RZ 3580 (# 99590) 02/07/2005
Computer Science 81 pages

Research Report

Efficient and Safe Networked Storage Protocols

Marc Kramis

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master Thesis WS 2004/05

Efficient and Safe Networked Storage Protocols

Marc Kramis

Abstract

After more than two decades evolving a variety of client/server-based distributed file systems (DFS), the recently emerging storage area networks (SAN) allow the former file-server to be split into a storage and a metadata component. Metadata servers perform file access coordination and metadata management, whereas storage devices directly serve the clients' read and write requests. The clear separation of duties, the straight data path, and the virtualization of storage result in better scalability, performance, and maintainability. Whereas the first generation of SAN-based DFS focused primarily on performance, the second generation aims at spreading its service to the organizations' desktops, where server-room trust-levels can no longer be presumed.
We identified a set of security threats that arise when SAN File System protocols for client to metadata server communication are opened to the insecure desktops. This work discusses and analyzes design modifications for client authentication, protocol encryption, and distributed lock recovery, which we partly implemented on a Linux-based SAN.FS environment. In addition, we cover quota management as well as lock scheduling, and introduce virtual machines as a valuable tool to support research within distributed storage systems.

Acknowledgements

I am grateful to Marcel Waldvogel who supervised this thesis first at IBM Research, then as professor from the University of Konstanz. He provided plenty of valuable input and was a much valued contact person.

Furthermore, I would like to say thank you to Patrick Droz, Roman Pletka, Robert Haas, Christian Cachin, and Charlotte Bolliger who supported me in the course of this thesis with their ideas, helpful suggestions and simply their time.

Finally, I am deeply grateful to my parents who supported me throughout my studies with patience and appreciation.

This thesis was carried out at

IBM Research GmbH, Zurich Research Laboratory

Contents

List of Figures

List of Tables

1 Overview

1.1 Introduction

The ever growing demand for storage capacity, unified and ubiquitous data access, as well as the requirement to administrate the storage conveniently if not autonomously, ask for the continuous development of distributed file systems (DFS[1]).

After more than two decades evolving a variety of client/server-based DFS, the recently emerging storage area networks (SAN) allow to split the former file-server into a storage and a metadata component (see Figure 1.1 on Page 2). Metadata servers perform file access coordination and metadata management, while storage devices directly serve the clients' read and write requests. The clear separation of duties, the straight data path, and the virtualization of storage result in better scalability, performance, and maintainability.

The first generation of SAN-based DFS brought up a sophisticated protocol, which relies on the absolute cooperation of all involved parties. Trust as a vital prerequisite however is highly questionable if clients outside server rooms under administrator control access the DFS over insecure networks.

The upcoming second generation of SAN-based DFS aims at spreading its service to the organizations' insecure desktops. Current research tries to achieve the necessary security following a two-pronged approach: First, the data path between clients and storage devices is secured. Second, the file access coordination and metadata services are hardened against malicious attacks. This work concentrates on the latter effort.

1.2 Task Description

The existing SAN.FS protocol has to be analyzed with the premise of potentially malicious clients. The focus lies on the client to metadata-server protocol, which supports the coordination of distributed file system access. Found issues have to be documented and suggestions to their elimination given. The impact of the suggested improvements on performance as well as other security issues should be covered.

In addition to the theoretical study and observation, a prototype implementation

[1]Please note that the acronym DFS used throughout this report does not have anything to do with the Microsoft Distributed File Service.

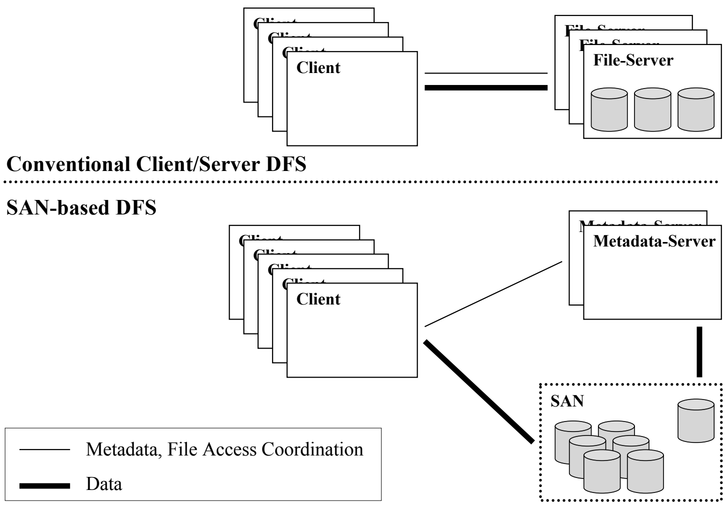

Client
File-Server
Conventional Client/Server DFS
SAN-based DFS
Client
Metadata-Server
SAN
Metadata, File Access Coordination
Data

Figure 1.1: *The figure shows the evolution from conventional client/server-based DFS to SAN-based DFS.*

based on the IBM SAN.FS is expected.

The complete task description is covered in Appendix A on Page 63.

1.3 Chapter Overview

This section summarizes the contents of each chapter.

Chapter 2, Distributed File Systems An explorative non-exhaustive journey including six different DFS to get an insight in commonly used techniques and critical areas.

Chapter 3, SAN.FS Protocol Give an introduction to the IBM SAN.FS protocol specification. This chapter provides the foundation to analyze the protocol with respect to security and performance.

Chapter 4, Security Issues Introduce several aspects of security and try to find issues from natural failure conditions as well as malicious operation.

Chapter 5, Design Modifications Suggest design modifications to a selection of the found issues and discuss the impact of the required changes related to security and performance.

Chapter 6, Virtual Machines Introduce virtual machines as a valuable tool for the research within distributed storage systems.

Chapter 7, Conclusions A final conclusion summarizes the results and motivates future work in this area.

2 Distributed File Systems

2.1 Overview

Over the last twenty years, there has been a lot of research activity around distributed file systems (DFS). This chapter tries to give a non-exhaustive explorative overview over different DFS to get an insight in commonly used techniques and critical areas. Among others, two well established distributed file systems being around in production environments for decades will be briefly discussed. The first one is the Network File System (NFS, [S$^+$03]) from Sun Microsystems, the second the Andrew File System (AFS, [Zay91]) originally developed at Carnegie Mellon University. Then a research prototype extending AFS known as Coda [SKK$^+$90] will be looked at, followed by a DFS called Echo from Digital Systems Research Center [MBH93]. A closely related distributed computing system will also be covered with Subversion [CSFP04], a version control system being the successor of the widely used Concurrent Versions System (CVS, [C$^+$04]). From the family of cluster file systems, Lustre [Bra03] will be briefly looked at. Finally, a conclusion will close this chapter.

There are many other DFS around, closely related or derived from one of the discussed systems. Many of them are at the level of research prototypes, trying to improve certain aspects as well as introduce and test new ideas. Many are commercially available but often not publicly documented. But the main concepts still remain the same and cannot contribute substantially to this overview. Nevertheless, there are some examples: Prominent descendants of NFS are Not Quite NFS (NQNFS, [Mac94]), introducing the concept of leases, and Spritely NFS developed at Berkeley [SM89] adding open and close calls to NFS. Other DFS try to overcome the limitations of centralized servers such as xFS with its roots at Berkeley [ADN$^+$96] or Frangipani from Digital Systems Research Center [TML97] or are tailored to work with cluster environments like the General Parallel File System (GPFS, [SH02]) from IBM. Last but not least, there are some heavily used DFS such as Server Message Block (SMB, [Mic04]) currently maintained by Microsoft who does not provide a complete public protocol specification or the recent Common Internet File System (CIFS, [SNI02]).

Note that since SAN.FS needs further investigation, a separate chapter is fully devoted to IBM's latest DFS [IBM03].

2.2 The Ideal Distributed File System

DFS allow to share data across multiple nodes in a network. [CDK01] gives an overview of the requirements such systems should meet (adapted):

Transparency with respect to access, location, mobility, performance and scalability.

Concurrent File Updates should not interfere with each other.

File Replication (or caching) for better scalability and fault tolerance.

Hardware and Operating System Heterogeneity to be highly interoperable and adapt to real-world scenarios.

Fault Tolerance to overcome transient communication or node failures.

Consistency to provide one-copy update semantics. [1]

Security for access control and integrity.

Efficiency for performance even under heavy load or with low bandwidth.

Unfortunately, any DFS has to overcome certain problems arising from its very nature of being distributed. E.g., nodes and their interconnecting networks cannot be trusted under all circumstances and they can become disconnected virtually at any time or moved to different places. Bandwidth and latency are often not guaranteed and can vary widely. Together with the idealistic requirements, this results in an inherent complexity of the protocols involved and also of implementing such a system. To alleviate this, many different systems have been developed, each tailored for specific use cases and environments.

All these different systems however share the distinction between data and metadata. The raw content of a file corresponds to the data, whereas all additional information such as the location the file is stored at (directory) or the type of the file and all assigned attributes belong to the metadata. The data itself is organized as a set of blocks.

2.3 NFS — Network File System

Having been launched in the late 1980's, today, Sun Microsystems's Network File System is a widely used DFS in academic and industrial environments where read access to files is far more frequent than write access and write access is well distributed and not concentrated on hot spots. [2] A high speed intranet is assumed to play the physical network layer. Early versions suffered from poor write performance as all blocks had to be written to disk synchronously. This restriction was loosened first with the help of additional hardware (battery-backed-up RAM), later with protocol modifications.

[1] File content seen by all processes reading or writing to it are as if there would be only a single copy of it.

[2] Specific files being frequently accessed for either reading or writing.

NFS [CDK01, S+03] is based on the following main concepts:

Client/Server The server is stateless. Note that some server implementations might keep state information to improve performance.

Remote Procedure Call (RPC) RPCs handle the client to server communication.

The stateless nature of the servers requires clients to identify themselves on each request ("Secure RPC"). Stateless operation on the other hand simplifies failure recovery and implementation. The following optimizations are found in NFS:

Server-Side Caching The server can either operate its cache in synchronous mode using write-through caching (the only option available in NFS 2) or in write-gathering mode where writes are bundled and written whenever convenient, or latest as a result of a client commit request.

Client-Side Caching To provide faster data access and reduce the load on the server and network each client maintains a cache. Consistency is ensured by polling the server on a regular basis for stale information in cached data or metadata. Writes from the client are either propagated to the server synchronously on each write or asynchronously whenever a file is closed or a sync operation is performed (write-behind). Note that both relaxations deviate from the strict UNIX one-copy file update semantics.

Piggy Packing To reduce network traffic, information about file attributes is sent to the client piggy packed on each file or directory operation request.

Read-Ahead and Write-Behind try to leverage the fact that most clients read or write consecutive data sequentially. The sequence is thus parallelized using several concurrent reader or writer threads.

Even though — or better to say because of the comparative simplicity and efficiency of NFS there are some issues to be discussed:

Locking is not natively supported and has to be achieved with separate protocols in NFS versions 2 and 3.

Global Name Space is not supported. It is not possible to have multiple servers to serve one common name space.

Hot Spots Files being frequently accessed quickly bring the servers to their performance limits because of the number of RPCs performed by clients to maintain their caches. Processing capacity turned out to be the limiting factor.

Consistency is not guaranteed under all circumstances because one-copy semantics is only approximated. Files shared via NFS should not be used for communication or close coordination between processes

POSIX Compliance There are several issues with POSIX semantics, e.g., the last close problem. [3]

[3]Delete an unlinked file after the last close.

Temporary Files also require interaction with the server. This could be optimized since temporary files are heavily used and mostly exist only for a short time on behalf of local applications [ODH+85].

Replication NFS only supports the replication of files in read-only mode. While this improves the availability of frequently accessed files such as system libraries that are not updated by clients, it does not help with files that actually are updated by the clients.

Special attention is paid to the NFS Lock Manager Protocol. This is a summary of the description found in [Cal00]. Some interesting notes about this locking protocol:

Loss of Server State It must be assured that clients can safely reestablish locks after server failures. The mechanism implemented uses a *grace period* and works as follows: after a failure, the server restarts and enters a grace period to allow clients to reclaim locks they hold before the crash. No new locks can be acquired during this time. Another name for this mechanism is *distributed lock recovery.*

Loss of Client State In case of a client failure, the server keeps the locks until the client explicitly reclaims them. To do so, it announces itself to the server after a restart with an increased client state number. Locks with a lower number are then released. Note that this can lead to wasted server resources and stale locks when the client is permanently down or removed.

No Caching for Locked Files to prevent data corruption or even deadlocks due to incompatibilities of the locks with the granularity of the paging system (blocks belonging to different locks could be placed on the same memory page).

Lock Types Locks are advisory[4] and can either be *exclusive* for write access or *shared* for multiple concurrent readers.

Deadlock Prevention Deadlocks are detected by checking dependency graphs of lock requests for cycles.

2.4 AFS — Andrew File System

AFS was originally designed to support the computing needs of Carnegie Mellon University with its 10,000 people which could not be met by existing DFS such as the NFS. To take it even further, AFS should no longer be constrained to local area networks (LAN) but also be of practical use in wide area networks (WAN) where latency is considerable and bandwidth scarce. AFS is based on the experiences made with NFS.

The design goals are discussed in detail in [Zay91] and the most important differences compared to NFS are found in the area of *scalability* and *name space.* AFS should scale better than NFS by an order of magnitude. Especially the clients per server rate should be substantially increased. AFS also provides a single name

[4]Non-cooperative clients can still read and write the data ignoring the locks. In contrast to advisory locks, mandatory locks enforce their semantics under any circumstance.

space to all users, which greatly simplifies collaboration and allows for full location transparency. To meet the ambitious goals, client-side caching was redesigned to support persistent *whole-file caching* together with *whole-file serving.*

To reduce client to server communication for caching purposes, open-to-close session semantics with *callbacks* have been introduced. Instead of the clients polling the server regularly even when nothing has changed, the server now directly notifies the clients if a file was updated. The downside is that servers now have to remember the clients to cache files persistently. If no callbacks are registered with the client for a defined period, the client checks whether the server is still alive to prevent losing synchronization caused by network partitions.

The callback mechanism allows local processes on one client to operate with one-copy update semantics. For distributed clients, the contract is, that only the update from the last close will be permanent and the other updates are lost.

2.5 Coda

Coda is a research prototype from the Carnegie Mellon University [SKK+90]. Being inspired by AFS from where it inherits many aspects (e.g., the global name space), it tries to overcome some of AFS's annoyances resulting from its large installation base with an increased probability of network and node failures. Coda should smoothly adjust to these numerous small failures and be more resilient. Two complementary mechanisms are provided to achieve this:

Server Replication for continuous read and write access even in case of a server failure.

Disconnected Operation to adapt for mobile computing and client or network failures.

From a production point of view, it has to be said that some aspects of Coda like reconciliation of conflicts or the replication algorithm seem to need more precise definitions and research.

Coda offers an approximation to the AFS consistency model. In case of no failure, processes sharing a file at a single node see exact UNIX semantics. Processes at different nodes see modifications at the granularity of the file open and close operations. In case of a failure, Coda prefers availability, which could lead to conflicting updates and therefore corrupted data over consistency. Coda assumes that this would rarely happen and require immediate detection and reparation. The Coda papers however remain quite obscure about the exact process of repairing such a conflict.

As with AFS, Coda uses the notion of a volume as the unit of replication. The replication strategy itself follows a *read-one, write all available* pattern which is a very good choice if reads occur more often than writes in the sense of scalability and communication overhead as recent research states [JPPMAK03]. The client initializes any action on behalf of replication. On writes, the client concurrently sends data to all available servers which leads to an overhead increasing with the amount of servers involved. On reads, the client compares the versions of all available servers. If one server is out of date, the clients kicks off a synchronization phase which is completed between the servers without any further client interaction. The client then fetches the data from the preferred server. This kind of lazy replication allows for offline servers and tries to minimize network utilization. It would be interesting to investigate how reliable this is and how big the probability is that a client gets stale data because the only server with the latest updates currently is down.

Disconnected operation is a trade-off between fully synchronized operation (client to server) and complete autonomy of the clients. Whenever a client (un)intentionally is disconnected from all servers, it fully relies on cached data which can be modified in any way. Reconnected again, the updates are propagated to the servers. During disconnected operation, the access to an uncached file will cause an unresolvable error. To prevent this, *hoarding* is used to prefetch files in connected operation on a simple empirical analysis of the user's behavior. As already stated, conflicts might arise on reconnection that must be resolved in some way.

2.6 Echo

The Echo DFS from the former Digital Systems Research Center [MBH93] has four major design goals:

Replication Servers and disks are replicated for availability.

Caching Clients extensively cache data and metadata for performance.

Global Naming Echo supports a globally scalable, hierarchical name space.

Distributed Security Clients are not trusted and authenticate themselves on behalf of the users logged in to them.

Echo tries to achieve full UNIX single-copy semantics. In contrast to NFS or other DFS, many UNIX file system details such as maintaining unlinked files as long as the file is still open with any process or guarantees to have enough storage capacity when flushing the cache are cared about. Shared read and exclusive write locks are applied on top of leases to prevent inconsistent caches even when network partitions occur. Caching is transparent as long as no network failures occur. Being disconnected from the network, the client blocks for at most two minutes to wait for automatic recovery and afterwards forwards the error to the applications. As a consequence of its highly synchronous design, Echo only works smoothly if network outages are rare.

