Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diploma Thesis WS 2003/04

Integrity Protection for Secure Networked Storage

Daniel Wagner

Abstract

Storage Area Networks (SAN) enable many computers to access a set of storage devices over a high-speed network. Because a SAN is typically used to provide direct access to shared storage, security and integrity protection of such storage devices are an important issue. Securing the contents of a file can be done by using cryptographic methods to encrypt the data. Encryption alone does not guarantee a secure filesystem because a malicious program could tamper a file on the SAN without being detected. It is assumed that although it is not possible to prevent this without slowing down the data path, it is at least possible to detect tamper attacks. In this diploma thesis the aspect of integrity protection is addressed.

Danksagung

An dieser Stelle möchte ich mich herzlichst bei Marcel Waldvogel bedanken, der die Diplomarbeit überhaupt ermöglichte und mich dann ausgezeichnet bei dieser Arbeit betreute.
Weitherhin möchte ich mich bei folgenden Personen für kleine und grössere Ideen oder Anregungen bedanken: Charlotte Bolliger, Christian Cachin, Patrick Droz, Caroline Dubois, Daniel Engeler, Robert Haas, Daniel Lutz, Danat Pomeranets.

This thesis was carried out at

IBM Research GmbH
Zurich Research Laboratory

Contents

List of Figures

List of Tables

Overview 1

1.1 Introduction

With the growing amount of data stored, the importance of stored data to individuals, companies and governments is also increasing. In addition to the traditional criteria, such as performance, capacity, and reliability, security is quickly becoming an important feature of storage systems. Storage space is typically provided by complex networked systems, in which clients communicate directly with a disk over a network (as in network-attached storage (NAS), or in storage-area networks (SANs)).

The main properties of a secure storage system are the following:

Confidentiality
: Protection against unauthorized disclosure, for example through eavesdropping on data.

Integrity
: Protection against illegal modification, which may involve only subtle changes to the stored data.

Availability
: Protection against accidental or intentional removal of the data.

A typical networked storage system consists of many and potentially distrusting clients, the data storages themselves (typically, these are standard disks attached to a network, but they may also offer advanced functions), and meta-data servers, which mediate access to the data storages. Unauthorized actions may occur on all three elements and on the network linking them. This means that security properties are at risk in two different locations:

Data in flight
: Unauthorized actions on data or on meta-data being transmitted between authorized clients, data storages, and meta-data servers. Such attacks are similar to problems encountered in other communication applications.

Data at rest
: Tampering with the data storages, either by physically accessing the data storages or by sending appropriate commands to the storage system. These commands may also be initiated by clients authorized to access other parts of the storage system. Such attacks are unique to storage systems.

The risk of attacks on data at rest is in general rated higher than that of attacks on data in flight; among other things, an attacker usually has more time and flexibility to access data at rest than data in flight.

Known mechanisms in the literature include using access-control mechanisms such as per-block access control [AJL$^+$03] and object-based storage [ADF$^+$03]; integrated security mechanisms

[MFLR02] with a file system based on PGP as one extreme [Hal99]; key-management issues [CCDP01, MKKW99], and tweak-able encryption for block-level security [HR03].

1.2 Integrity Protection

By using cryptographic measures on the data alone, i.e., without change to the disk interface, it is not possible to prevent unauthorized overwriting of data blocks. Alterations can be detected, but not prevented, by using cryptographic *integrity-protection* mechanisms. After detecting an illegal modification, the data must be restored from a backup. Mandatory versioning supported by the storage system can also allow automatic recovery.

1.3 Task Description

As part of this work, the existing SAN File System protocol [IBM03] has to be extended to support integrity protection. Tradeoffs between data structures and speed when accessing, modifying, and creating data have to be investigated. The goal is that incremental changes to or partial read from a file do not induce significant I/O or computation costs.

Issues to be evaluated include variable record sizes and the treatment of sparse files.

In addition to the theoretical study and observation, a prototype implementation based on the IBM SAN.FS as well as performance and functionality evaluations are expected.

1.4 Chapter Overview

Chapter 2, Introduction to SAN File System
: In this chapter a short introduction to the SAN File System is given. The general architecture of the system and the key features are listed. In addition, a more detailed introduction into data locks is given, which is essential for understanding the design decisions in the next chapter.

Chapter 3, Design
: This chapter will discuss the implementation-independent part of the design for integrity protection in the SAN File System.

Chapter 4, Protocol Design
: This chapter describes how the existing SAN File System Protocol can be extended to support integrity protection.

Chapter 5, Implementation
: This chapter covers some interesting implementation issues revealed by the prototype implementation. A general overview of the meta-data server implementation will be shown as well.

Chapter 6, Evaluation
: This chapter provides short evaluation of the proposed solution.

Chapter 7, Conclusions
: This chapter summarizes the results and provides an evaluation of this work.

Introduction to SAN File System

As part of this work the existing SAN File System has to be extended with integrity protection. This chapter introduces the key concepts of the SAN File System and summarizes those parts of the protocol specification document [IBM03] that are important for this thesis. The individual description has been taken almost verbatim from [IBM03].

2.1 SAN File System Components

Figure 2.1 shows a typical configuration of a SAN File System, consisting of client, storage devices and servers connected over an SAN and an IP-based network. The configuration consists of the following components:

- A set of SAN File System *clients*. These nodes read and write data to the storage devices.
- A cluster of *meta-data servers*. A meta-data server cluster manages all meta-data exported by the file system name space of that cluster.
- Some *storage devices*, labelled “data” in the figure. Here the storage devices are SCSI block-storage devices.
- A SAN to connect the clients, servers, and logical units (LU).
- One or more *administration consoles* for controlling the meta-data servers.
- An IP-based network to connect clients and servers.

The SAN File System supports heterogeneous clients, so that data on the SAN can be accessed from a Windows client or a Linux client. Although the SAN File System is a distributed and heterogeneous file system, the design goal is to provide applications running on clients with a single-site semantics for file-system access.

2.2 Protocols

As shown in Figure 2.1 the clients communicate with the meta-data servers over an IP-based network, whereas for access to the storage devices a dedicated SAN network is used, although it is also possible to use a single IP-based network such as Ethernet.

The various entities described above communicate with each other using several different protocols:

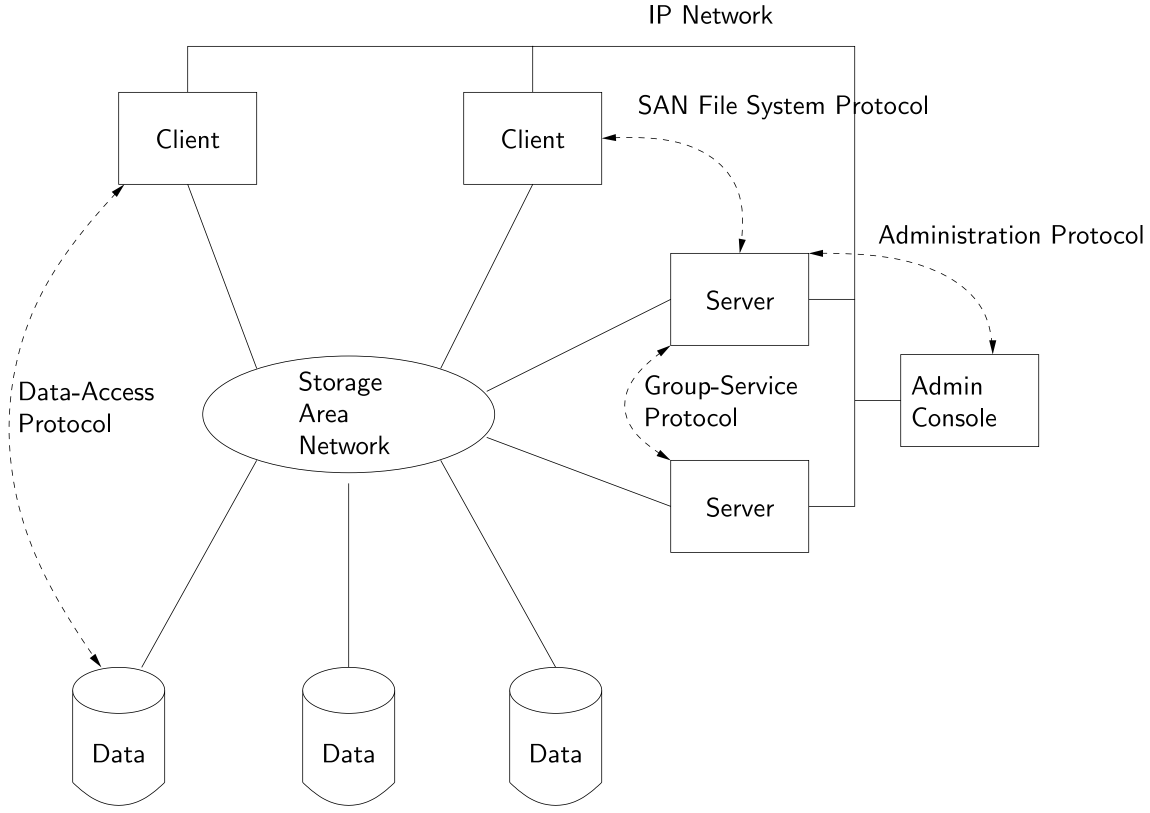

Figure 2.1: Entities in a file system. If the SAN is IP-based, the two networks may be merged.

SAN File System protocol
: Clients communicate with the meta-data servers using the SAN File System protocol. This protocol runs on an IP-network and can either use UDP or TCP as the underlying transport protocol.

Data-access protocol
: Clients access the block-storage devices using the data-access protocol which runs over iSCSI, SCSI over Fibre Channel, or SCSI over a parallel bus.

Cluster-group service protocol
: The nodes in a cluster of meta-data servers run this protocol to form a homogeneous view of the cluster, as individual nodes leave the cluster upon failure and rejoin when they recover. This protocol also allows the cluster to be managed as a single virtual entity.

Administration protocol
: This protocol is used by the administration console to communicate with nodes in the meta-data server cluster for configuration and administration purposes.

2.3 Key Design Features

Centralized client-server architecture
: The SAN File System is based on a client-server design. File-system clients accessing the global name space need not be aware of each other; they are only aware of one ore more servers that provide access to the name space.

Separation of meta-data from data
: File-system meta-data is managed separately from the data by the server cluster.

Centralized control of application synchronization
: All synchronization of application access to file-system objects is controlled by the server cluster.

Centralized control of client caching
: The SAN File System allows caching of meta-data and data on clients. Control of cached content on clients to maintain cache consistency is the responsibility of the servers.

Cross-platform file-system access
: The SAN File System allows cross-platform access to the file-system name space.

2.4 File-System Objects

The SAN File System consist of three types of objects:

Files
: A *file* is an unstructured ordered set of bytes, containing data that is accessed by clients. The content of a file is opaque to the file system itself.

Directories
: A *directory* is an interior node in a file-system name-space hierarchy, whose children are other file-system objects.

Symbolic links
: A *symbolic link* is a point in the client's file-system name tree at which the name lookup of on an object is redirected.

Every file-system object has a *global unique identifier* (referred to as "object ID" in the remainder of the document) that is assigned to it during object creation by the server processing the creation command. This object ID is immutable throughout the lifetime of the object. Object IDs are only intended for internal file-system use.

The 4-tuple <*clusterID.filesetID.objectID.versionNumber*> is a notation used to represent the unique object ID of an object. In Figure 2.2, the globally unique IDs of some file-system objects are illustrated. In the figure, the global root has the ID <*0.0.0.0*>, the cluster root has the ID <*4402.3.1.0*>, and so on.

The latest version of an object has the *versionNumber* 0.

2.4.1 File-sets

The SAN File System provides the concept of a *file-set*, which corresponds very loosely to the concept of a mountable file-system. Like the boundary between mountable file-systems, there is a directory, or a *file-set attach point*, that forms the boundary between two file-sets and each file-set corresponds to a subtree of the overall SAN file-system name space, as shown in the example in Figure 2.2.

2.5 Global Name Space

The SAN File System organizes all objects in the file system into a *global name space*. As in other file systems, the name space is organized as a tree.

