements persistent on-disk checksums for both on-disk data and metadata.

[1] Hash Tree http://en.wikipedia.org/wiki/Hash_tree

[2] Tree Hash EXchange Format http://www.open-content.net/specs/draft-jchapweske-thex-02.html

## 3.2 ZFS Integrity

To avoid data corruption that might be caused by a variety of sources, ZFS checksums all data and metadata blocks on write and verifies the hash value on every read before using the data. ZFS stores the checksum of each block in its parent block pointer, not in the block itself, to form a self-validating Merkle tree.

This structure provides fault isolation between data and checksum, which helps in auto detection of corrupted, misplaced, or missing writes by the filesystem. Errors that ZFS will detect and correct include:

- **Phantom writes**, where the write is not actually persistent on disk. This would likely be undetectable by T10 Protection Information (PI, formerly called T10 DIF) because there is no version number in the block, and the checksum and data are stored adjacent to the block it protects, so it would also not be overwritten.

- **Misdirected reads or writes**, where the disk addresses the wrong block and reads the wrong data, or overwrites other data. This is a possible failure scenario for T10 PI if the sector address has the same low 32 bit value. Single bit errors in the high 32-bits of a sector address would cause undetectable errors with T10 PI. Certain classes of software and firmware defects may cause sectors to be offset by a power-of-two value.

- **DMA parity errors** between the storage array and server memory, or corruption by the driver

- **Driver errors**, where data winds up in the wrong buffer inside the kernel.

- **Accidental overwrites**, such as swapping to a live file system.

In addition to allowing ZFS to detect data corruption using the block hash, ZFS is normally integrated with the underlying RAID redundancy and can immediately fetch and validate a different copy of the block data in case of errors. For critical filesystem metadata ZFS keeps *ditto copies* of the metadata to ensure they can be recovered even in the face of serious failures in the underlying storage.

Integrating ZFS as Lustre's backing filesystem will also enable Lustre to take advantage of the following existing ZFS features:

- **Tunable checksum** algorithms that can be selected for either performance (Fletcher-4) or robustness (SHA-256), and can be updated in the future.

- **Integrated data checksums** and RAID mirroring/parity, so there is never a RAID parity update hole where a RAID stripe undergoes a read-modify-write operation that might fail in the middle or hurt performance. ZFS only ever does full RAID stripe writes.

So far we have seen that that Lustre has a network level checksum while ZFS has a separate block level checksum. These two could work independently to provide a reasonable level of data integrity, but there would be 100% CPU overhead from duplicate checksum calculations on the server to first validate the network data, and then compute the block checksum for long-term storage.

### 3.3 Merkle Hash Tree

The hash tree shown in Figure 1: Hash tree calculation with multiple block sizes will not only allow the data hash to be computed in parallel on the client, leveraging multiple CPU cores if they are otherwise idle, but the interior hash values can also be used on both the client and the server, independent of the page size on the client and server, and independent of the filesystem block size. This means that the hash tree needs to be computed only once on the server in order to validate both the incoming network data and also for storage on disk.