Whenever one or more processes issue read or write operations, these must be scheduled to keep data structures consistent even when some write operation fails or data is lost during a crash. A similar technique for local file systems called *soft updates* is covered in [GMSP00].

Write-behind caching allows to bundle several write operations in a cache and commit them regularly or on request to the servers. This improves write performance significantly. The drawback of this approach is the difficulty of notifying applications or users about lost writes that are not made persistent due to client crashes or lease expiry in coincidence with server failures. Echo could not find an appropriate way to handle these situations. Operation ordering can only ameliorate but not eliminate this. Note that local UNIX file systems also suffer from this in case of crashes.

A lease protects cache consistency during network failures because it only depends on relative times on the server and the client assuming matching clock speeds. But the lease must be protected in case of a server failure. One option would be to ask clients after a server crash to reclaim their leases as done in Spritely NFS [SM89]; it is similar to distributed lock recovery. This performs well if no failures occur because the leases must not be made persistent during normal operation. But it is time consuming and risky after a crash. Malicious clients could make false claims. Echo chose to persist the leases and replicate them on the server side to provide faster and more reliable recovery.

2.7 Subversion

Subversion is a special kind of DFS used to manage files and directories over time with an emphasis on data and metadata versioning. It also distinguishes itself from others in that it does not completely hide its existence and operation from the user who has to issue proprietary commands to use it. Its preliminary use is for concurrent software development in LAN or WAN environments.

As a successor of the widely used Concurrent Versions System (CVS, [C$^+$04]), it adds several valuable improvements [CSFP04]:

Directory Versioning Not only files but also directory information is fully versioned.

True Version History also for copy, move, or delete commands.

Atomic Commits Either commit everything or nothing (important for network outages or node failures).

Versioned Metadata All attributes and the file location are also versioned.

Consistent Data Handling for text and binary data.

Efficient Branching and Tagging using a lazy-copy approach described below.

Hackability which means maintainability and easy integration.

Subversion is designed to be a client/server system where several clients concurrently access one or more central repositories where the versioned data and metadata is stored. Replication of the server-side repositories is not yet implemented.

The basic paradigm for distributed collaboration found in Subversion is the *copy-modify-merge* solution.[5] This is an optimistic approach where concurrent write

[5]Copy-modify-merge conceptually is a subset of operation shipping as described in [Lee00].

access to data is not prevented but resolved in case of a conflict. The choice for this solution was taken with the development process in mind. Two developers rarely work on exactly the same part of a source file, so it is perfectly reasonable to urge them to talk together and merge their work in such a case. Experience showed that this mechanism is satisfactory.

Copy-modify-merge has two important implications. First, it allows the client for asynchronous and therefore intermittently-connected operation on the shared data. This simplifies and accelerates the process for all involved parties at the cost of more intensive human interaction. Second, it offloads the server from maintaining each client's state and from coordination efforts among the concurrently running clients.

Several optimizations exist to increase performance by eliminating or reducing I/O operations:

Pristine Copy To avoid connecting to the server for every minor operation, the client keeps an untouched copy of the currently used files.

Lazy Copy Whenever a copy operation occurs on the server, the new files are created as hard links until the files change. This will trigger the actual copy operation and is the reason why branching and tagging are efficient operations in Subversion. Copy-on-write (COW) is another name for this technique.

Differential Copy Whenever data has to be moved from client to server or vice versa, only the differences are transmitted to reduce network bandwidth usage.

2.8 Lustre

The Lustre Linux Cluster file system from Cluster Filesystems, Inc. [Bra03] has its roots in Coda but also integrates features like leases or strict UNIX semantics from many other DFS that have already been described. Its strong requirement for availability and scalability led to a new architecture breaking up with the traditional client/server paradigm. While clients still play their traditional role as service requesters, the server infrastructure has been split into two parts, each requiring separate protocols for communication with the clients or among each other:

Metadata Servers (MDS) MDS handle client requests regarding metadata information such as directory information. MDS can be replicated for availability. Any communication from client to MDS is transactional and journaled.

Object Storage Targets (OST) File I/O from disks to clients, striping, and security enforcement is handled by the OST. An OST performs the block allocation for data objects[6] which is an important part of the Lustre architecture.

The advantage of this separation is the ability to replicate and scale out each part as required without touching other system components. There is no centralized place anymore that can become a bottleneck of the whole system. Lustre claims

[6]Lustre uses the notion of an object for a file which in turn can be seen as a set of blocks.

to be able to serve 10,000 clients with the maximum bandwidth supported by the underlying storage system and network by using a large set of OSTs. It would be interesting to know the clients per MDS or clients per OST ratio to provide reasonable performance.

2.9 Conclusion

Each of the discussed DFS adds some valuable concepts and shows different characteristics that are summarized in this section. The following aspects are compared, mostly chosen from the ideal DFS requirements in Section 2.2:

Architecture Three different types of architectures exist:

- *Serverless (S).* The DFS tries to eliminate the bottleneck of centralized servers in a peer-to-peer fashion or at least reduce the reliance on centralized servers substantially compared to other variants of DFS. None of the discussed systems uses this approach but it is mentioned for completeness.
- *Client/Server (C/S).* The DFS consists of one or more centralized servers and any number of clients. Servers provide data and metadata and manage concurrent access to it.
- *Client/Data Server/Metadata Server (C/DS/MDS).* The DFS consists of one or more centralized metadata servers and any number of data servers and clients.

(A)synchronous Operation Synchronous operation requires immediate availability of all involved partners such as servers, clients, and network links. Synchronous protocols will only perform well if all components of the system are highly available and provide low latency. These requirements are quite restrictive and not feasible for many real-world applications. Especially the latency of disk I/O operations are prohibitive for fully-synchronous protocols. Asynchronous operation tries to overcome this limitation by allowing for intermittent disconnection and high-latency components. Caches and queues are the concepts coming along with asynchronous protocols. Asynchronous protocols on the other hand cannot provide or only approximate semantics requiring synchronous operation. Any DFS therefore has to weigh the requirements and find an appropriate trade-off between synchronous and asynchronous operation.

Replication Three possible ways exist how replication appears in DFS:

- *None.* Neither data nor metadata nor state is replicated.
- *Read.* Files being replicated among several servers can only be modified on the master server.
- *Read/Write.* Files being replicated among several servers can be accessed and updated. Metadata and state information is also distributed over several servers for better availability.

Approximation of UNIX Single-Copy Semantics There are four possible modes where C is the set of all active clients and $C_{lockholder}$ is the subset of all active clients actually holding a compatible lock on some file f:

- *Strict.* At any time t it is guaranteed that at most one client $c_k \in C$ is allowed to update a file f or a portion of it. Note that this is a highly synchronous operation since c_k has to wait for all other clients $c_i \in (C_{lockholder} \setminus c_k)$ holding locks on f to agree and therefore release any read or write locks on f.
- *Asynchronous Callback.* At any time t it is highly probable that at most one client $c_k \in C$ is allowed to update a file f or a portion of it. This loosens strict semantics introducing asynchronous notification of all other clients $c_i \in (C_{lockholder} \setminus c_k)$ to agree without waiting for their acknowledgments. This implies that only the last client $c_n \in (C_{lockholder})$ closing f will see its data persisted. Note that processes on client c_k operate with strict semantics as long as $\{c_k\} = C_{lockholder}$.
- *Optimistic.* At time t_1 it is probable that at most one client $c_k \in C$ is updating a file f or a portion of it. If a conflicting update is found at time $t_2 > t_1$ (any client $c_i \in (C \setminus c_k)$ updated f or a portion of it concurrently) a reconciliation must be initiated.
- *None.* At time t, any number i of clients $c_{1...i} \in C$ can concurrently read and write a file f or a portion of it.

Lock State Recovery The global lock state[7] recovery after a server crash can be handled differently:

- *None.* The global lock state is not recovered or the protocol does not support/need locks.
- *Centralized.* The server makes the lock state persistent. This speeds up recovery time and assures secure lock recovery. The drawback is a loss of performance due to synchronous lock persistency writes on the server during normal operation.
- *Distributed.* The server keeps the lock state only in volatile memory to speed up normal operation. After a server failure, the server must recover the global lock state with the distributed lock state information from all clients. A grace period assures no new locks are handed out during this lock state recovery. This is both time consuming and subject to potential malicious operation.

Security Two levels of security are supported by the presented DFS besides authorization which is provided by all discussed DFS:

- *Authentication.* The DFS offers authentication facilities to assure that only verified users and/or clients get access to the file system. Often, this is achieved together with third party protocols such as Kerberos [CDK01].
- *Encryption.* Besides authentication, the DFS also encrypts data in flight (communication channels) or at rest (storage disks).

Protocol Layer The ideal DFS provides access transparency to make distributed access indistinguishable from local access. This can only be achieved with operating system plugins and is therefore intrinsically bound to operating-system specific access APIs or file system protocols. These are sometimes not sufficient or convenient for a specific DFS. So, application-specific protocols are introduced that require proprietary commands visible to the clients or users.

[7]The global lock state records which client holds which locks.

Networking Environment The design of a DFS binds it to a specific networking environment. Synchronous DFS are better suited for LANs whereas asynchronous DFS still perform well in a WAN.

Scalability A major goal of many DFS is to be scalable. It is interesting to compare the scalability with the other design decisions such as synchronous vs. asynchronous operation. Since there are no common performance figures and setups to compare (i.e., clients per server ratio with a given quality of service) only rough approximations are presented.

Resilience Another major requirement is the resilience of the file system from a client perspective. Resilience is mainly influenced by the availability of the servers (load factor, uptime) and the availability of the network. Replicating the servers can ameliorate both.

In the course of this chapter, six different DFS have been presented and certain aspects have been investigated. The final comparison in Table 2.1 on Page 16 shows that use cases and design decisions greatly influence scalability and resilience of these systems. There is a strong intention to support strict UNIX single-copy semantics. Only recent developments have brought along enough sophisticated techniques such as leases or distributed lock recovery to achieve this without sacrificing scalability requirements. Today, there is a lot of research activity in DFS due to the introduction of peer to peer (P2P) networking, storage area networks (SAN), and demand for security.

Table 2.1: *The DFS are compared regarding to the described aspects. Note that both columns* Synchronous *and* Asynchronous *are only approximations since most synchronous systems have asynchronous parts for performance reasons. Coda in connected mode uses callbacks and switches to optimistic semantics in disconnected mode. With NFS, only the additional locking protocol applies lock state recovery in a distributed fashion; the original NFS does not support locks. Also note that DFS designed for WANs work perfectly in a LAN environment. Subversion relies on external authentication and encryption facilities. Scalability is approximated and relative to NFS [CDK01], [BHJ+93], [SS96]. SAN.FS discussed in detail in Chapter 3 differs regarding replication. Data might be replicated on a lower level using RAID. Metadata servers are clustered which is a form of replication.*

	NFS	AFS	Coda	Echo	Subversion	Lustre	SAN.FS
Architecture							
S							
C/S	●	●	●	●	●		
C/DS/MDS						●	●
Operation							
Synchronous	●	●		●		●	●
Asynchronous		●	●		●		
Replication							
None					●		●
Read	●	●					
Read/Write			●	●		●	
UNIX semantics							
Strict				●		●	●
Callback		●	●				
Optimistic			●		●		
None	●						
Lock recovery							
None	●		●		●		
Centralized		●		●			
Distributed	●					●	●
Security							
Authentication	●	●		●		●	
Encryption						●	
Protocol layer							
Filesystem	●	●	●	●		●	●
Application					●		
Environment							
LAN	●			●		●	●
WAN		●	●		●		
Scalability		++	++	+	+++	+++	+++
Resilience	−	−	+	+	−	++	++

3 SAN.FS Protocol

3.1 Overview

The IBM TotalStorage^{TM} SAN File System (SAN.FS) Protocol Set is described in this chapter to obtain a good understanding of the concepts and techniques used. Some parts of this chapter are taken almost verbatim from the SAN File System Draft Protocol Specification [IBM03].

3.2 SAN.FS Components

SAN.FS is a DFS separating management from storage facilities for better scalability, performance and maintainability. It is a highly optimized system designed for LAN, actually SAN, environments providing many of the features discussed in Chapter 2 and supporting many common use cases very efficiently. The main components as seen in Figure 3.1 on Page 18 are:

Clients (C) Clients running on heterogeneous nodes provide SAN.FS services such as reading or writing data on the file system kernel driver level to the applications.

Metadata Servers (MDS) Several metadata servers are combined into a cluster for availability and performance reasons. The cluster uses a shared storage model for its state information. In the current version, only one cluster is allowed in a SAN.FS setup to provide the global namespace. Metadata servers manage the clients reading or writing data or metadata concurrently but do not store any data besides state information. Metadata servers are often simply called *servers*.

Storage Devices (SD) The SAN consists of numerous logical units (LU) such as single disks or RAID volumes. Data and metadata are stored in separate parts of the SAN for security reasons. The current protocol version supports SCSI block devices.

Note that there are also *admin consoles* giving administrators access to metadata server cluster nodes.

The interconnecting networks are conceptually separate but can easily be merged in practice, e.g., using iSCSI [SMS$^+$04] over an Ethernet-based LAN:

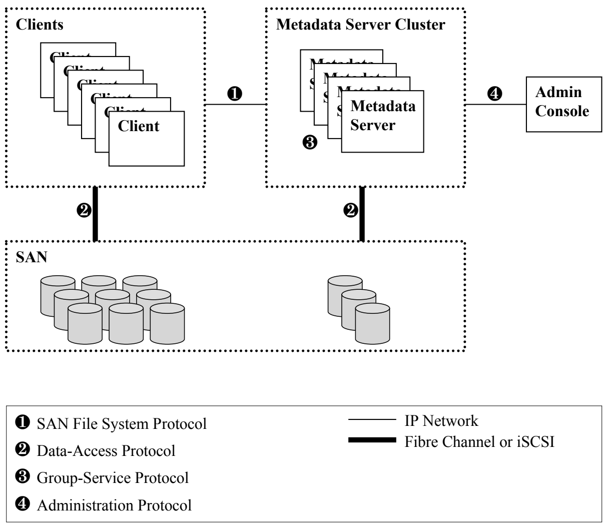


Figure 3.1: *Components of the SAN.FS and the required protocols. If the SAN is IP-based, the two networks may be merged. Note that the clients do not interact with each other and that the metadata servers use separate disks on the SAN.*

Management Network An IP-based LAN connects the clients with the metadata servers which is mainly used for management, concurrency control, or metadata exchange purposes. The admin consoles are also attached to this network.

Storage Area Network (SAN) A SAN connecting clients, metadata servers and storage devices.

In the current protocol version, all components and networks are fully trusted.

3.3 SAN.FS Protocols

Several protocols cover the communication needs of the SAN.FS components:

SAN File System Protocol The core protocol of SAN.FS specifies client to metadata server communication. It is layered on top of an IP network using either TCP or UDP.

Data-Access Protocol Clients directly access the block-storage devices using standardized protocols like iSCSI or SCSI over fibre channel.

Group-Service Protocol Several metadata servers leave and join the cluster dynamically, e.g., because of a failure. To provide a homogeneous view to the other components, a cluster group service protocol is used. Note that the SAN File System Protocol does not hide the existence of each single metadata server cluster node. As already stated, the cluster uses a shared storage.

Administration Protocol The administration consoles communicate with the nodes in the metadata server cluster for configuration and administration purposes.

3.4 Key Design Features of SAN.FS

SAN.FS provides strict UNIX semantics and is highly synchronous. This is why it depends on a low-latency, high-bandwidth LAN with rare node or network failures for smooth operation. To overcome potential limitations of this approach, a special architecture with several optimizations and techniques has been introduced that allows the protocol to scale better than most of the previously discussed DFS. In this section, the key design features of SAN.FS are presented, followed by the next section outlining some important concepts.

Centralized Client/Server Architecture Clients accessing the global name space need to communicate with the metadata servers and the storage devices but need not be aware of each other in a P2P fashion.

Separation of Metadata from Data While file data is stored in the SAN directly accessible to the clients, metadata is managed by the metadata servers. Metadata includes attributes and directory information of the files, symbolic links as well as a mapping of the file address space to physical blocks of the storage devices.

Centralized Control of Application Synchronization Client access to file system objects is controlled and coordinated by the metadata servers. To reduce the communication overhead, some control can be handed over to the client on behalf of the local applications.

Centralized Control of Client Caching To provide strict UNIX semantics, cache synchronization is also centralized on the metadata servers.

Cross-Platform File System Access SAN.FS allows cross-platform access to the global name space. Currently various UNIX flavors as well as Windows clients are supported.

3.5 Diving into SAN.FS

The following sections summarize important concepts found in SAN.FS. The following chapters rely on a thorough understanding of these mechanisms and are described in detail in [IBM03].

3.5.1 Global Name Space

The SAN.FS manages all file system objects within a *global name space* which is organized in a tree as follows (see Figure 3.2 on Page 21):

Root The *root* node is labeled *"/"*.

Cluster Nodes The *cluster nodes* are children of the root node. The current protocol version supports exactly one cluster labeled *STORAGETANK*.

