Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Security for the XtreemFS File System

D3.5.7

Due date of deliverable: 31/05/2008
Actual submission date: 30/05/2008

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: 3.5
Task number: 3.5.9

Responsible institution: Contributions by SAP Research, STFC, XLAB
Editor & and editor's address: Adolf Hohl

Version 2.4 / Last edited by Adolf Hohl / 30/05/08

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	√
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	11/02/08	Adolf Hohl	SAP	Initial Version
0.11	17/04/08	Erica Yang	STFC	Reorganize integration
1.0	27/05/08	Philip Robinson	SAP	Corrections after internal review

Reviewers:
Adrien Lebre (IRISA), Haiyan Yu (ICT)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.5.7	Integration	STFC, XLAB
T3.5.8	VO Lifecycle Management Systems	STFC, XLAB
T3.5.9	Security for Data Management	SAP*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

List of Figures

1 Executive Summary

XtreemFS is a distributed, object-based storage system, built on a network of Object-Based Storage Devices (OSD), made accessible via capablities issued by a central Metadata and Replica Catalog (MRC), enabling the creation of self-managed, heterogeneous, shared storage by moving low-level storage functions into the storage device itself and accessing the device through a standard object interface rather than a traditional block-based interface such as SCSI or IDE[7]. XtreemFS defines the concepts and mechanisms towards distributed data management within the XtreemOS project. The concern for XtreemFS in this deliverable is that of secure data management.

Besides robust security, we face the challenge of making this into a truly viable technology for storage and retrieval of mass distributed data objects, with frequent and sporadic access patterns. There are various applications in need of such data management technology, but each also require that the storage, access and transport of the application data be done with the assurance of confidentiality and integrity yet without imposing unacceptably on the performance of the application.

At the root of these problems is robust and efficient capability and key management. Capability management refers to how rights are issued to Clients by the MRC, requesting access to files that are stored on one or more OSDs. Capabilities must be correct, unique and unforgeable, such that availability, integrity and confidentiality of files can be assured and validated. Cryptographic keys and algorithms are the instruments and mechanisms employed in order to achieve robust capabilities. However, there is a performance penalty to be paid whenever cryptographic schemes are introduced. These performance bottlenecks can be introduced via more scalable capability and key management i.e. avoidance of exponential key explosion and/or by selecting a cryptographic algorithm with simple computations. Both avenues have been explored within the document. With respect to efficient capability management, the work from Leung [11, 12] and earlier work from Miller et al.[13] have been referenced. They suggest the usage of capability groups in order to avoid the capability explosion problem, which can negatively impact on issuing, validation and revocation procedures. The second suggestion we make for secure data management with XtreemFS is to explore the performance of capability management with less expensive cryptographic algorithms such as Elliptic Curves. Further investigation into the drawbacks of this technical approach are still to be conducted.

Finally, the deliverable discusses the way in which XtreemFS is integrated with the existing XtreemOS security services, using scenarios taken from job submission.

2 Introduction

XtreemFS is an object-based distributed file system. In addition to distributed file accessing, it also offers advanced file management features, such as facilities to enable inter-process communication for applications (Object Sharing Service), and fault-tolerant file replication service (Replica Management Service). Two challenges for implementing XtreemFS are making it robust yet efficient. However, the conflict between performance and security is a well-known dilemma for security engineering and administration, where the security mechanisms selected to protect a system must be done so in a manner that is aware of the performance expectations of the software. On selecting security mechanisms, they may still need to be tuned in order to satisfy the performance objectives of the system as well. It is this particular problem that we see as fundamental to the success of XtreemFS as a distributed, peta-scale object-based distributed system.

The correlation of security and performance can be expressed in a simplified way as in figure 1. The correlation itself is characterized by parameters of a solution, multiple available solutions and combinations of these.

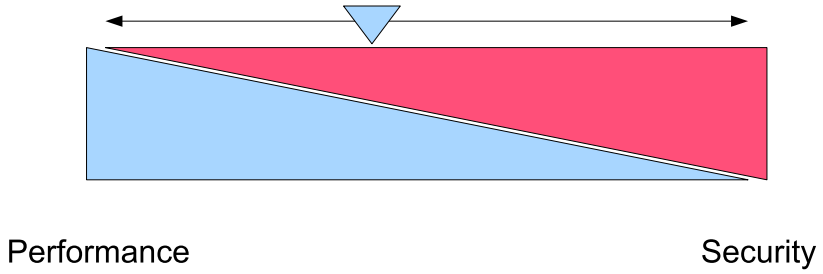

Figure 1: Security comes at a price

This deliverable is about providing several security solutions in the area of object based storage system which could be adapted for XtreemFS and be rated in terms of performance and security. The solution map given is a two dimensional space. Alternatives are either the concepts/variants and the algorithms which could be used for their implementation.

2.1 XtreemFS Architecture and Security Overview

2.1.1 XtreemFS Components

Three components are typically involved in accessing (i.e. create, delete, read, write) a XtreemFS file. They are:

- Metadata and Replica Catalog (MRC)

- Object Storage Device (OSD)
- XtreemFS client (XFS client)

To be self-contained, we have included the description of MRC, OSD and XFS client from the WP3.4's deliverable - D3.4.1 [4]. See also Figure 2 for the relation of these components.

The **MRC** *is responsible for maintaining all file system metadata, extended (user defined) metadata as well as information on replica locations. It also hosts access control policies and makes authorisation decisions.*

The task of the **OSD** *is to provide functionality for data access in the file system. It offers an object-based storage interface to hide the complexity associated with underlying block-based storage mechanisms. Capabilities of the component include read and write access, concurrency control and communication with remote storage hosts.*

(XtreemFS) Clients *are hosts running components of the access layer, i.e. the file system adapter or the XtreemFS library. Applications and user processes use the access layer to communicate with XtreemFS components. This can be done transparently to the application through the traditional Linux file system interface. XtreemOS aware applications can take advantage of the native XtreemFS interface through a library provided by the access layer.*

Figure 3 illustrates the interactions among MRC, OSD, and XFS client (aka. XFSc). The following interactions between XtreemFS components are originated from our discussion with WP3.4. Specifically, it was first presented in an XtreemOS internal workshop between WP3.5 (security) and XtreemFS (workshop took place in October 2007). In the figure, we have abstracted away the *internal details* of XtreemFS, such as how XtreemFS makes use of the Virtual File System (VFS) to implement a uniform file access interface and how it relies on the Filesystem in Userspace (FUSE) module to intercept file operations to the kernel. Instead, we continue to use XFSc to represent all these internal complexity of XtreemFS. Therefore, from the perspective of WP3.5, the following steps are involved in a typical file operation with XtreemFS:

1. A file (operation) request (e.g. create, open, delete, read, write a file), initiated by a user (application) process, comes into the XFSc via the Linux kernel[1].
   - 1.1 The XFSc requests local credentials from the FUSE daemon by passing in the PID of the calling process
   - 1.2 The XFSc receives UID/GIDs back

[1]This kernel must be compiled with the Filesystems in Userspace (FUSE) kernel module and support the Virtual File System. See the deliverable D3.4.1 [4] and D3.4.2 [5] for more details

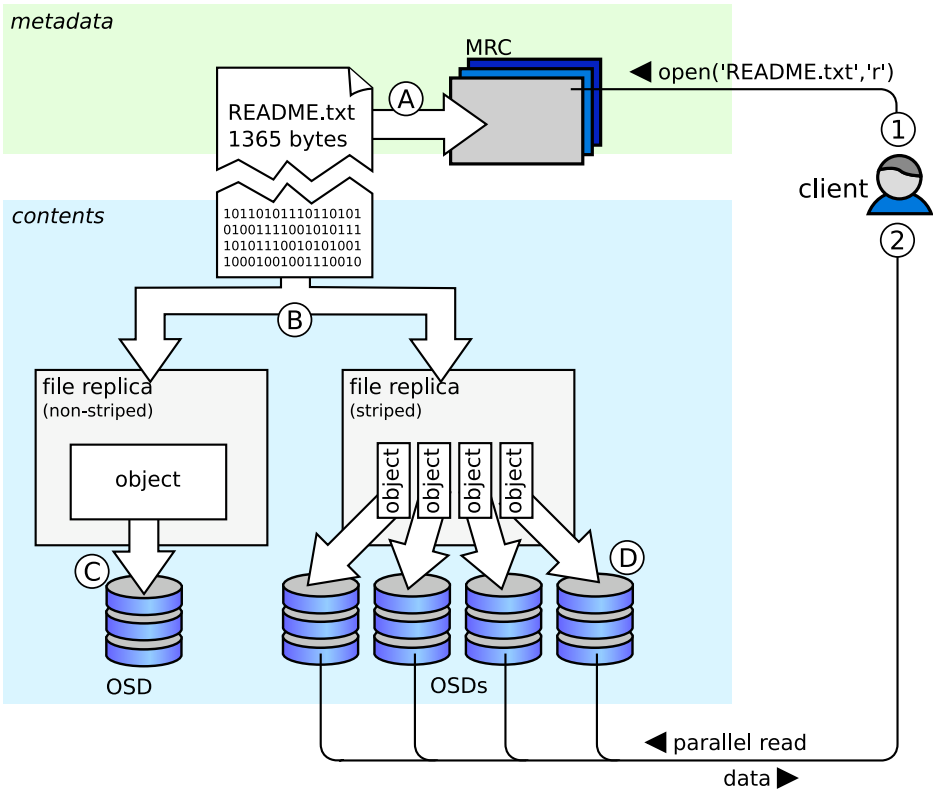

Figure 2: Overview of the components of the XtreemFS file system. It showcases also the architecture of most other approaches mentioned in this document.

