Architectural Model for Data Integrity

Version 0.26 rev 0

"Publication of this Working Draft for review and comment has been approved by the Data Integrity TWG. This draft represents a "best effort" attempt by the Data Integrity TWG to reach preliminary consensus, and it may be updated, replaced, or made obsolete at any time. This document should not be used as reference material or cited as other than a "work in progress." Suggestion for revision should be directed to http://www.snia.org/feedback/"

Working Draft

September 15, 2011

Revision History

Revision	Date	Sections	Originator:	Comments
.1	2/12/2009	Layout	Jim Williams	
.2	2/23/2009	Definitions	Bill Martin	
.3	2/26/2009		Bill Martin	Modifications to definitions from conference call
.4	4/23/2009	Definitions	Bill Martin	Additions to match dirm-graph from Martin Petersen
.5	4/23/2009	Definitions	Bill Martin	Changes agreed to on conferece Call
.6	5/28/2009	Definitions	Bill Martin	Modified definitions to be consistent with Martin Petersen's graphs. Removed all bridge nodes and made other changes to be consistent with terminology.
0.7	06/11/09	Definitions	Martin K. Petersen	Fixed a few typos and redundant wordings.
0.8	06/25/09	Model	Martin K. Petersen	Adapted model description from last face 2 face to use new definitions.
0.9	08/13/09	Introduction, Definitions, Model	Bill Martin	Changes from face-to-face meeting incorporated
0.10	08/13/09	Introduction, Definitions, Model	Bill Martin	Minor modifications from conference call
0.11	8/25/09	Protection Envelopes	Jim Williams	Definitions of terms
0.12	8/27/09	General	Bill Martin	Cleaned up formatting of clauses lost in 0.11
0.13	8/28/09	Use Cases	Jim Williams	Added definitions
0.14	9/3/09	Use Cases	Bill Martin Martin Petersen	Moved information around in the Use Cases clause and added additional Use Case details
0.15	9/3/09	Entire Document	Bill Martin	Removed change bars for review.
0.16	9/9/09	Entire Document	Martin K. Petersen	Added information on T10 PIM and DIX.

				Updated sections on scatter-gather and buffer management.
0.17	9/10/09	Introduction and Best Practices	Bill Martin	Modifications from conference call
0.18	9/30/09	Protection Information section	Jim Williams	Added initial content
0.19	10/29/09		Bill Martin	Added diagrams of Protection information flow from Martin Petersen
0.20	11/13/09	Data Integrity Metadata	Martin K. Petersen	Initial content
0.22	01/27/10	Application API	Martin K. Petersen	Notes about application programming interfaces
0.24	12/16/10	Application tag	Martin K. Petersen	Application Tag access control
0.25	01/26/11	Application tag	Martin K. Petersen	Notes from meeting on January 6th 2011
0.26	01/31/11	All	Bill Martin	Changes from 1/27/2011 Face-to-Face meeting

Suggestion for changes or modifications to this document should be submitted at http://www.snia.org/feedback/.

CONTENTS

FIGURES

Data Integrity Reference Model

1 Introduction

1.1 Scope

This document describes a model for how data integrity can be protected between an Application client and a storage device. If all nodes in an implementation follow this model then there will be end-to-end data protection for the following class of data integrity errors:

a. silent data corruption:

    a. in flight; and

    b. at rest;

b. wrong address on device; and

c. buffer corruption.

While this model does address most of the common data integrity errors, in the current embodiment it does not address all possible errors. Some examples of errors that are not handled are: stale data caused by lost writes or fractured writes greater than a block, some cases of systematic miss-addressing between nodes. The model does not attempt to detect errors that are detected by the underlying storage protocol.

1.2 Definition of Terms

1.2.1 **Address Stamp:** the information that validates the locale of the data within a specific domain. As applied to the T10 Data Protection model this is the Logical Block Reference Tag.

1.2.2 **Application Stamp:** information that is assigned by the application to validate the originator of the data. As applied to the T10 Data Protection model this is the Logical Block Application Tag.

1.2.3 **Blocking:** a property of a node in which the node, by design, does not pass Data Integrity Meta Data.

1.2.4 **Combined protection envelope:** multiple contiguous protection envelopes that allow data and associated Data Integrity Metadata to flow through them while allowing modifications of the Data Integrity Metadata.

1.2.5 **Converting:** a property of a node in which the node modifies the Data Stamp received to a different Data Stamp that is transmitted.

1.2.6 **Data Channel:** The logical and physical components that Data are transferred through.

1.2.7 **Data Consumer:** a component that is the final destination for data (e.g., application or storage device).

1.2.8 **Data Integrity Metadata:** consists of Data Stamp, Application Stamp, and Address Stamp.