Filesets The *filesets* are children of the cluster nodes and contain any number of filesets or file system objects (files, directories, or symbolic links). A fileset is identical with a *workload* from the metadata server point of view. A workload is managed by exactly one metadata server at any time and can be handed over in case of a metadata server failure. A metadata server can manage multiple workloads.

File 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.

Directory A *directory* represents a node in a file-system name-space hierarchy, whose children are other file-system objects.

Symbolic Link A *symbolic link* is a node in the file-system tree at which the name lookup of on an object is redirected.

Each file-system object is identified by a *global unique identifier* which is represented as a four-tuple <*clusterID.filesetID.objectID.versionNumber*>. The *clusterID* denotes the cluster the object is managed by. The *filesetID* marks the fileset or workload the object belongs to. The *objectID* is unique and immutable for the given file system object. The *versionNumber* belongs to the version of a whole fileset that can only be increased by the administrator-level operation *FlashCopy*. FlashCopy is used to make a consistent snapshot of a fileset for backup purposes. Note that FlashCopy uses a *copy-on-write* approach, where data is actually only copied on the first modification.

3.5.2 Lease Handling

In the SAN.FS, any locks or caches the client holds are protected by a lease. The client cannot perform any operation without continuously keeping the lease valid with the server. This happens either explicitly by sending a message to the server or implicitly while issuing requests to the server.

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

The lease allows the client to operate efficiently by reducing the communication overhead resulting from checking the server repeatedly for the validity of held locks or cached data.

3.5.3 Session Locks

Session locks protect the application state of a file object regarding open and close semantics. Because of the centralized control of application synchronization, a session lock must protect any operation on a file system object. The type of the

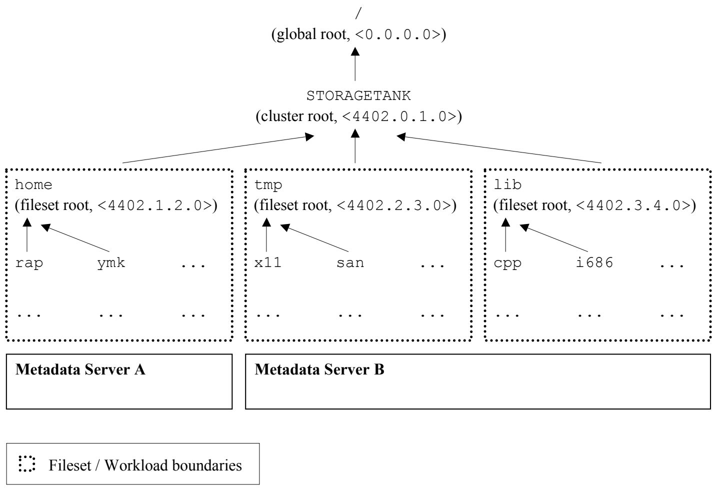

/
(global root, <0.0.0.0>)
STORAGETANK
(cluster root, <4402.0.1.0>)
home
(fileset root, <4402.1.2.0>)
rap
ymk
tmp
(fileset root, <4402.2.3.0>)
x11
san
lib
(fileset root, <4402.3.4.0>)
cpp
i686
Metadata Server A
Metadata Server B
Fileset / Workload boundaries

Figure 3.2: *Global name space of SAN.FS. The dotted lines denote fileset boundaries, which are identical with the workload boundaries. Workloads are distributed among the available metadata servers.*

session lock specifies, what operation the lock holder and other clients are allowed to perform. In other words, it tells the SAN.FS, what an application opens the file for. There are several existing semantics like POSIX or Windows semantics that are supported by SAN.FS. The client manages exactly one session lock per file on behalf of all local applications concurrently accessing it. The session lock must be as strong as the strongest lock required by an application. Session locks are *semi preemptable* to actually protect the application state of a file object. This implies two things: First, a client can refuse a metadata server request for a session lock on behalf of another client. Second, starvation might occur if the client keeps the lock for an undefined amount of time. The metadata server however can force a lock revocation for administration operations such as FlashCopy. This locking mechanism greatly reduces client to server communication and metadata server state-management overhead but nevertheless allows to maintain strict UNIX semantics.

3.5.4 Range Locks

On top of a session lock, a client application can acquire a range lock for any number of bytes — even behind the current end of file. This allows coordinating multiple (distributed) applications accessing specific parts of a single file concurrently. In contrast to session locks, the metadata server manages range locks at the granularity of applications. To reduce network traffic, it may delegate and revoke control for requested ranges to a client. A client can indicate that it does not immediately need a required range lock and thereafter waits for the server to grant it. Note that range locks smaller than the basic storage-block unit are automatically expanded to block size on the client side.

3.5.5 Data Locks

Side by side to a session lock, a client can acquire a data lock that actually allows for reading or writing a file system object or caching of (meta)data. Note that data locks apply to a whole file and cannot be combined with range locks. Data locks are *fully preemptable* because they do not have to protect an application file state. This implies two things: First, a client holding a specific lock must release it as soon as the server requests it (and might therefore have to commit dirty caches to the storage devices). Second: A client request for a lock will be granted as soon as all other clients holding a conflicting lock have released or downgraded the conflicting lock to a compatible mode.

For directories and symbolic links, the caching is read-only and only covers metadata. This data lock mode is called the *clean* (C) mode. If such an object must be modified, a transactional request is issued to the server. Caches are kept up-to-date using a publish-subscribe mechanism. Updates on the cached objects are propagated synchronously to all clients holding a data lock for it.

For files, the data locks are handled differently since both data and metadata are involved. The following modes are distinguished for file data locks:

Shared Read (SR) Multiple clients can concurrently hold this mode for read-only caching of data and metadata.

Shared Write (SW) Multiple clients can concurrently hold this mode for read-only caching of metadata. Data is read or written to using uncached direct I/O. This mode is useful for distributed applications such as databases, which implement their own caching and coordination model.

Exclusive (X) A single client can hold this mode for both reading and writing data or metadata.

3.5.6 Managing File Access

SAN.FS provides a powerful virtualized access to its data using a *block-allocation map*. The file address space is mapped on the metadata server to the physical blocks on the storage devices. This concept allows to save physical blocks for unused or empty parts in the file address space or to spread the blocks over many physical devices. A detailed description of the mappings and communication involved can be found in [Wag03, IBM03].

3.5.7 Failure Handling

Several scenarios exist, where well-defined failures must be handled and state must be recovered. One difficulty arises from the fact, that different failure types show the same behavior. Especially network partitions are not distinguishable in the short run from node failures and appropriate measurements such as leases or lock versioning have to be taken. The situation is potentially complicated because two separate networks are involved (i.e., an IP-based and maybe a fibre channel network).

A lock version is a two-tuple represented as <*filesetEpoch,lockVersion*>. The *filesetEpoch* is increased whenever the workload is handled by a new instance (on the same or a different physical machine) after a metadata server crash or shutdown. The *lockVersion* is increased whenever a client holding a lock should release it but its lease is expired. This is known as *lock stealing*. Additionally, data lock versions are increased, whenever (meta)data is modified. Note that for performance reasons, only lock version information is made persistent on the metadata server. Information about which client holds what locks in which mode is kept in volatile memory on the metadata server and on the clients.

Whenever the server or the IP network is down, the client will not be able to renew its lease. Before the lease actually expires, the client locally stops updates to the cache and tries to flush the cache to the storage devices. The cache sizes are chosen such that the flush operation will complete before the lease eventually expires. When the lease expires, all cached data or metadata is kept as is for later recovery but not read nor written. In case the client can acquire a new lease after the failure was resolved, it must also reacquire any session, range, or data locks it held. This is done using lock versioning as follows: A lock is reacquired and its version compared to the cached version. If they match, work is resumed. If they don't match, caches are cleared and a fresh lock is requested from the server.

Whenever a client or the IP network is down, the client lease will expire on the server. After an additional *safety period*, any locks held by the client can safely be transferred to other clients if requested (lock steal). The lock version is increased in this case to control the lock reacquisition of clients becoming available again later on.

Whenever the server itself fails, it eventually restarts and immediately enters a *grace period*. During this grace period, only lock reacquisitions are allowed. Thereafter, fresh locks can be acquired as usual. This approach is necessary to protect the current global state even after a metadata server crash. If the global state would not be restored accurately, client applications would likely have to report errors or at least to refresh large amount of (meta)data in their caches.

4 Security Issues

4.1 Overview

The first generation of SAN-based DFS expected other file-servers (e.g., NFS file-servers) to be their clients, i.e., that all components of the SAN-based DFS could be trusted. The second generation aims at spreading the SAN-based DFS directly to the organizations' insecure desktops. As a consequence, some components get deployed into environments that cannot be trusted or are connected via public networks. The rapidly growing number of computer devices and the various needs and intentions of their users introduce new aspects and problems not known or considered before. What used to be simple identifiers must be authenticated identifiers today. Data that was not sniffed at or modified by contract must be encrypted and its integrity assured. Protocols that provided access on a best effort basis must be hardened against abuse and their resilience – even under malicious attack – must be guaranteed.

This chapter gives a brief summary of security considerations, followed by a two-stepped approach to find issues with the SAN File System Protocol for client to metadata server communication as described in the task of this thesis: First, we looked at issues that arise during normal operation. Second, we took malicious operation into consideration. Since most of the found issues do not only affect SAN-based DFS but any DFS or local file system (FS), it is also mentioned in parenthesis with each issue, where it should be considered as well. Note that it is hard with failing nodes or applications and also in asynchronous systems, to keep apart normal from malicious operation. Finally, a conclusion will be drawn.

4.2 Security Considerations

Before one can look for security issues, it must be clear, first: what the environment is and second, what kind of security is expected.

4.2.1 Whom to Trust

A DFS consists of many different components, from which some are trusted and some are not. While trusting a component simplifies protocols and implementations and improves performance due to less complexity, it is often not practically feasible to extend the trust to the whole system. The trust levels for SAN.FS should look as follows (diverging from the assumptions of the current protocol version):

Network The underlying networks are not trusted.

Metadata Server Metadata servers are fully trusted. A trusted administrator can easily control the cluster of metadata servers in a safe place.

Storage Device Storage devices are fully trusted. A trusted administrator can easily control the storage racks in a safe place. Further security within SAN environments is not part of this paper and an active research topic (e.g., [AJL+03]).

Client Clients cannot be trusted because they could be tampered with malicious code either on the SAN.FS kernel driver or at the application level. Trusting users and applications is not in the scope of SAN.FS but must certainly be addressed by other means. Note that clients could also be partially trusted, i.e., the kernel is trusted but not the user space.

4.2.2 What to Expect

A wide range of concepts and expectations is covered by the word *security* ([AS99, CDK01]):

Authorization and Authentication

SAN.FS in the current version uses access control lists (ACL) to authorize an user to access files; authorization happens on the client side. Not only file access should be authorized, but any access to the system or its resources. Even more, any component of the system should authenticate itself to prevent forging identities; also trusted entities must prove who they are. The current SAN.FS protocol version does not support authentication.

Metadata Server Metadata servers should authenticate themselves to assure clients do not communicate with malicious third party metadata servers.

Client The client has to authenticate itself with the SAN.FS. This assures that only the expected set of clients can communicate with the SAN.FS.

User and Application Users and applications have to authenticate themselves on a layer higher than the SAN.FS. Whenever a user or application is requesting SAN.FS services on a client, it is assumed they already have been authorized and authenticated by other means. To get access to specific files they must still be authorized. This is a feature of the current protocol version and is implemented as access control lists (ACL).

Storage Device Storage devices are identified by a logical unit (LU). Authenticating LUs would be in the scope of the underlying protocols such as iSCSI. Note again that securing SANs is an active research topic and not part of this thesis.

Encryption

Preferrably, data must be encrypted to protect it in flight and at rest to prevent eavesdropping (adapted from [Wag03]).

Data In Flight Client to metadata server or storage device communication must be encrypted if confidential information is transmitted. Since encryption might adversely affect network throughput and latency, one might want to only encrypt parts of the transmitted protocol messages.

Data At Rest The stored data itself must be protected. One could argue that the virtualized mapping of the file address space to physical blocks provides a reasonable level of protection because a malicious client would need to know where the blocks of a specific file are stored to access them. This statement would require further investigation. Data at rest however is much more exposed to attacks since the attacker has more time and flexibility compared to attacking data in flight.

Integrity

Speaking of data security implies speaking of data integrity. It must at least be possible to detect malicious changes of the data. This aspect is covered in [Wag03]. The next step is not only to detect but also to recover such manipulation using ECC[1] or other techniques such as data versioning or simply backups.

Resilience

Besides the requirement for continuous operation even in case of node failures, it must be ensured that the services are not susceptible to any potential type of denial of service (DoS) attack.

Metadata Server Metadata servers can be clustered to increase resilience. But physical availability is not the only issue. Since metadata servers control distributed file access, they must keep state information such as leases and locks. It should not be possible at any time for a malicious client to prevent other clients from normal operation through consuming all metadata server memory and processing capacity.

Storage Device The availability of storage devices can be achieved with either deploying RAID systems as one LU or replicating the physical blocks using an extended virtualized file address space mapping. The RAID layer solution is currently used, which unloads the protocol from the burden of replication. The protocol level replication would probably introduce more communication overhead. On the other side, it could be more flexible and even show better resistance to network partitions because blocks could be widely spread.

[1]Error correcting code (ECC).

4.3 Natural Failure Conditions

The investigation of the protocol could not reveal any weaknesses for non-malicious fail-free operation. Several potential failure scenarios have to be covered however, from both the client and metadata server perspective since these are the active components in the system. For all given scenarios, it is important to remember that clients apply different types of interaction for either communicating with the metadata server or the storage device:

Client to Metadata Server Any interaction between a client and a metadata server is transactional and atomic. This is a synchronous mode where the transaction participants immediately see any failure.

Client to Storage Device Client to storage device interaction is direct for certain data lock types. In this case, client applications are assumed to implement their own specialized caching model. Other data lock types provide driver level caching. This cache is flushed at specific points in time, introducing asynchronous communication. As described in [BHJ+93], it is not always possible with asynchronous caches to report failures to the applications, e.g., when the application quit before the cache was flushed.

4.3.1 Client Perspective

From the point of view of a client, the following events must be analyzed:

Partitioned Storage Device(s) No data blocks can be read from or written to the storage device. While the data is still protected by locks, it may happen that dirty caches cannot be flushed. This will lead to unresolvable errors if the partition lasts for an extended period of time or another client requests the lock protecting a dirty cache.

Storage Device Failure Failing storage devices cannot be distinguished by the client from partitioned ones. Nevertheless, this scenario will potentially lead to a loss of data. A failure means the complete failure of the device.

Partitioned Metadata Server The client lease will expire but caches can be flushed to storage since clients cache at most as much data as they can flush to storage devices before the lease expires. When the lease eventually expires, any SAN.FS operation must be suspended and errors must be reported to the applications if the lease cannot be renewed within a specific amount of time.

Metadata Server Failure Failing metadata servers cannot be distinguished by the client from partitioned ones unless the workload is moved to another metadata server.

Partitioned Client When the client is disconnected from both, the metadata server and the storage device, the lease will expire, but the client will not be able to (completely) flush potentially dirty caches. This will lead to problems described above in *partitioned storage devices(s).*

Client Failure A crashing client will potentially lose data as any UNIX system does, if caches were dirty at the event of failure. If caches were partially written to the storage devices, other clients will see dirty data which they cannot distinguish from clean data.

Application Failure A rogue application on a client can harm session lock management since session locks are semi preemptable. As long as a session lock protects the application state, the client must deny a metadata server revoking the session lock from it. The only way the metadata server can forcibly revoke the stale session lock is during an administrator-level operation such as FlashCopy.

Distributed Application Misbehavior Since distributed applications are not aware of the underlying DFS and its optimizations, they might request the wrong lock modes from the SAN.FS point of view, e.g., a shared write data lock instead of a range lock. As a result, the client repeatedly has to acquire the same data lock and fill its cache, only to relinquish it on behalf of a metadata server request to release the data lock und flush the cache. The time the client can provide the lock to an application to actually perform some work is therefore reduced substantially. This is also known as *trashing*.

4.3.2 Metadata Server Perspective

From the point of view of a metadata server, the following events must be analyzed:

Partitioned Storage Device(s) No metadata blocks can be read or written from or to the storage device. Since the metadata server uses caching, it may work some short time without having to read or write from or to a storage device. Afterwards, or as soon as a synchronous write occurs, the metadata server has to cease operation.

Storage Device Failure Failing storage devices cannot be distinguished by the metadata server from partitioned ones. Nevertheless, this scenario will potentially lead to a loss of data.

Partitioned Client The client lease will expire and after an additional safety period, any locks formerly held by the client will be granted to other clients requesting it. The server cannot be certain, whether the client could safely flush all dirty cached data as described above.

Client Failure Same as described in *partitioned client*.

Failure of Client Application The metadata server will not be able to revoke a session lock from a client hosting a failing (i.e., hanging) application. As a result, starvation might occur if another client waits for this session lock, or at least the server has to keep stale lock state information.

Distributed Client Application Misbehavior The metadata server has to handle hot spots on certain data locks moving them back and forth between two or more clients. The lock is *trashed* between the clients causing noticeable overhead on the metadata server.