2. The XFSc asks the MRC whether the operation is permitted.
3. If the operation is allowed, the MRC returns with a set of capabilities. Otherwise, the operation is deemed as denied.
4. With the capabilities, the XFSc contacts the appropriate OSD(s) (Only one OSD is contacted in the diagram to illustrate the scenario. But in reality, a file may be stored in multiple OSDs.)
5. The OSD(s) transfers the file to the XFSc. If this process fails, errors will be reported.
6. XFSc presents the file back to the user process.

2.1.2 XtreemFS Security

In this section we list the known threats an access control mechanism in the XtreemFS file system has to be able to deal with. The assumption is at this point a trustworthy client. We don't consider the case of e.g. an illegal login from client or a rogue user process e.g. a trojan.

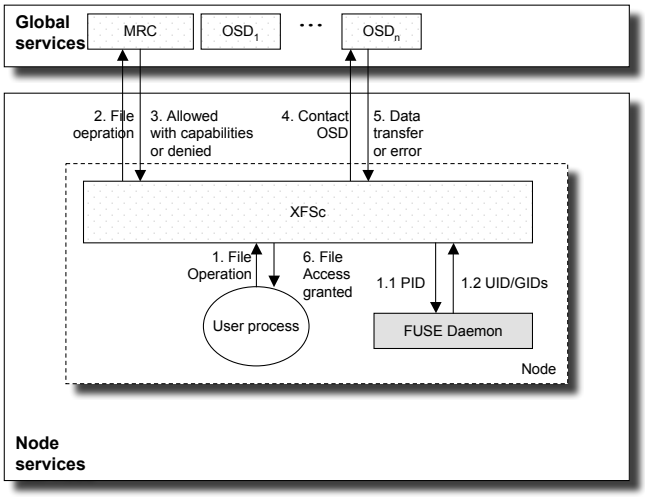

Figure 3: Interactions of an XtreemFS client with the MRC and the OSDs in the file system. The sequence of interactions shows the rough protocol flow in systems like these. It is assumed here, that a user is authenticated and checks via the MRC if he can get an authorization token to access data on the OSDs.

1. Man in middle or impersonation attacks that cause MRC to issue capabilities to a false client.

2. Man in middle attack between Client and MRC, such that capabilities can be intercepted, stolen or replaced in communication.

3. An attacker is able to gain access to an OSD and potentially corrupt its objects or read illegally.

4. Corrupted or malicious data is stored on OSDs and potentially returned to clients.

5. An attacker is able to read confidential contents or a user receives corrupted data without being able to validate it.

In order to achieve these goals, mechanisms from classic cryptography are applied, as listed below:

- Authentication of messages.
- Hashing of messages.
- Generating signatures.

- Encryption schemes.
- Replay detections.

There are also some preliminary assumptions and properties that must be explicitly ensured during system set up:

- All Clients are authenticated with the MRC such that the MRC knows all Clients and all Clients know the MRC based on a either a shared key $K_{C,MRC}$ or they have exchanged public keys K_C and K_{MRC} respectively.
- The MRC can "speak for" any Client in the file system to OSDs, such that an OSD believes that any capability signed by the MRC and issued to a Client C, really represents a valid access to an object.
- The MRC can "speak for" any OSD in the file system to Clients, such that any OSD returned as a response to an open file request is assumed to be authentic by the requesting Client.

2.1.3 XtreemFS Performance

The usage scenario of XtreemOS covers applications that exploit a huge amount of nodes and a huge amount of data. What this means in detail depends on the concrete application, but a scale in the region of thousands of nodes and data servers is a reasonable estimate. Thus XtreemFS needs to be fast in terms of throughput for data and computation intense applications. On the other hand the use cases describe the execution of database applications as well which need a very low latency in order to run fast enough.

The applications themselves may organize their data according to the design and implementation of their developers. There are no restrictions on the amount of files or their size. Thus we can conclude that the amount of files may be huge, ranging upwards to several millions of files, amassing a volume to the order of peta-bytes. Furthermore the metadata operations have to work very effectively to support handling this amount and volume of files efficiently.

Especially on the "first touch" of files, a metadata operation is necessary for the purpose of creating the capability required for granting access to the requested file and the requested privilege. It is very important, that the secure generation and issuing of capabilities in the metadata operations is fast for low latency and has a high throughput in terms serving capabilities to a huge amount of authorized requests. Consider that the amount of requests must scale with the number of clients, files, requests and replicas of files. The metadata server or servers are hence at risk of becoming the overall bottleneck in the file system, causing the throughput not to scale upwards.

2.2 Contributions of Deliverable

The document addresses the problem of efficient security for XtreemFS, as outlined in the introductory paragraphs. It is intended that this document supply knowledge that guides the continued architectural and technology decisions made within XtreemOS, in particular with regards to the integration of XtreemFS as the file system.

The first set of knowledge provided by the document surrounds the topic of capabilities and capability management. This topic covers related work in the area of generating, distributing, validating and revoking capabilities in distributed, object-based file systems. The topic of capability management could be addressed in a broader context such as distributed systems in general, but we chose to limit our discussion to closely related work in the area of object-based storage and networked storage devices.

Secondly, one means of achieving more efficient capability handling, besides the approaches discussed in the capability management section, is to change the actual cryptographic algorithms and mathematics being applied to one with lower computational demands. By considering various, existing cryptographic approaches that aim to be more efficient, we identified that ECC (Elliptic Curves Cryptography) is still a frontrunner with respect to speed and maturity. We therefore describe the way it works and consider past experimental results, in order to show that it can achieve the objectives set forth for a more efficient metadata server and capability handling.

Finally, the document describes how to integrate the XtreemFS file system with the existing VO and user management security services. This is done by considering integration with the Application Execution Management (AEM), component when doing job submission. This is an important integration use case to consider, as XtreemFS will need to scale with the number of jobs and volumes of data the AEM components handle. Security should not be a bottleneck in these processes, given the service level agreements that will be in existence.

3 Capabilities and Capability Management

In this section, the requirements for smart capability management for object storage are described. Secondly, related work that further validates these requirements and offers some solutions are provided.

The classical challenge facing access control in distributed parallel file systems is the large amount of numbers that have to be multiplied to guess the complexity of access control decisions and expressions in the form of a capability.

The amount of capabilities in a distributed parallel file system are characterized by the amount of:

- **Users**: capabilities are traceable to unique User identities
- **Clients**: capabilities may be bound to Client identities
- **Files**: capabilities are issued for particular files and particular access modes
- **OSDs**: capabilities may be bound to particular OSDs
- **Objects per file**: capabilities may be issued at a finer grain for access to objects as opposed to coarse-grained access to files or volumes

In the evolved design of using stateless OSDs to store data and the MRC or Metadata Directory Server (also referred to as "MDS" in most other approaches) mitigating between Client requests and the OSDs the MRC/MDS is at risk to become the bottleneck.

3.1 Requirements

Since the capabilities have a semantic meaning and are used to express that its owner is allowed to perform a set of actions with a certain object, there are a set of mandatory requirements. They have to be:

- Provably bound to client and MRC identities.
- Issued at the granularity of object blocks on a particular object storage device; capabilities should be for the most part self-contained such that there is little need for an OSD to reference many other tables or data structures when executing client requests. All meta-data is maintained by the MRC, which should be capable of maintaining the object map.
- Uniqueness and freshness such that no replay is permitted.
- Fast generation and scalable storage.

- Revocation should be implicit (i.e. time out) as opposed to explicit broadcasts by the MRC or constant polling from OSD to some central capability management server (which may or may not be the MRC).

A lot of research was and is being done in the field of distributed parallel file systems with the advent of GRID computing and the needs for processing very large data sets. However security in these distributed system built using cost efficient off the shelf hardware seemed to be neglected. Since large GRID systems exceed a single trust domain and span over multiple administrative domains and since distributed file system approaches find their way into the datacenter, security becomes a major issue.

The major findings of these efforts are presented in the related work section.

3.2 Related Work

This section presents the major findings of research done in the field of secure access to distributed parallel file systems. Due to the large extent distributed file systems can grow and due to the fact of their distribution this task is more difficult than securing a file system in the past.

Substantial work was done by Miller et al.[13]. They formed the term of a Secure Network Attached Disks (SNAD) and evaluated three different schemes;

SNAD 1 The first SNAD scheme provides security on each block of data. Writes in this scheme encrypt each data block, compute a hash over the entire data object (including the metadata), and sign the hash using the user's private key. This hash can then be verified by anyone with the user's public key. In particular, the disk can recompute the hash and compare it against the hash signed by the user who sent the block. If they match, the disk successfully verifies the provided signature, and the user has the permission to write the file, the SNAD server writes the block to disk. The block security information for this scheme thus consists of a signed secure hash. Reads in this scheme require no operations by the SNAD server CPU, but do require that the client CPU check the hash and signature just as the SNAD server did on a write. Additionally, the client must decrypt the data.

SNAD 2 Scheme 2 replaces the SNAD serverŠs signature verification with a HMAC (Hashed Message Authentication Code)[2]. In this scheme, the client performs a cryptographic hash on the block and signs it. However, this signed hash, which is stored with the secure block, is only verified by the client when it reads the block. The client also calculates an HMAC on the secure block using the secret HMAC key it shares with the server and sends the HMAC to the SNAD server. The SNAD server computes an HMAC using the shared secret key from

the certificate object and checks it against the HMAC received from the client. Recalculating the entire hash including the HMAC key would be time-consuming; instead, the client simply performs an HMAC over the hash. The replacement of a signature verification by an HMAC reduces the load on the SNAD disk CPU, but does not reduce the load on the client CPU, which still must perform signatures on writes and verifications on reads.