1.2.9 **Data Producer:** a component that is the initial source of data (e.g., application or storage device)

1.2.10 **Data Stamp:** the information (e.g., checksum, CRC, or hash code) that validates the data within the protection envelope. As applied to the T10 Data Protection model this is the Logical Block Guard.

1.2.11 **Error Handling Domain:** a portion of a Protection Envelope that includes an Error Handling node, a Verifying node, and a Generating node. The Error Handling Domain is instantiated by an Error Handling node when it originates a protected I/O request and is deinstantiated upon return of completion status to the Error Handling node. See 8.2.

1.2.12 **Error Handling:** a property of a node in which the node receives error notification from a Verifying node and processes data integrity errors.

1.2.13 **Generating:** a property of a node in which the node generates Data Integrity Metadata.

1.2.14 **Initiating:** a property of a node in which the node requests a protected transfer.

1.2.15 **node:** a component of a data integrity environment.

1.2.16 **Passing:** a property of a node in which the node passes data and Data Integrity Metadata un-modified.

1.2.17 **Protection Envelope:** all of the components that allow data and associated Data Integrity Metadata to flow through them without any modification of the Data Stamp & Application Stamp.

1.2.18 **Protection Information Channel:** The logical and physical components that Data Integrity Metadata are transferred through.

1.2.19 **Receiving:** a property of a node in which the node receives data and Data Integrity Metadata and is the termination of the Combined protection envelope.

1.2.20 **Translating:** a property of a node in which the node modifies the Address Stamp in the Data Integrity Metadata received to a different Stamp in the Data Integrity Metadata that is transmitted.

1.2.21 **Verifying:** a property of a node in which the node verifies the Data Integrity Metadata that is associated with the data and passes errors to the Error Handling node of the Error Handling Domain.

1.3 INCITS T10 Protection Information Model

The T10 Protection Information Model is an optional set of features in the SCSI Block Commands protocol (SBC) that provide 8 bytes of extra storage per protection information interval. The 8 bytes are arranged into a 16-bit CRC called the guard tag (Data Stamp), a 16-bit application tag (Application Stamp) and 32-bit reference tag (Address Stamp). The format of the guard tag and the reference tag are defined in SBC, allowing initiators and targets to verify that data integrity has been preserved.

1.4 Data Integrity Extensions (DIX)

The T10 Protection Information Model only concerns itself with communication between a SCSI initiator and a target. The Data Integrity Extensions define a mechanism that allows an operating system to exchange Data Integrity Metadata with a SCSI initiator, thus enabling end-to-end data integrity protection.

Note that DIX is a specific implementation and may not map into this model. See http://oss.oracle.com/projects/data-integrity/dist/documentation/dix.pdf for more information on DIX.

2 Model Description

2.1 Overview

The purpose of the data integrity model is to define a set of mechanisms which ensure preservation of the data integrity of an I/O request as it is transferred from the application to the physical storage and vice versa.

The I/O stack consists of an arbitrary number of connected nodes: User applications, system libraries, filesystems, logical volume managers, host adapters, storage network switches, storage arrays and disk drives are examples of nodes.

Data integrity is protected by attaching additional information to each I/O request. This extra information, known as Data Integrity Metadata, consists of a set of parameters that can be used to verify that the data buffer and the control path are in agreement and that the integrity of the data buffer has been preserved. There may be a handoff of the Data Integrity Metadata between protection envelopes, because it may not be feasible for the content of the Data Integrity Metadata to be identical between connected nodes.

A handoff is the process that occurs with the Data, Address, and Data Integrity Meta Data of an I/O transaction in a node that is both a Receiving node and a Generating node (i.e., Converting node or Translating node).

A protected handoff is a handoff where Data Integrity Meta Data that is to be transmitted is generated before the validation of the received Data Integrity Meta Data in such a way that any intervening data corruption is detected.

An unprotected handoff is a handoff where it is possible that neither of the protection envelopes detects an error introduced during the handoff (e.g., the data is outside the protection envelope during the handoff).

A combined protection envelope is two or more adjoining protection envelopes where a protected handoff is performed by the nodes in each pair of adjoining envelopes.

The full path of an I/O request between the application and the storage may be protected by 1 or more protection envelopes. These protection envelopes may provide 0 or more combined protection envelopes. If this path consists of exactly 1 protection envelope or exactly 1 combined protection envelope with all nodes in the I/O stack included, then data integrity is protected over the full path.

Each node in the I/O stack can have one or more data integrity properties. The properties are specific to a particular I/O request and a node can have a different set of properties for other I/O requests. The properties that a node may have are:

a) Blocking;

b) Converting;

c) Error Handling;

d) Generating;

e) Initiating;

f) Passing;

g) Receiving;

h) Translating; and

i) Verifying.