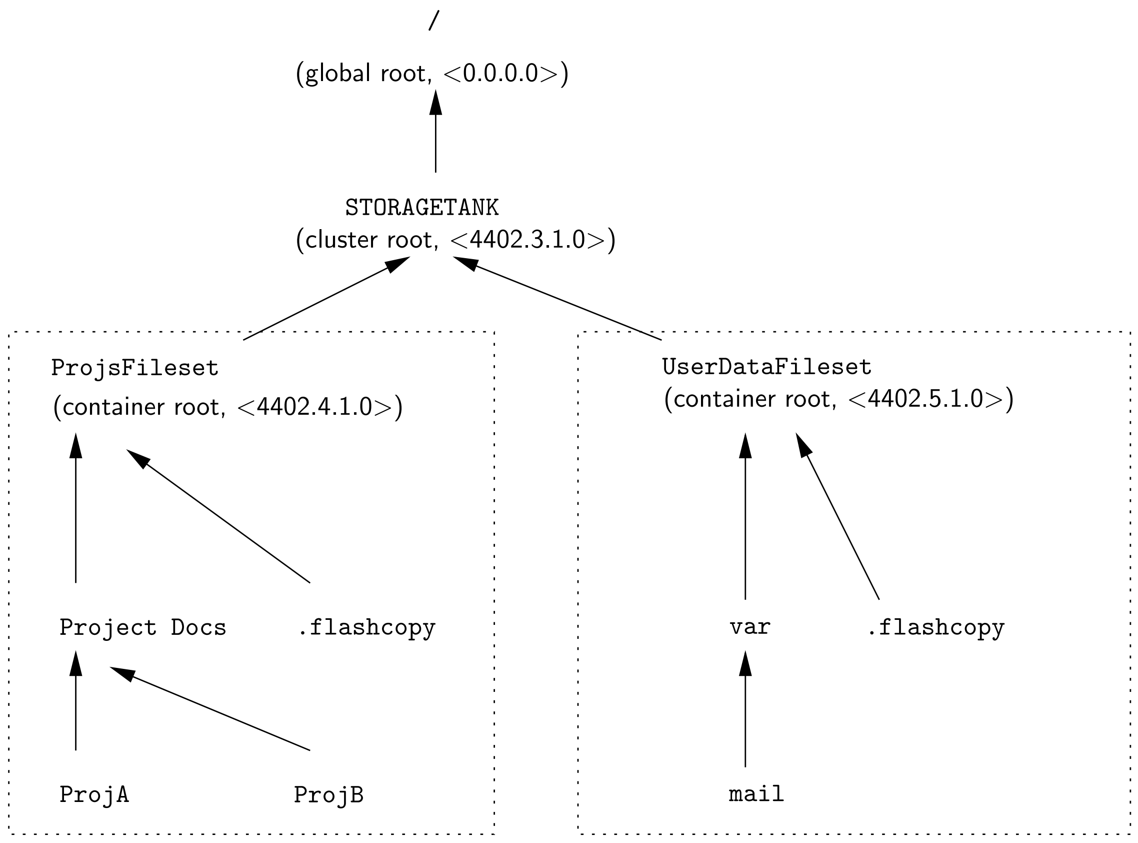

Figure 2.2: The dotted lines denote file-set boundaries. The `ProjsFileset` and `UserDataFileset` directories are file-set attach points.

2.6 Leases

In the SAN File System, all cached meta-data and the maintenance of application-level locks on objects that a client manages are protected by various types of file-system locks. The prerequisite for holding such locks and performing the operations permitted by those locks is that the client continuously maintains a valid lease with the server.

A *lease* is a timed contract handed out by the server in which the server promises to honor the client's locks for a specific period of time that they both agree on.

A lease allows a client to operate efficiently when serving local file-system requests. As long as a client has a valid lease with a meta-data server, it can access data, cache content, and assume that the files opened by local applications are protected, without having to continuously check with the server which owns the objects that are being operated on.

2.7 Synchronization of Client Access to File-System Objects

The server that owns the load unit[1] in which an object resides provides a central point of control for the management of how those file-system objects are being accessed at any point in time by

[1]Conceptually, objects are grouped into load units, and each load unit is served by exactly one server at any given time

the various clients in the network.

SAN File System provides a *session-lock* feature for clients to mediate their object open states through the server, both independently and on behalf of local applications.

The UNIX and the Windows platform also provide APIs to applications for explicitly locking access to an entire file or to a range of logical addresses in a file. The SAN File System provides a byte range locking feature for implementing these application semantics across machines.

The SAN File System supports aggressive caching of meta-data and data on clients, when application-access modes indicate that such caching would be useful[2].

2.7.1 Session Locks: open/close

Several different *modes* are defined for a session lock. The semantic of a session lock mode defines the operations the lock holder is allowed to perform on the object, and the operations on that object that other lock holders are allowed to perform.

The SAN File System protocol is designed to minimize network traffic for such lock management. The client is not required to communicate with the server for every application "open" call. Instead, the client is expected to acquire a session lock in a mode compatible with an initial application "open" call. It is allowed to hold the lock even after the application "closes" the file and to reuse the lock if another application "opens" the file in a compatible mode, *without communicating with the server*.

2.7.2 Range Locks: Access Coordination

The SAN File System provides a range-lock feature for files to allow clients to map application requests to acquire byte-range locks on a file to SAN File System range locks, and to request the server to allocate the lock after checking for conflicts from other (applications on other) clients. A range lock on a file is subordinate to a session lock; therefore the client must prove that a session lock is held in some mode before it can submit a range lock request to a server.

2.7.3 Data Locks: Caching

The SAN File System provides a *data lock* feature for clients to coordinate the caching of directories, symbolic links, and files.

For directories and symbolic links where caching is read-only, the data lock provides "publish-subscribe" semantics in which any change to the object's meta-data (including directory content) results in notifications being sent to other clients that also hold that lock.

The locks on files are more complex. There are two shared modes for caching, one in which both data and meta-data can be cached in read-only mode, and the other for non-cached (direct) I/O. In each case, multiple clients can hold the lock and cache meta-data in read-only mode. There is also an exclusive lock mode that allows a client to maintain dirty meta-data and data caches.

A client can hold at most one data lock per object. Thus, a data lock is used to control the caching of an *entire* object.

A client acquires data locks to perform two key functions:

[2]Applications such as database management systems manage their own caches, and typically do not benefit from additional caching of *data* in the underlying file system. However, caching of meta-data such as block-allocation maps is still useful.

Table 2.1: Data Lock Summary

Mode	Lock holder cache ops				Stronger than	Weaker than	Comp. with	Lock semantics
	RM	RD	WM	WD				
C	+	−	−	−	−	−	C	Clean mode. Directory and symbolic links only.
SR	+	+	−	−	−	X	SR	Shared read. Files only, cached I/O.
SW	+	−	−	−	−	X	SW	Shared write. Files only, direct I/O (no data cache).
X	+	+	+	+	SR, SW	−	−	Exclusive. Files only. No data cache for direct I/O.

- To cache data and meta-data for all types of objects, and to protect the cache by relying on the property that the cached content will not be modified by other client without all lock holders being notified by the server.
- To read and write the contents of files.

A data lock is used for both data and meta-data. Furthermore data locks are completely independent of session locks, although they are related in the sense that a client typically holds a “matched” pair of such locks together.

Several different *modes* are defined for data locks. These are summarized in Table 2.1. The mode of a data lock determines the types of cache operations the lock holder can perform. The following four categories of operations are permitted on client-side caches:

1. Cache meta-data in read-only mode (RM).
2. Cache data in read-only mode—applicable only to files (RD).
3. Cache meta-data in read-write mode, allowing updated meta-data to be buffered in the client and thus to be temporarily inconsistent with the server’s state (WM). This is only allowed for files.
4. Cache data in read-write mode—applicable only to the content of the files (WD).

Clients acquire locks either explicitly or as a side effect of creating or finding an object. Certain commands also result in the server granting locks (both session and data) to the requesting client in an “opportunistic” manner, with the expectation that clients will need to use these locks in timely fashion.

When a client requests a data lock on an object in a certain mode, the request will not be denied. However before the lock can be granted, the server must coordinate changes to the data lock (and corresponding cache) states of all other clients as necessary so that they are compatible with the lock mode being granted to the new client. A server sends a `DemandDataLock` message [IBM03, Section 6.6.3] to a client to demand that a client downgrade its data lock to a mode compatible with what is being requested by another client. Data locks are *fully preemptible*: because a data lock does not protect application state, the SAN File System protocol rules state that a client must honor the demand and downgrade the lock.

Data Locks for Directories and Symbolic Links

Directories and symbolic links are "meta-data-only" objects. All content associated with these objects is managed by the server. Clients cache directory and symbolic-link content in "read-only" mode. The SAN File System protocol requires that *all* updates to the basic attributes of such objects, such as the modification time, to security-related aspects such as permissions, and to their content have to be performed by submitting transaction requests to the server.

As neither attributes nor content can be modified locally, none of the complexities of managing "dirty" caches in exclusive mode need to be addressed for directories and symbolic links, resulting in a simple set of rules for managing cache coherency for these object types.

The SAN File System provides a *clean* or *C-mode* data lock that all clients must acquire to protect their "clean" caches of directories and symbolic links. The C-mode data lock provides a publish-subscribe mechanism. Whenever one client submits a transaction request on a directory of symbolic links that would result in modifications to the object's attributes or contents, the other clients that hold a C-mode data lock on that object can be certain that they will receive notifications regarding such changes from the server.

Data Locks for Files

File-related caches are fundamentally different from those of directories and symbolic links in two ways. First, unlike meta-data-only objects, files consist both of data and meta-data, which are kept in separate block storage devices and accessed through separate paths. Therefore, the rules for caching data versus meta-data have to be considered separately, as reflected in Table 2.1. Secondly, most attributes of files can be modified locally by a client (under the appropriate data lock) without issuing a transaction between a client and a server for managing meta-data, in contrast the protocol described for directories and symbolic links. There is no publish-subscribe model of data locks for files for maintaining cache coherency via notifications from the server. Instead, a single client can modify meta-data locally under an exclusive-mode lock. Other clients can compete for the lock and each client should eventually have the lock granted to it. When the lock is granted to a client, the client can view and cache the consistent meta-data for that object.

Three data-lock modes are provided for managing file caches:

Shared Read (SR)
: This mode allows clients to cache meta-data and data for read operations. The SR-mode lock can be concurrently held by multiple clients.

Shared Write (SW)
: In this mode, clients *cannot cache data*, but can cache meta-data in read-only mode. The SW-mode lock can be concurrently held by multiple clients.

Exclusive (X)
: This exclusive mode allows a *single* client to cache both data and meta-data, which the client can read and modify.

2.7.4 Direct I/O

Certain applications such as database-management systems implement sophisticated cache-management systems for operation on top of either a block device or a file system. In parallel implementations, they use proprietary protocols to coordinate their access to disjoint ranges of a block device of files. For such applications, the SAN File System supports a *direct I/O* mode. In this

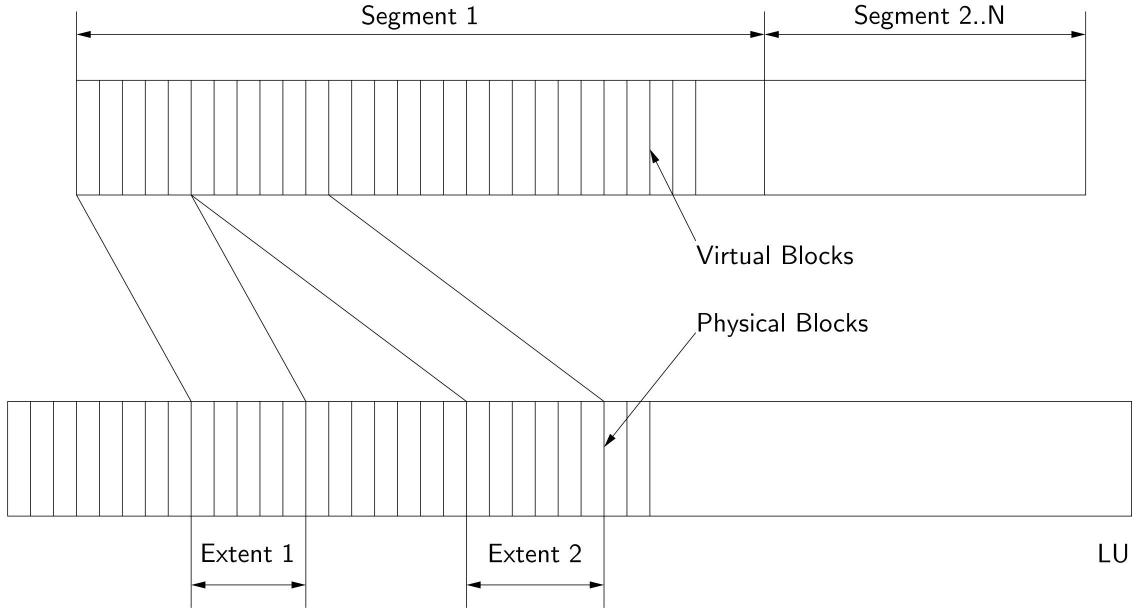

Figure 2.3: Block Storage Allocation.

mode, no file-system data cache is maintained for the file. Meta-data (e. g., block-allocation maps) is cached in read-only mode[3].

2.8 Managing File Access

The architecture of the SAN File System can accommodate several *strategies* for accessing the contents of files on the SAN. In the current version of the protocol, only two data-access strategies are defined:

Strategy #0
: Also known as the *null* access strategy. Data for a file tagged with this strategy number cannot be accessed by clients. Essentially, it is a no-op strategy used for directories and symbolic links

Strategy #1
: This strategy is used to access data in SCSI logical units discovered on the SAN, where a file's data is organized into units of fixed-size blocks, and higher-level organizational units—segments and extents—are built on top of blocks. Currently this strategy is used for all files.The details of this strategy are explained in Section 2.8.1.

2.8.1 File Data Layout for Block Disk Strategy #1

Block disk strategy #1 supports file access over a full 2^{64}-byte-long address range. It also supports sparse files, e. g. files where parts of address space have never been written to and do not occupy storage space on disk.