SNAD 3 The third scheme uses a keyed-hash (HMAC) approach to authenticate a writer of a data block and verify the blockŠs integrity. HMACs differ from signed hashes in that a user able to verify a keyed-hash is also able to create it. Scheme 3 still uses public-key authentication for key objects because writing key objects, while slower with public-key controls, is very infrequent. Write operations in this scheme require the client to encrypt the secure block and calculate an HMAC over the ciphertext. This information is then sent to the disk, which authenticates the sender by recomputing the HMAC using the shared secret key from the certificate object. If the write is authentic and the user has the permissions to modify or create the secure block, the SNAD disk commits the write to disk, updating structures as necessary. Note that the disk does not store the HMAC because it must recalculate a new HMAC if the reader is a different user from the user who wrote the block. Unlike the previous two schemes, this scheme requires the SNAD disk to perform a cryptographic operation on a read: the disk must calculate a new HMAC using the key from the user requesting the data. The data object, along with the new HMAC, is then sent to the client requesting the data. If the disk were forced to write blocks without the proper encryption key, a client could detect this during a read by recomputing the non-linear checksum over the cleartext and comparing it to the stored checksum.

Aguilera et al.[1] implement a capability-based schema for network-attached disks (NADs) with capability groups for managing revocation and Bloom filters for replay detection. NADs are storage devices that accept block read/write requests over a network. Object Storage Devices (OSDs) are similar to NADs from a networking perspective, but NAD requests refer to blocks on the disks as opposed to object identifiers. OSDs are therefore more portable than NADs, as block identifiers are bound to the platform. Nevertheless, the security problem remains the same in both systems. In the work of Aguilera et al., they assume that servers and disks are trusted, while clients connecting to the NADs may be compromised. Secondly, they assume that the network is not secure, which is one of the assumptions that we also make as the basis of our design. One of the major concerns of capability-based approaches to security is the demands on RAM usage and access. Depending on the granularity, access frequency of objects and

lifetime of capabilities, large amounts of RAM might be required for storing capabilities. In addition, they may have to be persisted on the storage devices (NADs, OSDs), cached at Clients or at the Filesystem Server (i.e. MRC) in order to maintain security and performance of the distributed filesystem. The capabilities used by Aquilera et al. have the following schema:

```
CAP c = {
    capID: <identifier for the capability>,
    groupID: <group to which capability belongs>,
    diskID: <target disk>,
    extents: <physical block range or block-map>,
    mode: <read, write, rw>
}
```

The protocol for interaction between the components in the NADs system they assume is shown below, but this can be generalized for any storage area network including OSDs. NADs and OSDs are represented as Disks (D_1 - D_n) in the protocol, while blocks and objects are referred to as targets (B) and the MRC is the server. The protocol is discussed afterwards.

1. Client $\rightarrow$ MRC: (open (target, req))/$K_{MRC,Client}$;
2. MRC $\rightarrow$ Client: (CAP(target, req), s = h(CAP, $K_{MRC,D}$)) /$K_{MRC,Client}$;
3. Client $\rightarrow$ D_1: (req, CAP, h(req, s))/$K_{D_1,Client}$;
4. ON valid(CAP): D_1 $\rightarrow$ target: execute(request);
5. Client $\rightarrow$ D_n: (request, CAP, h(req, s))/$K_{D_n,Client}$;
6. ON valid(CAP): D_n $\rightarrow$ target: execute(request);
7. D_1 $\rightarrow$ Client: (response, h(resp, s))/$K_{D_1,Client}$;
8. D_n $\rightarrow$ Client:(response, h(resp, s))/$K_{D_1,Client}$;

Step 1 shows the Client making a request to the MRC, where it is assumed that there is a key $K_{MRC,Client}$ shared between the Client and the MRC in order to create a secure channel. Otherwise, if may be assumed that the network between the Client and the MRC is secured. In one configuration, every network domain may have an on-site, trusted MRC. In the case that they have a shared filesystem, it is however expected that there is one MRC. In step 2, the MRC responds to the Client with a capability CAP(target, request) that gives the Client the permission to execute the request on the stated target. In addition, in step 2, the key $K_{MRC,D}$

is a shared key between the MRC and a disk D, while a secret s is generated at the MRC by performing a keyed hash of the capability CAP using the key shared with D. This serves to generate both a capability that is unforgeable, proves that it is created by the MRC (or by the disk) and can be validated by the disk D, given that $K_{MRC,D}$ has not been compromised. Steps 3 and 5 are parallel requests to target disks D_1 to D_n. It is currently assumed that the Client has shared keys with D_1 to D_n, $K_{D_1,Client}$ and $K_{D_n,Client}$ respectively. Note that the additional keyed hash or MAC `h(req, s)` is done in order to prove that the Client has really been issued with the capability CAP and has possession of s, which can be determined by the OSD. Steps 4 and 6 are the parallel executions of the requests (read, write, delete) on the target blocks or objects, corresponding to steps 3 and 5 respectively, while 7 and 8 are the respective responses. The disks also perform a hash with `s` on the response, to prove that this is a valid response to the Client's request.

The work from Aguilera et al.[1] has been presented as it provides some basic requirements and approaches for handling cryptographic capabilities for distributed storage. However, the work of Miller et al[13] has had more influence on the standardization of ANSI T10 SCSI OSD, described in the next section.

3.2.1 ANSI T10 SCSI OSD Extensions

The OSD Technical Work Group of the Storage Networking Industry Association (SNIA) consisting of various industry and academic partners have defined the ANSI T10 SCSI OSD command set and works on further extensions. First, a rough sketch of the architecture is presented before security related questions are addressed. The shared secret security architecture is described by Factor et al. in detail in [6].

The goal is to provide a solution which combines the best attributes of block oriented Storage Area Networks (SAN) and file oriented Network Attached Storage (NAS). While SANs provide excellent performance for most access patterns they suffer from sharing. This is the domain of NAS systems. The new approach should provide:

- Improved storage management
- Data sharing among clients
- Aggregated performance
- Security

The goals are addressed by leveraging the interface of disk drives from a pure block layer access model to an object based access model where data is not addressed by block number, but by an unique identifier of an object. The object

itself contains the Object Identifier (OID), metadata and attributes and its physical location on the disk.

Factor et al.[6] describe a set of requirements, design tradeoffs for a security protocol to support the ANSI T10 Object-based Storage Devices (OSD) Standard, also discussed in more detail below in subsection 3.2.1. They refer to the main outcome as "the OSD Protocol" in their paper. The protocol is based on a secure capability-based model that enables fine-grained access control that protects both entire storage device and individual objects from unauthorized access. This variability in granularity and protection coverage is one of the features that is important in XtreemFS security. The protocol defines three methods of security based on the applicationsŠ requirements with different security assumptions:

1. CAPKEY: basic mechanism for protecting credential integrity, focusing on binding a credential to a particular secure channel between a client and an object store. The security goal here is to protect against replay of credential over another secure channel created by a rogue Client. This therefore relies on an underlying security transport mechanism such as IPSEC or TLS

2. CMDRSP: protects the integrity of commands and their arguments. It is not reliant on the underlying security transport mechanism for verification of credential integrity but still relies on the secure transport for transported data integrity. That is, it adds built-in command integrity to CAPKEY instead of relying on the lower layers of the system

3. ALLDATA: protects the integrity and confidentiality of credentials and data without relying on an assumption of a secure network. If it is implemented over a VPN or within a closed, trusted network, then it might be considered redundant.

They also place an emphasis on quick key management such that normal operations are not disrupted. Their key management schema also introduces a key hierarchy, which is covered in section 3.2.1. At the time of publication referenced, they had not yet determined how to deal with complex capabilities for applications that need to access and perform operations on multiple objects in one request. We however adapt their basic capability schema as shown below:

```
CAP c = {
    SecurityInfo {
       h(K, PRN, c),
       Algorithm
    };
    ObjectDescription {
```

```
    }
    Permission {
        [read],
        [write],
        [delete],
        [etc]
    }
    Expiration: dd/mm/yyyy:00:00:00;
    [Audit: <optional field>]
}
```

Four different objects are defined in the standard, which correspond to the purpose and types of keys in the standard:

- User Objects: Objects which are created on behalf of users, e.g. their applications.
- Collection Objects: A group of user objects with some attributes in common.
- Partition Objects: A partition object contains user and collection objects. They have same security and space management attributes.
- Root Object: Represents the object storage device and contains partition objects.

At the time of writing this deliverable, there was no concept for differentiating between different types of objects and enclosures in XtreemFS. Objects in XtreemFS are therefore assumed to be semantically equivalent to User Objects in the ANSI T10 standard. As security and group management becomes more advanced in XtreemFS, we should further consider adopting this particular classification of object types and corresponding keys, as further concerns such as illicit information flows and violations of digital rights arise.

Keys in the Standard

For this set of different objects an extended SCSI command set is defined. Of special interest are the security related commands to set keys, e.g. for an object or a master key for an entire OSD and the check of the validity of object based operations such as read, write, append, create, ...

The following lists the defined shared secret keys and the possible operations which could be performed:

Master Key: allows for unrestricted access to the drive and is used to bootstrap the system's security. It is associated with the drive owner and used to set a root key:

→ **Root Keys**: used to create OSD partitions on a single device and to set the partition keys. Allows full access apart from initializing the drive and changing the master key.

→ **Partition Keys**: for creation of partitions (i.e. a form of root key for partitions) and to derive working keys:

→ **Working Keys**: Frequently changed keys for the generation of capability keys for accessing objects. Keys are versioned and up to 16 refreshed keys can be marked as valid to prevent invalidation of all cached credentials.

It has to be noted, that the standard does not say how the keys have to be maintained. A scheme such as Kerberos may do.

Protocol flow

In order to access an object with a certain operation the following protocol has to be processed. It ensures security by only performing actions of integrity check commands. A parameterized integrity checksum in the form of a 160 bit keyed HMAC-SHA1 is used to identify commands containing the object and the allowed operations. Note that the decision as to if some principal has access to a certain object is taken at the level of a separate policy manager, which implements the desired semantic.