The following examples use Data Integrity Extensions and INCITS T10 Protection Information; however, other protection information models may be used for any of the protection envelopes.

Figure 1 shows the different nodes in an example system and the properties of those nodes when only the INCITS T10 Protection Information Model is implemented.

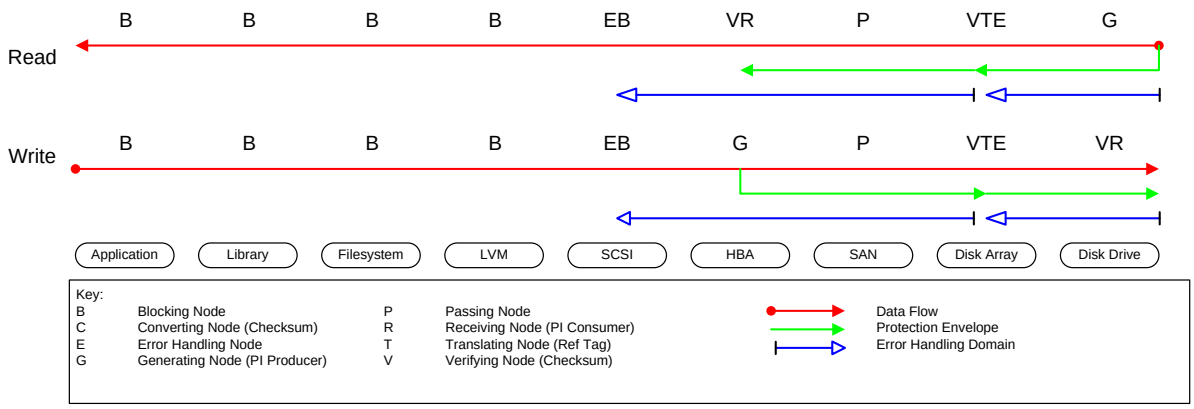

Figure 1 INCITS T10 Protection Information Model

Figure 2 shows the different nodes in an example system and the properties of those nodes when only the Data Integrity Extensions are implemented.

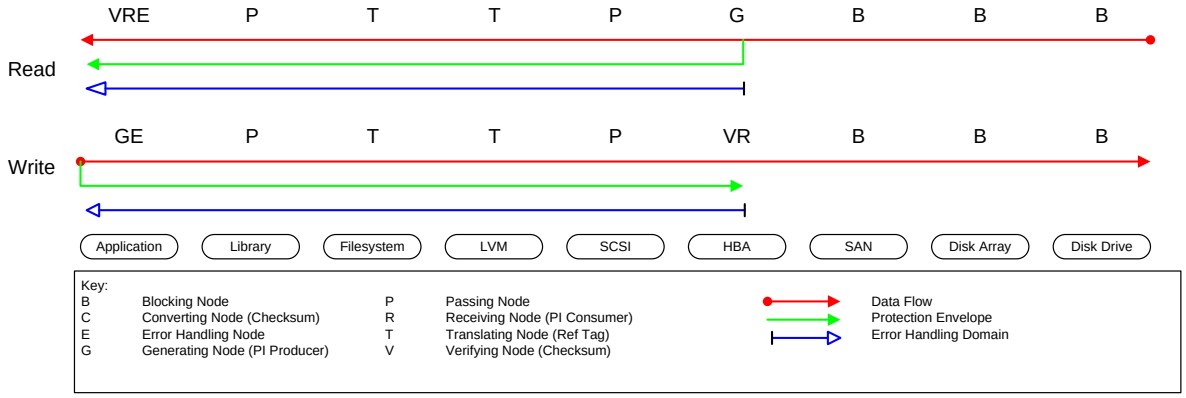

Figure 2 Data Integrity Extensions

Figure 3 shows the different nodes in an example system and the properties of those nodes when the Data Integrity Extensions are implemented in the server and the INCITS T10 Protection Information Model is implemented internally in the disk array.