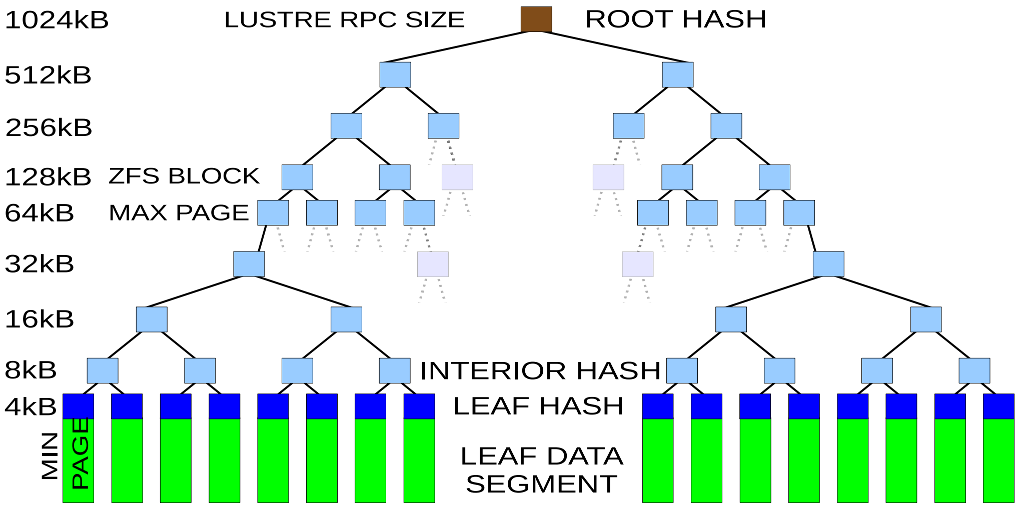
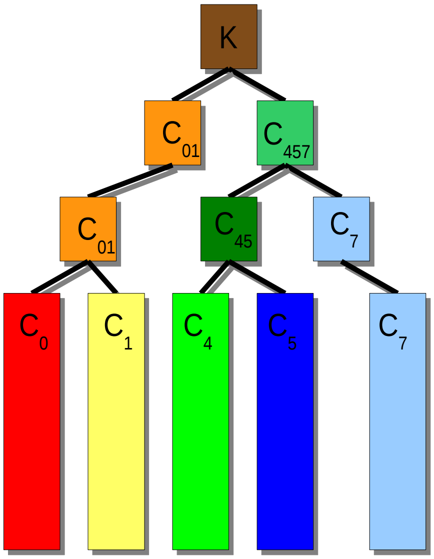


*Figure 1: Hash tree calculation with multiple block sizes*

The Merkle hash tree is nothing but a tree of hashes, where data segments form leaf nodes and each intermediate node is the hash of its children nodes' hash digests. To construct a Merkle hash tree for a data-set, first divide the given data-set into leaf data segments using a constant segment size. Hash each data segment separately, say *leaf-data-segment-0* and *leaf-data-segment-1*, to form hash digest values $C_0$ and $C_1$ respectively. Concatenate the adjacent hash digest values $C_0$ and $C_1$, and hash the resulting byte stream to form interior hash $C_{01}$. Continue this to form the resultant Merkle tree structure and to get root hash value $K$ as shown in Figure 2: Hash Tree for discontiguous data segments, below.



LH(x) = hash of data x (leaf)
IH(x) = hash of data x (interior)
x+y = concatenation x and y

$C_0$ = LH(leaf data segment 0)
$C_1$ = LH(leaf data segment 1)
$C_4$ = LH(leaf data segment 4)
$C_5$ = LH(leaf data segment 5)
$C_7$ = LH(leaf data segment 7)

$C_{01} = IH(C_0 + C_1)$
$C_{45} = IH(C_4 + C_5)$

$C_{457} = IH(C_{45} + C_7)$

$K = IH(C_{01} + C_{457})$ = ROOT hash

*Figure 2: Hash Tree for discontiguous data segments*

1. The size of the data segment is chosen to be some small common data size, such as 4kB. Since the cost of computing the individual leaf hash values is proportional to the total amount of data, this step does not add any overhead compared to a simple full-data hash function.

1. If the number of leaf segments is not a power-of-two value, and/or the leaf segments are not aligned on a power-of-two offset in the file, or are discontinuous, then the hash value of the unaligned segment is promoted unchanged up the hash tree until it matches a power-of-two alignment and is combined into the tree. For example, in Figure 2: Hash Tree for discontiguous data segments, hash $C_7$ does not have any immediate siblings with which to be combined to calculate the parent hash, so $C_7$ is promoted up the tree without being rehashed, until it can be paired with an aligned segment.

Properties 1 and 2 increase the chance that the interior nodes of the hash tree can be used unchanged on the peer, independent of the page size on the client and server, and also independent of the backing filesystem block size. Only a single hash value needs to be computed on the client and server in order to validate both in-coming network data and also for persistent storage on disk. Also this structure provides fault isolation between data and checksum by storing the checksum seprately from the data itself.

The hash tree begins with a standard hash algorithm such as Fletcher-4, SHA-256, or Tiger that is computed independently on the leaf data segments. For each level up the hash tree, an interior hash function, which is commonly the same as the leaf hash function with a different seed, is computed on the concatenated hash digest values of each pair of subsidiary nodes. There is no requirement, however, that the interior hash value be the same as the leaf hash, so long as both the client and server agree on the algorithm being used.