A file's address space is split into *segments*. Each segment consists of exactly 256 *blocks*, where the block size is variable and can be any power of 2 between 4096 bytes (4 KB) and 262144 bytes

[3]Shared Write (SW) data lock

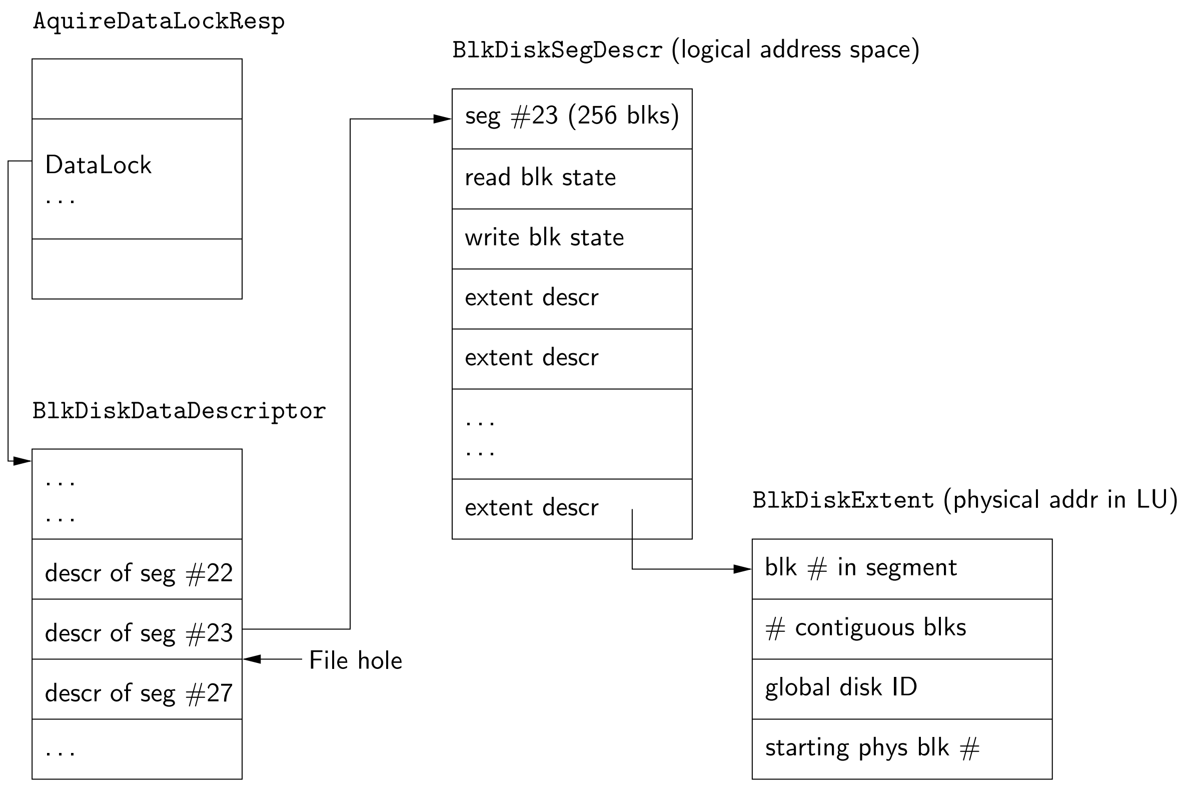

Figure 2.4: Block-Allocation Map.

(256 KB). For a given byte offset n in a file, the corresponding segment is $\lfloor n/(256 \cdot \mathit{blocksize})\rfloor$.

The segments that are part of the logical address space of the file are mapped into extents, which are part of the physical address space of a block device of the SCSI logical unit (LU) on the SAN. In this context, an *LU* is a block-addressable unit of storage.

Each extent consists of a *contiguous* range of physical blocks on an LU. The block sizes for a segment and for an extent have to be identical because the extent represents the physical storage corresponding to a contiguous region of the file's address space that consists of the same number of blocks. The block size used for a given file is one of the attributes of the file and is fixed for all extents of a file.

As a segment is exactly 256 blocks long, each segment requires 1 through 256 extents to store its data, and because an extent can consist of 1 to 256 contiguous blocks, with a 4 KB block size, an extent can be at most 1 MB long.

Blocks are always completely used or unused.

2.8.2 Block-Allocation Maps

`BlkDiskExtent`, `BlkDiskSegDescr` and `BlkDiskSegDescr` [IBM03, Section 4.14] are the key data structures used to represent the mapping between logical segments and physical extents, also referred to as the *block-allocation map*. A granted data lock allowing file access contains a `BlkDiskDataDescriptor` structure which includes a list of `BlkDiskSegDescr` structures. Each segment descriptor represents a segment in the logical address space of the file, and the structure in turn contains a number of `BlkDiskExtent` structures that describe how contiguous blocks inside a segment are mapped to the same number of contiguous blocks on an LU identified by the global disk ID.

2.9 Object Versioning and File System Level FlashCopy

The SAN File System supports the creation of multiple versions of an object, all but one of which is designed to be immutable. In the current version of the protocol, the scope of such versioning is at the file-set level. It also supports the creation of a FlashCopyImage of a file-set as an administrative operation. The term FlashCopyImage refers to an immutable "snapshot" or "point-in-time copy" of an entire file-set. When a FlashCopyImage is created, the name-space subtree in the file-set boundary, without any wormholes[4], is recreated under the `.flashcopy` directory under the name assigned to the FlashCopyImage during the creation operation.

A FlashCopyImage corresponds to an immutable *version* of a file-set. Every FlashCopyImage creation results in a monotonic increase in the version number of every object that belongs to that image. Only the portion of a file-set that is not under the `.flashcopy` directory contains modifiable objects, and is referred to as the *primary image*. The version number of objects in the primary image is 0.

FlashCopyImage creation is not an expensive operation, because creation of the "replica" name space *does not involve copying file data*. However, meta-data updates are required to represent the new name space, to reflect new versioning information corresponding to objects in the new image, and to update the block-allocation maps of files in the primary as well as in the new FlashCopy image.

Immediately after the creation of a new FlashCopyImage, any given file in the primary image and the corresponding FlashCopyImage have identical content. However, once the object in the primary image is modified by a client, its contents diverge from that of the FlashCopyImage (as well as from that of older images). Each file in a FlashCopyImage shares data (physical blocks on LUs) with the primary image until the latter is modified. On modification of a file object, *copy-on-write* techniques are used to preserve the immutability of the physical blocks that are being referenced by older versions of the object.

[4]attachment points of other file-sets

3 Mechanism Design

This chapter describes how the integrity-protection feature can be added to the SAN File System. The proposed protocol extension does not change the existing protocol in any way. Existing clients and servers without the protocol extension should be able to connect to clients or server that have this extension. A policy for mixing clients and servers with or without the protocol extension must exist.

3.1 Benefits of Distributed File Systems

Existing local file-systems for the various clients (e.g. Windows, Linux or AIX) all share the same advantages and disadvantages: On the positive side are the performance and the integration into the operation system[1]; on the negative side is the lack of support for sharing data stored on the local disks.

In contrast, distributed file-systems such as NFS [Now89] or CIFS [CIF] provide applications running on multiple (client) machines with access to the same file-system name space. In such systems, a single server coordinates the access to the data.

The challenge in designing of a distributed file-system such as the SAN File System is to build a file system that can exploit shared access to SAN storage, and provide sufficient coordination capabilities such that individual clients on multiple machines can access the same shared storage in a consistent manner.

The key benefits of the SAN File System is that it enables distributed file system-style sharing while allowing clients to access a large amount of file-system data directly from SAN as in a local file system. It can avoid some of the performance overheads of distributed file-systems in which all access to file system objects, both data and meta-data, have to go through a single server. The ultimate goal of the SAN File System is to provide distributed file-system sharing semantics with local file system-like performance.

3.2 Performance versus Integrity

Adding integrity operation at storage-device access level would be a very simple solution. Of course, all accesses would need to be first authorized by the meta-data server. This could be done very efficient by handing out a token to the client and to the storage device. The main drawback is the additional workload added to the storage device. Furthermore, the integrity of a file must only be guaranteed at the end of a data lock phase.

Instead, if the client has to do integrity operations, then also the client is charged with the additional workload. Furthermore, the client will be able to optimize the integrity operations such

[1]For example ACL support.

as lazy generating of integrity information. Furthermore, the data path to the storage device is not slowed down.

3.3 Integrity Information

Integrity can be ensured by storing a cryptographic hash of the file contents in a secure place. If a client wants to ensure that the data has not been changed, the client computes a hash for the file and compares it with the value retrieved from the secure place. As the meta-data server is ultimately trusted, it is the natural place to save the additional meta-data there. Section 3.4.2 discusses the reasons for storing the hashes on the meta-data server.

All message send between the meta-data server and clients have to use an authentic channel. Note that it does not need to be an encrypted channel.

Hash functions such as MD5 [Riv92] (Message Digest 5) and SHA-1 [EJ01] (Secure Hash Algorithm 1) have two important properties:

- They map an input string (pre-image) of variable length to a fixed-length output (image).
- They are collision-free[2].

Because of these two properties, they are the perfect candidates for building a secure checksum. First, it is necessary to map the input data to a (short) fixed-length value because we do not want to store the complete data on the meta-data server again. Second, the resulting computed value should be distinct so that manipulations on the data can be detected.

MD5 and SHA-1 are both one-way functions. This property is not mandatory for integrity protection because the content of a file still is readable from any client and therefore must be secured by other means, like ciphering.

Furthermore, integrity protection must allow one to read and/or modify only parts of a potentially large file. It is important to use data structures that allows storing of hash values as optimal as possible for any size of a file and for any operation. A data structure that has these properties is the *hash tree*, also known as *Merkle Tree* [Mer80].

3.4 Hash Tree

The description below has been taken almost verbatim from [Mic00].

Recall that a binary tree is a tree in which every node has at most two children, hereafter called the *0-child* and the *1-child*. A *Merkle tree* [Mer80] with security parameter n is a binary tree whose nodes store values, some of which are computed by means of a collision-free hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ in a special manner. A leaf node can store any value, but each internal node should store a value that is the one-way hash of the concatenation of the values in its children. That is, if an internal node has a 0-child storing the value u and a 1-child storing a value v, then it stores the value $H(u\|v)$. Thus, because H produces n-bit outputs, each internal node of a Merkle tree, including the root, stores an n-bit value. Except for the root value, each value stored in a node of a Merkle tree is said to be a 0-value if it is stored in a node that is the 0-child of its parent, a 1-value otherwise.

[2]A hash function is collision-free if collisions are hard to find. The function is weakly collision-free if it is computationally hard to find a collision for a given message x. That is, it is computationally infeasible to find a message $x \neq y$ such that $H(x) = H(y)$. A hash function is strongly collision free if it is computationally infeasible to find any messages x, y such that $x \neq y$ and $H(x) = H(y)$. [Sch96]

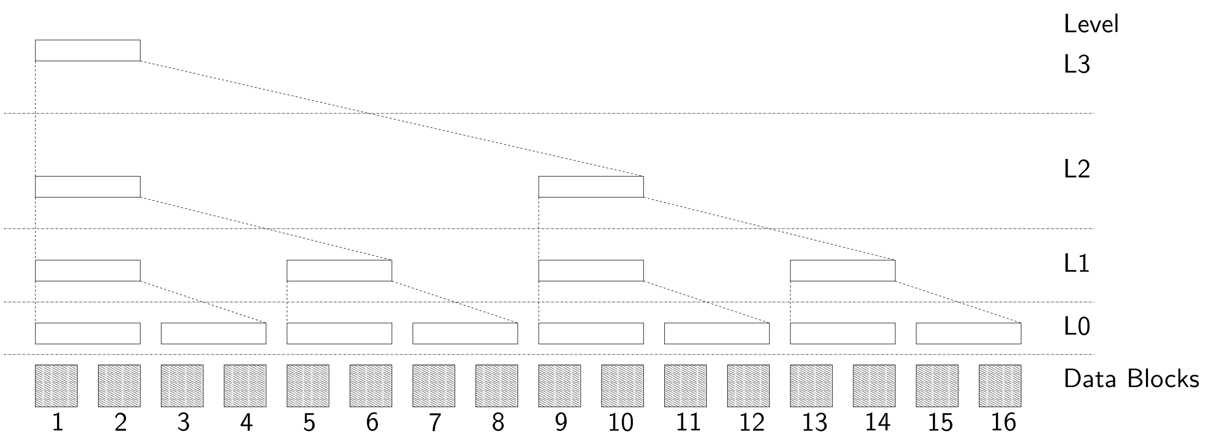

Figure 3.1: Binary Hash Tree. The white boxes are the *hsegments* (hash segments) and the hatched boxes are the data blocks. A normal hsegment (the figure shows only a binary tree) can cover from 1 up to 256 data blocks or hsegments. A hsegment is indexed with the virtual block start address and the level.

The crucial property of a Merkle tree is that, unless one succeeds in finding a collision for H, *it is computationally hard to change any value in the tree (and, in particular, a value stored in a leaf node) without also changing the root value.* This property allows a party A to commit to L values, $v_1, \ldots, v_L$ (for simplicity assume that L is a power of 2 and let $d = \log L$), by means of a single n-bit value. That is, A stores value v_i in the i-th leaf of a full binary tree of depth d, and uses a collision-free hash function H to build a Merkle tree, thereby obtaining an n-bit value, R, stored in the root. This root value R "implicitly defines" what the L original values were. Assume in fact that, at some point in time, A gives R, but not the original values, to another party B. Then, whenever at a later point in time, A wants to "prove" to B what the value of, say, v_i was, A may just reveal all L original values to B, so that B can recompute the Merkle tree and then verify that the newly computed root value indeed equals R.