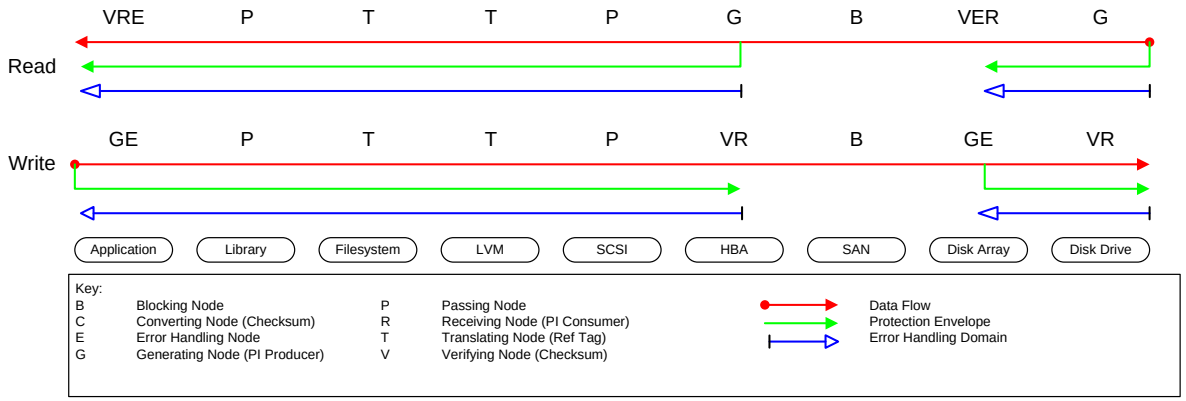

Figure 3 Data Integrity Extensions with T10 Protection Information inside array

Figure 4 shows the different nodes in an example system and the properties of those nodes when the Data Integrity Extensions are implemented in the server and the INCITS T10 Protection Information Model is implemented between the HBA and the disk array and the INCITS T10 Protection information is passed to the disk drive by the disk array controller. This provides protection from the application to the physical media where the data is stored.

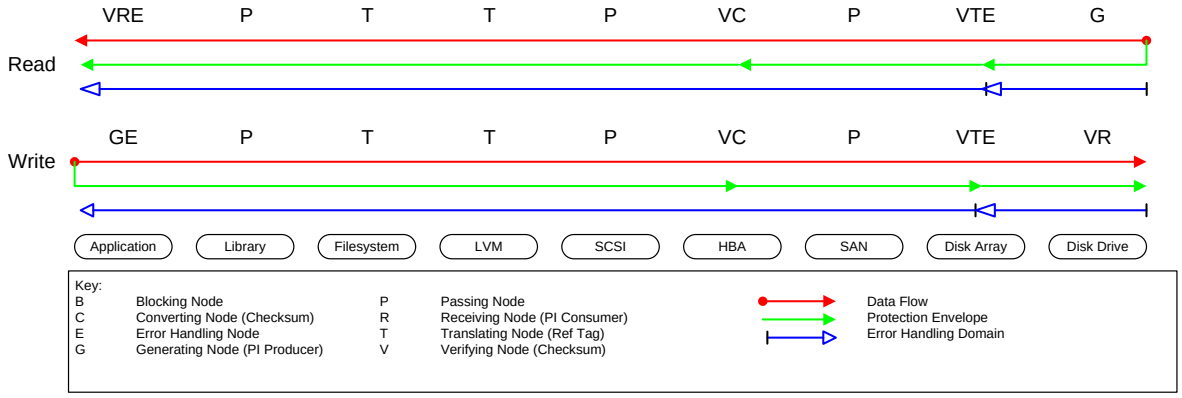

Figure 4 Full protection

3 Use Cases

3.1 Overview

Multi-core processors and virtualization are leading to many big benefits within the industry. This also is driving software to become more complex in dealing with multiple contexts, distributed work, and multi-tenant capabilities. With these new complexities comes a greater potential for silent data corruption (SDC) within these systems. This added flexibility makes it virtually impossible to verify the many configuration

permutations possible, leaving a potential for SDC. This model is being developed to provide mechanisms to eliminate sources of SDC.

3.2 Description of Data Integrity Errors

3.2.1 Overview

There are three attributes associated with Data Integrity errors. These are:

a) Time;

b) Type; and

c) Location.

3.2.2 When does data corruption occur

There are a number of times that data corruption can occur. Each of these times is associated with a reference (e.g., T_{write}) for the purpose of categorizing error scenarios. These times include:

a) Latent sector errors (i.e., Application is not able to read once valid data – I/O returns an error), which are not addressed by the SNIA Data Integrity Architectural Model;

b) Silent data corruption (i.e., Data read by application is not what was last written), which is addressed by the SNIA Data Integrity Architectural model include:

    a. Corruption that occurs when an application writes data to storage (T_{write});

    b. Corruption that occurs when an application reads data from storage (T_{read}); and

    c. Corruption that occurs while data is at rest (T_{rest}).

3.2.3 Types of data integrity errors

There are a number of types of data integrity errors. Each of these errors is associated with a reference (e.g., CE) for the purpose of categorizing error scenarios. The types of data integrity errors include:

a) Misplacement Error ($ME_{(a,d)}$) – data is stored or retrieved from the wrong location or device. There are two variations; wrong address and wrong device;

b) Content Error (CE) – data content is changed during a read or write;

c) Lost Operation (LO) – data write operation was acknowledged written but was not really written; and

d) Mis-ordered Operation (MO).