Involved in the protocol is a client who wants to perform actions on a certain object storage device. A metadata server mediates the requests by the client. The metadata server also performs the authentication of a client via the use of an identity database and grants or denies the authorization request using a policy/authorization database.

1. Exchange of a shared secret between security manager and OSD.
2. Client requests access to object (requests a capability) at a certain object storage device, in a certain partition object.
3. Metadata server performs client authentication via identity database.
4. Metadata server determines client authorization via authorization database.
5. If client is authorized, a credential (capability plus integrity checksum) is issued to the client

6. Client uses the capability to perform the requested action at the object.

Summary The ANSI T10 SCSI OSD standard is a purely symmetric key based approach to create capabilities for user requests. A major drawback is the establishment and the maintenance of the security associations and the respective key material. Since it builds on the SCSI protocol, it implicitly defines the interconnect types. Today, the SCSI protocol is used in DAS (Direct Attached Storage) environments, in iSCSI based SAN (Storage Area Networks) environments and in SRP (SCSI RDMA Protocol) using the RDMA (Remote Direct Memory Access Protocol) over Ethernet or Infiniband as interconnects. They have all in common, that these protocols are usually used in closed environments. For the secure establishment of keys additional identification/authentication information and protocols are necessary.

3.2.2 pNFS - NFSv4.1

For the well known file sharing protocol NFS, which is well adopted in industrial and productive environments, efforts are going on to extend it to fulfill future needs. Major architectural changes were made in the version 4.1 of the NFS protocol. This version is also called parallel NFS or pNFS. This name indicates the major changes where a set of data servers, coordinated my a metadata server aggregates its performance and enables the parallel use of a set of data servers at the same time. The NFS server is no longer one single entity and separates data and metadata. Thus it follows a similar setting than most object oriented storage architectures and their trust model.

The major difference of pNFS in regard to other approaches is the ability to handle files, blocks and objects on a set of distributed data servers. In the following we will review the latest specification for object handling. Here the standard assumes the presence of ANSI T10 compliant devices and provides the interfaces.

Please note, that although pNFS may look similar in terms of its architecture as other object oriented storage architectures the application scenarios may look very different. Up to now there is no information available which allows estimates on the ability to scale it into large dimensions.

As other approaches, pNFS splits the former NFS protocol in two parts and decouples metadata from content. Therefore the protocol for metadata has to be enriched by a set of commands which reflect the consequences of this design. However, pNFS is designed to handle objects, files and blocks and thus have to

provide different ways of addressing and mediating access to these objects.

A client system mounts a pNFS file system at the pNFS MDS. There it can also get a list of used storage devices, their IDs and their hostnames. After a mount of the remote pNFS file system the usual protocol starts with at the client side with issuing an `OPEN` request with a subsequent `LAYOUTGET` request. The MDS (MRC in XtreemFS) replies to geometry of how the requested file is distributed over the set of NFS servers. The response of a `LAYOUTGET` request represents the right to access that specific file and thus the capability. In order to get a capability, the client has to be authenticated and his permissions for that object have to be checked. More precisely the `LAYOUTGET` returns in case of a success a set of capabilities for the objects which are associated with the file.

Via a `LAYOUTRETURN` remote procedure call a client can free a `LAYOUT` which will not be used any more. Also a MDS can request a `LAYOUT` back via the callback `CB_LAYOUTRECALL` remote procedure call.

Up to now the file-based access method is the only mechanism which is described in detail in the standard. When this access method is used, the storage devices use the NFS version 4 protocol to deliver files to the client.

In case the access method is object based, the object storage protocol must implement the security aspects described in the ANSI SCSI T10 OSD extensions in the chapter before.

To recall, these are NOSEC, CAPKEY, CMDRSP, and ALLDATA. As the name says, the NOSEC variant does not provide any security. In order to provide a minimum level of security the CAPKEY variant or better has to be used. From a key management perspective not only the object keys which have to be managed by the security manager in the form of the MDS have to be maintained. Since it is a symmetric protocol, additional steps have to be performed in the setup phase to negotiate on secure channels to setup and maintain keys. These tasks should be achieved using a framework such as GSS_API.

3.2.3 Ceph

Leung [11, 12] describes the implementation of a security protocol called Maat, which is intended to address the peta-scale security problem, identifying that the problem lies in the tie between the number of security operations to the number of devices and requests. The expected scale is hundreds of thousands of clients and storage devices, which is a lot more than traditional notions of storage networks. The main feature of their work is the idea of *extended capabilities*, where binary Merkle Trees are used to create tokens that can authorize I/O for any number of clients to any number of files, based on the ability of a capability to prove the integrity and authenticity of any client and request within the capability's tree structure. Secondly, the implement automatic or implicit revocation by allowing capabilities to expire within a lifetime t. They also feature secure delegation as a means of enabling scalable, cooperative computation across many devices.

Key Analysis of the Approach from Olsen and Miller

Olsen and Miller [14] presents a secure capability protocol for an object based distributed file system. This has a comparable understanding of objects to XtreemFS, although there is an emphasis on the massive use of mapping and hashing functions for the statistical equalized distribution of data. Thus their mapping function may be replaced by a query to a map. Their work is embedded into the work on the CEPH petascale filesystem.

Their initial, non secured protocol looks like this (where U denotes the user on behalf this request is made):

1. `Client → MRC: U,(open (path, mode))`
2. `MRC → Client:  H`
3. `Client → D(H,i):  read(oid,(H,i),bno)`
4. `D(H,i) → Client:  data`
5. `...`

From the overall protocol flow it can be compared to other approaches with the difference that the OSD is referenced by the mapping function `D(H,i)`. They consider a cascade of metadata servers for high availability reasons. A MRC may forward the initial response to an authoritative MRC first.

Starting from this point, they introduce security mechanisms into the setting and show different variants. The protocol uses symmetric key cryptography for all communications to ensure integrity and authenticity. It is assumed, that a shared

key between the Client and the MRC(K_{CM}) is provided by an appropriate mechanism.

Regarding XtreemOS, the certificate infrastructure could be reused for the distribution of certificates. However it should be noted, that the storage infrastructure itself may be used outside the context of VOs and may have a longer persistence than VOs itself. This has the consequence that an infrastructure supporting the distribution of these shared secrets should not logically be bound to VOs itself.

It should be noted, that the MRC trusts in the authenticity of the user U, given that they have access to the Client. The integration of a user identity infrastructure is therefore not considered. It is also assumed, that the MRC is keeping the shared secret K_{CM} with the client and shared secret with the OSD K_{MD} secret. A further assumption is that a reasonable synchronous time information is available among all OSDs and MRCs.

The protocol then looks like this (where C is a security token from an authentication service which is associated with U and τ={H,perm,K_{CD},T_s,T_e}K_{MD} where T_e, T_s expressing the validity period):

1. `Client` $\rightarrow$ `MRC: C,{open (path, mode)}`K_{CM}
2. `MRC` $\rightarrow$ `Client:` `{H,` K_{CD}`,` T_s`,` T_e`}`K_{CM}`,` τ
3. `Client` $\rightarrow$ `D(H,i):` τ`,{read(i,bno),` T_1`}`K_{CD}
4. `D(H,i)` $\rightarrow$ `Client:` `{`T_1`, data}`K_{CD}
5. `...`

Regarding a fine/coarse grained key assignment this protocol can best be compared with a master key or a partition key in the ANSI T10 SCSI OSD standard, where, a key is used to access or initialize a disk. Note that no object-based keying scheme is used, as the disk gets an individual, unique key for each Client.

The protocol is designed to protect honest clients from clients which were rendered dishonest. This could happen by malicious software or by a malicious user. In such a case the K_{CM} key can be misused and actions on the metadata channel on behalf of the belonging user can be performed. Also the shared key K_{CD} and the connection with the OSD can be misused. The timeframe in which this can occur is limited by the validity time of the ticket τ.

The following figure visualizes the security associations for this protocol. As in the protocol, the clients are denoted with C, the OSDs with D and the MDS/MRC with M. All keys are symmetric. This implies that an authentication procedure with the establishment of secure channels between the Clients and the MDS and the MDS and the OSDs has to take place in advance. These associations are not taken into account here. For the applicability of this scheme to XtreemFS it would be desirable to connect the MDS into the tree or trust chain for one to several VOs. This makes it possible to authenticate users and to perform access control decisions. A different scheme may be used to authenticate the MDS with the OSDs.

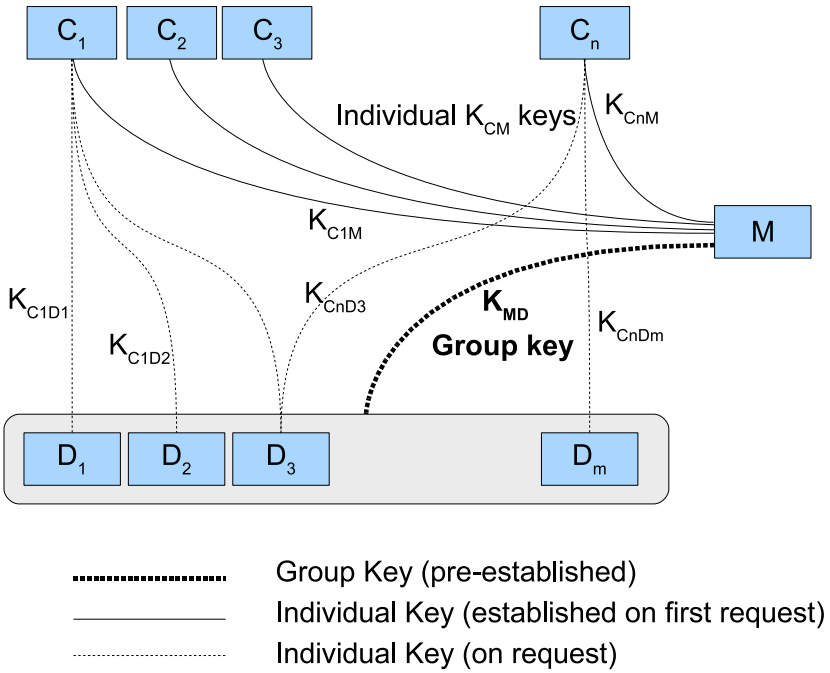

Figure 4: Security associations in their first approach based on symmetric key.