3.4.1 Applied Hash Tree

A *hsegment* stores hashes for each data block or segment it covers. On top of those hsegments a hash tree is builded. For simplicity, the inner nodes of this tree also use hsegments as their data type. Therefore, a hsegment has a context-(level)-depending semantic:

Level 0

The input string for the hash function is read from the data blocks. The pre-image size is *segment_size*$/2^n$ with $n = 0 \ldots 8$.

Level 1. . . 5

The input string comes from the hsegment (i. e. their hash values) of the level below the current one. The pre-image size in comparison to the Level 0 pre-image size, is rather small. The smallest segment size is 1 MB (256 blocks · 4 KB) and the normal hsegmet size is 5 KB for a SHA hash function with 20-byte hash values (256 blocks · 20 bytes).

Within a hsegment the pre-image size for the hash function is constant, i. e. a hsegment is generated out of 2^n for $n = 0 \ldots 8$ equal-sized pre-image data blocks or hsegments input data. It is not possible that a pre-image can cross a hsegment border.

If hsegment hashes 256 sub hsegments, then the hash tree has at most six levels. With the smallest hash block size of 4 KB (12 bit) and the number of blocks per segment (8 bit), 44 bit of the address range are left.

$$\lceil 44/8 \rceil = 6$$

3.4.2 Storing the Hash Tree

One of the main properties of a hash tree is that the hash value stored in the root node implicitly defines the entire file. Therefore, it would be sufficient to store the root node safely, that is the entire tree could be stored on the same storage device as the file (and also exposed to the same attacks). This approach has an advantage if the SAN network is faster than the IP network because reading and writing involve network transfers, which has to be carefully considered. Furthermore, the hash tree is separated from the root value. A small disadvantage is that real meta-data is stored on the data disks, so that the normal data space is used. Of course one could designate a special storage space that is not used for normal data. The real disadvantage of this approach lies in the additional logic needed for the FlashCopyImage (Section 2.9) feature of the file-system and buffer space management. The client duplicates server features.

Another solution is to store the complete hash tree on the meta-data server. This makes perfect sense if the SAN network is based on the same IP network as used by the clients to communicate with the meta-data server. This commonly is the case if iSCSI is used as the SAN network protocol. The advantage of this strategy is that the meta-data stay together and do not require additional space on the data storage. Another good reason for storing the hash tree on the meta-data server is that the logic for FlashCopyImage is much simpler. For more details see the implementation details in Section 5.1.3.

3.5 Data Locks

Section 2.7.3 introduced the basic concept of data locks and their use. The data locks allow multiple clients to access the file system and serialize their data manipulations.

As long as a client holds a data lock in SW or X-mode, it can access the storage device and change data. During this period it is not important that the hash values are updated immediately. Before releasing the data lock, the client has to update the normal SAN File System object meta-data. If integrity is supported, the client can also update the integrity information on the meta-data server after the normal meta-data update.

For supporting integrity protection, extending the semantics of the data locks have to be considered.

The C-mode semantic does not have to be changed, because the data always is stored in the meta-data server. Therefore, it is guaranteed that this information is never changed accidentally or deliberately. A client needs the appropriate access rights to change data, and the meta-data server checks the validity before executing any command.

The SR-mode is also very simple to handle because neither data nor meta-data can be changed. The client only has to fetch hash values from the meta-data server for checking.

In the X-mode, only one client holds a data lock. No inconsistency can arise from this situation because all changes on data and meta-data are done as transactions. So fetching the integrity information can be done when needed. The integrity information has to be updated when the lock is released. This allows the client to build the hash values only at the end of an X-mode session, and no additional overhead for each write operation is needed. Only at the end of a lock session do the hashes have to be generated. Therefore, a client can write the same block several times without having to generate the hash values for it. As the client will cache the writes updated anyway, the overall operation overhead for the integrity protection feature is as small as possible.

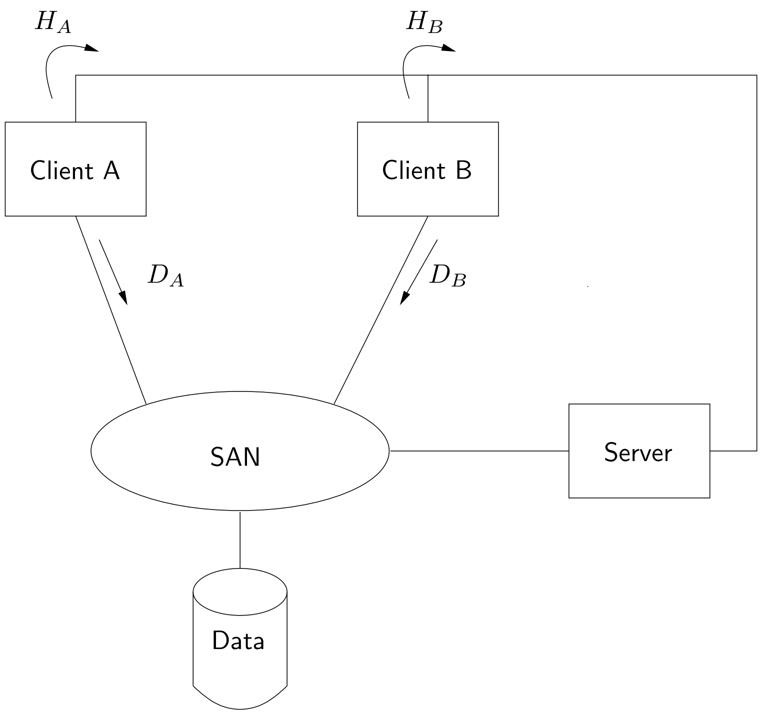

Figure 3.2: A race condition exists for the SW-W1 mode if the meta-data server handles H_A and then H_B while the data storage handles first D_B and then D_A

The SW-mode has to be considered in more detail, which is done next.

3.5.1 Race Conditions in SW-mode

All data lock modes except the SW-mode are fairly common and straightforward solutions to protect data from race conditions. But the SW-mode introduced in SAN File Systems has to be considered more carefully because in the SW-mode a simple update of the hash values (`BlkDiskSetHash`, see Section 4.3.4) leads to a race condition.

Let us consider the case of two clients A and B who are both writing to the same block. Client A sends the data block D_A to the storage device and the hash values H_A to the meta-data server. Client B does the same at the same time (D_B, H_B). The storage device executes first D_A and then D_B, while the meta-data server executes first H_B and then H_A.

The SW-mode is used for direct I/O (see Section 2.7.4). Unfortunately the read-only property of the meta-data conflicts with the need to store hash values on the meta-data server. First, we have to drop the requirement of the read-only property for the meta-data in SW-mode. Second, we have to introduce a hash property for the SW-mode. Here is a brief list of possible solutions:

SW-NP mode

No-op. The SW-mode is not supported at all. No integrity checks when reading. No updating.

Obviously, any SAN File System implementation which does not provide integrity protection is using this mode.

SW-FA mode

Fail. Trying to acquire the SW-mode will always fail for integrity-protected files.

The advantage of this mode is, that in general integrity protection is supported but not for the SW-mode.

SW-UP mode
: Upgrade to X. The SW mode is updated to the X-mode.

  The advantage of this mode is, that also for the SW-mode integrity protection is available and it is very simple to handle. The disadvantage is that the concept of the Direct I/O mode for application with sophisticated locking in user land is completely lost.

SW-W1 mode
: Update integrity information on write. This is an optimistic operation because it may result in the race condition described above.

  The advantage of this mode is that the concept of Direct I/O is not removed as with the SW-UP mode. The disadvantage is that a race condition exists.

SW-W2 mode
: Update integrity information on write, check integrity on the meta-server. This mode is based on the SW-W1-mode. Each client has to deliver both the old hash value for the written block as well as the new one. The meta-data server compares the old hash value with the stored value. If these two values match, the new value can be stored. In case of a mismatch, the block is marked as uncertain.

  The advantage of this mode is that the race condition of SW-W1 mode is removed. The disadvantage is that a client can mark all blocks as uncertain and so destroy the integrity information. Furthermore, an additional state has to be stored for each block.

SW-W3 mode
: Update integrity information on write, check integrity on the meta-data server. This mode is based on the SW-W2 mode. Again, each client provides the old hash value and the new hash value. The meta-data server rejects the operation if the old hash value does not match the stored hash value. The client needs to resubmit the change later on. This scheme guarantees that in each round at least one client can update the meta-data.

  The advantage of this mode is that blocks are not marked as uncertain and it is guaranteed that all updates are successful eventually. The disadvantage is that it is possible that clients start thrashing with X-mode lock.

SW-RM mode
: Allows clients to acquire the SW mode, but relinquishes integrity information for each block written.

  This mode is not desirable, because a faulty client can very easily destroy the integrity information by accident or on purpose.

For the SAN File System protocol extension the SW-W3 mode is used. Generally, concurrent writes to the same block are not expected because Direct I/O is only useful for application with sophisticated locking in first place.

4 Protocol Design

This chapter describes how the existing SAN File System protocol can be extended to allow access to and modify the hash tree described in Chapter 3. A design goal was that the existing protocol is not altered. Furthermore, the new protocol should be as efficient as possible.

To achieve the design goals, several new data types have to be introduced as well as five new messages.

4.1 Data Types

4.1.1 Integrity Block States

Even if integrity protection is available and activated, not all files should protected by the integrity-protection mechanism because of performance reasons. On the file level, this can be achieved by using the extended meta information about the file-like security information.

Also on the block level, it is desirable to enable or disable integrity protection on a per block basis. For example, sparse files normally have larger parts of the file address space for which no blocks have been allocated. For these ranges it is not necessary to create hashes. Other examples are files opened in SW mode: Some blocks might not need integrity protection because they are used only for synchronization when the application is running.

Therefore in addition to the allocation map, each block needs a new state which shows whether the block has a *defined* hash value.

The "defined" state can be altered through an fcntl() system call. The underlying SAN File System will then emit a `BlkDiskSetHash` (see Section 4.3.4) or `BlkDiskUpdateHash` (see Section 4.3.6) message with either a valid or an invalid (empty) hash value for the specified block.

All other states can be computed, e. g. if a client reads one of the following blocks:

sparse block, not allocated and not defined
: Allocation for this block has not been done, therefore no hash value is available. When this block is allocated, a hash value should be computed.

allocated but not defined
: The block has been marked as not to be included in integrity checks.

allocated and defined
: Perform normal integrity operations on read and write operations.

not allocated but defined
: Not allowed.

The minimum hash block size equals the minimum data block size (4 KB). The maximum hash block size is the size of an entire segment.

4.1.2 Basic Data Types Overview

In the SAN File System protocol, all integral quantities must be transmitted in *big-endian form*.

The following list describes the basic data types that constitute the building blocks for SAN File System protocol messages:

Vectors of Fixed-Length Data Items
: To encode variable-length lists of data item of *identical* size, the data type `Vector` is defined. See Section 4.2.1 in [IBM03].

Lists of Variable-Length Data Items
: The encode variable-length lists of data items of *variable* size, the data type `List` is defined. See Section 4.2.2 in [IBM03].

Capability Bit Mask
: To enable SAN File System clients and servers to discover and negotiate optional capabilities, the data structure `FnBitmap` is defined. See Section 4.3 in [IBM03].

Opaque Strings
: Several message contain fields whose contents are variable-length arrays of bytes which are opaque to the server. For these arrays, all bytes (including the zero byte) are legal, and no terminating zero is required. For such opaque arrays, the data structure `ArbString` is defined. See Section 4.5 in [IBM03].

4.1.3 `BlkDiskHashPair`

Byte	Type	Description
1 – 8	`Uint64`	Segment number (for levels > 0, this is the number of the first segment covered by this hash)
9 – 9	`Uint8`	Level number within the hash tree
10 – 10	`Uint8`	Block number within this segment
11 – 11	`Boolean`	Old hash defined (0 → not, 1 → is)
12 – 12	`Boolean`	New hash defined (0 → not, 1 → is)

The old hash and the new hash (each only if defined) follows directly the `BlkDiskHashPair` header.

4.1.4 `BlkDiskRejectedHash`

1 – 8	`Uint64`	Segment number (for levels > 0, this is the number of the first segment covered by this hash)
9 – 9	`Uint8`	Level number within the hash tree
10 – 10	`Uint8`	Block number within this segment
11 – 11	`Boolean`	Hash defined (0 → not, 1 → is)

The meta-data server's value of the hash directly follows the `BlkDiskRejectedHash` header, if defined.

Table 4.1: New message types.