3.2.4 Location of Data Integrity Errors

There are a number of locations where data integrity errors can originate. Each of these locations is associated with a reference (e.g., L_{app}) for the purpose of categorizing error scenarios. The locations include:

a. Application layer (L_{app});

b. Operating System (L_{os});

c. Host Hardware (L_{host});

d. I/O Controller (or any storage interface) (L_{ioc});

e. Storage Fabric (L_{fabric});

f. Storage Array (L_{array}); and

g. Hard Disk Drive (L_{disk}).

3.3 Causes of Data Integrity errors addressed by this model (real world causes)

3.3.1 Operating System memory mapping failure leading to wrong data going to an LBA

$E\{ T_{write}, ME_a, L_{os}\}$

Modern operating systems depend on a memory management unit (MMU) in the host processor for segmenting system memory into user memory. The MMU translates virtual memory references made by user applications to system memory references. When an application performs a write I/O operation, it prepares a buffer in user space (virtual) that an I/O controller ultimately accesses for carrying out the write I/O operation. I/O controllers do not generally have access to the processor's MMU and they either implement their own MMU, or

depend on the operating system kernel for translating virtual (user) addressable I/O buffers to system addressable addresses. The code implementing these translations is a potential source of bugs and there are many documented cases of operating system bugs pertaining to translation errors leading to the wrong buffer being written to a device.

This type of error will be detected by a mis-match of the Data Stamp.

3.3.2 Operating System memory mapping failure leading to wrong data presented to an application while reading a LUN.

$E\{ T_{read}, ME_d, L_{os}\}$

An operating system memory mapping failure leading to a wrong block being presented to an application is similar to the failure described in 3.3.1 with the difference being that the memory mapping failure occurs during a read operation rather than a write operation. In the path between I/O controller and virtual (user) memory, a translation bug, as described in 3.3.1 can cause the wrong block of memory to be presented to an application during a read operation.

This error will be detected by a mis-match of the Data Stamp.

3.3.3 Misplacement of valid data

$E\{ T_*, ME_d, L_*\}$

Errors in hardware and logic can lead to I/O buffers being delivered to unintended locations. These errors can lead to good data being overwritten, or invalid data being presented as good data. Areas where this may happen include:

a. I/O controller memory;
b. Server Memory addressing;
c. Spindle misplacement; and
d. Storage controller firmware errors.

This error will be detected by a mis-match of the:

a) Data Stamp if this error is at a granularity below the block level; or
b) Address Stamp.

3.3.4 Host hardware failures

$E\{ T_*, CE, L_{host}\}$

Errors in hardware and logic can lead to incorrect data being delivered. These errors can lead to invalid data being presented as good data. Causes of this include:

a. I/O controller memory failures; and
b. Cache coherency failures.

This error will be detected by a mis-match of the Data Stamp.

3.4 Causes of Data Integrity errors not covered by this model

3.4.1 Lost write caused by storage array

E{ T_{write}, LO, L_{array}}

Storage arrays commonly use a cache memory for buffering I/O operations between internal storage devices and the array's I/O controllers, providing connections to external hosts. Array caches are also used for improving array performance by keeping frequently accessed data in memory. Data residing in an array's cache usually also resides on the array's storage devices. This means that data blocks, may exist in multiple places inside of an array. When there are multiple copies of a block in the array, the array firmware must keep track of which version of a data block is current. For example, just after receiving a block write operation from a host, the copy of the block in the cache may be current, while the version on the storage device(s) is out of date. One way in which a lost write is manifested is if a bug in the array firmware loses track of which version of a data block is current and inadvertently returns an older version of a block on a subsequent read operation. Another way in which a lost write is manifested is a failure of a volatile write cache.

3.4.2 Storage array mapping algorithm failure

E{ T_{*}, CE, L_{array}}

Storage arrays may provide a mapping mechanism between the LBA provided to a LBA on the physical medium. If this mapping mechanism fails, then data may be written to a physical device with the correct Address Stamp for that physical device; or data may be read from the wrong address on the physical device and the Address Stamp presented to the upper layer will reflect the address that was requested even though the wrong data is presented.

Storage arrays may provide a mapping mechanism between the Application Stamp provided and an Application Stamp on the physical medium, causing similar undetectable translation errors to occur.

3.4.3 Operating System bug over-writing wrong LUN on system crash

E{T_{write}, ME_{d}, L_{os}}

Operating systems often reserve a storage device, or a portion of a storage device for storing the memory contents of an operating system after a crash. A LUN reserved for an application can be corrupted, if through an incorrect configuration the reserved device was not large enough for the contents of the

operating system memory, and because of an operating system bug the memory dump operation can overwrite a normal application LUN.