The overhead of computing the tree of interior hashes on the leaf hash digests is only a small fraction of computing the leaf hashes:

$$hash\,tree\,overhead = \frac{((2 \cdot hash\,digest\,length - 1) \cdot data\,segment\,length)}{(total\,data\,length)}$$

This is only about 1.6% overhead for a 256-bit (32-byte) Fletcher-4 digest for a 4096-byte data segment length, compared to computing the Fletcher-4 digest for the entire data length.

The hash tree allows a single hash value to be computed and stored for each page on the client, and a single hash value to be passed over the network. Being able to re-use the leaf hash values for both the network and the disk avoids a 100% overhead if the leaf hash needs to be recomputed (e.g. to translate between network and disk block sizes), or to store a digest value for each data segment.

## 3.4 End-to-end data integrity

To provide full end-to-end data integrity from client memory to server disk and back, integration between Lustre network checksum and ZFS block level hash is required. The proposed solution is to implement the identical hash tree algorithm for Lustre RPCs and for the ZFS block level hash.

With implementation of a Merkle tree, a client will compute a single per-RPC hash which will be sent to the server along with the write request. This single hash value from the client can then be verified on the server by regenerating the hash tree. The interior hash values corresponding to the block size for that object will be passed to ZFS.

Conversely, for a read request Lustre will use the on-disk block hash value computed by the ZFS DMU to compute the single RPC checksum and send it to the client. The client can then verify the hash on receive by constructing an equivalent Merkle tree and comparing the root hash. The client can then attach the interior hash values corresponding to its page size to each page in the read, and could optionally reverify the hash value after copying the data to userspace if it has been kept in the client cache for an extended period. By integrating the Lustre checksum with the ZFS checksum can detect and correct all of the errors described in Section 3.2 - ZFS Integrity.

It should be noted that ZFS could also be modified to use T10-DIF when interfacing with hardware that supports it, but it does not depend on this functionality in order to ensure data integrity. Also, the planned ZFS end-to end checksumming capability does not depend on any hardware support so if T10-DIF does not become available as soon as expected the Lustre checksumming will be unaffected. Table 1: T10-DIF vs. Hash Tree summarizes the characteristics of T10-DIF in comparison to the Hash Tree. If the SHA-256 hash function is used then virtually all data corruption can be detected, even if deliberately introduced.

| T10-DIF | Hash Tree |
|---|---|
| **CRC-16 Guard Word**<br>• All 1 bit errors<br>• All adjacent 2 bit errors<br>• Single 16-bit burst error<br>• $10^{-5}$ bit error rate | **Fletcher-4 Checksum**<br>• All 1- 2- 3- 4 bit errors<br>• All errors affecting 4 or fewer 32 bit words<br>• Single 128-bit burst error<br>• $10^{-13}$ bit error rate |
| **32-bit Reference Tag**<br>• Misplaced write != 2n TB<br>• Misplaced read != 2n TB | **Hash Tree**<br>• Misplaced read or misplaced write<br>• Phantom write<br>• Bad RAID reconstruction |

*Table 1: T10-DIF vs. Hash Tree*

# 4 Logic Specification

This section will discuss the end-to-end checksum solution in details using read and write options.

## 4.1 Client Write:

The Lustre hash tree implementation will use 4kB as data segment size. Figure 3: End-to-end checksum on Server Shows an example with a 64kB page size, so there are 16 4kB data segments per page. For each *data-segment-n*, a hash digest value $C_n$ will be calculated using the leaf hash when the data buffers are copied into the kernel pages to avoid the need to touch the data pages again when the RPC is being generated. The leaf hash values are concatenated and hashed using intermediate hash, until the root hash *K* is calculated.

After we compute the root hash value the client will send the data pages using bulk RDMA and the root hash value *K* using a write RPC to the server.