Code	Messages	Class	Response	Fwd
0x80 3A	GetSecurity	Transact.	GetSecurityResp	N
0x40 3B	GetSecurityResp	Response	-	-
0x80 3C	SetSecurity	Transact.	SetSecurityResp	-
0x80 3D	SetSecurityResp	Response	-	-
0x80 66	BlkDiskGetHash	Transact.	BlkDiskGetHashResp	N
0x40 67	BlkDiskGetHashResp	Response	-	-
0x80 68	BlkDiskSetHash	Transact.	BlkDiskSetHashResp	N
0x40 69	BlkDiskSetHashResp	Response	-	-
0x80 6A	BlkDiskUpdateHash	Transact.	BlkDiskUpdateHashResp	N
0x40 6B	BlkDiskUpdateHashResp	Response	-	-

4.1.5 `BlkDiskHashDescr`

1 – 8	`Uint64`	Segment number (for levels > 0, this is the number of the first segment covered by this hash)
9 – 10	`Uint16`	Number of hashes that follow this structure header; can be zero
11 – 11	`Uint8`	Level of this hash in the hash tree ($0 \rightarrow$ hash over block contents, $i \rightarrow$ hash over “one segment” of hashes at level $i-1$)
12 – 12	`Uint8`	Shift count for hash-block size ($0 \rightarrow$ 1-to-1 mapping, $1 \rightarrow$ 2 blocks per hash, $2 \rightarrow$ 4 blocks per hash, $3 \rightarrow 8, \ldots, 8 \rightarrow$ all 256 blocks in a single hash)
13 – 44	`Uint8[32]`	Occupied bitmap. ($0 \rightarrow$ block does not have a defined hash, $1 \rightarrow$ block has a defined hash value). Hash[0] represented of the MSB of byte 0.

The hashes come directly after the `BlkDiskHashDescr` header.

4.1.6 `ExtendedSecurityAttributes`

1 – 2	`Uint16`	Hash algorithm ($0 \rightarrow$ undefined/not, $1 \rightarrow$ SHA-1)
3 – 4	`Uint16`	Encryption algorithm ($0 \rightarrow$ not used)
5 – 6	`Uint16`	Signature algorithm ($0 \rightarrow$ not used)
7 – 14	`ArbString`	Signature
15 – 22	`ArbString`	Encryption Key

The extended security attributes also contain fields for file encryption. As file encryption is not covered in this thesis not all fields have yet been completely defined.

In Appendix A, a different approach is shown. However, for simplicity the solution proposed here has been considered for implementation because the flexibility of the first proposal is not needed.

4.2 Message Types

4.3 Messages

In this section all SAN File System protocol extensions are listed.

Fields marked with the word *Hint* have a special meaning: The meta-data server is free to ignore the information provided by the client. This optional information allows the server to perform certain optimizations.

4.3.1 `Identify`

This is the first message a client sends to a server, to identify itself and obtain a lease [IBM03, Section 6.1.1]. The `Identify` message contains a capability bit mask to discover and negotiate optional capabilities. If Bit 0 is set, the integrity protection is supported. Up to now, no policy has been decided how to handle situation where an entity does and another does not support integrity protection.

4.3.2 `BlkDiskGetHash`

Type of message

Transaction command, client to server.

Description

Request hash values for an object.

Response

The server immediately acknowledges this request with a network-layer Ack message. After the server finishes processing the transaction, there are two possible outcomes:

- The transaction completed successfully and a `BlkDiskGetHashResp` is sent.
- An error occurred. In this case, the server responds with a `ReportTxnStatus` message. Typical errors include "Object not found".

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 60	`ObjId`	Unique object ID [IBM03, Section 4.9.2] of file.
61 – 68	`Uint64`	Number of first hash segment in the file for which hash descriptors are being requested.
69 – 76	`Uint64`	Hint: additional number of hash segments (after the required first segment) to be returned.
77 – 77	`Uint8`	Lowest level of hashes in the hash tree requested. The least significant 8· level bits of the segment number need to be zero (= aligned).
78 – 78	`Uint8`	Hint: additional number of higher levels desired.

4.3.3 `BlkDiskGetHashResp`

Type of message

Transaction response, server to client.

Description

This is a server's positive response to the `BlkDiskGetHash` command. This message contains a list of `BlkDiskHashDescr` (Section 4.1.5).

Response

The client acknowledges this message with a network-layer Ack message.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 52	`List`	Returned list of block hash structures `BlkDiskHashDescr`.
53 –		Variable data for 1 list.

4.3.4 `BlkDiskSetHash`

Type of message

Transaction command, client to server.

Description

Set hash values for an object. This command should be only used if the client holds an X-mode data lock.

If the client does not provide a hash value for the block specified, the hash value stored on the meta-data server for this block is deleted.

Response

The server immediately acknowledges this request with a network-layer Ack message. After the server finishes processing the transaction, there are two possible outcomes:

- The transaction completed successfully, and a `BlkDiskSetHashResp` is sent.
- An error occurred. In this case, the server responds with a `ReportTxnStatus` message. Typical errors include "Object not found".

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 60	`ObjId`	Unique object ID [IBM03, Section 4.9.2] of file.
61 – 72	`List`	List of block hash structures `BlkDiskHashDescr`.
73 –		Variable data for 1 list.

4.3.5 `BlkDiskSetHashResp`

Type of message

Transaction response, server to client.

Description

This is a server's positive response to the `BlkdDiskSetHash` command.

Response

The client acknowledges this message with a network-layer Ack message.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].

4.3.6 `BlkDiskUpdateHash`

Type of message

Transaction command, client to server.

Description

Before the client writes to a file, the client should modify the server's hash tree, because this prevents that a client has to write the same block several times before it receives an successful `BlkDiskUpdateHashResp` response. This command requests that existing hash values be updated, and is useful for clients in SW mode, because a race condition may arise if the client were to use a simple `BlkDiskSetHash`. See Section 3.5.1 for more information on the race conditions.

Response

The server immediately acknowledges this request with a network layer Ack message. After the server finishes processing the transaction, there are three possible outcomes:

- The transaction completed successfully, and a `GetSecurityResp` is sent.
- An missmatch occurred. The old hash value did not match the actual hash value on the meta-data server. In this case the client should first try to update its hash-value caches, and resend the message. After repeated mismatches, the client should upgrade its SW-mode lock to a X-mode lock and update the server.
- An error occurred. In this case, the server responds with a `ReportTxnStatus` message. Typical errors include "Object not found".

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 60	`ObjId`	Unique object ID [IBM03, Section 4.9.2] of file.
61 – 72	`List`	List of block hash pairs to be updated (`BlkDiskHashPair`).
73 –		Variable data for 1 list.

4.3.7 `BlkDiskUpdateHashResp`

Type of message

Transaction response, server to client.

Description

This is a server's response to the `BlkDiskUpdateHash` command. There are three possible outcomes:

- The returned vector is empty, which indicates a successful transaction.
- The returned vector is non-empty, which indicates unsuccessful transaction due a hash value missmatch. The vector contains the current information from the meta-data server.
- An error occurred. In this case, the server responds with a `ReportTxnStatus` message. Typical errors include "Object not found".

Response

The client acknowledges this message with a network-layer Ack message.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 48	`Vector`	Vector of rejected hash updates. If empty, the `BlkDiskUpdateHash` was successful. If non-empty, all updates were rejected due to the listed mismatches in old hashes (current values are being returned).
49 –		Variable data for 1 vector.

4.3.8 `GetSecurity`

Type of message
: Transaction command, client to server.

Description
: Request security information of an object.

Response
: The server immediately acknowledges this request with a network-layer Ack message. After the server finishes processing the transaction, there are two possible outcomes:

- The transaction completed successfully, in which case a `GetSecurityResp` is sent.
- An error occurred. In this case, the server responds with a `ReportTxnStatus` message. Typical errors include “Object not found”.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 60	`ObjId`	Unique file-object identification.

4.3.9 `GetSecurityResp`

Type of message
: Transaction response, server to client.

Description
: This is a server’s positive response to the `GetSecurity` command. This message contains a list of `ExtendedSecurityAttributes` (Section 4.1.6).

Response
: The client acknowledges this message with a network-layer Ack message.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 62	`ExtSecAttr`	Extended security attributes.

4.3.10 `SetSecurity`

Type of message
: Transaction command, client to server.

Description
: Sets security information of an object.

Response
: The server immediately acknowledges this request with a network-layer Ack message. After the server finishes processing the transaction, there are two possible outcomes:

- The transaction completed successfully, and a `SetSecurityResp` is sent.
- An error occurred. In this case, the server responds with a `ReportTxnStatus` message. Typical errors include “Object not found”.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].
41 – 60	`ObjId`	Unique file-object identification.
61 – 82	`ExtSecAttr`	Extended security attributes.

4.3.11 `SetSecurityResp`

Type of message

Transaction response, server to client.

Description

This is a server’s positive response to the `SetSecurity` command.

Response

The client acknowledges this message with a network-layer Ack message.

Message format

1 – 40	`TxnMsgHdr`	Transaction message header [IBM03, Section 5.2].

4.4 `fcntl()s`

With POSIX flags for opening a file (e. g. O_RONLY) it is not possible to express all options the proposed integrity protection offers. The general workaround for this kind of problem in UNIX is to introduce new fcntl. Of course, any fcntl that is not covered by the POSIX standard is highly system-specific and not known to arbitrary programs. Therefore, the default behavior of the file-system should be what the user would expect. That means that if an integrity error is detected by the file system the corresponding read call should return an error.

The following fcntls could be needed (the list is not exhaustive):

SetVerifyLevel

Set verification level.

never

Ignore all integrity-protection information for defined blocks. If the block is written, existing integrity-protection information has to be deleted.

for defined blocks

Check integrity only for defined blocks and ignore all undefined blocks. This should be default behavior.

always

Check integrity of all defined blocks. For undefined blocks return an error.

Defaults behavior or default values can be set in an appropriate `/proc` entry or similar.

SetBlkHashAggrSize

Set block hash aggregation size. The default value is one block per hash. Global default values could be changed with an appropriate `/proc` entry or similar.

SetAutoAggr

Should the hash block size be updated on write or read actions to that segment?

0 no autosize

1 autosize on new writes (append)

2 autosize on overwrites (shrink only if significant)

3 autosize on reads (if the client also has write permission) (shrink only if significant)

SetEncrypted
: Set the encryption state of new files.

5 Implementation

This chapter covers some implementation issues of interest and the general meta-data server design. The client has not yet been implemented.

5.1 Tank Server

The meta-data server implementation by IBM is called *tank*. It is written in C++, and object-oriented design approaches are used. The tank server is divided into several components. Only those components that are important for this implementation are briefly described here.

Object Meta-Data Manager (OM)
: The instance of an Object Meta-Data Manager class is a singleton object, that manages all Standard Containers (SC) registered with this server.

Standard Container (SC)
: A Standard Container object creates a Standard Container Schema Manager. Each instance of those standard (user-writable) container classes manages the schema for a particular container.

Protocol Transaction Manager (TM)
: The Protocol Transaction Manager instance is also a singleton. It handles transaction requests from clients.

Database (DB)
: The btree database component is the storage for the meta-data. Data members are called *records*. All records are stored in pages, and each record has to fit into a page.

5.1.1 Meta-Data Storage and Load Operations

An important class is the *Segment Table Entry (STE)*. All relevant information about the block mapping in a segment is encapsulated in instances of this class. Of course, when the Segment Table Entry object is written to the database (DB) only relevant information (in a compacted form) is stored. The Segment Table Entry (STE) also handles the FlashCopyImage feature.

Each record stored in the database has a key to allow its retrieval.

5.1.2 Transaction message example

When a client initiates a transaction, a transaction object is created for the message (transaction messages). This transaction object is then executed as `dspAction`[1]. Depending on the nature of

[1] `dspAction` stands for dispatch action.

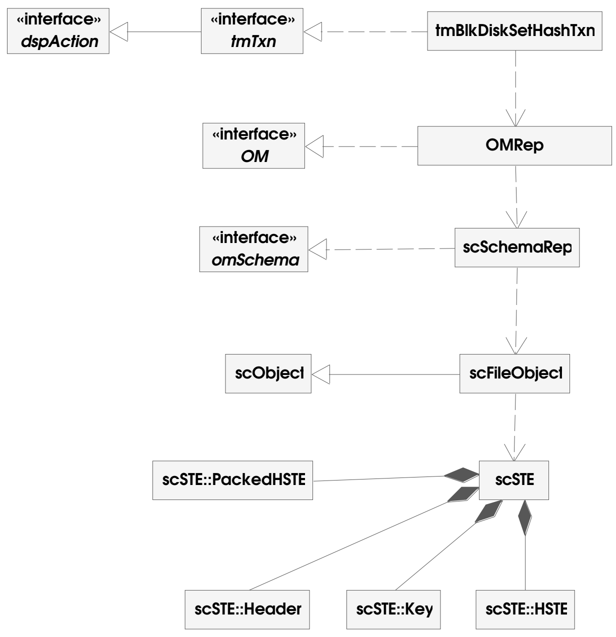

Figure 5.1: This UML class diagram shows a simplified overview of the classes that have to be edited in order to extend the existing SAN File System Protocol. Only the `BlkDiskSetHash` message implementation is shown here. Similar changes are needed for transaction messages such as `BlkDiskGetHash`.

the transaction, it uses different parts of the server. Here, we are only looking at block-disk strategy messages. Therefore, the Transaction Manager (TM) calls the Object Meta-Data Manager (OM), which delegates the work to the appropriate Standard Containter (CS). The schema manager looks up the file object id[2] in its database. With this data, a new `scObject`[3] is created—`scFileObject` in this case—which then reads all necessary data from the internal database. If attributes of an object have to be changed the `scFileObject` already contains the information needed. If segment manipulations have to be performed an instance of the Segment Table Entry (STE) is created. Loading and storing of STEs is done by the STE itself[4].