3.4.4 Administrative Error

$E\{T_{write}, ME_d, L_{app}\}$

A frequent cause for the loss of data integrity is through an administrative error. Many people in an I/T organization have operational privileges enabling them to execute destructive actions that can lead to a loss of data. The problem is exacerbated by the fact that I/T configurations are usually extremely complex, and the tools used by administrative users often provide little or no safeguards against accidentally corrupting data. Three common examples of administrative errors that lead to data loss include:

1. Double allocating a storage device that was previously allocated to another application to a new application;

2. Initializing the wrong storage device causing all data to be lost; and

3. Disabling an I/O path between a server and its storage devices.

4 Best Practices

4.1 Increasing Soft Error Rate (SER) within Servers

ASIC's and memories are rapidly increasing in density and speed. With the current trend, process geometries are shrinking. With this geometry shrink comes increased soft error rates. Where in the past, systems only needed to deal with memory bit flips, which is where error correction circuits (ECC) technology has been very helpful. With the smaller geometries, flip flops and combinational logic are also now adversely affected, and providing similar error correcting circuits can be complex and expensive.

When a particle strikes a sensitive region of an SRAM cell, the charge that accumulates could exceed the minimum charge that is needed to flip the value stored in the cell, resulting in a soft error. The smallest charge that results in a soft error is called the *critical charge* (Q_{crit}) of the SRAM cell. Sources of these disruptions can come from cosmic rays or alpha particles.

Alpha particles have been problematic for memories for quite some time, and can come from internal process materials. Shielding, better materials and other techniques have helped to control this. Cosmic rays, on the other hand, cannot practically be controlled by the same techniques. Before the geometry shrinks, chips weren't as affected by cosmic rays, thus the techniques used to control alpha particles was sufficient. Now, for chips that are being developed in these geometries, an active awareness is needed and

extra measures taken to identify SER and control the potential for silent data corruption (SDC).

SER is calculated for each individual chip within a system, and is additive for all the chips within the system. The potential for SDC within a server then increases with each additional device. A target goal approaching zero SER for SDC in a single device is what should be sought, and this Data Integrity model will help to achieve this goal.

4.2 Scatter-Gather List Separation

Operating systems traditionally work on buffers that are multiples of 512 bytes. Some of this is a result of the processor page size (which is usually 4KB) and filesystem block size. As a result it is difficult to support a non-exponent of 2 sector size required by the T10 Protection Information Model.

To overcome this, a different approach that involves separating the data and protection information buffers at the operating system level can be taken. The OS data path is unchanged and the accompanying T10 protection information is pinned to the I/O requests as separate buffers which are subsequently described by extra scatter-gather lists passed to the HBA.

When writing, the HBA will interleave the data and protection information buffers resulting in non-exponent of 2 (e.g., 520-byte) sectors being sent to the T10 target that supports protection information. When reading, the HBA will split the sectors into a data portion and a protection information portion that are transferred to different areas in host memory.

Separation of the data and protection information buffers enables end-to-end data integrity support without requiring intrusive changes to the operating system memory management subsystem and I/O stack.

4.3 Buffer Management

On UNIX systems it is common to have a buffer or a page cache that sits in front of a storage device, enhancing both read and write performance. For writes the cache is used to buffer requests for a number of seconds before they are sent to the disk. This artificial delay allows the filesystem space allocators to pick an optimal location for the data, and it allows the I/O scheduler to coalesce adjacent requests, reducing head seeks and command processing overhead. However, areas on disk that contain filesystem metadata are often written with a frequency higher than the buffer cache flush delay. It is even possible for the buffer to be updated by the filesystem at the same time it is being transferred via DMA by the host adapter. This practice has traditionally not been problematic because the buffer in memory is marked dirty during the update, and as a result a new write is issued immediately, overwriting the portion on disk that was changed while in flight. Once data integrity is enabled, however, changing

a page in flight will cause the data stamp check to fail and the I/O to be aborted. Other operating systems may have similar issues. This is also a problem in existing deployments when the iSCSI digest checksum is enabled.

The solution is to ensure that the operating system never modifies a buffer that has been issued for I/O.

A similar problem exists for applications that use the direct I/O facility. Direct I/O allows the application to bypass the buffer cache and write straight to disk without intermediate buffering. In this case it is up to the application to ensure that the buffer is not modified until I/O completion is received.

4.4 Cache Management

For storage devices that contain volatile cache a cache failure may result in a lost write. To maximize the effectiveness of the Error Handling Domain it is recommended that volatile caches be disabled for protected write requests.