Figure 5 shows the keys which has to be maintained at each entity in the first approach of Olsen and Miller.

A Refined Protocol

A refined protocol is presented in the following. This protocol introduces public key signature scheme to authenticate the capability. Scalability and the reduction of failure domain are the reason for this change. The authors state that within a set of 1000 and more OSDs a shared secret can no longer be maintained effectively. A compromise of one OSD reveals the secret which is shared with all other OSDs

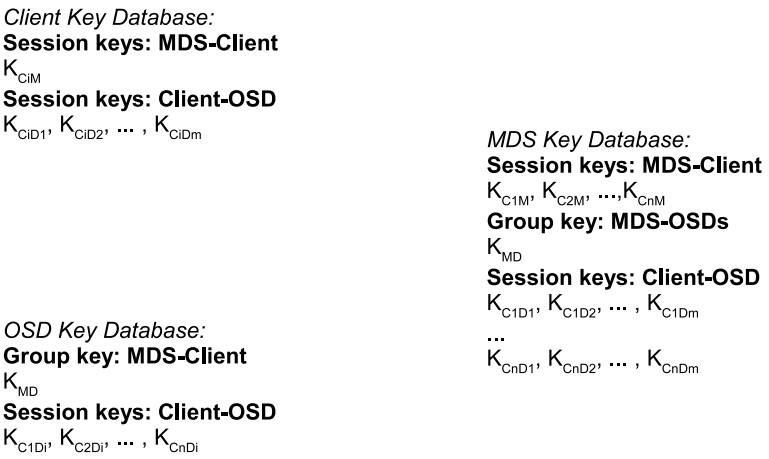

Figure 5: Key repositories in their first approach.

and the MDS. This would allow an attacker to forge capabilities to other non-compromised OSDs. Now, a public key scheme is used to build individual security associations between the MDS and the OSDs. A compromise of one OSD will not lead the secret to compromise other OSDs as well.

The capabilities which were signed in the former protocol using the secret shared key are now signed using the privat key of the MDS. The MDS has a private denoted as K_M^R and a corresponding public key denoted as K_M^U.

It is assumed that the shared key K_{CM} between the clients and the MDS as well as the shared key K_{CD} between the client and individual OSDs is kept secret. This also holds for K_{MD} between the MDS and the OSDs. Similar as in the protocol before τ={P,H,perm,T_s,T_e}K_M^R where T_e, T_s expressing the validity period, H the file handle, $perm$ the permissions. P denotes a set of principals which are allowed to access the same handle as well. It is powerful concept to reduce the set of capabilities to create and maintain.

The tupel ι={C,G,$K_{C,D(H,i)}$,T'_s,T'_e}$K_{M,D(H,i)}$ forms a ticket which is handed to the client by the MDS. It states that any client, knowing $K_{C,D(H,i)}$, is user C, a member of group G within the timeframe starting from T'_s to T'_e will be able to perform actions with the provided capability at the OSD. Since this ticket is encrypted symmetrically with the key $K_{M,D(H,i)}$ of the security association between the MDS and an OSD, only they can decrypt it. The ticket keeps confidentially the key $K_{C,D(H,i)}$ which the client must use to pass the actual request to the OSD.

Regarding XtreemFS with a representation of rights in the form of access con-

trol lists, such a concept may be applied as well by maintaining a reverse mapping stating all qualified users for an object with a certain right. Another aspect is the reduction of load at the MDS since a capability may be reused instead of creating a new one. However the use of this feature makes this approach unsuitably to implement a coarse grained locking scheme.

1. Client → MDS: C,{open (path, mode)}K_{CM}
2. MDS → Client: {τ}K_{CM}
3. Client → $MDS_{D(H,i)}$: C,D(H,i)
4. $MDS_{D(H,i)}$ → Client: {$K_{C,D(H,i)}$,D(H,i),T_s,T_e}K_{CM},ι
5. Client → D(H,i): ι,{τ,read(i,bno), T_1}$K_{C,D(H,i)}$
6. D(H,i) → Client: {T_1, data}$K_{C,D(H,i)}$
7. ...

Figure 6 shows the security associations between the entities in the distributed file system. It shows the cipher type (shared key or public key) with which these associations are maintained.

Figure 7 shows the key repositories that have to be maintained at each entity in the distributed file system. It also includes the capabilities and the tickets. Note that there is still the assumption of pre-authenticated users on the clients.

Again the protocol is stateless for the OSDs which is beneficial for fault tolerance and recovery. By caching the capabilities and the tickets the OSD may be seen as stateful. Successfully verified capabilities and tickets may be cached for the lifetime or a shorter period and provide a shortcut for the verification process. However this information can be reconstructed at any point in time with the sole consequence of a performance reduction. An evicted, cached, but still valid entry may be reconstructed at the OSD by extracting the symmetric key for the client security association in the capability.

In contrast to the former protocol, this one needs two more iterations. This increases the time gap of the first request to the time when data is read or written by at least two additional network round trip time units (assuming the data processing at the MDS is negligible in regard to the round trip time).

Focussing on the MDS itself, the additional communication significantly increases the overall MDS traffic. Thus the MDS is at risk to become a bottleneck itself. The MDS also maintains the security association for the clients and the OSDs.

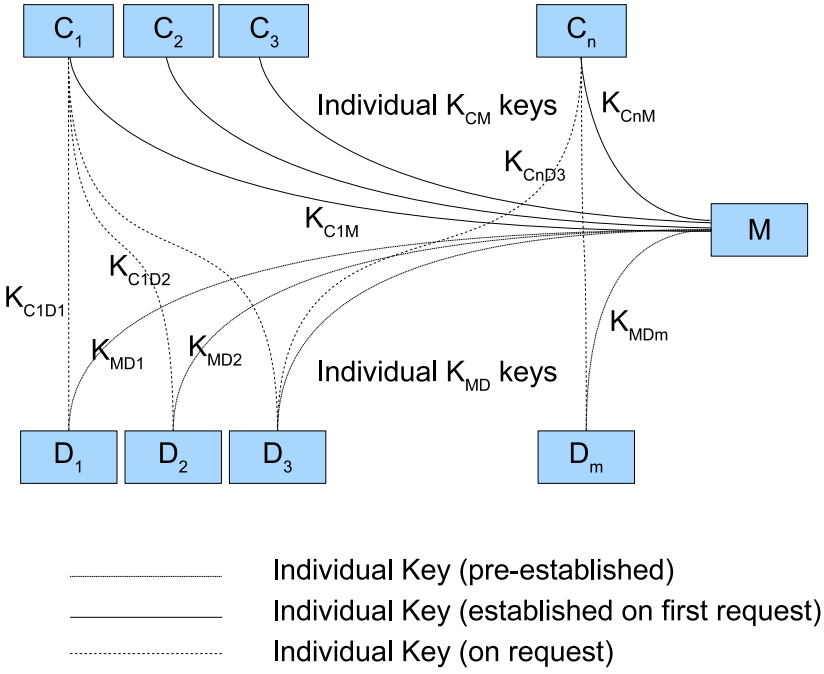

Figure 6: Security associations in their second approach with symmetric and asymmetric keys

Client Key Database:
Public Key:
K_M^U
Session keys: MDS-Client
K_{CiM}
Session keys: Client-OSD
$K_{CiD1}, K_{CiD2}, \dots, K_{CiDm}$

OSD Key Database:
Public Key:
K_M^U
Group key: MDS-Client
K_{MD}
Session keys: Client-OSD
$K_{C1Di}, K_{C2Di}, \dots, K_{CnDi}$

MDS Key Database:
Private Key:
K_M^R
Session keys: MDS-Client
$K_{C1M}, K_{C2M}, \dots, K_{CnM}$
Group key: MDS-OSDs
K_{MD}
Session keys: Client-OSD
$K_{C1D1}, K_{C1D2}, \dots, K_{C1Dm}$
...
$K_{CnD1}, K_{CnD2}, \dots, K_{CnDm}$

Figure 7: Key repositories in their second approach

According to Olsen and Miller[14] the refined protocol solves these issues by letting the clients maintain their security associations with the OSDs themselves.

Further refined protocol

In order to tackle latency issues and to avoid the risk of making the MDS or the set of MDS's a bottleneck a further refinement of the protocol is proposed by Olsen and Miller[14]. The major difference in regard to the former protocol which uses a ticket as a token of authorization for users and their group relationship is that the ticket and the credential can be used in a more decoupled way.

While the ticket is issued in the former protocol in the mid of the protocol, it can be transmitted at the beginning of the protocol explicitly. Having a longer lifetime of the ticket this process is performed not frequently. Thus the remaining part of the protocol does not need these steps for most cases any more and one round-trip can be saved in most cases.

A further detail of this protocol is the use of public key cryptography to establish a security association between the client and the OSD. This association is maintained still via symmetric algorithms.

The MDS does not maintain the security association between the client and the OSD. It just sends an initialization vector within the ticket. Based on this, the client computes a symmetric key which he transmits to the OSD encrypted with the OSDs public key $K^U_{D(H,i)}$. The OSD can rebuild the used symmetric key in order to check that it was generated by C. Therefore it uses the inverse of the used keys for exponentiation. These are the public key of the client and the private key of the OSD itself.