5.1.3 FlashCopyImage

Section 2.9 described the basic functionality of FlashCopyImage at user level. The FlashCopyImage feature of the SAN File System demands a careful handling of block allocation maps.

If a snapshot is taken, the current primary image is marked with the current epoch number and all blocks are marked as read-only. Then the epoch for this file is increased. If the client wants to update a block, a COW segment is created and only the block which is newly written to gets a new entry in the COW segment. The previous primary segment and the COW segment together contain the current mapping for the file.

When another snapshot is taken, all COW segments must first be merged with the previous primary image and a new primary image has to be created for this ending epoch. If during an epoch a segment has not been altered no new primary image has to be created, then several epochs can share the same primary image. Therefore, several lookups in the database could be needed until a valid segment is found (iterating of all epochs in the worst case).

[2]There also exist other objects, such as symlink or directory objects
[3]`scObject` is the generic SAN File System object representation.
[4]The STE handles FlashCopyImage and inline loading/storing of meta-data also, see Sections 5.1.3 and 5.1.5

The current implementation of the prototype does not support the FlashCopyImage feature. For supporting FlashCopyImage a similar scheme to the existing primary/COW segment image has to be implemented.

If a client has an object ID with the version number of 0, then it will allways have the current object.

5.1.4 Hash Segments

Data Type 5.1.1 A hash segment

```
struct HSTE
{
   Header header;
   uint8 level;
   uint8 hash_block_size;
   OM::HashState state;
   OM::Hash hash[ Max_HSTE_Hashes ];


   void insert_hash( int index, const OM::Hash *hashP );
   void remove_hash( int index );
   OM::Hash *find_hash( int rbn );

   void setLevel( const uint8 level );
   void setHashBlockSize( const uint8 size );

   uint8 getLevel( ) const;
   uint8 getHashBlockSize( ) const;

   void getHashState( OM::HashState *hashStateP );

   void getKey( Key *keyP, const OM::VirtSegNo &segmentNo,
                const uint8 level ) const;
};
```

Data Type 5.1.2 A packed hash segment

```
struct PackedHSTE
{
   Header header;
   uint8 block_size;
   OM::HashState state;
   OM::Hash hash[ Max_HSTE_Hashes ];
};
```

The Segment Table Entries (STE) are the key data structure for all file-system objects containing per block information. Therefore, we chose it as the most natural place to add support for storing the hashes. New data subtypes have been added to STE: `scSTE::HSTE` (see Data Type 5.1.1) and `scSTE::PackedHSTE` (see Data Type 5.1.2).

All nodes in the hash tree are called *hash segments*, short *hsegments*. There is no distinction between the leaf nodes and the inner nodes. The only difference between leaf and inner nodes is that the input for the leaf nodes are data blocks whereas the input for the inner nodes are hash segments.

`scSTE::HSTE` is used for an hsegment when it is in the main memory. All hashes are stored in a simple fixed-sized array. This makes it impossible to use a digest algorithm that produces an image larger than `Max_Hash_Size` bytes without truncation.

`scSTE::PackedHSTE` is used for storing a `scSTE::HSTE` into the database. Although the field `hash` is defined as a fixed-size array, only defined hashes are stored, leaving no gap between them. The `state` field contains the information which block is defined and where to put it. The total size of this structure without the hashes is 225 bytes.

5.1.5 Store Strategy for Hashes

All meta-data is stored into a *btree*-based database. To prevent serious performance hits when also hash values are stored and retrieved from the database, the hash values should be placed physically close to the corresponding STE. The database stores data according to a key that contains three fields which are considered according to their priority.

Field Member	Meaning	Order
Virtual Segment Number	The segment number of the STE	increasing order
Type	The type of the STE (either primary, COW or HSTE level)	decreasing order
Version	Type-specific PIT[5] version number	
	Primary: birth epoch of the STE	decreasing order
	COW: revert count[6]	decreasing order

The primary field is the segment number. So all segments are clustered by segment number. Inside a cluster, the segments are clustered by type again in decreasing order. The order of the types is as follows: primary[7], COW[8], HSTE_L0, HSTE_L1, HSTE_L2, HSTE_L3, HSTE_L4, HSTE_L5. Thus the hash segments are sorted according to how often a type occurs.

The root hash value is stored in the file-object attributes rather than adding another hash segment level to the hash tree.

5.1.6 Inlining

The existing Tank Server implementation takes into account that for small files (≤ 1 segment) the overhead for storing two records, first, the file-object attributes and second, the segment block mappings, is too large. To prevent this, the block mapping for small files is stored *inline* into the file-attribute data and no additional database access is needed for loading the segment data.

The restricting factor for storing data inline in the file-object attributes is the additional space needed. The current scheme allows the segment to be stored inline together with the hashes if it fits into one page[9].

[5]Point-In-Time

[6]The revert count is used for PIT cleaner which job is the remove unused STEs. Unused STEs appear when a FlashCopyImage is removed.

[7]The primary image of the file-system object. See FlashCopyImage, Section 5.1.3

[8]Copy On Write (COW) segment

[9]A page is the smallest entity with which the database operates. However, several records can be stored into a single page.

6 Evaluation

6.1 Building Cost

A performance-limiting factor is clearly how expensive it is to build a tree. There are various scenarios that have to be considered. Building or updating a hash tree is done at the client so that the client and not the server can be charged for the building cost. Furthermore, this strategy ensures that the server is not unnecessarily loaded with work. The server only stores, loads, and compares hash values, which should not be expensive operations.

The building cost of the tree depends on the size of a file. In general, the time needed to hash a pre-image is proportional to the length of the image. For the following calculation, hashing a data block costs $C_{\text{datablock}}$ and hashing a complete hsegment costs C_{hsegment}. Furthermore we assume a constant hash image size of s_{image}.

Let k be the degree of a node (for block-disk strategy #1 it is always 256; see Section 2.8.1) and e be the number of data blocks used by the total size of the file.

In the dense/contiguous case, where the entire data is stored in one contiguous block without a gap, the total costs are the following:

$$e = \left\lceil \frac{\mathit{filesize}}{\mathit{blocksize}} \right\rceil$$

$$D_c(e) = C_{\text{datablock}} \cdot e + C_{\text{hsegment}} \cdot \sum_{\ell=1}^{\lceil \log_k e \rceil} \left\lceil \frac{e}{k^\ell} \right\rceil \tag{6.1}$$

In the sparse case, where the data is stored with “gaps” between data, the costs are the following[1]:

$$f(b) = \begin{cases} 0, & \text{if } b \geq e \text{ or the block is free,} \\ 1, & \text{if } b < e \text{ and the block is used.} \end{cases}$$

$$S_c(e, f(\cdot)) = C_{\text{datablock}} \cdot \sum_{b=0}^{e} f(b) + C_{\text{hsegment}} \cdot \sum_{\ell=1}^{\lceil \log_k e \rceil} \underbrace{\sum_{s=0}^{\left\lceil \frac{e}{k^\ell} \right\rceil} \left(1 - \underbrace{\left(\prod_{b=s\cdot k^\ell}^{(s+1)\cdot k^\ell - 1} (1 - f(b)) \right)}_{0 \text{ if at least one block is used}} \right)}_{\text{all hsegments on a level}} \tag{6.2}$$

As Equation (6.1) implies, two terms contribute to the total costs. The first is the cost of hashing all data blocks and the second is the cost of building the hash tree. Hashing the data blocks will always have the complexity of $O(e)$.

[1]Blocks are numbered starting at 0

Let c be $\lceil log_k e \rceil$ and $k > 1$:

$$\begin{aligned}
\sum_{\ell=1}^{\lceil \log_k e \rceil} \left\lceil \frac{e}{k^\ell} \right\rceil &= \sum_{\ell=2}^{c} \left\lceil \frac{e}{k^\ell} \right\rceil \\
&\leq \frac{e}{k^2} + \frac{e}{k^3} + \frac{e}{k^4} + \cdots + \frac{e}{k^c} + (c-1) \\
&= \frac{e}{k^2} \cdot \frac{\left(\frac{1}{k}\right)^c - 1}{\frac{1}{k} - 1} + (c-1) \\
&= \frac{e}{k^2} \cdot \frac{\frac{1-k^c}{k^c}}{\frac{1-k}{k}} + (c-1) \\
&= \frac{e}{k^2} \frac{k \cdot (k^c - 1)}{(k-1) \cdot k^c} + (c-1) \\
&= e \cdot \frac{k^c - 1}{k^{c+2} - k^{c+1}} + (c-1)
\end{aligned}$$

$$\lim_{e \to \infty} \left(e \cdot \frac{k^c - 1}{k^{c+2} - k^{c+1}} + (c-1) \right) = \lim_{e \to \infty} \left(e \cdot \frac{k^{\lceil \log_k e \rceil} - 1}{k^{\lceil \log_k e \rceil + 2} - k^{\lceil \log_k e \rceil + 1}} + (\lceil \log_k e \rceil - 1) \right) = e \tag{6.3}$$

As Equation (6.3) shows, the total costs for building the hash tree without the leaves are of complexity $O(e)$, which is expected as in the worst case if $k = 2$ (binary tree) the total number of inner nodes is $e - 1$. For a reasonably large k, the costs of building the hash tree drop significantly.

Equation (6.2) uses function $f(b)$, which returns 0 if the data block number b is free and 1 if the data block is used (allocated). The formula is divided into two halves. The first term sums up the costs for hashing data blocks and the second those for hashing all hsegments. The product is 0 if at least one data block between $[s \cdot k^\ell, (s+1) \cdot k^\ell - 1]$ is used (allocated). The inner sum sums up all those hsegments on a level ℓ that are actually needed (see product). As a result of this calculation, there will always be at least one hsegment that covers the entire file size even though the file is completely sparse.

$$\sum_{\ell=2}^{\lceil \log_k e \rceil} \sum_{s=0}^{\left\lceil \frac{e}{k^\ell} \right\rceil} \left(1 - \left(\prod_{b=s \cdot k^\ell}^{(s+1) \cdot k^\ell - 1} (1 - f(b)) \right) \right) \leq \sum_{\ell=2}^{\lceil \log_k e \rceil} \left\lceil \frac{e}{k^\ell} \right\rceil \tag{6.4}$$

Estimation (6.4) shows clearly the obvious fact that if a file is sparse the building costs are less or equal as in the dense/contiguous case.

6.2 Space Usage

The functions that describe the space usage are very similar to the build-cost functions of the hash function. This is not very surprising because the number of nodes (leaves and inner nodes of the hash tree) that have to be stored equals the number of hashes calculated. Therefore, the cost constants $C_{\text{datablock}}$ and C_{hsegment} can be substituted by S_{hash} which is the size of the hash plus some additional space for the internal data structure (pointers etc.).

In the dense/contiguous case the formula looks like

$$D_s(e) = S_{\text{hash}} \cdot \left(e + \sum_{\ell=1}^{\lceil \log_k e \rceil} \left\lceil \frac{e}{k^\ell} \right\rceil \right), \tag{6.5}$$

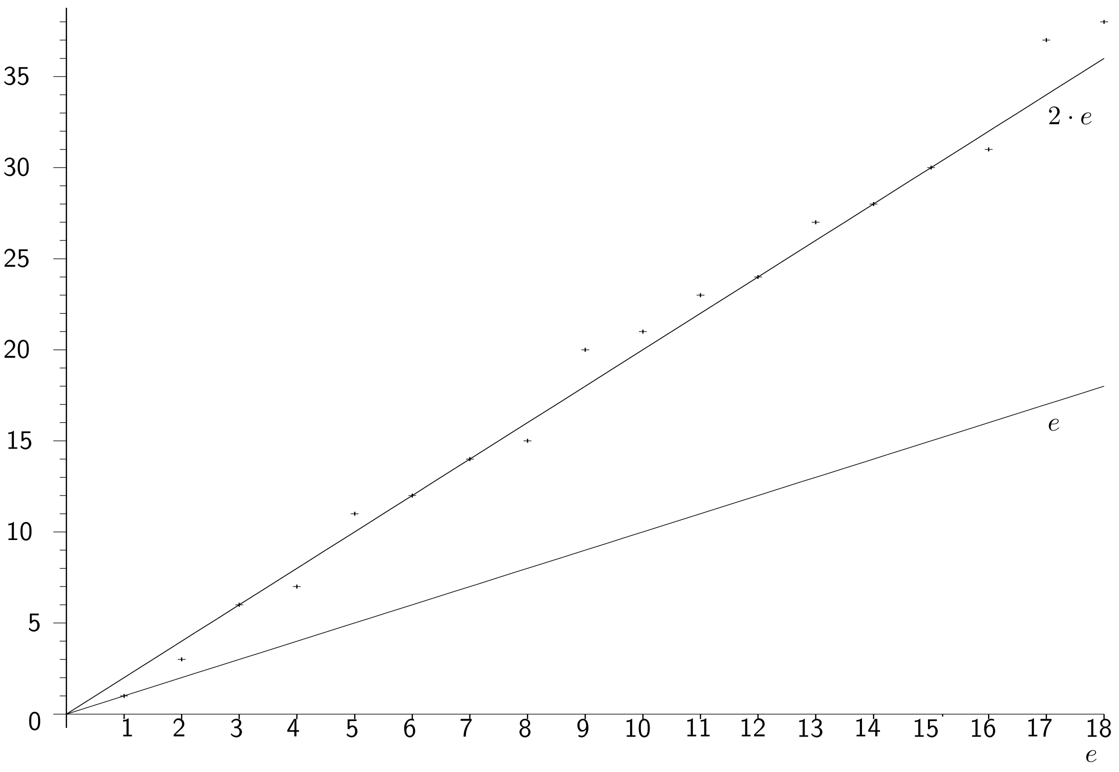

Figure 6.1: Building costs for a hash tree with the following properties: $k = 2$, $C_{\text{datablock}} = 1$, $C_{\text{hsegment}} = 1$, and $e \in [0, 18]$. For $e \neq 2^n$, where $n \in N$, the total cost can be higher than $2 \cdot e$ because the binary tree is not completely filled. The maximum derivation is reached for $e = 2^n - 1$, when to a completely filled tree another leaf is added. The tree then needs a new leaf, a new top root and $log_k e - 2$ inner nodes.

and in the sparse case the space usage is as follows

$$S_s(e, f(\cdot)) = S_{\text{hash}} \cdot \left(\sum_{b=0}^{e} f(b) + \sum_{\ell=1}^{\lceil \log_k e \rceil} \sum_{s=0}^{\lceil \frac{e}{k^\ell} \rceil} \left(1 - \left(\prod_{b=s \cdot k^\ell}^{(s+1)\cdot k^\ell - 1} (1 - f(b)) \right) \right) \right). \tag{6.6}$$

The facts stated in section 6.1 about Equations (6.1) and (6.2) also apply here.