5 Protection Information Channel

As defined, a Protection Envelope contains data and Data Integrity Metadata. In the INCITS T10 Protection Information model, the Data Integrity Metadata is an eight byte field that is added to a protection information interval on a storage device. In the T10 model, storage device blocks increase in size with the addition of the Data Integrity Metadata. In the T10 Protection Information model both Data Integrity Metadata and actual data move through any communication path as a single unit.

One can consider that two logical paths exist, one for data and the other for Data Integrity Metadata. The data path is called the Data Channel and the Data Integrity Metadata path is called the Protection Information Channel.

5.1 Movement of Protection Information

An important purpose of this specification is to facilitate extending the Protection Envelope from the host-storage interface to the application endpoint. Without extending the protection envelope, the Protection Information Channel terminates at the host-storage interface and only the Data Channel continues to the application. Consequently, a key element of this architecture describes how the Protection Information Channel may be extended to the application endpoint.

Because of the nature of the SCSI specification, co-locating the Data Channel with the Protection Information Channel on the same physical and logical path is a reasonable approach. However, because of several factors associated with current practices and implementations, co-locating the Data Channel and Protection Information Channel may not always be practical (e.g., operating system layers, virtualization layers, and non-SCSI storage devices). These reasons are largely related with the fact that co-location of the two channels would lead to significant software structuring disruption and

operational inefficiencies. Consequently, it is considered a “best practice” that implementation of the Protection Information Channel be separated from the Data Channel inside the operating system and application.

Separation of the channels means that for data moving from a storage device towards an application, the host-storage interface will transfer Data Integrity Metadata into a separate buffer that moves in parallel with the data buffer towards the application.

For data moving from an application towards a storage device, at the point of origination for the Data Integrity Metadata, the Protection Information Channel is created. This means that the Data Integrity Metadata is created and stored in a buffer that moves in parallel towards the host-storage interface. From the host-storage interface to the storage device, the Data Channel and Protection Information Channel are co-located. A common point where separation and aggregation of the Data Channel and Protection Information Channel will occur is in the host-storage interface device driver inside the operating system. It is conceivable that the host-storage interface may have special hardware to make separation and aggregation efficient.

At the point of separation and aggregation of the Data Channel with the Protection Information Channel, there may be a translation of the Data Integrity Metadata. A common example is to use a checksum algorithm in the operating system and application layer that is more efficient for that environment. The Data Integrity Metadata is potentially translated at other places in the path (e.g., at the Logical volume manager, or the host-storage interface).

6 Data Integrity Metadata

6.1 Overview

The Data Integrity Metadata consists of one or more stamps (e.g., Data Stamp, and Address Stamp) that each protect one or more properties of a portion of the Data Buffer. In the T10 Protection Information Model there are three stamps associated with each Logical Block transferred to and from the storage device. However, the Data Integrity Model allows great flexibility in terms of the stamps that can be exchanged, as long as adjacent nodes in the I/O path know how to convert from one convention to another. The current version of this architectural model focuses on the stamps provided by the T10 Protection Information Model.

6.2 Components

6.2.1 Address Stamp

The Address Stamp (e.g., Logical Block Reference Tag in T10) contains a value that allows nodes in the I/O path to verify that information is sent/received to/from the right location (e.g., Logical Block Address or virtual address space).

The T10 Protection Information Model defines 3 types of Protection Information. With Type 1 the Reference Tag contains the lowest 32 bits of the target Logical Block Address. Type 2 Protection Information, however, uses the Reference Tag as an incrementing counter for the logical sequence with the initial value passed in the command (i.e., consequently, the locality property of the Address Stamp is not used to its full extent).

The meaning of the Address Stamp changes as a request traverses the I/O path. At the application or filesystem level it is likely to correspond to a linear sequential view of the file being read or written. As the request gets closer to the hardware level the tag may be translated to take on a value that has extra meaning in relation to the subsystems involved or the hardware. For instance a partitioned disk or a Logical Volume Manager may want to translate the Address Stamp to a number that corresponds to the target LBA on the underlying storage.

6.2.2 Data Stamp

The Data Stamp consists of a checksum, Cyclic Redundancy Check, or similar and is used to guarantee the integrity of a portion of the Data Buffer (e.g., the Logical Block Guard in the T10 Protection Information Model).

The Data Stamp can be converted from one format to another by a node. Such a conversion should be done as a protected handoff (See 2.1).

The Data Stamp may be changed from one format to another for a variety of reasons. Some choices of checksum may provide better protection against multi-bit error but may be prohibitively CPU-intensive to calculate. In such cases it may be advantageous to use a weaker checksum to protect part of the I/O path.