Partitioned Metadata Server The metadata server ceases operation as soon as all leases expired due to a network partition. Whenever the metadata server is reconnected, clients will start to reacquire leases and locks without the metadata server entering a grace period, which could lead to clients losing session locks, which consequently must be reported as an error to the applications. This is a clear violation of the global state recovery, which demands grace period semantics.

Metadata Server Failure A crashing metadata server will potentially lose any cached metadata or state information. When restarted, it will enter a grace period to prefer lock reacquisitions over fresh lock acquisitions to reestablish the current global state.

4.3.3 Issues From Natural Failure Conditions

The following issues should be addressed for natural failure conditions. The issues are ordered by their impact on other file systems. Note that an issue affecting SAN.FS can also affect any DFS and an issue affecting a DFS can also affect any local FS:

Non-Atomic Disk Writes (FS) Without atomic disk operations, application or client crashes and lease expiration combined with network outages might lead to dirty persisted data on storage devices without having a possibility for detection or recovery. In contrast to local operation where such errors mostly occur when the application itself crashed too, distributed applications might see dirty persisted data unexpectedly and therefore misbehave or report errors.

Grace Period Usage (DFS) Metadata servers do not enter a grace period after being partitioned from the clients. This deviates from the semantics defined for a metadata server crash — the client should see consistent behavior since it cannot distinguish crashed from partitioned metadata servers. The omission of the grace period might lead to an inaccurate global state recovery with a lot of communication overhead or even application errors because a client cannot reacquire a session lock stolen by another client.

Data Lock Full Preemptability (SAN.FS) The trashing of data locks under certain conditions causes a lot of overhead and inefficiency that should be avoided or cured.

Session Lock Semi Preemptability (SAN.FS) Clients will protect the state of an application by holding session locks even in the case this application hangs. This will lead to “zombie” session locks that cannot be reclaimed by the server during normal operation until the client is shut down or the application is killed. Only the next administrator-level operation such as detaching a fileset or restoring a FlashCopy image will force the release of a all session locks.

4.4 Malicious Operation

A malicious user could harm SAN.FS in two fundamentally different ways: First, he could simply want to disrupt the service. Second, he could either read or write data without authorization. Comparable to criminal stories, our malicious user will always choose the most economic path to achieve his goals. SAN.FS has to apply the appropriate counter-measurements step by step also in an economic order. This will require the malicious user to invest ever-growing resources and therefore successively limits the number of potential malicious users.

At least two questions arise for a malicious user: First, how is the set of clients connected to the SAN.FS service organized and protected? Second, is it possible to attack the service from unprivileged user space? If the set of clients is well protected, the malicious user probably has to break into existing hosts. If unprivileged access is not sufficient, he must obtain root rights, which is even more difficult.

4.4.1 Denial of Service (DoS)

Denial of service tries to bring the system to its limits and eventually render it useless for others by using up all resources such as memory, processing capacity, network bandwidth, or disk capacity [CDK01, LRST00]. The following attacks can be performed (ordered by difficulty):

Byzantine Messages The malicious client tries to confuse the metadata server or another client by sending arbitrary messages causing byzantine errors. Since it is not clear, what impact some message combinations have, it must be tried out for each metadata server or client implementation. Depending on the checks performed, this might lead to quick and therefore cheap results.

Disk Space Without having to analyze network traffic or hijacking connections to send fictitious protocol messages, it is feasible for only one client to consume all available disk space in a given fileset. It only must create one or more files and acquire block after block. The impact of this attack depends on the granularity of quotas on the filesets. The finer it is, the less harm it does. Note that SAN.FS supports file sizes of up to $2^{64} - 1$ bytes and that block allocations are expensive operations.

Session Locks The semi preemptability of session locks makes them an interesting target for a denial of service attack because only administrator-level operations will allow the metadata server to forcibly reclaim session locks. A malicious client could now start to acquire as many session locks as possible. The more clients follow this pattern, the faster the service is unusable for compliant clients, which cannot obtain session locks for specific files as required anymore. It is an important issue that clients can acquire a session lock on *any* file regardless of the ACLs protecting it, since authorization is done on the client side. Note that this attack requires an adaptation of the client or a SAN.FS protocol simulation if locks need to be stolen on objects the client would theoretically have no access to.

Data Locks The full preemptability of data locks makes them an interesting target for a denial of service attack if the accompanying session lock was first acquired by a good client and allows for shared access. A malicious client could then continuously acquire a data lock and release it again on demand, which would lead to trashing and an unusable service for the good client. Adaptation of client code is required for the same purpose as described for *session locks*.

Lock Reacquisition Since the global state is mostly kept in volatile memory and must be recovered after server crashes from the client memories, this offers the possibility for maliciously reacquiring locks formerly held by other clients or maliciously upgrading locks. Furthermore, it would be possible for malicious clients to reacquire locks from disconnected or crashed clients forging their identities. Both would break the global state and potentially cause good clients to report errors.

Other Fancy Attacks With the current protocol, some other highly specific attacks might be imaginable such as man-in-the-middle attacks where a malicious client would hijack an existing metadata server to client TCP connection and try to break the global state or make the service unusable. But all must rely in one way or the other on security issues discussed later on.

4.4.2 Read or Write Data

A malicious client has two ways to read or write data on the storage devices: First, it can learn the exact storage device and block numbers by querying the metadata server. Second, it can sniff on the network.

Via Metadata Server

Without having the possibility to sniff on the network, a malicious client must perform four steps to manipulate data:

1. Locate the required file, i.e., find the associated id (see Section 3.5.1).
2. Acquire a session and a data lock for read access on the file.
3. Fetch the block addresses from the metadata server.
4. Access the blocks on the storage device without performing authorization.

The malicious client must simulate the protocol to circumvent authorization.

Direct Access

If the malicious client somehow learned about the exact location (i.e., storage device and blocks) or if the client just wants to overwrite data randomly on the storage devices, it can directly access them without first requesting a lock. This is possible because locks are only advisory in the current protocol. Note that this currently is an operation involving only clients and storage devices but still should be mentioned here since the locking basically is in the domain of the metadata servers.

4.4.3 Issues from Malicious Operation

The following issues should be addressed during malicious operation where clients and their network connections are no longer trusted. The issues are ordered by their impact on other file systems. Note that an issue affecting SAN.FS can also affect any DFS and an issue affecting a DFS can also affect any local FS:

No Authentication (FS) If the partners are not authenticated, it is particularly easy to forge an identity or dynamically exchange identities. E.g., locks are bound to a client identifier and an IP address which must not be authenticated. The set of clients accessing the service can therefore hardly be controlled.

No Encryption (FS) The client to metadata server communication must be encrypted if confidential information is transferred (e.g., symmetric keys for data encryption as mentioned in [Wag03], or the contents of some home directory) or untrusted clients could easily eavesdrop on the communication and gather metadata information.

Spare Resource Limitation (Quotas) (FS) To prevent unlimited damage to the system, certain resources must clearly be limited per client or even per user. These are the number of session, data and range locks a client can hold, as well as the storage space a client or user can allocate. The current protocol version supports disk space quotas on a per fileset basis where the chosen quota granularity renders the SAN.FS more or less vulnerable to DoS attacks exploiting disk capacity. The finer the granularity, the less vulnerable it is. A trade-off must be chosen however since storage devices are statically assigned to filesets. A more coarse quota granularity leads to multiplexing gains[2] in storage space. Another aspect is the manageability of the system. The finer the granularity, the less manageable it becomes.

Advisory Locking (FS) The current advisory locking is not sufficient to prevent direct access from malicious clients to storage devices circumventing metadata servers and therefore potentially breaking the global lock state. Mandatory locking should be introduced to enforce locking semantics under all circumstances. This could require a protocol modification on both client to metadata server and client to storage device communication and maybe require a cooperation of metadata servers and storage devices.

Frugal Misbehavior Penalty (FS) There are some cases (especially if authentication is introduced), where malicious operation can easily be detected due to protocol violations. In most cases however, malicious operation cannot clearly be distinguished from normal or faulty operation. Nevertheless, there are indications such as trashing data locks or attempts to exceed quotas. All these events must be monitored and immediate penalties or deferred administrative actions must be triggered (e.g., reinstall a rogue client) according to configurable policies. Note that the current protocol already defines some incidents where the client lease is expired as a penalty.

Distributed Lock Recovery (DFS) Since SAN.FS uses a distributed, unauthenticated lock state recovery, malicious clients can make false lock claims after a failure and maliciously steal or upgrade locks which breaks the global lock state.

[2]Multiplexing gains are known from network traffic where the combination of multiple sources into one channel leads to efficient data transmission because of the varying bandwidth utilization of each source over time.

Client Side Authorization (SAN.FS) Authorization performed on the client side is conceptually weak if clients are not trusted. It specifically allows a client to acquire any lock type in any mode without having to proof, the user requesting it would finally be allowed to do so. This is important together with the described lock issues.

Session Lock Semi Preemptability (SAN.FS) makes session locks an attractive target for DoS consuming all available session locks in exclusive mode without having to hand them back before the next administrator-level operation. This is even aggravated taking the issue with *client side authorization* described above into account and will lead to the starvation of good clients waiting for a session lock.

Data Lock Fully Preemptability (SAN.FS) makes data locks an attractive target for DoS against a specific client holding data locks because data locks must be handed back to the metadata server on request. An attacker could enforce trashing by continuously requesting the same data lock. If the intention is not to prevent clients from accessing a file but keeping the clients and the metadata servers busy, trashing can be achieved by first acquiring a shared session lock and then moving the data lock accompanying it back and forth between the malicious and the good client. Trashing is also very effective if applied within two malicious clients.

4.5 Conclusion

SAN.FS currently is a DFS trusting all cooperating parties. This allows many optimizations such as client side authorization or the lack of authentication and encryption. Introducing malicious client activity, there is a shift of the interest from high performance towards sustainable long-term operation. Mechanisms such as distributed lock recovery must be rethought and adapted.

Not only during malicious, but also during fully trusted normal operation, there are some issues. Special failure scenarios must be addressed. It has to be said that these normal-operation issues have a low probability of occurrence. Nevertheless, a highly reliable DFS should be hardened even for rare failure events.

The next chapter tries to eliminate a selection of the found issues suggesting possible design modifications and evaluates their impact on the overall SAN.FS performance and security.

5 Design Modifications

5.1 Overview

This chapter addresses some of the issues found. For each, a possible design modification is suggested. First, a short overview is provided, discussing alternatives. Then, the mechanism, and where necessary, the protocol design or an implementation prototype is presented. Finally, an evaluation of the security and performance impacts on SAN.FS as well as a discussion of the remaining open issues conclude each suggested design modification.

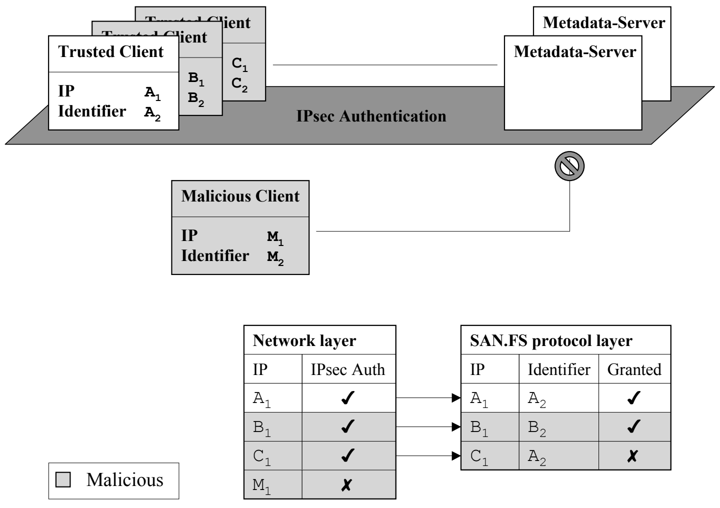
5.2 Authentication and Encryption

The set of possible attacks can be dramatically reduced if clients and servers have to authenticate themselves and communication is encrypted. Out of the seven Open Systems Interconnection (OSI) Reference Model layers [Tan96], this is commonly achieved on either the network (IP) or the transport (TCP/UDP) layer. Whatever solution is chosen, it is expected, that the runtime performance of SAN.FS is adversely affected by introducing cryptographic operations and that the key or certificate management can become an issue for large-scale installations.

The minimal requirement is to mutually authenticate clients and metadata servers to prevent unauthorized access. Encryption on top of authentication is not necessary to protect the current protocol: First, malicious clients that cannot authenticate themselves with the metadata server, are excluded from the protocol. Second, trusted but malicious clients can decide on the global protocol state by legally challenging the metadata server. On the other hand, encryption is a prerequisite, if the protocol is extended to transport confidential metadata such as secret keys or if the metadata itself must be protected; without encryption, a malicious client can learn the metadata by eavesdropping on the authenticated communication in the long run. An intermediate step could be only to encrypt selected parts of the protocol messages.

A promising candidate is the IP Security Protocol (IPsec, [TDG98]). It provides network-level integrity, confidentiality and authenticity on an end-to-end basis; end-to-end means host-to-host and not user-to-user. IPsec is optional for IPv4, mandatory for the next generation Internet Protocol IPv6 and available today for all major operating systems. Its main advantage from a SAN.FS point of view is, that its setup does not require touching existing SAN.FS client or server implementations and can be plugged-in transparently on the lower layer. Non-SAN.FS applications can automatically leverage IPsec. This would not be the case with a transport layer security (TLS, [APS99]) based solution.

5.2.1 Mechanism Design



Network layer	
IP	IPsec Auth
A_1	✓
B_1	✓
C_1	✓
M_1	✗

SAN.FS protocol layer		
IP	Identifier	Granted
A_1	A_2	✓
B_1	B_2	✓
C_1	A_2	✗

Figure 5.1: *IPsec allows to restrict access to the metadata server to trusted clients. It is not possible for a malicious client (trusted or not) to forge the identity of a trusted client. Note that trusted clients can become malicious after they have been hacked. As a consequence, trusted but malicious clients can still access the metadata server. Nevertheless, the bar to access the metadata server is substantially higher with authentication.*

The set of clients is divided into a set of trusted clients and a set of untrusted clients. This separation is configurable by an administrator on the network (IP) layer. The metadata server shares a different secret with each of the trusted clients and requires all communication with trusted clients to be authenticated with IPsec. Non-IPsec traffic is not accepted. Neither is traffic with bad authentication. This guarantees that only trusted and authenticated clients can access the metadata server (see Figure 5.1).

The set of trusted clients must not necessarily be disjoint from the set of malicious clients. It is assumed that trusted clients could be hacked and become malicious without violating the authentication of IPsec. Note that the ease of hacking a trusted client depends on the administration and security policies applied within an organization. IPsec however states, that it is not computationally feasible for a malicious client to forge the identity (i.e., the IP address) of a trusted and authenticated client or to modify the sent packets in any other way.

On the SAN.FS protocol layer, the metadata server identifies each client with an unique *client ID*, which is chosen by the client itself and is valid for the lifetime of a lease (see [IBM03], Section 4.7.3, Client ID). Note that the client ID and the lease expire when the IP address of the client changes. Whenever the metadata server receives a protocol message, it assures that the IP address and the client ID of the message match the IP address and client ID stored in volatile memory. As soon as a specific client initializes these values on the metadata server by sending it an *Identify* message, the client ID is authenticated by the IP address which in turn is authenticated by IPsec.

5.2.2 Implementation

A firewall (e.g., iptables on Linux systems) is configured to drop non-IPsec TCP or UDP traffic to a specific port on the metadata server. This reduces the set of clients accessing the metadata server to the trusted clients because IPsec traffic is only accepted on the IPsec layer, if it is successfully authenticated.

The prototype uses Internet Key Exchange (IKE) implemented by the KAME[1] based racoon daemon to automatically exchange Security Associations (SA) based upon a common configuration. Racoon also makes sure to regularly exchange the symmetric key to increase security. *HMAC SHA1* is used for authentication, *AES* for encryption. This configuration is suggested by the IPsec implementation because it offers the best performance and security trade-off. The set of trusted clients is controlled by the metadata server IPsec-configuration listing the pre-shared keys for all clients. If a client is added to the set, its pre-shared key is added to the list and removed from it, whenever the client is removed from the set. In addition, each client stores the pre-shared key of the metadata server.

It is worthwhile to note that there is some critique about IPsec [FS00] especially because of its complexity. The implementation prototype is taking this into consideration and uses the Encapsulating Security Payload (ESP) in tunnel mode for encryption and authentication. Transport mode or authentication alone is not used because it is only a special case of tunnel mode and does not work together properly with the current iptables of Linux.

5.2.3 Evaluation

IPsec performance was measured by [MIK02]. Some own measurements were made to verify these results. The measured performance drops compared to non-IPsec operation depend on the test setup (i.e., network bandwidth and minimal available CPU power on both hosts). While the latency of IPsec is not affected throughout different setups (see Figure 5.2 on Page 38), the throughput clearly is (see Figures 5.3 and 5.4 on Pages 39 and 40). The reason lies in the fact that the available network bandwidth is the bottleneck in non-IPsec operation while it usually is CPU power in IPsec operation. The two setups chosen for the final measurement consisted of:

Slow Scenario An IBM xSeries 335 server with a XEON 2.8GHz and an average 1.8GHz Pentium4 desktop connected over a 100Mbit Ethernet. While the server showed a CPU utilization of less than 20%, the desktop CPU was

[1]www.kame.net

utilized by over 99%. Note that the slow host dictates the overall system performance.