6.3 Update Operations

Updating data blocks involves updating the hash tree. Besides rehashing the data blocks, $\lceil log_k e \rceil$ hsegments[2], and one hash root value have to be rehashed. If only a block is modified the total costs of updating the hash tree is $O(log_k e)$. If more then one block is updated, the equations for the sparse case can be used. Function f has to be modified. f is 1 if the block has been changed.

[2]Because the hash tree has a height of $\lceil log_k e \rceil$.

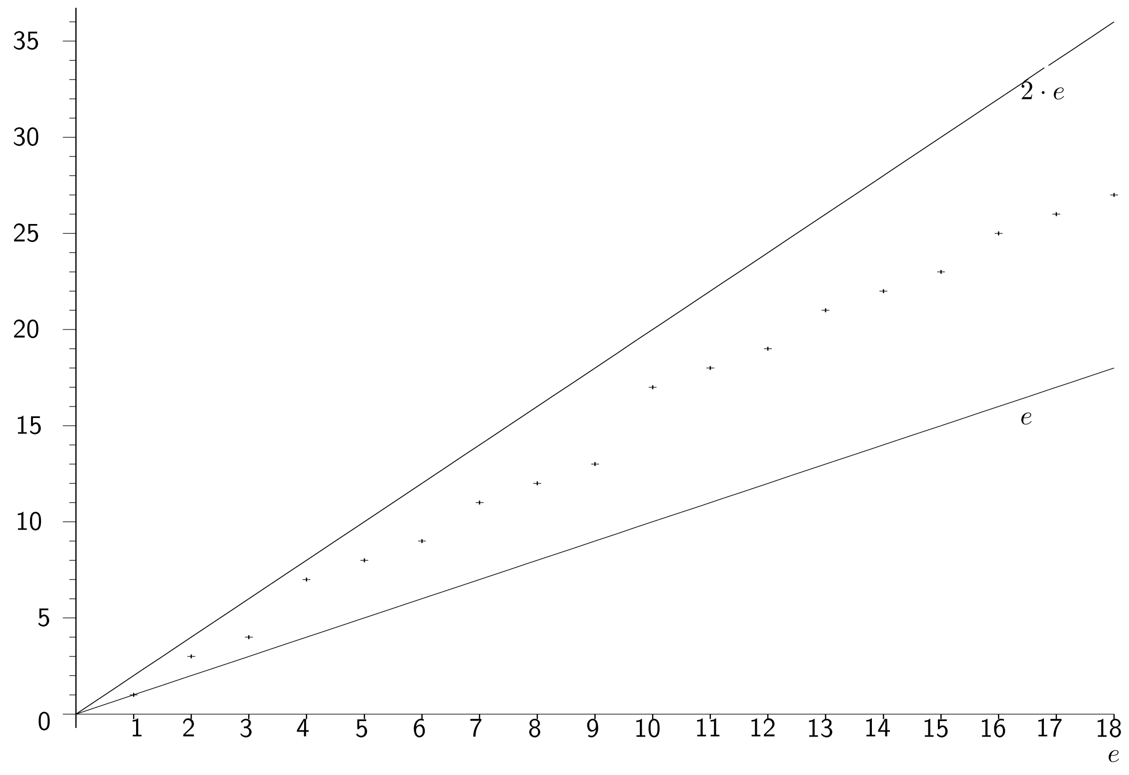

Figure 6.2: Building costs for a hash tree with the following properties: $k = 3$, $C_{\text{datablock}} = 1$, $C_{\text{hsegment}} = 1$, and $e \in [0, 18]$. For a reasonably large k, the total number of inner nodes decreases rapidly and the main costs for building the hash tree results from hashing the data blocks.

6.4 Examples

For illustration purposes, here are some examples. Let $k = 256$, $S_{\text{hash}} = 20$ bytes, and the hashing function has a throughput of 15 MB/s[3]; $C_{\text{datablock}} = C_{\text{hsegment}} = 15$ MB/s. $C_{\text{datablock}}$ and C_{hsegment} have the units MB/s because the constants represent the throughput of the hashing algorithm. The block size is 4 KB.

Files that are smaller than one segment (≤ 1 MB)

For a normal user installation, the scenario should be quite accurate.

A small survey over a small set of workstations (8 Unix installations, Red Hat 7.3, 2 × Red Hat 9, 2 × Debian 3.0, Mac OS X 10.3, 2 × SunOS 5.9) has shown that the average file size is significantly smaller than 1 MB. 2.04 million files had a total size of 143.5 GB, which is roughly 70 KB on average. Note that the provided numbers are only for

Files with a size of 6 MB

This scenario should be applicable to systems that have audio files (songs) (in audio-codec-like format MP3, OGG Vorbis, ACC, etc.) stored.

Files with a size of 700 MB

For systems with slightly larger files, assuming CD images are stored, this scenario should be applied.

[3]A Pentium III 900 MHz computer achieves approximately 15 MB/s throughput with SHA-1. This was measured by the author.

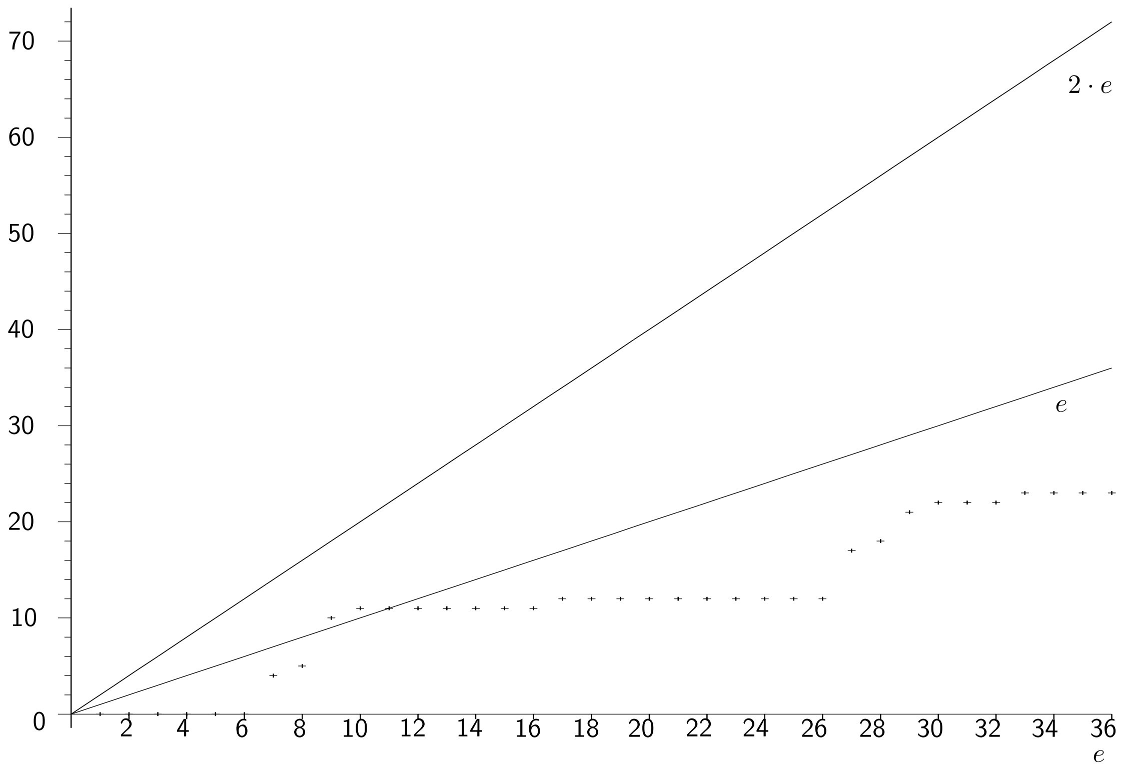

Figure 6.3: Building costs for a sparse hash tree with the following properties: $k = 2$, $C_{\text{datablock}} = 1$, $C_{\text{hsegment}} = 1$ and $e \in [0, 36]$. $f(b) = \begin{cases} 1, & \text{if } (5 < b < 10 \text{ or } 25 < b < 30) \text{ and } b < e \\ 0, & \text{otherwise.} \end{cases}$

Files with a size of 2 TB

In the area of science, especially in physics, large files are quite common.

Largest possible file: 18 EB

Finally this scenario shows the “worst” case. Because of the data structures used in strategy #1, the maximum size of each LU used by the SAN File System is 2^{44} bytes or 16 TB when using a block size of 4 KB, and up to 2^{51} bytes or 1 PB when using 256 KB-sized blocks. But it still is interesting to see the numbers of the theoretical limit.

The results of the calculation can be found in Table 6.1.

6.5 Implementation

6.5.1 Unmarshalling

The layouts of the `BlkDiskHashDescr` descriptor (Section 4.1.5) and the `PackedHSTE` (Section 5.1.2) are basically the same. Only the headers differ slightly. This allows a very efficient unmarshalling of hash descriptors: nothing at all has to be done. Furthermore, only one copy operation is necessary when the data from the message has to be copied into the database or vice-versa. Unfortunately, the current implementation does not take advantage of it because the code is still in very early development stages.

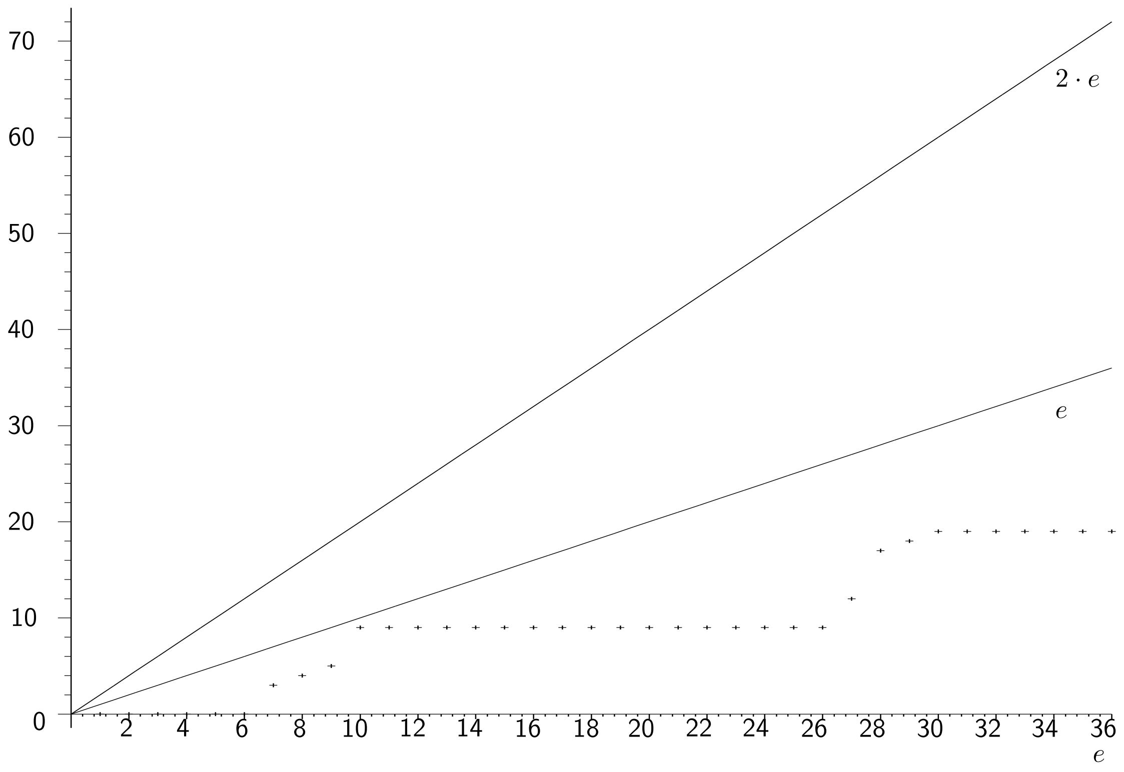

Figure 6.4: Building costs for a sparse hash tree with the following properties: $k = 3$, $C_{\text{datablock}} = 1$, $C_{\text{hsegment}} = 1$ and $e \in [0, 36]$. $f(b) = \begin{cases} 1, & \text{if } (5 < b < 10 \text{ or } 25 < b < 30) \text{ and } b < e \\ 0, & \text{otherwise.} \end{cases}$

6.5.2 Inlining

Inlining is an optimization to minimize the number of database accesses. As described in Section 5.1.6 the restricting factor for storing data inline in the file-object attributes is the page size of the database. There are two criteria which every record has to meet:

- At least two keys must fit into a page.
- A key and its corresponding data (record) must fit completely into a leaf. The size of the leaf is the size of a page.