1. `Client` $\rightarrow$ `MDS: C,request ticket`
2. `MDS` $\rightarrow$ `Client:` $\{\tau\}K_{CM}$
3. `Client` $\rightarrow$ $MDS_{D(H,i)}$`: C,{open (path, mode)}`K_{CM}
4. `MDS` $\rightarrow$ `Client:` $\{\iota\}K_{CM}$
5. `Client` $\rightarrow$ `D(H,i):` $\{K_{C,D(H,i)}\}K^U_{D(H,i)}$, $\{\iota, \tau, read(i, bno), T_1\}K_{C,D(H,i)}$
6. `D(H,i)` $\rightarrow$ `Client:` $\{T_1$, `data}`$K_{C,D(H,i)}$
7. `...`

The following figure 8 shows the security associations which have to be maintained with this protocol. Note that since the trust model is still the same, the amount of associations is pretty much the same as in the former protocol.

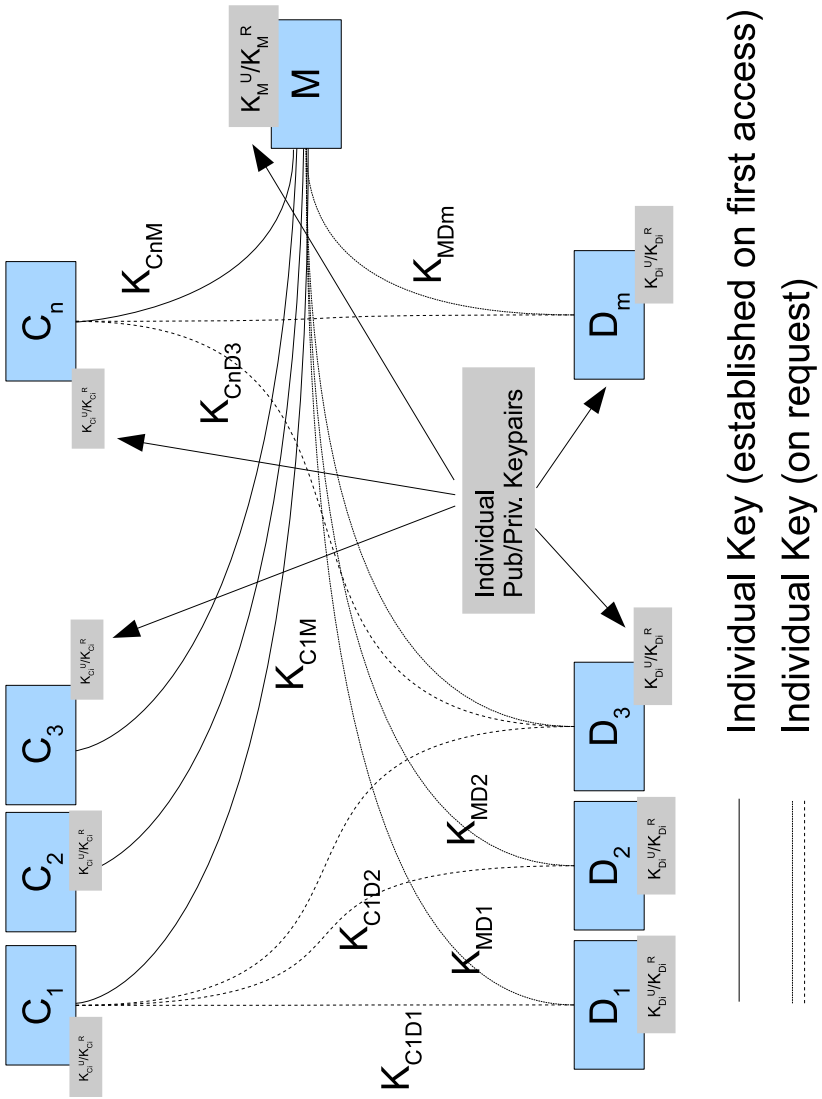

Figure 8: Security associations in their third approach with focus on asymmetric keys

In figure 9 there is however a notable difference. Since the MDS delegated the maintenance of the security associations between the clients and the OSDs to the entities involved in the association, the MDS is significantly relieved of this additional overhead.

Regarding the applicability of this protocol to XtreemFS this protocol may fit as well. The address space of objects is flat and does not contain a group or hierarchy of objects as the ANSI T10 OSD standard does.

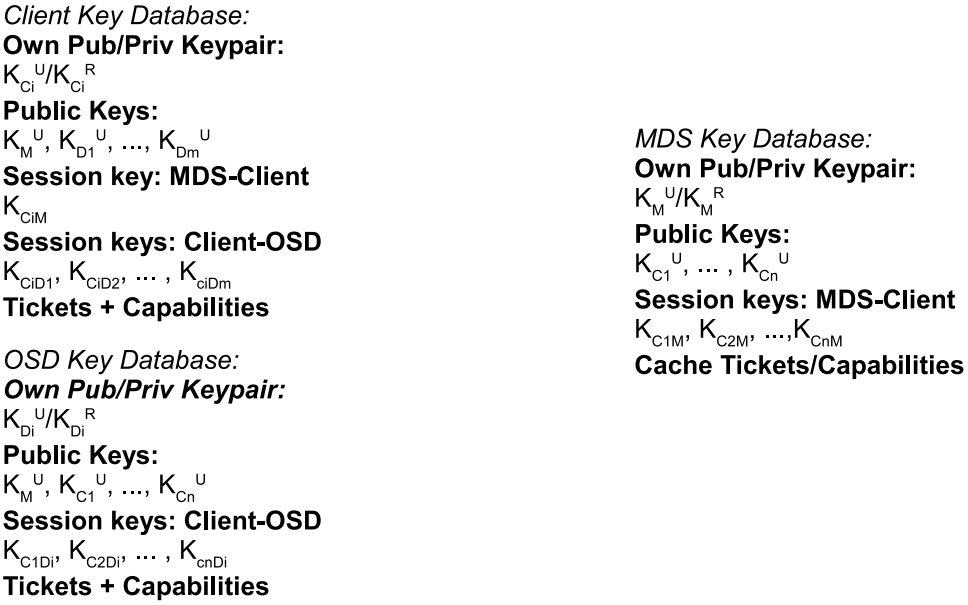

Figure 9: Key, ticket and capability repositories in the third approach of Olsen and Miller. The key repository in the MDS shrinks by the introduction of public key cryptography for the client-OSD communication.

However, with the massive use of public key cryptography, although this makes the maintenance of security associations a lot easier, requires additional mechanisms to reduce CPU load, as it consumes much more resources than symmetric cryptography.

From the current point of view there are two ways to achieve this:

1. Reduce the amount of public key creation and verification operations that need to be done.

2. Use public key crypto algorithms that consume less resources.

Since effective crypto algorithms are discussed in the next section, we first place emphasis on an approach to reduce the amount of capabilities in a distributed filesystem. The approach taken is to broaden the scope within which capabilities are valid.

It should be noted, that the following approaches require a thorough understanding of users at the MRC/MDS and their relationships among each other. The following facts within current XtreemOS design make it unfeasible to apply this concept.

- Users are maintained at a the VO level which up to now does not have a tight relationship with the MRC/MDS and

- The semantic of grouping may not known or not be defined *depending on the VO model.* It also may get complicated in case the MRC/MDS has to deal with mapped users.

Independent of these issues the architecture of XtreemOS is evolving and may provide the chance to apply these promising approaches. In the following the rough idea should be sketched to show the achievable advantages.

Coarse Grained Credentials

The basic idea behind reducing the amount of capabilities is to extend their validity. An easy approach would be to extend just their lifetime. However the gains by that are limited especially when a number of different files are accessed just once. Leung et al. [12] follow a different approach which is visualized in figure 10. Instead of issuing a capability for just a single user on a single client for a single file or object they group entities with the same attributes.

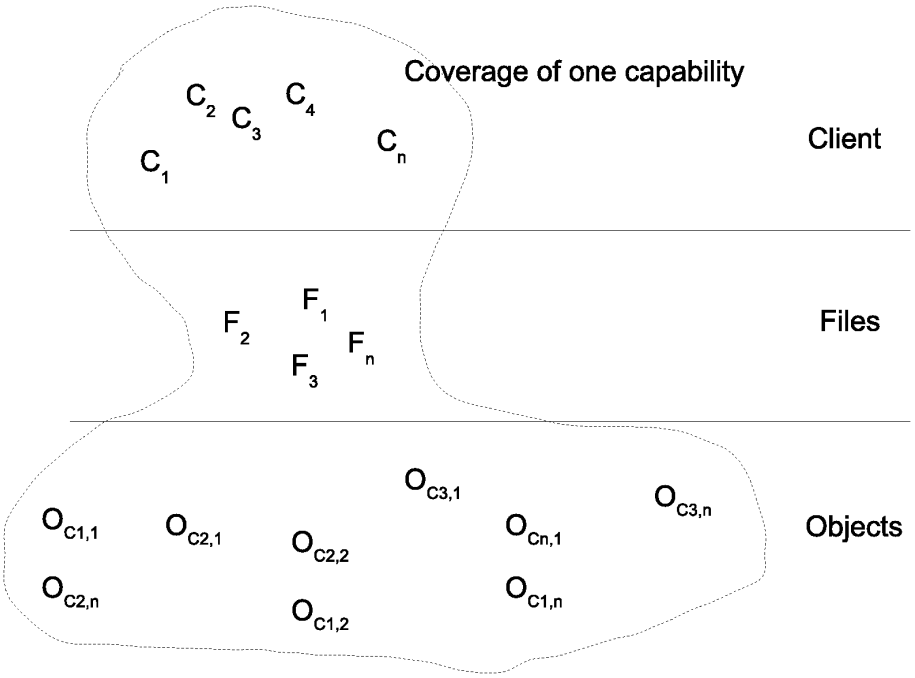

Figure 10: Extending the validity of a capability to span across multiple users and files.