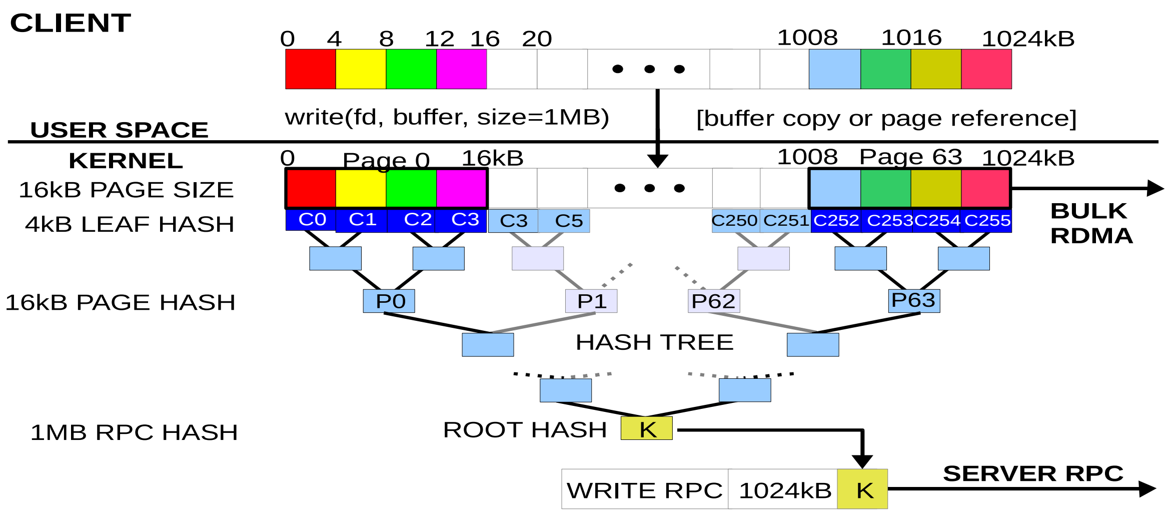


Figure 3: End-to-end integrity for client write

## 4.2 Server Write

As shown in Figure 4: End-to-end integrity for server write the server will receive the write RPC and bulk RDMA transfer of the pages from the client. On the server side the data in pages will be divided into the data segments of size 4kB as done on the client. In short as the total amount of data is not changed the number of data-segments will be the same regardless of the page size on the client and server.

The server will perform the same process as the client to generate the hash tree and root hash value K`. If K (i.e. root hash value received by RPC from the client) == K` (i.e. the root hash value calculated on the server) then we know that the set of checksum values of leaf data segments $\{C_{0-N}\} = \{S_{0-N}\}$ and and as a result also that $C_0 = S_0$, $C_1 = S_1$, ..., $C_N = S_N$. In turn, we know that each segment of data on the client is the same as the segment of data on the server.

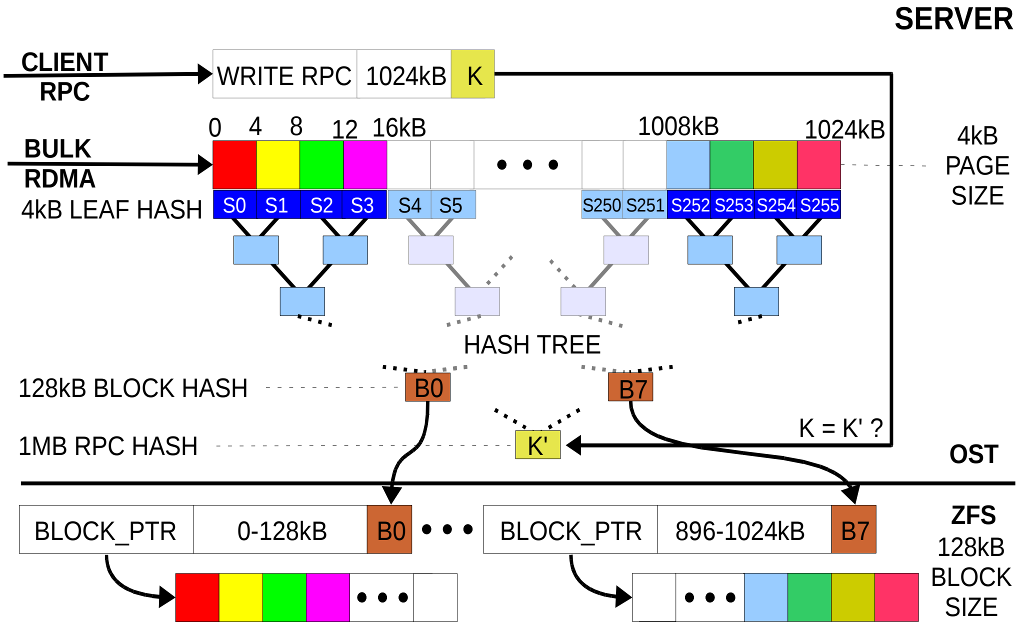


*Figure 4: End-to-end integrity for server write*

So far we have covered the network validation part of write operations on the server side. The reason for the extra complexity of the tree checksum algorithm becomes clear when we examine writing the  checksum to the backing storage. The block size of each ZFS file is variable (up to 128kB today, possibly larger in the future) and is not known to the client in advance. The ZFS block size will not necessarily align with the page size of either the client or server, and may increase as the file size grows.  By computing and tracking the checksums $S_{0..N}$ in 4kB segments on the server data buffers, we can compute a new block hash $B_n$ for each file block using only the data-segment checksums, without having to recompute the checksum on the entire block's data.