The current implementation of the database uses a net page size of 3960 bytes. The size of a key is 16 bytes, which means a record cannot be larger than 3944 bytes. The largest packed STE is 3264 bytes, which clearly fits into a page.

However, with the additional hashes (20 bytes for SHA-1) the page is too small to hold both the packed STE and the packed HSTE ($255 + 256 \cdot 20$ bytes $= 5375$).

Of course, these are the worst cases—256 extents needed for 256 blocks and a hash for each block. The anticipated case is that there will a small number of extents. Moreover, if not all blocks are used the total size of all meta-data should still fit into the database, to benefit from of the performance gain of having to access the database only once. The additional size checks cost much less than an database access.

Table 6.1: The results show that for small files sizes (≤ 6 MB) the build and store costs are reasonable small.

File size [bytes]	1,048,576	6,291,456	734,003,200	2,199,023,255,552
#Data blocks	256	1,536	179,200	536,870,912
#Hashes	257	1,543	179,904	538,976,289
Space Usage [bytes]	5,140	30,860	3,598,080	10,779,525,780
% of the data	0.49	0.49	0.49	0.49
Build Time [s]	0.067	0.400	46.668	139,812.810

File size [bytes]	18,446,744,073,709,551,616
#Data blocks	4,503,599,627,370,496
#Hashes	4,521,260,802,000,000
Space Usage [bytes]	90,425,216,040,000,000
% of the data	0.49
Build Time [s]	139,812.810

6.5.3 Hash Size, DB Page Size, Network Message Size

For performance reasons, a SAN File System network message is limited to a constant size of 8 KB. This also means that the maximum size per hash is limited. Whereas SHA-1 hashes (20 bytes) still fit into one message, SHA-256 will exceed the limits. Increasing the network message limit to 16 KB solves this problem. Using SHA-256 (32 bytes image size) for hashing should provide sufficient security for the foreseeable future.

The database page size limit is also exceeded when all 256 hashes have to be stored even for SHA-1. Possible workarounds would be to trim hashes, or to introduce more key types and split the hashes into to parts, e.g. `HSTE_L0_0` for the first part of the hashes and `HSTE_L0_1` for the second part of the hashes of a segment. Of course, this is not very elegant, because for SHA-256 we would have to split a hash segment into three parts and introduce new types again.

A more elegant solution is to split the message transparently into the number of necessary blocks using another abstraction layer between the database interface and the STEs. The abstraction layer would split up an oversized record into smaller records and tag them accordingly their relations. During retrieval the abstraction layer would glue the smaller record together to an oversized record. The advantage is that existing code has not to be touched and no special handling is needed for any record size. But the disadvantage is that most likely overhead is added to the normal records which do fit into a single page.

7 Conclusions

The extension to the existing SAN File System protocol fulfills the requirements of the expected goals: It does not change the existing protocol. It merely is an add-on extension with three new message types for integrity protection and two more for transmitting security information such as the digest algorithm.

One design goal was to keep the workload for the server as small as possible. Because all hash-tree calculations and integrity checks are done on the client side, the server is only used as a secure storage space. This additional workload is rather small. The longest operation on the server is accessing the database, which normally involves a disk access over a SAN. Because the server is heavily multi-threaded, only more threads (actions executing threads) are required, which then will wait until the database operations have been completed.

The entire hash-tree building and compare logic is on the client side. This reduces the network access substantially. The client can for example decide on its own whether the current hash-block size for a segment is too small and can optimize if necessary. As long as a client holds an X-mode data lock, no communication whatsoever is needed. Only before the data lock is released do the modified segments have to be transferred.

The SAN File System provides a set of data locks that allows efficient caching of file-system objects. Among more established locks such as SR or X-mode locks, the shared write lock (SW-mode) is more unusual. Its purpose is to allow Direct I/O operations, e. g. writing data to the storage devices without caching. In the SAN File System protocol specification it is clearly written that meta-data is cached as read-only when a client accesses a file-object holding a SW-mode lock. But the proposed extension allows modification on the meta-data while the SW-mode lock is held. This introduces a new kind of problem: Concurrent updates of the meta-data which could potentially lead to a race condition. Several solutions were provided to this problem.

Unfortunately, the current SAN File System protocol limits the maximum size of the hashes because the longest network message is limited to 8 KB. Digest algorithms such as SHA-1 will work, but not SHA-256. A simple increase of the this limit to 16 KB solves this problem. Of course, this could introduce some performance penalty on the client and the server side as well as entail significantly heavier temporary buffer usage.

Speaking of limits, the building of complete hash trees for small files (in the MB range) does not take much time and space. But for larger files (in the GB range), the time needed to build a complete hash tree is noticeable. For huge files (in the TB range), the capacity needed for building the trees can be excessive. Although with sufficient hardware support, larger hash blocks aggregations it could still be possible to handle this large amount of data. One should not forget that the real data also has to be transfered to the client. Of course, it is possible to disable integrity protection for block ranges. If integration protection is disabled then the costs for hashing the data blocks can be committed, but not those for the inner nodes.

It has been shown that the hash tree is an adequate data structure for either dense files or sparse files. The size and access costs are logarithmic to the base k, which normally is 256 (block disk strategy #1).

A client can create and modify hash trees, therefore one might think that the client can modify data without authorization. This is not a problem, because as long as a client is authorized to access the meta-data server, the server must approve all operations. The aim of integrity protection is to detect modifications that were done without involvement of the meta-data server. This can clearly be detected by the proposed protocol.

On the practical part of the work, the server was extended and a client simulator written. Because of limited time and resources as well as the utter underestimation of the project size, it was not possible to implement the client as well. Therefore, no test results and benchmarks can be shown. The proposed further work clearly is to complete the implementation and to measure the performance.

Note that this extension is not an IBM product.

A Extended Security Attribute Variation

Section 4.1.6 describes how security attributes can be encoded in the SAN File System Protocol. This appendix provides another solution. This solution was developed first but then replaced by the simpler approach mentioned in Section 4.1.6. The solution proposed here provides a flexibility which is not needed for the implementation of integrity protection.

A.1 `ExtendedSecurityAttributes`

Common header:

1 – 2	`Uint16`	Class/type of attribute. For most attributes, this is also the index in the `List`.
3 – 4	`Uint16`	Mode/algorithm used (0→none; > 0 defined by individual algorithm).
5 – 6	`Uint16`	Criticality flags for unsupported attributes.
7 –		Parameter-specific values, see below.

The criticality flag is a bit field with the following values. Each bit set indicates that not understanding this security attribute prevents one from performing the operation in question.

0x00 01	Prevents data reads unless understood
0x00 02	Prevents data writes unless understood
0x00 04	Prevents entering SW mode unless understood
0x00 10	Prevents entering this directory unless understood
0x01 00	Prevents updating Unix ACL unless understood
0x02 00	Prevents updating Windows ACL unless understood
0x04 00	Prevents security attribute updates unless understood
0x10 00	Allow ACL updates, but mark this attribute as invalid
0x20 00	Allow data writes, but mark this attribute as invalid
0x40 00	Allow security attribute updates, but mark this attribute as invalid
0x80 00	This security attribute has been marked invalid by a non-conforming client

ESA[0] is EncryptionAttributes, ESA[1] is IntegrityAttributes, ESA[2] is SignatureAttributes; they all follow the common header.

A.2 AESEncryptionAttribute

1 – 2	`Uint16`	Class=0
3 – 4	`Uint16`	Algorithm=1
5 – 6	`Uint16`	Criticality=0x00 01
7 – 8	`Uint16`	Key length (in bytes)
9 – *s*	`Key`	Key data

A.3 SHA1IntegrityAttribute

1 – 2	`Uint16`	Class=1
3 – 4	`Uint16`	Algorithm=1
5 – 6	`Uint16`	Criticality=0x00 02 (or, for permissive mode, 0x20 00)

A.4 `MiscAttr` extensions

0x00 04 00 00	Enhanced security attributes present (deny access to clients that do not support them, as identified during the `Identify` [IBM03, Section 6.1.1] phase)

B Project Plan

Week	Begin	Goal	Achievement
1	Nov. 3	Getting an overview of the topic	Done
2	Nov. 10	Project plan General design decisions Document the design decisions Identify interesting spots in the source code	Done Done Done
3	Nov. 17	Define data structures Define functions or identify functions that need to be changed Get in touch with the environment (sysadmin view and compiler environment) → Milestone	Data structures defined. SAN.FS protocol extended (new message types introduced). Milestone reached.
4	Nov. 24	Start to implement (SW-NP mode) Hash tree Meta-data server	Started to implement server message marshaling
5	Dec. 1	Client	Still implementing server message marshaling
6	Dec. 8	Ditto	Ditto (sic!)
7	Dec. 15	Test and review (maybe change) implemented code → Milestone.	Fixing compiling environment and syntax errors fixed Started to work on documentation
8	Dec. 22	Write mid-thesis report *Christmas*	Gearing up debugging environment Finished writing chapter Introduction to SAN.FS
9	Dec. 29	*New Year's Eve*	Writing documentation
10	Jan. 5	Implement special SW cases and test them Start working on dynamic hash block size	Writing mid-thesis report Got server started
11	Jan. 12	Ditto	Got debugger working on server Started to debug
12	Jan. 19	Benchmarks	Setup of a dedicated new platform Fixing obvious bugs
13	Jan. 26	Use insights from the benchmarks to optimize behavior	Added mixing code to client simulator and server Created clean patch of all work done
14	Feb. 2	Ditto Mini Benchmark	Writing documentation

15	Feb. 9	Write documentation Talk preparation	Writing more documentation
16	Feb. 16	Ditto	Ditto
17	Feb. 23	Reserve	Corrected documentation Cleansed Patch Talk prepared
18	Mar. 1		Talk given End of thesis.

C Abbreviations

SAN
: Storage Area Network

LU
: (SCSI) Logic Units

SCSI
: Small Computer System Interface

PIT
: Point In Time

hsegment
: Hash Segment.

segment
: Virtual block address ranges on a file

extent
: Physical block ranges on a SCSI LU

pre-image
: A variable-length input string for a hash function ([Sch96, Chapter 2.4])

wormholes
: Attachment points of other filesets

FlashCopyImage
: Multiple version of a file object but all immutable

Bibliography

[ADF$^+$03] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *Proceedings of the Conference on Mass Storage Systems and Technologies (MSS'03)*, pages 165–177, 2003.

[AJL$^+$03] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Mike Burrows, Timothy Mann, and Chandramohan Thekkath. Strong security for network-attached storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, January 2003.

[CCDP01] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[CIF] Common Internet File System (CIFS) Technical Reference. `http://www.snia.org/tech_activities/CIFS`.

[EJ01] Donald E. Eastlake and Paul E. Jones. SHA-1: US Secure Hash Algorithm 1. Technical Report 3174, September 2001.

[Hal99] Andrea Halter. PGP File System mit flexiblem Gruppenmanagement. Diploma thesis, TIK, ETH Zürich, February 1999.

[HR03] Shai Halevi and Phil Rogaway. Tweakable enciphering modes for sector-level encryption (extended abstract). In *Advances in Cryptology - Crypto'03*, volume 2729 of *LNCS*. Springer-Verlag, Berlin Heidelberg, 2003.

[IBM03] IBM. IBM TotalStorage SAN File System Draft Protocol Specification, April 2003.

[Mer80] Ralph C. Merkle. Protocols for public key cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[MFLR02] Ethan L. Miller, William E. Freeman, Darrell D. E. Long, and Benjamin C. Reed. Strong security for network-attached storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, January 2002.

[Mic00] Silvio Micali. CS proofs. *Siam Journal on Computing*, 30(4):1253–1298, 2000.

[MKKW99] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the Symposium on Operating Systems Principles*, pages 124–139, 1999.

[Now89] Bill Nowicki. NFS: Network File System protocol specification. Technical Report 1094, March 1989.

[Riv92] Ronald L. Rivest. The MD5 message digest algorithm. Internet RFC 1321, April 1992.

[Sch96] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, New York, second edition, 1996.