In order to group entities, the MRC/MDS needs to be able to query its database of access policies for *who else is allowed to access object o with rights r?*. The same has to be done for the other constellations of subjects and rights[2]. Note that in the CEPH petabyte file system [16] in which this approach should be implemented a file may consist of several object. These are addressed by a mapping function.

A similar situation occurs in XtreemFS, where files may consist of several objects as well, due to striping. However, currently, the MRC/MDS maintains parts of files.

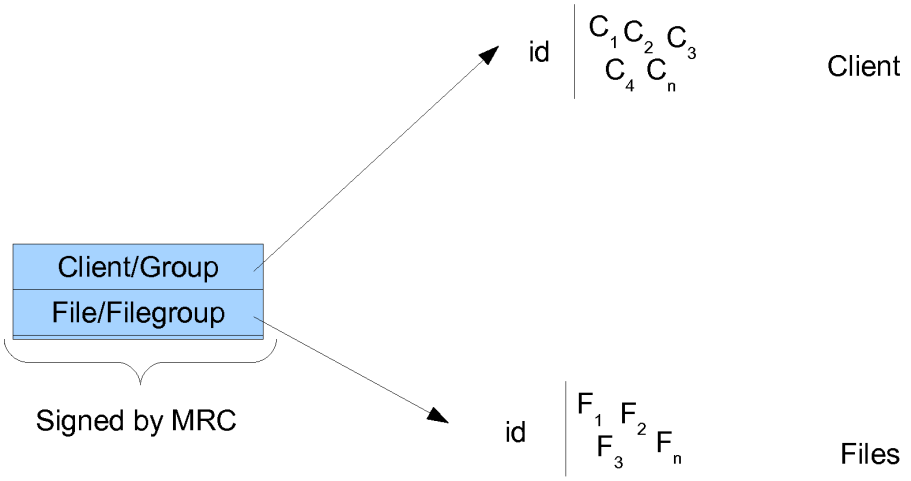

Figure 11: Capability used to have a constant size for its identifiers. Dealing with list to describe groups would be cumbersome. Here, only the identifiers for the client and the file are shown of the capability.

The question then remains: how can these groups be efficiently expressed through capabilities? For a solution see figure 11. Transmitting just a list of identifiers instead of just one would increase the size of the capability and more important would bind the understanding of a group to the lifetime of a capability.

Instead of transmitting a list of identifiers in the capability an efficient mechanism is proposed to identify the group with a fixed size identifier and maintain the groups in a condensed way using a Merkle tree [10]. All identifiers now describe a subject or object by its hashed value.

The Merkle tree itself is a tree[3] which keeps a hash of its elements in the

[2]Also, the use of public key cryptography for signing a capability is a way of grouping. It allows to verify a capability on any OSD with a trust relationship with the MRC/MDS.

[3]A binary tree in that case.

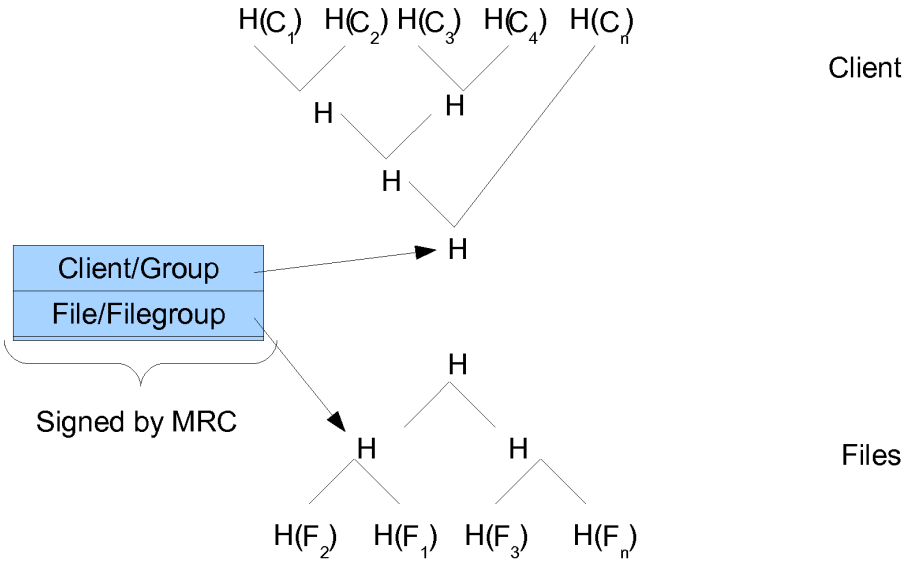

Figure 12: Using fixed size identifiers for groups using Merkle trees. The trees itself are updated at the OSD in case the group identifier is not known.

leafs. The layer above is generated by applying a hash function to the neighboured elements. Usually the hashes of the above layer is calculated hashing the concatenation of the hashes below.

When the MRC/MDS now issues a capability with identifiers not known by the OSD, the OSD needs to update its database of group information. The advantage of using Merkle trees here is that updates or larger groups can be expressed by already known groups to the OSD since two joined Merkle trees will result in a Merkle tree again.

The savings in capabilities using this scheme depend heavily on the ability to find group of users with the same (or a significant union) of access rights to a set of files. If this can be achieved often the amount of capabilities can be divided roughly speaking by the average group size of user and file identifiers.

3.3 Applicability to XtreemFS

In this section we will summarize the applicability of the approaches for XtreemFS. This was already covered in the descriptions but will be presented here in a condensed form:

- Seperation of VO certificates and certificates and capabilities of the XtreemFS file system: In XtreemOS a lot of attributes are expressed using X.509 cer-

tificates and with an extended attribute set. While this container is designed for expressiveness and flexibility, the tickets and capabilities in the file system have to be designed for high performance and low latency generation and verification. This can currently not be achieved with X.509 containers:

  - The extended X.509 or VO certificates may be used to authenticate the user once, e.g. the user via the FUSE client to the MRC.
  - Very different lifetimes for VO certificates and file system capabilities and tickets.
  - Seperation of duties of the MRC/MDS and the VO certificate and user management.
  - For all further interactions a very efficient scheme has to be used in order to keep latency low and throughput high.

- The architecture of the presented approaches and their understanding fit to the architecture of XtreemFS or at least a subset may be mapped to it. Within XtreemFS the MRC/MDS mediates access from clients to objects within one namespace.

  - The SNAD schemes which influenced the NAD and finally formed the basis for the ANSI T10 SCSI OSD standard uses an object hierarchy and thus different namespaces. However focussing on the working keys of the ANSI T10 SCSI OSD standard this is applicable to XtreemFS.
  - The approach for ceph uses a flat namespace for file handles and objects. Thus their architecture fit exactly the one of XtreemFS. Their schemes can be applied omitting the mapping function $D(H, i)$ which directs a client with a known file handle and the i-th byte to the OSD keeping the object. Since in XtreemFS this job is in the responsibility of the MRC/MDS, it has to provide the exact object identification instead of a file handle.

- Scalability: From an architecture point of view the approaches of SNAD, NAD, pNFS , ceph and XtreemFS they are all similar. However pNFS (which may use the ANSI T10 SCSI OSD standard) is still a datacenter approach. It features parallel access to data servers using NFS protocol version 4 on the client to the data server path. Traditionally there are a few reliable data servers in the data center. Currently there is no hint that the amount of data server is considered to scale up to the often mentioned thousands of nodes as in other approaches.

We consider the use of public/private key cryptography as a reasonable way to simplify the key management. At the same time it solves untackled mutual authentication questions in the pure symmetric capability shemes.

The use of public/private key cryptography turns also around the way a signature is created. Using symmetric schemes with all individual security associations, the signature asserts: *"this capability is signed for OSD x by the MRC* as opposed to *this capability is signed by the MRC and valid for all OSDs*. This simplifies the maintenance of security associations by the number of OSDs. This is visualized in figure 7 in regard to 9.

4 Efficient Cryptography

In order for XtreemFS to be both scalable and secure, more efficient cryptographic mechanisms should be considered. In this section we discuss alternative cryptographic algorithms other than the mainstream algorithms i.e. RSA (integer factorization) and DSA (discrete logarithms). This is done with the expectation of achieving more efficient performance of the secure XtreemFS protocols. Efficient protocols and algorithms should provide better scalability with respect to storage, communications and CPU usage. This suggests smaller keys, smaller cipher text and less complex cryptographic algorithms. Nevertheless, the robustness of security should remain with the selected solution.

Previous sections have already described approaches to more efficient key management. This section further supports the improvement of security efficiency in XtreemFS by investigating more efficient cryptographic mechanisms other than the traditional public-key crypto-mechanisms used. In doing so, the Elliptical Curves Cryptography (ECC) and NTRU algorithms are investigated.

4.1 Elliptic Curves Cryptography

The strength of any crypto algorithm is determined by the complexity or difficulty of solving a solution to a mathematical problem. If all parameters are known then the problem is trivial but solving for unknowns is complex. Cryptographic keys are therefore the values that must be kept secret such that the mathematical problem is just an arithmetic calculation.

Most of the processing power and effort when using RSA for signatures or encryption is attributed to the complexity of computing modular exponentials and the usage of large integers. RSA' one way function capability is based on the fact that given b, e and m in the equation $c \equiv b^e(modm)$, finding c is trivial; however, given c, b and m, the determination of e is non trivial. Work towards more efficient crypto mechanisms is based on finding mathematical problems that have a sufficiently strong one-way function property without having the high computational requirements. There are as well other efforts towards using parallelization or more linear modular exponentiation algorithms, but these are not mainstream and typically require specialized hardware. The presence of such specialized hardware cannot be assumed for XtreemFS nor for XtreemOS.