Additional benefits can be realized when the data is being cached in RAM on the client or server, and the per-segment hashes are kept with the data.  Each time the data is read by a client, the network RPC checksum only needs to recompute the hash tree for the segments being read instead of for the entire data, resulting in about a 98% CPU reduction for each cached read compared to recomputing the data checksum for each read. Also, keeping the checksums with the data in cache on the client and/or server will allow memory corruption to be detected long after the data was read from disk without any additional CPU overhead.

## 4.3 Read

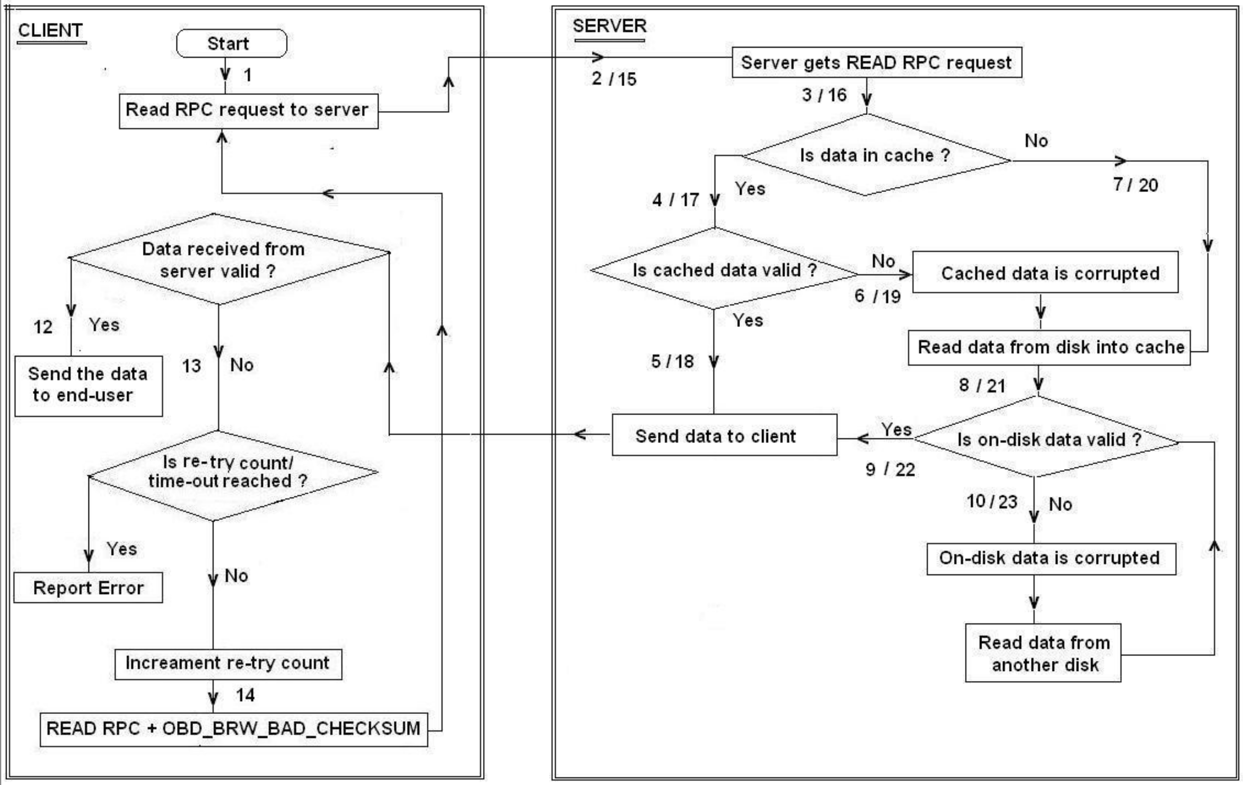

CLIENT
Start
1
Read RPC request to server
Data received from server valid ?
12
Yes
Send the data to end-user
13
No
Is re-try count/ time-out reached ?
Yes
Report Error
No
Increament re-try count
14
READ RPC + OBD_BRW_BAD_CHECKSUM
SERVER
2 / 15
Server gets READ RPC request
3 / 16
Is data in cache ?
No
7 / 20
4 / 17
Yes
Is cached data valid ?
No
6 / 19
Cached data is corrupted
Yes
5 / 18
Read data from disk into cache
8 / 21
Send data to client
Yes
9 / 22
Is on-disk data valid ?
10 / 23
No
On-disk data is corrupted
Read data from another disk

*Figure 5: End-to-end checksum Read decision tree*

The read operation will be described using the flow diagram as shown in Figure 5: End-to-end checksum Read decision tree followed by description at every stage.

1. Clients sends READ RPC request
2. Server receives the READ RPC request
3. Server checks if the required data is present in cache
4. Server will verify the reliability of the cached data by calculating checksum on cached data and comparing it with the checksum stored on arcbuf
5. If the checksum mention in step 4 matches then the server will conclude that the cached data is reliable and send it to the client.
6. If the checksum mentioned in step 4 does not match, then server will discard the the data present in arcbuf and read the data from disk into cache and will go on to step 8
7. Continuing from step 3, where we have determined that the data is not in cache, we will read it from disk into cache.
8. To check the validity of the on-disk data, the server will calculate the checksum of the on-disk data and compare it with the checksum value stored on disk.

9. If the checksum mentioned in step 8 matches then the on-disk data is valid and the server will send it to the client.
10. If the checksum mentioned in the step 8 does not match then then on-disk data has gotten corrupted and the DMU needs to fetch the another copy of the data (i.e. if another disk exists) and again check the validity of the data.
11. Client received the required data from the server. Now the client will check if the data it received from the server is valid by checking the calculated checksum value with the checksum value received from server.
12. If the checksum values mentioned in step 11 match the client knows that the data it got from the server is not corrupted and it will pass it to end-user.