Fast Scenario Two IBM xSeries 335 servers with a XEON 2.8GHz connected over a 1Gbit Ethernet. Both servers showed a CPU utilization of over 90%.

On both hosts, the actual Linux operating system was running in a virtual machine. The results are listed in Table 5.1 on Page 39. Note that encryption requires slightly more CPU power than decryption.

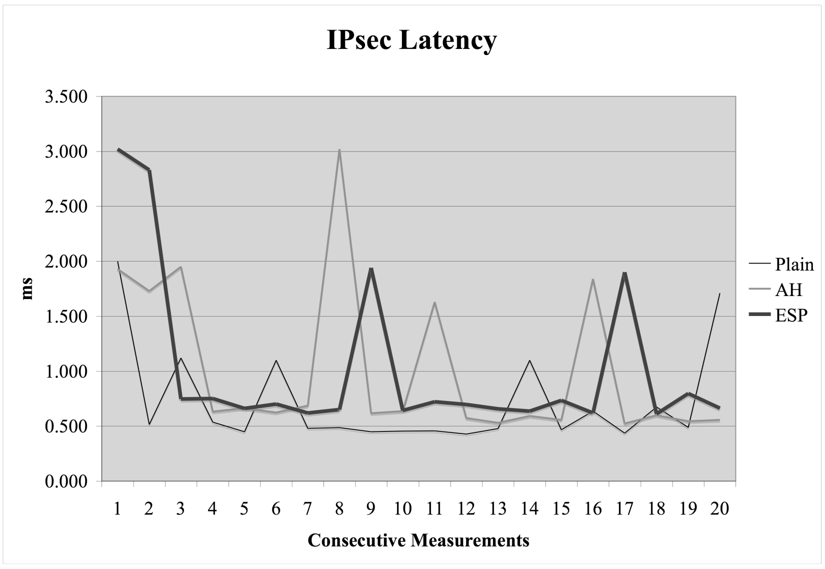


Figure 5.2: IPsec latency for non-IPsec (Plain), IPsec with authentication (AH), and IPsec with authentication and encryption (ESP)

The SAN.FS protocol intermittently transmits small packets of mostly less than 512 bytes. Each transmission is accompanied by some server-side work-time and potential reads and writes on storage devices. This protocol characteristics support the assumption, that the IPsec performance penalty is not that severe if applied to the SAN.FS protocol (see Figure 5.5 on Page 41).

The PostMark benchmark [Kat97] was applied for several setups to verify this. PostMark was chosen because it offers a good mix of metadata and data operations – it simulates the workload of a mail, news, or e-commerce server. It is not clear, why the results contain a serrated pattern. But all measurements – even if taken after some time – lie within a narrow interval. The setup consisted of a storage and a metadata server running in a virtual machine on a IBM xSeries 335 server with a XEON 2.8GHz. The client running in a virtual machine on an average 1.8GHz Pentium4 desktop was connected with the server over a 100Mbit Ethernet. All hosts and virtual machines were equipped with sufficient memory not to start swapping and no other tasks were running. Both, the storage and metadata server showed CPU utilizations of mostly less than 50%. The client CPU was utilized by over 90%. Network bandwidth was never saturated. It follows, that the client was the

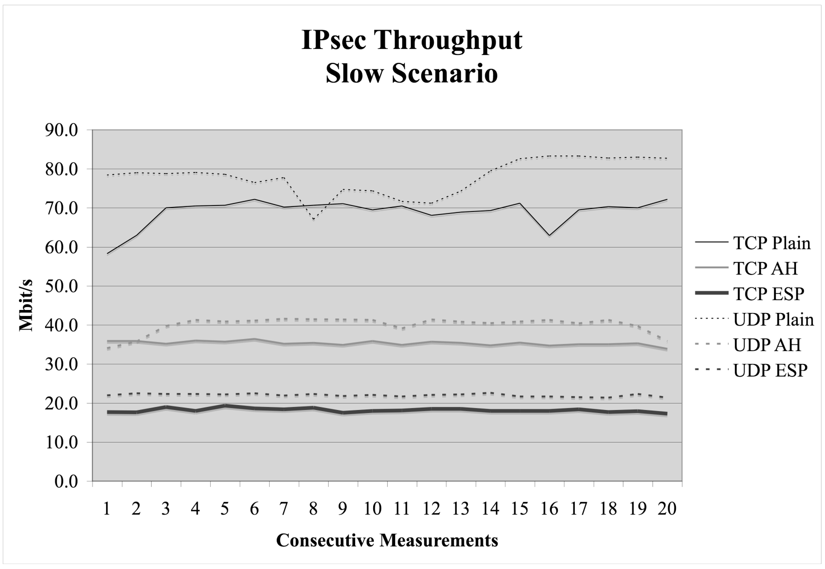


Figure 5.3: *IPsec throughput in a slow scenario for TCP and UDP.*

Table 5.1: *IPsec throughput evaluation. As a rule of thumb, two fast hosts achieve a throughput with IPsec comparable to two slow hosts without IPsec. The values are in Mbit/s.*

	UDP Plain	UDP AH	UDP ESP	TCP Plain	TCP AH	TCP ESP
Slow Scenario						
Min	67.1	34.1	21.4	58.3	33.9	17.3
Avg	77.9	40.0	22.0	69.0	35.3	18.2
Max	83.3	41.6	22.6	72.2	36.4	19.3
MDev	4.6	2.2	0.4	3.5	0.6	0.5
Conf_{95}(Avg)	±2.0	±1.0	±0.2	±1.5	±0.2	±0.2
Conf_{99}(Avg)	±2.6	±1.3	±0.2	±2.0	±0.3	±0.3
Fast Scenario						
Min	515.0	177.0	71.1	455.0	159.0	71.1
Avg	524.5	186.5	80.3	478.5	189.3	72.9
Max	541.0	211.0	90.1	502.0	226.0	76.1
MDev	9.0	10.6	6.3	15.8	12.7	1.3
Conf_{95}(Avg)	±4.0	±4.7	±2.8	±6.9	±5.6	±0.6
Conf_{99}(Avg)	±5.2	±6.1	±3.6	±9.1	±7.3	±0.7

bottleneck – a clear indication of SAN.FS scalability. Table 5.2 on Page 40 lists the results.

Three cases were identified (see Figure 5.5 on Page 41):

Best Case The performance of SAN.FS is not affected by introducing IPsec. This case occurs when the client file system operation does not involve any com-

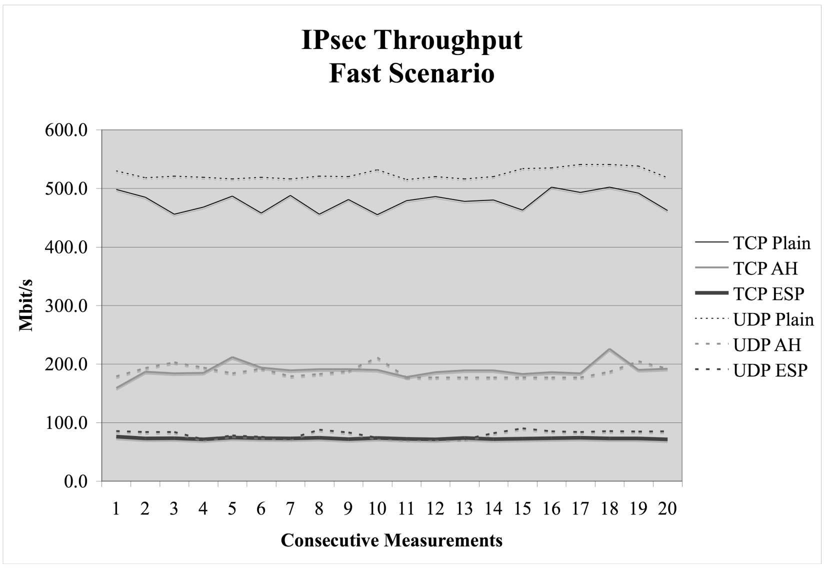


Figure 5.4: *IPsec throughput in a fast scenario for TCP and UDP.*

Table 5.2: *The PostMark benchmark applied to several SAN.FS setups. The number of files tests was 1'000; the number of transactions 10'000. For each setup, 20 runs were performed. Runtimes varied between 81 and 124 seconds.* AH *stands for IPsec authentication in transport mode and* ESP *for IPsec authentication and encryption in tunnel mode. All provided values denote the number of transactions per second.*

	UDP Plain	UDP AH	UDP ESP	TCP Plain	TCP AH	TCP ESP
Min	96.0	90.0	81.0	102.0	90.0	81.0
Avg	107.5	99.9	95.9	113.4	101.4	92.3
Max	117.0	107.0	107.0	123.0	111.0	99.0
MDev	7.2	6.1	7.6	7.4	6.3	5.3
Conf_{95}(Avg)	±3.1	±2.7	±3.3	±3.2	±2.8	±2.3
Conf_{99}(Avg)	±4.1	±3.5	±4.4	±4.3	±3.7	±3.1

munication with the metadata server. E.g., reading a file from which the metadata is already cached on the client.

Average Case In the expected average case, IPsec does only slightly affect SAN.FS's performance (11% decrease on average). This is because other operations such as metadata server-side operations or operations on the storage consume comparably more time than the client to metadata server communication. The PostMark benchmark with mixed metadata and data operations is a good example.

Worst Case The performance drop of SAN.FS equals the performance drop of IPsec which is 72% on average. In this case, the client exchanges a huge amount of data with the metadata server. E.g., reading of a directory containing many ten thousands of files.

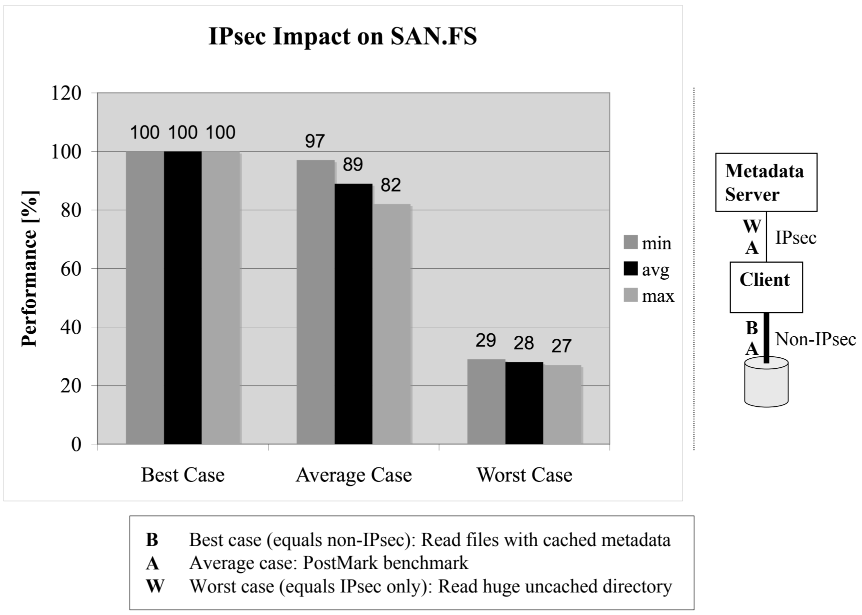


Figure 5.5: *IPsec SAN.FS performance evaluation. Applying IPsec on client to metadata server communication affects the overall SAN.FS performance in a wide range. Best, average and worst-case scenarios have been identified. Notably, the expected average loss in performance is far less dramatic than the preceding IPsec throughput performance evaluation indicated.*

If only the times to send a message are considered, the performance characteristics obey this idealized equation:

$$f(t_s, t_m) = \frac{t_s + t_m}{t_s + 3.57t_m} \qquad | t_s, t_m \in \mathbb{R}, t_s, t_m \in [0; 1], (t_s + t_m) = 1 \tag{5.1}$$

Here, t_s is the fraction of communication time between client and storage; t_m is the fraction of communication time between client and metadata server. For the best case, the tuple (t_s, t_m) would be $(1.0, 0.0)$ with a performance of 100%. For the worst case, it would be $(0.0, 1.0)$ with a performance of 28%.

The following is recommended for the IBM SAN.FS:

Non-IPsec For the current non-IPsec setup, it is recommend to use *TCP* for client to metadata server communication. TCP offers about five percent better throughput on average compared to UDP. The corresponding 95% confidence intervals hardly overlap.

IPsec For IPsec scenarios, it is recommended to use the *ESP tunnel* mode over *UDP*. This provides authentication and encryption. The average loss from non-IPsec operation to IPsec operation with authentication is about seven percent. The additional average loss from IPsec with authentication to IPsec with authentication and encryption is only four percent. The 99% confidence intervals of IPsec with authentication and IPsec with authentication and encryption overlap by 50%. Given that encryption prevents eavesdropping on metadata information, this price is more than justified. UDP outperforms TCP in this case on an average of four percent.

5.2.4 Open Issues

The following issues are not covered by the implementation prototype:

Large Scale Management The preshared key method is not feasible as soon as multiple metadata servers or more than some few clients have to be managed. In this scenario, a switch to the Kerberos based method is highly recommended. With Kerberos, all keys are managed centrally.

Dynamic IP Addresses There is an issue with dynamically assigned IP addresses. With the preshared key method, IP addresses must be static.

Dynamic Configuration Because of a flaw in the current Linux IPsec implementation, a client cannot be dynamically removed from the metadata server IPsec configuration without restarting IPsec. This affects normal operation because some clients might have to start lock recovery procedures since their locks were expired.

5.3 Secure Distributed Lock Recovery

Centralized lock recovery puts all the responsibility for persistently maintaining the lock state onto the server. This effectively requires the server to perform a synchronous write to stable storage on whenever a lock is acquired or relinquished. These writes significantly slow down the lock request process and severely limit the performance and scalability of the file system. The advantage is, that the global state can quickly and securely be recovered after failures at the server without involving the clients. There is no need for a grace period and the clients can immediately resume work after the server has sent them their locks. Note that authentication is required to provide an appropriate level of security.

Distributed lock recovery in contrast only persists a fraction of the global lock state. SAN.FS for example found an optimal trade-off and only stores rarely changing information such as the file system object identifier, the epoch and the lock version on the disk. Frequently changing information such as client identifier or lock mode is only kept in volatile memory – redundantly on both servers and clients. This approach favors scalability and performance over security and short recovery times. Recovery should be a rare process, as such recovery performance is not a prime criterion for normal operation, but may become critical in real-time environments. Recovery times are affected because of the inescapable restoration of the global lock state on the server after its failure. The complete trust and reliance upon the client cooperation is no longer tolerable without appropriate precautions if malicious client operation must be taken into consideration.

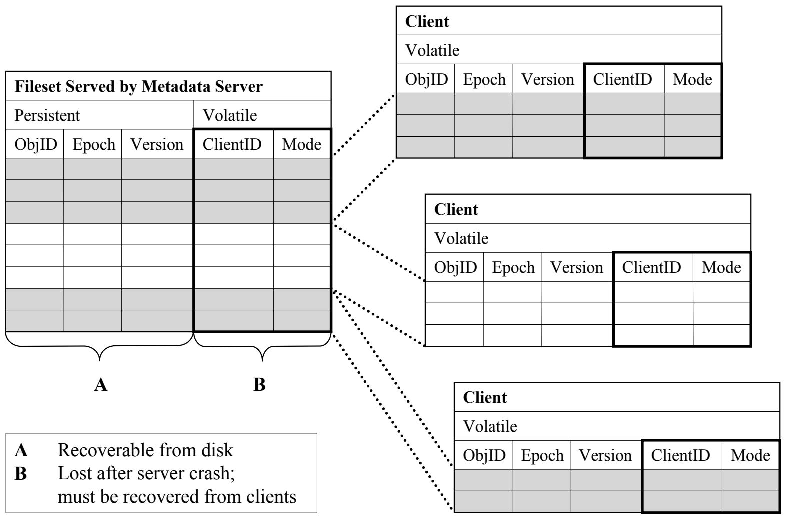

Fileset Served by Metadata Server
Persistent
Volatile
ObjID
Epoch
Version
ClientID
Mode
A
B
A Recoverable from disk
B Lost after server crash; must be recovered from clients
Client
Volatile

Figure 5.6: Distributed lock recovery. The lock state is handled at the fileset level. Epoch and lock versions are used for failure handling as described in Section 3.5.7 on Page 22. After a metadata server crash, the global lock state must be recovered from the clients.

The following sections suggest a mechanism to secure the distributed lock recovery in a way that does not impose substantial overhead and keeps the advantages of scalability and performance.

5.3.1 Mechanism Design

All three types of locks, namely session, data and range locks are involved with distributed lock recovery. Session locks must be protected in all modes since they themselves protect the application state. Data locks must be protected in the shared and exclusive modes because they themselves protect the distributed caching mechanism. Clean mode data locks do not need protection because of their read-only nature; modifications on the corresponding file system objects (i.e., directories and symbolic links) are based on a separate publish-subscribe mechanism. Range locks are protected by session locks and the overhead to protect the range locks too is not justifiable, especially because their handling is comparatively much more complex.