For instance the T10 Protection Information Model uses a 16-bit CRC that is expensive to calculate without dedicated hardware support. In that case two capable nodes can instead decide to use a lightweight checksum such as the one used in the Internet Protocol or a CRC that can be calculated in hardware.

6.2.3 Application Stamp

The purpose of the Application Stamp is to prevent overwriting blocks claimed by another application, filesystem, or volume manager. The Receiving node will reject any portions of a request that contain an incorrect Application Stamp. In the T10 Protection Information Model the Application Stamp corresponds to the Application Tag.

T10 SBC-3 provides an Application Tag Mode Page that defines expected Application Tags for ranges of logical blocks on the device. T10 SPC-4 provides an Application Tag Mode Page Enabled bit (ATMPE) to indicate that the Application Tag Mode Page is valid, and a Reject Write Without Protection bit (RWWP) to prevent writing data without protection information.

In order for a write request to succeed the Application Tag for each block in the received Data Integrity Metadata must match the predefined Application Tag for the target LBA. A write request that does not include the correct Application Tag will be rejected by the storage device. The LBA ranges and their matching Application Tags are set on behalf of the application or filesystem by the operating system via the Application Tag Mode Page. The ATMPE bit and the RWWP in the Control Mode Page shall both be set to one to indicate that the Application Tag shall be compared on all reads and writes

The Application Tag write prevention mechanism in T10 SBC-3 only protects write access to a device. It is possible for other initiators with access to the target device to clear the RWWP bit in the Control Mode Page, modify the Application Tag Mode Page, or overwrite user data. It is therefore imperative that access to the target device be controlled. It is recommended that storage administration interfaces default to disallow initiator access to newly created volumes when T10 Protection Information is enabled. Access to the target device should be explicitly allowed by the storage administrator using appropriate mechanisms (e.g., LUN masking or zoning).

7 Mechanisms for passing protection information between nodes/components

7.1 Overview

At the T10 protocol level the interface for querying and determining Protection Information Capabilities is well defined. However, the remaining nodes of the I/O stack do not have a standards-based way to communicate their data integrity capabilities to adjacent nodes. The interfaces between nodes in the I/O stack are likely to be different. For instance, the interface between an application and operating system I/O library is different from the interface between a filesystem and a Logical Volume Manager. For the Protection Information Channel to be established nodes in the stack must have some way of communicating their abilities.

7.2 Information required to be passed

A node will need to advertise its capabilities to adjacent nodes. The interface is not necessarily the same at both sides of the node. The node will have to communicate:

a) The size and format of the Address Stamp, Application Stamp, and Data Stamp;

b) How much data in the Data Buffer goes with each Data Integrity Metadata (e.g., the protection information interval); and

c) Whether each of the stamps are returned as written

7.3 Full discovery requirement

Since stamps may contain information that can not be recreated on the fly. It is therefore important that the Initiating node knows whether the stamps will be returned as written. Consequently it is required that the entire protection envelope goes through discovery before a request with Data Integrity Metadata is submitted. The process for discovery is outside the scope of this architectural model.

8 Data integrity aware Application Programming Interfaces

Application to operating system programming interfaces vary between platforms. Even within a single operating system there may be many different ways for an application to submit I/O, each with different buffering and completion characteristics. It is outside the scope of this architectural model to explicitly define operating system I/O interfaces. This model does, however, provide a set of recommendations for a data integrity aware interface.

8.1 Protecting and validating a data buffer

Since applications generally have no knowledge about the characteristics of the underlying storage, including logical block sizes, protection information type, etc., it is recommended that the data integrity-aware programming interface provide a set of functions that allow applications to protect and verify buffers without knowing the actual format of the Data Integrity Metadata. These functions can be called explicitly by the application, or they can be called transparently by the I/O library or operating system kernel.

If a node in the I/O stack receives an unprotected I/O request, and it knows that the next node in the direction of the I/O request is integrity-capable, it may assume the role of an Initiating node and request that the buffer be protected.

8.2 Error handling

Data writes are most frequently performed with the operating system doing intermediate buffering which allows the application to keep running while the storage is working in the background. A drawback to this approach is that the application will get I/O completion as soon as the data has been buffered by the operating system. A subsequent error between the operating system and disk could leave the data unwritten, and there generally are no means for providing the application with a deferred error notification.

In the Data Integrity Architectural Model it is the responsibility of the Error Handling node to act upon data integrity errors. Completion shall not be signalled to the Error Handling node before the storage device has returned completion status[1]. If the Error Handling node is the application, the operating system must provide a programming interface that enables the application to get comprehensive I/O completion status. On many systems this involves using either unbuffered I/O or asynchronous I/O submission routines.

[1] If the storage device contains volatile cache, failure of that volatile cache may result in a lost write.