13. If the checksum values mentioned in step 11 do not match then the client will determine if the re-count or time-out is reached, if yes then report error.
14. If not then increment the re-try count, set the OBD_BRW_BAD_CHECKSUM flag and send the READ RPC again.
15. Server again receives the READ RPC request.
16. Server checks if the data is in cache.
17. If the data is in cache then check the validity of the data by comparing the in cache checksum value with calculated checksum value.
18. If the checksum values match then it might be network corruption or client cache corruption. This is considering the fact that we got checksum value miss-match at client side, for data received from server. Since the checksum matches, send the data to the client.
19. If the checksum values mentioned in step 18 do not match, then cached data is corrupted and server will discard the the data present in arcbuf and read the data from disk into cache and go to step 21, it is the same as step 6.
20. Continuing from step 16, if the data is not in cache then get the data from the disk and check the validity of the data as mention in next step.
21. To check the validity of the on-disk data, server will calculate the checksum on the on-disk data and compare it with the checksum value stored on disk (This is same as step 8, only difference is that we have OBD_BRW_BAD_CHECKSUM flag set).
22. If the checksums mentioned in the step 21 match then the on-disk data is valid and the server will send it to the client.
23. If the checksums mentioned in the step 21 do not match then the on-disk data has become corrupted and the DMU needs to fetch the another copy of the data (i.e. if another disk exists) and again check the validity of the data.

# 5 Use Cases

The following section describes the partial block read and write.

## 5.1 Partial block write

In a partial block write, the client has sent the data to be updated or modified. The server will receive the data through a write RPC.

1. Server will read the whole block into memory as shown in Figure 6: Partial block write integrity verification.
2. Server then divides the data into data segments and calculate the leaf hash values $C_0$ to $C_7$ using the leaf hash function. Using the intermediate hash function, intermediate hash values will be calculated to form a tree structure and root hash $C_{0.7}$
3. The $C_{0.7}$ root hash value will be compared with the disk checksum value to verify the data.
4. Now, the whole block is copied to a new buffer in memory.
5. Place the updated data to into the new buffer.
6. Compute the new root hash value $BA_{0.7}$.
7. After that data verification for write will compare buffer *b* matches *B,* buffer *a* matches *A*, and hash $M_{2.5}$ matches the RPC root hash *K* sent by the client for that block. Note: If we want to be 100% strict we need to actually compute checksums for the copied data $B_0$ == $C_0$; $B_1$ == $C_1$; $A_6$ == $C_{6\backslash}$; $A_7$ == $C_7$ to verify that the coped data did not change from original.
8. The modified block will now be written to the disk along with the root hash value $BA_{0.7}$.