The key idea to making distributed lock recovery secure is to hand out a secure token to a client requesting a lock (see Figure 5.7 on Page 44). The token is secure because only the server is able to generate it from the lock state known to both the server and the client. This is achieved by introducing a server-side secret. On lock reacquisition, the client has to hand in the token together with the lock state it claims to recover. The server can now simply compare the provided token with the token derived from the provided lock state. The lock is granted again only if the tokens match.

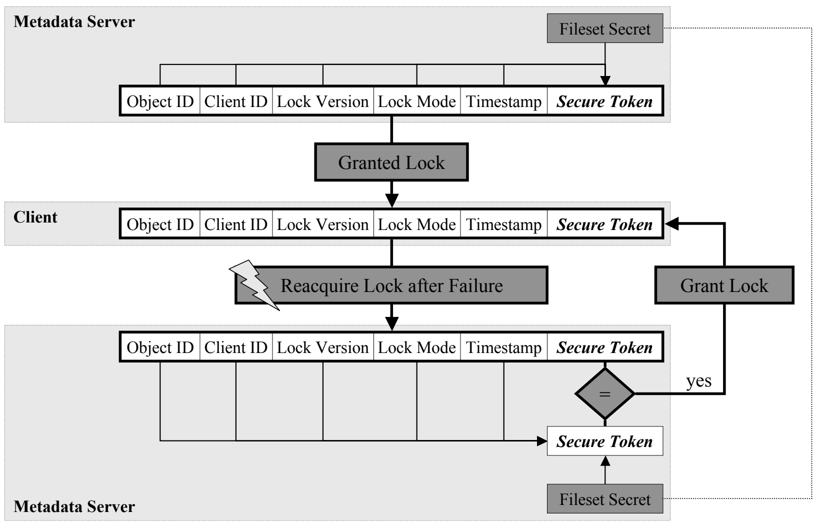


Figure 5.7: *The secure distributed lock recovery is based on a secure token that can only be generated at the metadata server for a given set of lock attributes (such as object identifier, lock mode etc.). On a lock reacquisition request, the metadata server can simply check the integrity of the attributes provided by the client. A client can thus no longer maliciously reacquire any locks.*

The authentication mechanism described in Section 5.2 guarantees the authenticity of the client identifier contained in the token. Thus, it is not possible for a third client to replay an eavesdropped token. Furthermore, the server provides a timestamp that is also contained in the token and changes whenever the lock mode changes into an incompatible mode (e.g., another client requests the data lock in exclusive mode). Only the client(s) with the latest timestamp will be allowed to recover the lock. If a malicious client tries to acquire an old lock, the server has its lease expired after the grace period, if another client reacquired a newer lock. Since a client is not allowed to perform any other action than lock reacquisition, this is safe. Naively, this would require waiting until the end of the grace period before actually granting any locks. As the SAN.FS protocol allows the revocation of locks by the server ”for administrative reasons”, the lock can be opportunistically granted and later revoked. Some implementations might also choose not to allow lock reacquisitions for locks older than t hours. Authentication and timestamps make the token replay-safe. Note that this implies a common accurate notion of time within the metadata server cluster.

The actual cryptographic method for the token is chosen such that the number of exchanged messages is not increased and the demand for message size and server processing capacity is minimal. Nevertheless, it must be strong enough to quench possible attackers. Keyed-hashing for message authentication (HMAC, [KBC97]) based on the secure hash algorithm (SHA-1, [EJ01]) offers several important properties:

Confidentiality It is impossible to figure out what data generated that digest.

Integrity It is essentially impossible to find another set of data to generate the same digest.

Size The digest has a fixed size, which is only a fraction of the digested data.

Performance The digest can be computed in reasonably fast time.

SHA-1 based HMAC digests can be applied as secure tokens as follows: The lock state information is hashed together with a persistent per-fileset secret key on the metadata server currently handling it. The format of the hashed data is given in Table 5.3.

Table 5.3: *The following format is hashed using SHA-1 based HMAC. The resulting digest has a size of 20 bytes. All types are described in [IBM03].*

Byte	Type	Description
1 - 20	`ObjID`	File system object identifier
21 - 28	`Uint64`	Client identifier
29 - 41	`[Sess|Data]LockVersn`	Epoch and lock version
42 - 42	`Uint8`	Lock mode
43 - 50	`Uint64`	Timestamp
51 - 66	`Uint8[16]`	Per-fileset secret
1 - 20	`Uint8[20]`	SHA-1 based HMAC digest

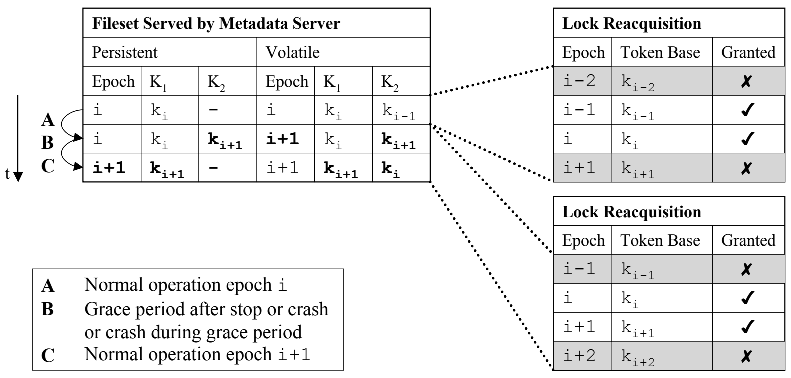


Figure 5.8: *The figure shows the different stages of fileset secrets for token generation and verification.*

Each fileset persistently stores two 128 bit secret keys K_1 and K_2. K_1 contains the value k_i for the current epoch i. K_2 is *null* if the last grace period successfully ended or k_{i+1} else.

Whenever a new instance of a metadata server starts to serve the fileset, the epoch number is incremented in memory. If K_2 is null, a new value k_{i+1} is generated and written to stable storage. If not, k_{i+1} is read from K_2.

During the grace period following the service startup, lock reacquisitions with the former epoch number i can be checked using k_i. Lock reacquisitions with an epoch number of the current epoch number can be checked using k_{i+1} and are an indicator, that the server crashed during the grace period. Other epoch numbers are not accepted for lock reacquisitions. If the lock is granted, the metadata server publishes a fresh token based on k_{i+1}.

If the server fails during the grace period, a new instance will restart the process again as described above and will be able to recover k_{i+1} from stable storage. When the grace period successfully ends, k_{i+1} is written to K_1 together with the new epoch number $i + 1$ and K_2 is set to *null* on stable storage.

Since according to the protocol, clients can reacquire locks also *after* the grace period (unless the lock was legitimately acquired by another client after the grace period had ended), the metadata server can still check tokens from epoch i with k_i kept in volatile memory. This behavior reflects the current protocol lock reacquisition semantics. Note that an implementation might choose to cache generated tokens to reduce the digest calculation overhead.

5.3.2 Protocol Design

Consecutively, all adapted or added protocol messages are listed and described. The original message formats can be found in [IBM03], Chapter 6, Message Formats. Some messages where split to reduce either the message size for the usual case or to save one token generation. An alternative could be to use a variable-size token type. This would later on allow exchanging the secure hash algorithm easily.

AcquireSessionLock

The fields containing the lock version and alternate client ID could be removed, because lock reacquisition is no longer handled with this message.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 60	`ObjID`	File system object identifier
		[DROPPED] Epoch and lock version
61 - 61	`Uint8`	Desired session lock mode
62 - 62	`Uint8`	Opportunistic correlated data lock mode
		[DROPPED] Alternate client identifier
63 -		Variable data

ReAcquireSessionLock

The new fields *timestamp* and *secure hash* are required for lock reacquisition.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 60	`ObjID`	File system object identifier
61 - 61	`Uint8`	Desired session lock mode
62 - 73	`SessLockVersn`	Epoch and lock version
74 - 74	`Uint8`	Opportunistic correlated data lock mode
75 - 82	`Uint64`	Alternate client identifier (under former lock)
83 - 90	`Uint64`	**[NEW] Timestamp**
91 - 110	`Uint8[20]`	**[NEW] Secure hash**
111 -		Variable data

AcquireSessionLockResp

The new fields *timestamp* and *secure hash* are required to propagate a new token to the client.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 53	`SessionLock`	Granted session lock
54 - 54	`Boolean`	Locking internals
55 - 172	`DataLock`	Granted correlated data lock
173 - 180	`Uint64`	**[NEW] Timestamp**
181 - 200	`Uint8[20]`	**[NEW] Secure hash**
201 -		Variable data

PublishLockVersion

The new field *secure hash* is required to propagate a fresh token to the clients. Since only the lock version was incremented, there is no need to transmit a fresh *timestamp*.

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
53 - 64	`SessLockVersn`	Epoch and lock version
65 - 84	`Uint8[20]`	**[NEW] Secure hash**

DemandDowngradeSessionLock

The new fields *timestamp* and *secure hash* are required to propagate an updated token opportunistically to the client. If the client denies the request, it must keep the old timestamp and secure hash. In case it complies, there is no need to transmit another message for the token.

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
53 - 53	`Uint8`	Desired session lock mode
54 - 54	`Uint8`	Locking internals
55 - 62	`Uint64`	**[NEW] Timestamp**
63 - 82	`Uint8[20]`	**[NEW] Secure hash**

DemandReleaseSessionLock

If the client complies, it must not be provided with a fresh token and timestamp. If it does not comply, it keeps the lock and the old token.

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
		[DROPPED] Desired lock mode
53 - 53	`Uint8`	Locking internals

DowngradeSessionLock

The message format does not need to be changed. The message semantics does not need to be changed if the downgrade is server-initiated by a Demand[Release | Downgrade]SessionLock message. In this case, the client already obtained the new secure hash.

When the client could spontaneously downgrade the lock to a weaker mode, it does not send a message to the server and keeps the old lock state. This avoids an additional message from the server to the client providing a fresh secure hash. If another client requests the lock in a stronger mode, the server will have to demand the lock first. This is known as being *lazy*. The hope is, that this rarely happens and the lock is either used in a stronger mode on the same client again or then completely released.

In case the client could spontaneously release the lock and therefore does not need a fresh secure hash, it can aggregate several locks and economically send a bulk release message to the metadata server (see BatchReleaseSessionLock).

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
53 - 53	`Uint8`	Desired new lock mode

BatchReleaseSessionLock

This message is used to release a bunch of locks at once. It efficiently aggregates several DowngradeSessionLock messages with the purpose to completely release a lock; the amount of messages and bytes transferred can thus be considerably reduced. The amount of locks released is implementation specific. The suggestion is to aggregate as few locks as possible (at least two) and send it as early as possible to allow for faster lock acquisitions by other clients and reduce server state but on the other hand to send as few messages as possible to save network bandwidth.

Byte	Type	Description
1 - 32	`MsgHdr`	[NEW] **Message header**
33 - 40	`Vector`	[NEW] **ObjIDs for which to release session lock**
41 -		[NEW] **Variable data of type:**
		[NEW] <`ObjID`>

AcquireDataLock

The fields containing the lock version and alternate client ID could be removed, because lock reacquisition is no longer handled with this message.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 60	`ObjID`	File system object identifier
61 - 61	`Uint8`	Desired data lock mode
62 - 62	`Boolean`	Locking internals
		[DROPPED] **Epoch and lock version**
		[DROPPED] **Alternate client identifier**

ReAcquireDataLock

The new fields *timestamp* and *secure hash* are required for lock reacquisition.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 60	`ObjID`	File system object identifier
61 - 61	`Uint8`	Desired data lock mode
62 - 62	`Boolean`	Locking internals
63 - 74	`DataLockVersn`	Epoch and lock version
75 - 82	`Uint64`	Alternate client identifier (under former lock)
83 - 90	`Uint64`	[NEW] **Timestamp**
91 - 110	`Uint8[20]`	[NEW] **Secure hash**

AcquireDataLockResp

The new fields *timestamp* and *secure hash* are required to propagate a new token to the client.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 158	`DataLock`	Granted data lock
159 - 166	`Uint64`	[NEW] **Timestamp**
167 - 186	`Uint8[20]`	[NEW] **Secure hash**
187 -		Variable data

DemandDowngradeDataLock

The new fields *timestamp* and *secure hash* are required to propagate an updated token opportunistically to the client. If the client is allowed to defer the request, it must keep the old timestamp and secure hash. Opportunistic demands must be handled with a *DemandReleaseDataLock* message since they do not require a fresh timestamp or secure hash.

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
53 - 53	`Uint8`	Desired data lock mode
54 - 54	`Uint8`	Opportunistic flag
55 - 62	`Uint64`	[NEW] **Timestamp**
63 - 82	`Uint8[20]`	[NEW] **Secure hash**

DemandReleaseDataLock

Opportunistic data lock demands must also be handled with this message.

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
		[DROPPED] **Desired data lock mode**
53 - 53	`Boolean`	Opportunistic flag

DowngradeDataLock

The message format does not need to be changed. The message semantics does not need to be changed if the downgrade is server-initiated by a Demand[Release | Downgrade]DataLock message. In this case, the client already obtained the new secure hash.

When the client could spontaneously downgrade the lock to a weaker mode, it does not send a message to the server and keeps the old lock state. This avoids an additional message from the server to the client providing a fresh secure hash. If another client requests the lock in a stronger mode, the server will have to demand the lock first. This is known as being *lazy*. The hope is, that this rarely happens and the lock is either used in a stronger mode on the same client again or then completely released.

In case the client could spontaneously release the lock and therefore does not need a fresh secure hash, it can aggregate several locks and economically send a bulk release message to the metadata server (see BatchReleaseDataLock).

Byte	Type	Description
1 - 32	`MsgHdr`	Message header
33 - 52	`ObjID`	File system object identifier
53 - 53	`Uint8`	Desired data lock mode
54 - 54	`Boolean`	Is access time that follows is valid
55 - 62	`Timestamp`	Optional new access time
63 -		Variable data

BatchReleaseDataLock

This message is used to release a bunch of locks at once. It efficiently aggregates several DowngradeDataLock messages with the purpose to completely release a lock; the amount of messages and bytes transferred can thus be considerably reduced. The amount of locks released is implementation specific. The suggestion is to aggregate as few locks as possible (at least two) and send it as early as possible to allow for faster lock acquisitions by other clients and reduce server state but on the other hand to send as few messages as possible to save network bandwidth. Note that the variable data consists of a complex type containing the additional information about access times as listed in DowngradeDataLock.

Byte	Type	Description
1 - 32	`MsgHdr`	**[NEW] Message header**
33 - 40	`Vector`	**[NEW] ObjIDs for which to release session lock**
41 -		**[NEW] Variable data of type:**
		[NEW] `<ObjID, Boolean, Timestamp>`

BlockDiskUpdateResp

The new field *secure hash* is required to propagate a fresh token to the client. Since only the lock version was incremented, there is no need to transmit a fresh *timestamp*.

Byte	Type	Description
1 - 40	`TxnMsgHdr`	Transaction message header
41 - 109	`BasicAttr`	File system object attributes
110 - 129	`Uint8[20]`	**[NEW] Secure hash**

5.3.3 Evaluation

The message sizes of the old and new messages are listed in Table 5.4 on Page 52. It is also listed whether a message requires the generation of a new token on the server side (to transmit it to the client or check the integrity of provided attributes). Note that *PublishLockVersion* messages are potentially sent to multiple clients simultaneously so that only one token must be generated for the whole set. The expectation is, that the overall network traffic is slightly increased. Nevertheless, the impact of the secure distributed lock recovery on message throughput over an IPsec-secured network should be very moderate.

Table 5.4: *Overview of adapted messages. Sizes are in bytes. Negative change indicates a decrease in message size.*

Message	Old Size	New Size	Change	New token
AcquireSessionLock	≥ 82	≥ 62	$\leq -24\%$	no
ReAcquireSessionLock	≥ 82	≥ 110	$\leq 35\%$	yes
AcquireSessionLockResp	≥ 172	≥ 200	$\leq 16\%$	yes
PublishLockVersion	64	84	31%	yes
DemandDowngradeSessionLock	54	82	52%	yes
DemandReleaseSessionLock	54	53	−2%	no
DowngradeSessionLock	53	53	0%	no
BatchReleaseSessionLock	≥ 106	≥ 80	$\leq -25\%$	no
AcquireDataLock	82	62	−24%	no
ReAcquireDataLock	82	110	35%	yes
AcquireDataLockResp	≥ 158	≥ 186	$\leq 18\%$	yes
DemandDowngradeDataLock	54	82	52%	yes
DemandReleaseDataLock	54	53	−2%	no
DowngradeDataLock	≥ 62	≥ 62	0%	no
BatchReleaseDataLock	≥ 124	≥ 80	$\leq -35\%$	no
BlockDiskUpdateResp	109	129	18%	yes

Both the client and the metadata server have to store additional data in volatile memory. The overhead is 28 bytes per lock, consisting of an eight-byte timestamp and a twenty-byte secure hash. This increases the *SessionLock* data structure of 13 bytes by 315% and the *DataLock* data structure of 118 bytes by 24%. The overhead increases linearly with the number of locks n. Note that a data lock mostly accompanies a session lock.

$$f(n) = 2n(28) \qquad [byte] \qquad | n \in \mathbb{Z}^+ \tag{5.2}$$

With 200’000 data and session locks, for example, the memory consumption on a client and metadata server would grow from 12.49MB to 17.83MB each. Lock quotas as discussed in Section 5.4 could set reasonable limits.

The worst-case scenario for performance is lock reacquisition after a server crash. Since the metadata server cannot verify the tokens from its memory, it must first generate a token to check the request and second generate a fresh token to transmit it back to the client. Measurements on an IBM xSeries 335 server with a XEON 2.8GHz showed an average token generation rate of 462’616 per second (token size is 66 bytes). This theoretically results in 231’308 possible lock reacquisitions per second. Measurements showed that without token generation or IPsec, the average session lock acquisition rate per second on a metadata server is 503, which is substantially slower. It follows, that the impact of token generation on the metadata server is in the per-mill-range if compared to the already existing workload.

5.3.4 Open Issues

The following is not covered and left for future work:

Secure Range Lock Recovery The complex management of range locks with implicit range splitting and merging on both the client and metadata server does not allow to apply the token mechanism without further analysis. Either an advanced mechanism can be found or it must be proven that it is not necessary to secure range lock recovery at all.

Message Probability Distribution There is no in-depth knowledge about the probability distribution of events such as lock release or (re)acquisition (on the same or another client). Without this information, the protocol cannot be optimized properly. Furthermore, it is not possible to analyze the impact of the increased message sizes.

Implementation Prototype A prototype for the IBM SAN.FS could not be implemented in the course of this work.

5.4 Quota Management

Quotas are a simple means to restrict access to resources on a per user or per client base. Sophisticated mechanisms can assure that the users are not too limited in their daily work without widely opening the door for malicious intentions. Quotas should be applied on all three lock types as well as on disk blocks. The described mechanism focuses on disk blocks only.

Basically, three types of quota management exist:

Static With static quota management, the limits are fixed by the administrator. The limit is either set for everybody (client or user) or can be configured individually. Each request to increase the quota results in a manual action on behalf of the administrator. This approach is simple to implement, efficient with respect to CPU and memory utilization, but it is neither flexible nor is it scalable.

Dynamic Dynamic quota management allows to automatically grow the limits over a certain amount of time. Implementations could use the token or leaky bucket algorithm described in [Tan96]. This allows limiting the growth to amounts acceptable for the whole system without requiring too many administrator interventions or reducing the flexibility of each client or user.

Elastic Elastic quotas are described in [LNZ+02] and are an interesting approach listed for completeness where the file system itself reclaims useless or temporary files according to an user-defined policy, if disk space becomes scarce.

5.4.1 Mechanism Design

The following algorithm falls in the category of dynamic quota management. To prevent a consecutive acquisition of an imaginary number of disk blocks followed by an immediate release through a file deletion, the rate, at which disk blocks can be acquired, must be limited. The upper limit of the rate is given by the amount of data a client can actually write to disk; the administrator can choose lower limits on a per client basis. Note that it should still be possible to allocate new disk blocks in (limited) bursts.

A leaky bucket fills a token bucket per client with tokens allowing the client to acquire a multiple of the block size. The leaky bucket generating the tokens assures, that the maximum rate of allocations is limited to the rate of the leaky bucket itself. The token bucket serves as a buffer to deal with peak allocation requests and prevents an unbound growth in case no tokens are used at all.

If the token bucket is empty, no disk block allocation requests are accepted anymore. The client must put its write operations on hold until it can proceed with allocating new disk blocks.

5.4.2 Implementation

For demonstration purposes and due to the limiting time constraints, only a small Java program was written to show, that a single client could consume all available disk blocks.

5.4.3 Evaluation

It could be observed that the acquisition of disk blocks is an expensive operation quickly bringing the metadata server to its limits. It could also be shown that a single client can consume all available disk blocks. If the disk block allocation happens at the rate the client actually can write the blocks (or even lower), the CPU utilization on the metadata server can be reduced.

5.4.4 Open Issues

The following is left to future work:

Protocol Design The described mechanism must be integrated into the existing protocol.

Implementation The metadata server must be extended to support quota management. The quotas should be configurable on a per user or per client base. It would be nice to make the quota policies pluggable to facilitate testing and evaluation.

Evaluation The impact of the quota management not only on the overall SAN.FS performance (which is to be considered as low) but also on the flexibility of the users must be evaluated. It must also be discussed whether a combination of dynamic and static quota management would be useful. The rate at which the leaky bucket generates the tokens as well as the size of the token bucket must be specified for several use cases. A realistic trade-off between flexibility and security must be found.

5.5 Data Lock Scheduling

Data lock trashing is a security issue because it basically leads to a denial of service. It has been observed, that file access on a ”trashed” file was prolonged by one or more orders of magnitude and that the CPU utilization on the metadata server was over 80% if only two clients were involved in such an attack.

5.5.1 Mechanism Design

The idea is to integrate a scheduling algorithm on the metadata server. Each request for a data lock must be queued. Only if the lock-scheduling policy agrees, it may continue and start the lock acquisition process. A client is not allowed to queue more than one request for a specific lock (see Figure 5.9).

The scheduling algorithm itself might vary depending on what should be achieved:

No Scheduling This is the current variant. If no trashing occurs, the lock can immediately be handed out to the requesting client. If multiple clients court for the same data lock, it is handed back and forth (trashed) between the requesting clients. If this happens, no client will really be able to continue its work. Even though the metadata server does not have to implement any scheduling algorithm or keep any additional state, it is kept busy by moving the lock around.

Fixed Scheduling Each data lock acquisition request is delayed on the metadata server for a fixed amount of time, if the lock is already held by another client. This allows a client to proceed immediately, if it is the only one to acquire a specific lock. The implementation is kept simple because the delay is static. The hope is, that the ratio of working to waiting time can be increased and that the metadata server is no longer under heavy load since it can work on other operations when a client was put on hold.

First-Prioritization Scheduling The client which first requested the data lock for a specific file is prioritized over all other clients. This means, that the delay for a lock acquisition is very small (or zero) if it is from the prioritized client or big, if not. The advantage is, that a good client can almost not be attacked if

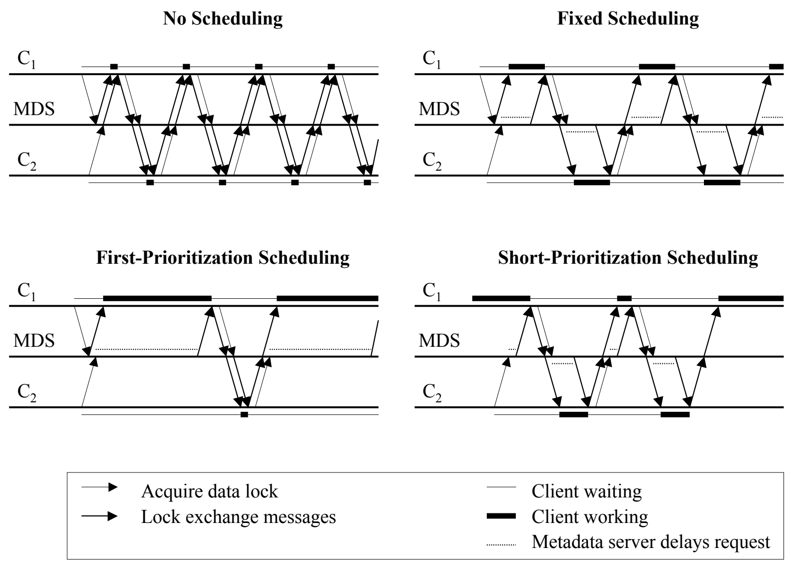


Figure 5.9: *Out of the possible variants for data lock scheduling, four are listed: First, the current non-scheduling protocol. Second, the fixed time scheduling. Third, the first-prioritization scheduling. Fourth, the short-prioritization scheduling. Each variant has its strengths and weaknesses.*

it was the first to acquire the lock. If the malicious client was first to acquire the lock, the scheduling will behave worse than in the no-scheduling case.

Short-Prioritization Scheduling Here, the idea is to favor the small (or short) jobs over the long ones to increase the overall throughput of jobs. This policy is also known as *least attained service* (LAS). The idea comes from TCP scheduling in routers described in [RBUK04]. From [ODH$^+$85] we know, that most files are only worked on for a very short amount of time, so it should do a good job for our purpose. Since it cannot be predicted how long a job will last neither on the client nor on the metadata server, another mechanism must be used. The idea is to keep a counter in the metadata server memory, which reflects the number of lock acquisitions a client made on a lock during a time interval. The next request is delayed inversely proportional to the counter of the current request. In other words, young counters are assigned a longer working time than old counters. It follows, that short jobs are favored over long jobs.

5.5.2 Implementation

Due to the limiting time constraints, scheduling could only be simulated for the fixed and first-prioritization scheduling scenario by delaying a data lock acquisition on the client. For demonstration purposes, a good client plays a movie and a Java program on a malicious client repeatedly acquires an exclusive data lock on the movie file, which causes data lock trashing and the film to flicker or freeze.

5.5.3 Evaluation

Even though no extensive performance measurements could be done, the impact of scheduling on the overall SAN.FS performance is considerable. The observation showed, that the current protocol version without scheduling resulted in freezing movies and heavy loads on the metadata server. As soon as the clients delayed their acquisition requests, the movie was playing slow but not freezing and the load on the metadata server dropped from roughly 80% to less than 10%.

5.5.4 Open Issues

The following is left to future work:

Scheduling Theory There was no in-depth theoretical evaluation of the different scheduling algorithms. It should be possible to find an optimal scheduling algorithm under the basic conditions of comparably long switching times and malicious operation.

Implementation The metadata server must be extended to support scheduling for data locks. It would be nice to make the scheduling policy itself pluggable to facilitate the evaluation and prepare the autonomous selection of the policy depending on the workload (e.g., many short jobs or only jobs from a single user).

Evaluation Each scheduling algorithm should be evaluated with different workloads and delay parameters to get a thorough understanding of the impact on the overall SAN.FS performance.

6 Virtual Machines

This chapter gives the motivation to introduce virtual machines (VMs) to the research within distributed storage systems and describes a specific product that was used in the course of this work.

6.1 Work With Distributed Storage Systems

It turned out, that the work with distributed storage systems was not effective and flexible enough. The setup of the distributed environment was time-consuming and tightly bound to the selected physical hosts. The sensitivity of the components regarding installed kernel versions, patches and other software was considerable. The fact that several people worked on the same set of hosts actually prevented them to develop, test or run performance experiments concurrently. Note that only one metadata server or client can be run on a single physical host. This resulted in trying out VMs, which were soon accepted as a convenient, flexible and swift tool.

6.2 A Convenient Tool

For the development of Linux-based SAN.FS prototypes, VMware Workstation 4.5[1] was used to host the actual Linux operating system. A set of VMs was produced, each containing a next step in the evolution towards a full client, metadata or storage server. These stages can conveniently be copied whenever a new VM is required in a specific stage. To emerge a running VM out of a selected stage, it can simply be copied to another directory. A commonly used feature is the snapshot and the corresponding revert operation to save a currently running VM and continue its operation later on – maybe on another physical host.

The following experience was made with the VMs:

Physical Requirements Per VM, 1GB or more of physical disk space is required, depending on the setup of the contained Linux operating system. At least 128MB of physical RAM must be available to run one VM without starting a time-consuming swapping process on the physical host. Depending on the workload of one VM, up to four VMs can be run on an average desktop computer. The limiting factor mostly is the available physical memory.

[1] `http://www.vmware.com`

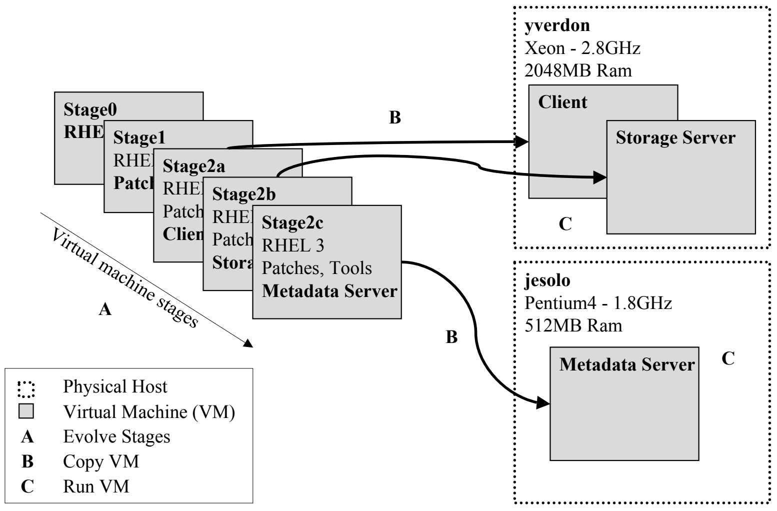


Figure 6.1: *Virtual machines can speed up research within distributed storage systems significantly.*

Long Term Stability A client, metadata and storage server VM setup was up and running for more than three days, without noticing any remarkable memory or CPU utilization on the physical host. It can be assumed that the VMs will be stable in the long run if the contained operating system is too. Note that the overall loss of performance is in the area of 10%.

Snapshot Mechanism The snapshot mechanism should not be used to make a running distributed system of clients, metadata and storage servers persistent. If the VMs are not reverted properly, all kind of errors might occur. It is recommended to stop running services such as iSCSI or metadata servers and only then perform a snapshot operation. Care must be taken not to overwrite an old snapshot in the course of some tests; only one snapshot can be taken per VM.

Specific Hardware The VMs can only offer a subset of the currently available hardware. E.g., 64bit support is only experimental and special SCSI RAID adapters are not supported.

Flexibility If VMware is installed on a set of physical hosts, the VMs required to run a specific configuration of clients, metadata or storage servers can be distributed flexibly based on the daily availability and needs of the whole group of developers. This feature turned out to be timesaving for setting up different performance measurement setups.

It can be concluded, that VMs are a valuable tool and can significantly speed-up the work with distributed storage systems. As it is with each tool, the operator must know, what it can do and where it's limits are.

7 Conclusions

As with each project, some answers can be found for a given question or task. In the majority of cases however, answers raise new questions and might open the door for new interests. This chapter summarizes the achieved work and projects some ideas onto the future.

7.1 Achievements

We identified a set of security threats that arise when SAN File System protocols for client to metadata server communication are opened to the insecure desktops. This work discusses and analyzes design modifications for client authentication, protocol encryption, and distributed lock recovery, which we partly implemented on a Linux-based SAN.FS environment. In addition, we cover quota management as well as lock scheduling, and introduce virtual machines as a valuable tool to support research within distributed storage systems.

IPsec was applied to provide authentication and encryption to SAN.FS on the network layer. Securing the network layer for client to metadata server communication assures that no one can learn or manipulate the global protocol state. Furthermore it is no longer possible to forge the identity of a trusted client. Performance evaluations state that the expected loss for the average use case is on the order of 11%.

Securing the distributed lock recovery with a replay-safe and efficient secure-token mechanism prevents malicious clients from breaking the global lock state after failures. It can be shown that there is no need to move to a centralized lock recovery mechanism, which performs much worse during normal operation, because the secure distributed lock recovery has a minimal impact on server-processing requirements, memory consumption and message sizes.

Both, lock scheduling and quota management could help on making SAN.FS more secure. Due to the time constraints, it could only be shown that data lock trashing and block acquisitions can be used for denial of service attacks – and that scheduling and quotas could limit the damage on the system.

Virtual machines can speed up research within distributed storage systems substantially. Once time-consuming setups can be deployed from appropriately prepared virtual machines within minutes. This allows multiple researchers to work on the same physical hosts with different setups of the distributed storage system at the same time. Varying performance measurements can be set up in virtually no time as well.

We conclude that there *are* security issues in SAN-based DFS that cannot be neglected if malicious operation must be taken into consideration. However, security *can* be improved considerably at a moderate price.

7.2 Future Work

Besides the open issues described in Chapter 5, the following ideas could motivate further research:

Protocol Simulator A protocol simulator could assist in quickly evaluating different protocol workloads (be they real or random) as well as protocol versions under varying environments. It would be instructive to now about the occurrences of the involved protocol messages and about the probability distributions of specific events (e.g., "the probability that a file is reopened on the same client after *t* seconds."). This knowledge would help to parameterize and optimize the protocol for general or specific workloads. Optionally, the simulator could test a protocol version for correctness.

Autonomous Protocol Adaption When the parameterizable portions of the protocol are known (e.g., data lock scheduling algorithm, number of locks released by a bulk lock release, etc.), the protocol could adapt itself autonomously to provide the best performance for specific use cases such as video streaming, mail servers or databases. Note that such optimizations could be applied at the fileset granularity.

Policy-Based Block Allocation The current block allocation algorithm does not take the characteristics of the storage devices into consideration (e.g., performance for varying access patterns or block sizes; long-term storage stability). Neither does the block allocation algorithm distribute the blocks in a high-level RAID fashion over the available disks to gain either additional performance or availability.

Mandatory Locking The current advisory locking is not sufficient to protect the data from malicious manipulation. It could be a challenge to find a scalable, well performing and secure distributed protocol that provides mandatory locking to enforce file access coordination.

A Task Description

A.1 Introduction

With the amount of data being stored, also the importance of this data to individuals, companies and governments is increasing. In addition to traditional criteria, such as performance, capacity, and reliability, security is rapidly becoming an important feature of storage systems. Large-scale storage systems, such as those used at CERN to record and evaluate experiment results, require huge amounts of fast storage space, which is typically provided by complex networked systems, where clients communicate directly with a disk over a network (as in network-attached storage (NAS), or in storage-area networks (SANs)).