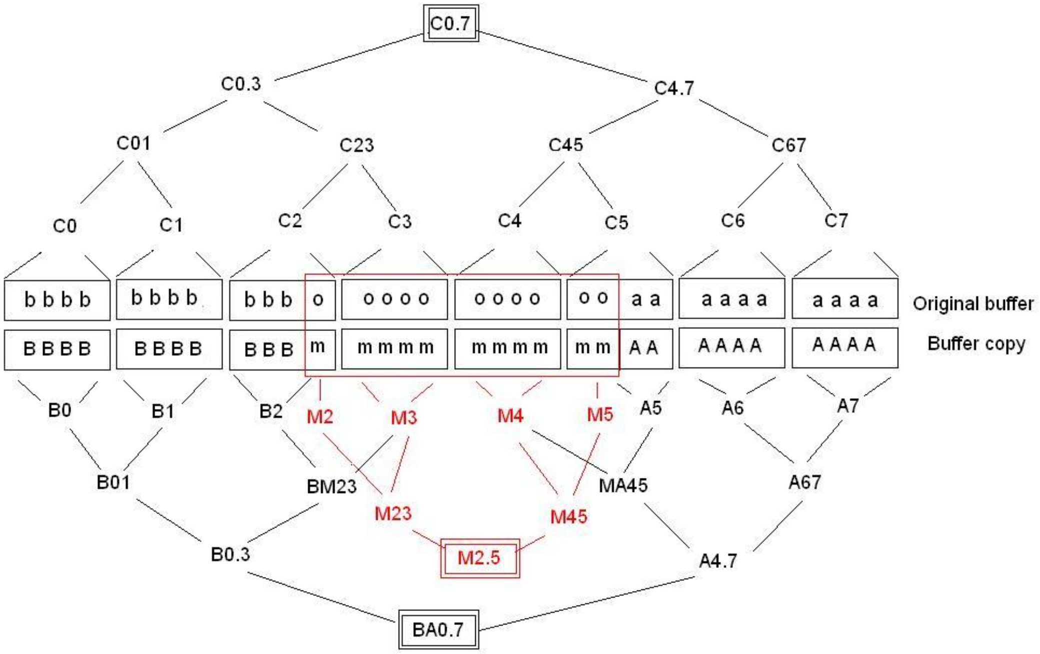


*Figure 6: Partial block write integrity verification*

Note here in step 6 we have computed the root hash value $BA_{0.7}$, the reason to compute the checksum on the written data AFTER the checksum of the modified buffer is to ensure that there is no window between the verification of the received data checksum and the computation of the new block checksum where the received data could be modified.

## 5.2 Partial block read:

In case of partial block read, server will compute the checksum of the partial-segment first, e.g. consider the data segment containing *m* is the partial segment as shown in Figure 6: Partial block write integrity verification. In that case checksum value will be calculated as $M_{2.5}$

Then checksum for whole block is computed and compared against the original checksum to ensure that the whole block is valid. This will also ensure that the checksum calculated on partial segment is also valid.

If every thing is fine then data segment along with the checksum value will be sent to the client.

# 6 State Management

## 6.1 Scalability & Performance

Performance of the proposed solution will be mainly dependent on selection of the checksum algorithm.

## 6.2 Recovery Changes

There will not be any recovery changes needed.

## 6.3 Disk Format Changes

There will be ZFS disk format changes to handle the new hash tree format.  ZFS already supports different checksums on a per-block basis, so adding a new checksum type even to an existing filesystem is possible.

## 6.4 Wire Format Changes

There will be Lustre wire format changes to hold a 32-byte checksum instead of the current 4-byte checksum.

## 6.5 Protocol Changes

There is no need for any protocol changes.

## 6.6 API Changes

New API will be added to ZFS's DMU for ZFS Lustre integration.

## 6.7 RPC Order Changes

RPC order will not change