To achieve maximum performance in such a system, access to the storage devices should be as direct and unfettered as possible. A successfull approach is to separate data and control planes, such as used in SAN.FS [IBM03]. There, the disks are accessible to the clients through a SAN, whereas metadata and mutual exclusion are communicated through a (cluster of) metadata server(s).

In general, SANs and related systems were (and still are) designed to be used in closely controlled, trusted environments, such as single server room under common management. The power of personal workstations and thus the need to move data quickly to many users' desks are increasing fast. CERN, for example, plans to give all users direct access to their multi-petabyte storage system, both from their sensors and data collectors and from the computational grid or workstations. All storage will be made accessible directly through iSCSI [SMS+04] (SCSI over IP). This will change the entire SAN paradigm, as no longer all machines can be considered fully trusted: Viruses, worms and other forms of malware are much more likely to appear on end-user machines; also the possibility of hacker attacks (insiders or outsiders) increases.

Therefore, a typical networked storage system has to be considered as consisting of three main components: multiple and potentially distrusting clients, the data stores themselves (typically, these are standard disks attached to a network, but they may also offer advanced functions), and metadata servers, which mediate access to the data stores. The data and metadata storage is still considered to be trusted, under common administration. However, clients and the network connecting the components have to be viewed as posing a potential security risk.

A.2 Protocol Issues

In earlier work, the problem of data integrity in such an environment has been investigated [Wag03]. Together with ongoing work on confidentiality protection in such an environment, that Diploma thesis addressed the data-related issues. What remains to be investigated are protocol-related issues, which is the area of this Masters thesis.

As the protocols were typically designed for trusted environments and stringent performance requirements, they also lack appropriate safeguards. For example, the locking/consistency protocols, which include a generalized file open/close management, enable clients to write-behind the data. While this improves performance, it requires full cooperation from all clients at all times. Malicious clients could navigate the metadata server into a state in ehich it no longer has control over the file. Because the system is asynchronous and clients may need to write-back (flush) large caches when the metadata server requests returning the lock, it is impossible to distinguish a slow client from a malfunctioning or malicious client.

Another aspect includes denial of service (DoS). Clients could overwhelm the servers with requests or cause them to establish state to exhaust their computational, networking, and storage resources. In a dynamic environment such as the one at CERN, static partitioning of resources is wasteful, in terms of both administrative overhead as well as storage space.

These problems are not limited to SAN.FS; similar problems also arise in NFS, including version 4 [S$^+$03], AFS [Zay91], and other networked file systems.

A.3 Task Description

As part of this work, you will investigate these protocol issues and evaluate their handling in different distributed file system protocols. You then propose mechanisms that help solve or at least mitigate these problems. You will evaluate the performance and reliability of the proposed mechanisms, taking into account three scenarios: (1) All client machines behave as they should (classical scenario), (2) a single machine becomes hostile after a break-in (”insecure desktop”), and (3) several machines can be hostile at different points in time (”Internet scenario”). How do the solutions differ? Potential workloads might be modeled after CERN with mixed real-time/best-effort environment. You are also requested to provide a description of the optimal misbehavior strategy the client(s) should follow to maximize service disruption to well-behaving clients (e.g., Byzantine fault model [CDK01]).

As part of your study, also think of the risks of incorporating these solutions (e.g., could the change cause protocol violations for good clients?) and their impact (performance, complexity, failure modes, recovery times, ...). Can anti-DoS mechanisms introduced into other protocols (e.g., TCP SYN-Cookies [Ber96]) help? What is the impact of your proposed changes onto the other security mechanisms?

Besides the theoretical study and observations, a prototype implementation based on the IBM SAN.FS as well as performance and functionality evaluations are expected.

A.4 General Comments

- Present a project time line after two weeks.
- There will be weekly meetings with the supervisor.
- Prepare a short intermediary report of 1-2 pages by mid-thesis.
- At the end of your thesis, the following has to be handed in: Source codes and a report (in English) including a one-page summary in both German and English.
- Source code is to be presented in machine-readable form (ASCII or UTF-8).
- The report should follow the rules of a scientific publication and include performance measures of the prototype. Two paper copies and a PDF document of the report have to be submitted. The summary is to be submitted in ASCII, UTF-8, or HTML as well.
- The final presentation of the project at both IBM and ETH is an integral part of the thesis.
- Development environment: Mainly Linux and C/C++, with sprinkles of Java and Python.
- Intellectual property and confidentiality rights and duties are handled in a separate contract.

A.5 Administrativa

Responsible Professor: Prof. Dr. Gustavo Alonso, ETH Zrich

Supervisor: Dr. Marcel Waldvogel, IBM Research GmbH

Start: 26 July 2004

End: 25 January 2005

B Mid Thesis Report

B.1 Where We Are

After being introduced to the IBM Zurich Research Laboratory during the first week that was seamless and comfortable I concentrated my work on theoretical research about distributed file systems including IBM SAN File System until the end of the second month. The goal was to get a thorough understanding of the problems and solutions in this field and to obtain the ability to find security issues in the SAN.FS. I also tried to immediately write down what I was doing and to prepare chapter by chapter in a pragmatic and logical way to the final paper.

During the third month I concentrated on finding security issues in the SAN.FS trying to systematically analyze failure conditions and malicious operation. Not all found issues however can be dealt with within this thesis regarding possible improvements or implementations; some of the options must be chosen for further investigation. We decided to start with these two tasks:

Encryption Implement IPsec and firewall rules to provide client to metadata server communication confidentiality and authentication and make a performance analysis. This task is completed to 75% by writing this report.

Data lock abuse Implement a scenario to show how a malicious client can interfere with normal operation of a trusted client, given that a malicious user got privileged control over a once "trusted" client and find, compare, and implement remedies. This task is not started yet.

On the practical side of the thesis I had to dive into the world of Linux, which was completely new for me and therefore challenging. Furthermore I have not before met such a large project written in C and C++. Even though I feel to advance much slower than I would have in a Java enterprise environment which I'm used to, the experience and lessons learned are invaluable for my future work in computer science.

B.2 Next Steps

For the remaining three months, the emphasis lies on finding theoretical remedies for the issues mentioned above, comparing and analyzing them as well as writing an implementation prototype to practically test them. The priorities are as follows (depending on the progress, the last ones might be dropped):

Encryption Complete the task described above. The remaining step is to implement the firewall rules.

Quotas The current quota granularity on a per fileset basis is not sufficient for many use cases. Try to find a conveniently simple but flexible solution to introduce quotas. This will be based on preliminary research and mainly consist of these two steps:

1. *Find concept* Find the best concept from literature for quotas in the SAN.FS and compare them if there are more than one solution applying criteria discussed above.
2. *Implement prototype* Implement the best found solution as a prototype.

Data lock abuse Complete the task described above following these steps:

1. *Proof of weakness* Setup environment and prove malicious operation influences normal operation. The basic idea is to have a trusted client showing a movie stored in SAN.FS while a malicious user tries to trash data locks by repeatedly request a lock interfering with the lock necessary for the movie file.
2. *Find remedy* Theoretically describe a solution or algorithm to avoid the trashing and compare them, if several exist, regarding complexity, protocol adaptation effort and performance impacts.
3. *Implement prototype* Implement the best found solution as a prototype.

Session lock abuse Try to find ways to prevent or reduce the possibility of abusing session locks following these steps:

1. *Proof of weakness* Setup environment and proof a malicious client (or even faster a set of them) can interfere with normal operation. The basic idea is to eagerly aggregate session locks on all files and not hand them back.
2. *Find remedy* Theoretically describe a solution or algorithm to avoid this and compare if several exist regarding complexity, protocol adaptation effort and performance impacts. As a side effect, the issue with session locks and normal failure conditions should be fixed.
3. *Implement prototype* Implement the best found solution as a prototype.

Finally, the thesis paper must be completed, together with the preparation for the final presentations.

C Summary

After more than two decades evolving a variety of client/server-based distributed file systems (DFS), the recently emerging storage area networks (SAN) allow the former file-server to be split into a storage and a metadata component. Metadata servers perform file access coordination and metadata management, whereas storage devices directly serve the clients' read and write requests. The clear separation of duties, the straight data path, and the virtualization of storage result in better scalability, performance, and maintainability. Whereas the first generation of SAN-based DFS focused primarily on performance, the second generation aims at spreading its service to the organizations' desktops, where server-room trust-levels can no longer be presumed.

We identified a set of security threats that arise when SAN File System protocols for client to metadata server communication are opened to the insecure desktops. This work discusses and analyzes design modifications for client authentication, protocol encryption, and distributed lock recovery, which we partly implemented on a Linux-based SAN.FS environment. In addition, we cover quota management as well as lock scheduling, and introduce virtual machines as a valuable tool to support research within distributed storage systems.

IPsec was applied to provide authentication and encryption to SAN.FS on the network layer. Securing the network layer for client to metadata server communication assures that no one can learn or manipulate the global protocol state. Furthermore it is no longer possible to forge the identity of a trusted client. Performance evaluations state that the expected loss for the average use case is on the order of 11%.

Securing the distributed lock recovery with a replay-safe and efficient secure-token mechanism prevents malicious clients from breaking the global lock state after failures. It can be shown that there is no need to move to a centralized lock recovery mechanism, which performs much worse during normal operation, because the secure distributed lock recovery has a minimal impact on server-processing requirements, memory consumption and message sizes.

Virtual machines can speed up research within distributed storage systems substantially. Once time-consuming setups can be deployed from appropriately prepared virtual machines within minutes. This allows multiple researchers to work on the same physical hosts with different setups of the distributed storage system at the same time. Varying performance measurements can be set up in virtually no time as well.

We conclude that there *are* security issues in SAN-based DFS that cannot be neglected if malicious operation must be taken into consideration. However, security

can be improved considerably at a moderate price.

D Zusammenfassung

Nach über zwanzig Jahren Entwicklung an Client/Server basierten, verteilten Datei Systemen (VDS), erlauben die neulich eingeführten Storage Area Networks (SAN), den Server in eine *Metadaten* und eine *Daten* Komponente aufzuteilen. Metadaten Server synchronisieren den Dateizugriff und verwalten die Metadaten. Die Daten Einheiten bearbeiten Lese- und Schreibzugriffe. Die klare Aufgabentrennung, die direkten Datenpfade und die Virtualisierung des Speichers resultieren in verbesserter Performanz, Skalierbarkeit und Wartbarkeit. Nachdem sich die erste Generation von SAN-basierten VDS vorrangig auf die Performanz konzentrierte, will man in der zweiten Generation die Sicherheit berücksichtigen, um möglichst vielen Clients den Zugriff auf das VDS gestatten zu können.

Eine Reihe von Sicherheitsproblemen taucht auf, wenn man SAN-basierte VDS im Bereich der Client zu Metadaten Server Kommunikation über viele, unsichere Rechner verteilt. Diese Arbeit diskutiert und analysiert Anpassungen im Bereich von Client Authentifizierung, Protokoll Verschlüsselung und Lock Wiederherstellung, die wir teilweise in einer Linux SAN.FS Umgebung implementiert haben. Zusätzlich streifen wir Quota Management und Lock Scheduling und führen Virtuelle Maschinen als wertvolles Hilfsmittel ein.

Um Authentifizierung und Verschlüsselung zu ermöglichen, wurde IPsec auf der Netzwerk Schicht aktiviert. Dies stellt sicher, dass niemand den globalen Protokoll-Zustand in Erfahrung bringen oder manipulieren kann. Des weiteren ist es nicht mehr möglich, die Identität eines vertrauenswürdigen Clients zu fälschen. Eine Evaluation der Performanz hat gezeigt, dass der zu erwartende Verlust für den durchschnittlichen Anwendungsfall im Bereich von 11% liegt.

Die verteilte Lock Wiederherstellung kann mittels eines Token Mechanismus gesichert werden, der die Manipulation des globalen Lock Zustandes nach einem Fehler verhindert. Es kann gezeigt werden, dass ein Wechsel zur langsameren zentralisierten Lock Wiederherstellung unnötig ist, weil der vorgeschlagene Mechanismus auf Rechenleistung, Speicherverbrauch und Kommunikation praktisch keinen Einfluss hat.

Virtuelle Maschinen (VM) können die Forschung mit VDS massgeblich beschleunigen. Vormals zeitraubende Installationen können mittels vorbereiteten VMs praktisch innert Minuten aufgesetzt werden. Das erlaubt die gleichzeitige Arbeit mit unterschiedlichen Installationen auf einer physikalischen Maschine. Unterschiedlichste Szenarien können ebenfalls ohne grossen Zeitverlust aufgesetzt werden.

In SAN-basierten VDS existieren zusammenfassend Schwachstellen, die böswillig ausgenutzt werden können. Die Sicherheit kann aber nachhaltig verbessert werden – bei moderaten Kosten.

Bibliography

[ADN$^+$96] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14:41–79, February 1996.

[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.

[APS99] Apostolopoulos, Peris, and Saha. Transport layer security: How much does it really cost? In *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*, 1999.

[AS99] Yair Amir and Jonathan Stanton. Lecture 12, 1999.

[Ber96] D. J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1996.

[BHJ$^+$93] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Palo Alto, CA, USA, October 1993.

[Bra03] Peter J. Braam. The Lustre Storage Architecture. `http://www.clusterfs.com/docs/lustre.pdf`, 2003.

[C$^+$04] Per Cederqvist et al. Version management with CVS. Technical report, Concurrent Versions System, 2004.

[Cal00] Brent Callaghan. *NFS Illustrated*. Addison-Wesley, 2nd edition, 2000.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 3rd edition, 2001.

[CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. 2004.

[EJ01] D. Eastlake and P. Jones. Us secure hash algorithm 1 (SHA1). Technical Report 3174, September 2001.

[FS00] N. Ferguson and B. Schneier. A cryptographic evaluation of IPsec. Technical report, 3031 Tisch Way, Suite 100PE, San Jose, CA 95128, USA, 2000.

[GMSP00] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18:127–153, May 2000.

[IBM03] IBM. IBM TotalStorage SAN File System. `http://www.ibm.com/servers/storage/support/virtual/sanfs.html`, April 2003.

[JPPMAK03] Ricardo Jimenez-Peris, Marta Patino-Martinez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28:257–294, September 2003.

[Kat97] J. Katcher. Postmark: a new filesystem benchmark. `http://www.netapp.com/tech_library/3022.html`, 1997.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical Report 2104, February 1997.

[Lee00] Yui-Wah Lee. *Operation-based Update Propagation in a Mobile File System.* PhD thesis, The Chinese University of Hong Kong, Department of Computer Science and Engineering, January 2000.

[LNZ$^+$02] Ozgur Can Leonard, Jason Neigh, Erez Zadok, Jefferey Osborn, Ariye Shater, and Charles Wright. The design and implementation of elastic quotas. Technical Report CUCS-014-02, Columbia University, June 2002.

[LRST00] Felix Lau, Stuart H. Rubin, Michael H. Smith, and Ljiljana Trajovic. Distributed denial of service attacks. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 2275–2280, Nashville, TN, USA, October 2000.

[Mac94] R. Macklem. Not quite NFS: Soft cache consistency for NFS. pages 261–278, San Francisco, California, January 1994.

[MBH93] Timothy Mann, Andrew Birrell, and Andy Hisgen. A coherent distributed file cache with directory write-behind. Technical report, Digital Systems Research Center, Palo Alto, 1993.

[Mic04] Microsoft. Microsoft SMB protocol and CIFS protocol overview. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/microsoft_smb_protocol_and_cifs_protocol_overview.asp`, 2004.

[MIK02] S. Miltchev, S. Ioannidis, and A. Keromytis. A study of the relative costs of network security protocols, 2002.

[ODH$^+$85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 15–24. ACM Press, 1985.

[RBUK04] Idris A. Rai, Ernst W. Biersack, and Guillaume Urvoy-Keller. Size-based Scheduling to Improve the Performance of Short TCP Flows, November 2004.

[S$^+$03] Spencer Shepler et al. Network file system (NFS) version 4 protocol. Internet RFC 3530, April 2003.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[SKK$^+$90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. CODA – a highly available file system for a distributed worksation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SM89] V. Srinivasan and J. D. Mogul. Spritely NFS: Experiments with cache-consistency protocols. pages 45–57, Litchfield Park, Arizona, December 1989.

[SMS$^+$04] Julian Satran, Kalman Meth, Costa Sapuntzakis, Efri Zeidner, and Mallikarjun Chadalapaka. Internet small computer systems interface (iSCSI). Internet RFC 3720, April 2004.

[SNI02] SNIA. Common internet file system (CIFS) technical reference. `http://www.snia.org/tech_activities/CIFS`, 2002.

[SS96] Mirjana Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, 1996.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[TDG98] R. Thayer, N. Doraswamy, and R. Glenn. Ip security. Internet RFC 2411, November 1998.

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[Wag03] Daniel Wagner. Integrity protection for secure networked storage. Diploma thesis, Swiss Federal Institute of Technology Zurich, 2003.

[Zay91] Edward R. Zayas. AFS-3 programmer's reference: File server/cache manager interface. Technical Report FS-00-D162, Transarc Corporation, Pittsburgh, PA, USA, August 1991.