ECC is a public-key cryptographic scheme based on arithmetic operations on elliptic curves over a finite field [3, 15]. A finite field containing q elements exists if and only if q is a power of a prime number. In addition, for each field element q there is precisely one finite field denoted as $\mathbb{F}_q$. There are then two types of finite fields - prime finite fields and 2-finite fields. The prime finite field is described below:

Time to break in MIPS years	RSA/DSA key size	ECC key size	RSA/ECC key size ratio
10^4	512	106	5:1
10^8	768	132	6:1
10^{11}	1,024	160	7:1
10^{20}	2,048	210	10:1
10^{78}	21,000	600	35:1

Table 1: Key Size Equivalent Strength Comparison from Certicom [3]

Prime finite fields $\mathbb{F}_p$: $q = p$ and p is an odd prime number and the field is the set of integers $(0, 1, ..., p-1)$. If $a, b \in \mathbb{F}_p$ then the *addition modulo* p is expressed as $a+b = r$ such that $r \in [0, p-1]$ is the remainder of $(a+b)/p$; otherwise stated, $a + b \equiv r(modp)$. Similarly the *multiplication modulo* p is expressed as $a.b = s$ such that $s \in [0, p-1]$ is the remainder when $(a.b)/p$; otherwise stated, $a.b \equiv s(modp)$. Determining the additive inverse and multiplicative inverse is as follows:

- If $a \in \mathbb{F}_p$ then the additive inverse $(-a)$ of $a \in \mathbb{F}_p$ is the solution $a + x \equiv 0(modp)$
- If $a \in \mathbb{F}_p, a \neq 0$ then the multiplicative inverse a^{-1} of $a \in \mathbb{F}_p$ is the solution $a.x \equiv 1(modp)$

ECC's point multiplication out-performs RSA's modular exponentiation and requires smaller key sizes for the same relative cryptographic strength, as shown in Table 1.

The only real drawback with ECC is the relative lack of available implementations and libraries in comparison to RSA and DSA. Sun microsystems has however produced a release of OpenSSL that now includes ECC[4]. For Java implementations, Bouncy Castle[5] is a noted provider.

4.2 NTRU

NTRU [8] is another very efficient crypto-algorithm. It is known to only require $O(N^2)$ operations to encrypt or decrypt a message of size N, while key lengths are $O(N)$. It uses a mixing system based on polynomial algebra and reduction of modulo two numbers p and q. However, NTRU has been commercialized since the initial release of its algorithm, such that this cannot be suggested as an option.

[4]Sun's OpenSSL with ECC available from research.sun.com/projects/crypto/

[5]www.bouncycastle.org

4.3 Applicability to XtreemFS

Using an efficient cryptographic mechanism in XtreemFS would contribute to better scalability, alongside a more efficient capability management approach. Given that a 160-bit ECC public key is as robust as a 1,024 bit RSA/DSA public key, devices with limited storage resources can still store 7 times more capabilities, given the formats remain the same. A second point to note is that XtreemOS is intended to support many different flavors of machine architectures with different capabilities. These are expected to range from embedded and mobile devices to large-scale cluster and supercomputers. This would provide far more flexibility with respect to types of devices and disks that can be used as OSDs or act as Clients to XtreemFS. Furthermore, increasing the CPU throughput at the MRC would provide less risks of it becoming a bottleneck as the amount of users, files, objects, replicas and accesses increase.

The main disadvantage of using ECC is that it has less technical support than the more established DSA and RSA algorithms. However, as discussed above, this is changing. Additionally, based on the fact that ECC is more recent and less deployed than DSA and RSA, it has not been subjected to the same extent of testing. A further disadvantage for ECC is that there are more parameters that need to be considered than in the more classical algorithms. As there are more parameters, the amount of bootstrapping and negotiation communication is larger. For settings that require negotiation per-message or per-requests, ECC would be the wrong choice. However, considering the types of scenarios considered for integration, as presented in the next section, per-message negotiation is not often the case, such that ECC is a viable solution for efficient capability creation and verification.

5 Integration

XtreemFS is a distributed file system that directly leverages the VO support features embodied in XtreemOS, a Grid-aware operating system. In contrast to the conventional middleware approaches that require constant awareness of the very existence of distributed files (e.g. users are often required to shift their data around themselves and explicitly make sure the outputs are stored at stable storage), the direct integration with operating system level mechanisms provides a possibility to reduce user awareness of distributed Grid environment and enhance the transparency of using the Grid as a 'black-box' computing resource.

The aim of this section is to describe such integration in the context of the XtreemOS project. Section 5.1 describes two XtreemFS usage scenarios that motivate the integration and highlight the typical settings of such usages. The remaining sections present the necessary steps that are required to setup and operate such scenarios. Especially, we will look at how XtreemFS could leverage the security services developed by WP3.5 to support secure file access and volume mounting.

5.1 Motivating Scenarios

The job submission is a process which involves a VO user's preparation of the job description, then submitting it along with the user's certificate into the AEM, and finally, if all goes well, executing the job on the node designated by the AEM. The user needs to register with one or more VO(s) hosted by a VOHost system before the job submission process can be started.

Figure 13 shows this process. The user obtains her certificate from the CDA service, and she states the job details and requirements in a JSDL document. AEM has components running on the target nodes which use the user's certificate to provide the PAM authentication.

In this section we focus on the details of the interaction between the job executed on the node and the XtreemFS volumes. The user is a member of a VO, and the job submission occurs within the VO. The XtreemFS access should therefore occur within the VO as well. To clarify these interactions, we introduce two scenarios:

- Scenario 1: file system support for job submission in static mounting (volumes mounted before a job is submitted), start a job, given the path, the JobManager to specify the input and output files on a mounted volumn of XtreemFS.

- Scenario 2: file system support for job submission in dynamic mounting (volumes mounted on-the-fly while/after a job is submitted)

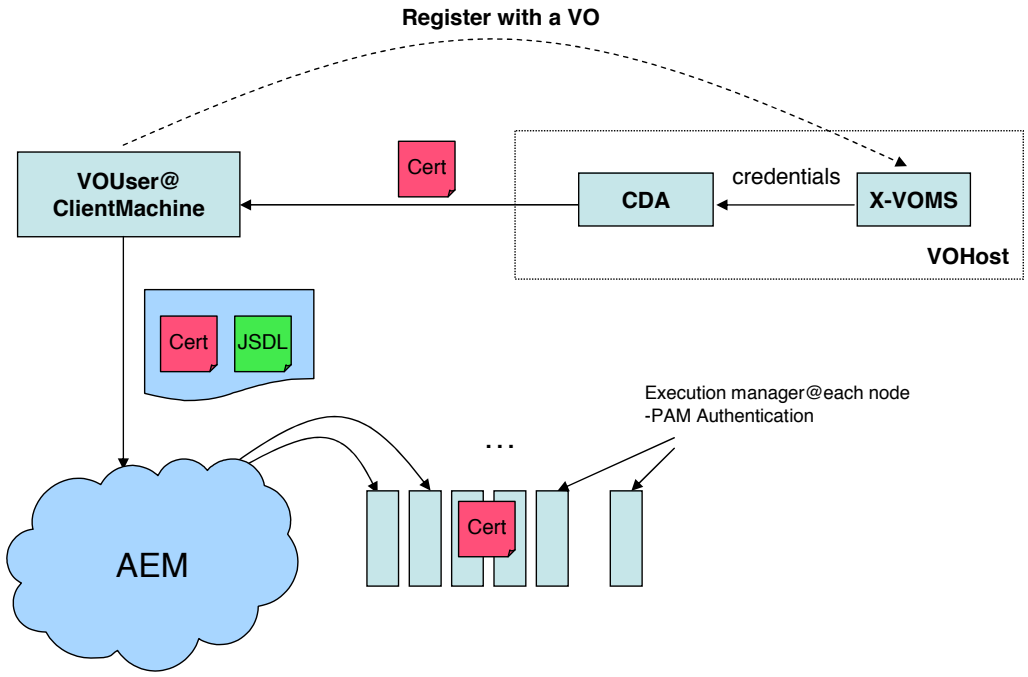

Figure 13: Basic AEM Job Submission Process. The dash line means that users need to join a VO hosted by the VOHost system before the job submission process can be started.

Each of the scenarios implies certain requirements that need to be heeded in order for scenario to be feasible. The requirements stem from the way the user (i.e., the person submitting a job) sets up the job submission, the way the XtreemFS services pass the submission and execute the job, and provide a way for the user to collect the job results.

5.1.1 Static mounting

A convenient way, as far as the user is concerned, is to provide a path to a binary in a way that is independent of the node that would actually execute the file. A default mount point, with a known volume name, is used to mount XtreemFS volume(s) on the nodes, along with the machine booting process, to reduce the complexity of interacting with XtreemFS. Such volumes are pre-mounted once the nodes are ready. As they are being pre-configured, and available for use without users' intervention, this kind of mounting is referred as static. This scenario is illustrated in Figure 14.

Assuming the volume has been statically mounted, the user does not have to express specifically which volume to use. However, to make such a usage possible, the following requirements need to be fulfilled:

- **Designated volumes.** Upon creation of a VO, certain volumes need to be designated to be assigned to the VO only, each of these volumes to be used for specific purpose (e.g., binary and library storage or data storage). According to the current agreements between WP3.3 and WP3.4, this assumption will be followed.

- **A unique mount point.** The mount point of the volumes will have to be the same on all the nodes that execute jobs within a VO. This means that its path and name should not clash with any existing and possible future paths used on the node. We could ensure that by agreeing on a root path to all mount points (e.g. /mnt/xtreemfs), following by the VO's ID. However, using the ID might disagree with the following requirement.

- **A well-known mount point path.** The users submitting the job need to know the exact path to the volume's data as mounted on the nodes. This means that the path would have to either be an obvious one (e.g., /mnt/xtreemfs/voSurveys_bin for a volume containing binaries in the VO named voSurveys), or there should be some means for the members of the VO to list these default mounting points. The system should therefore provide the means to map the volume's ID into a generic, user-friendly name.

- **Mount before the first use.** The node designated for an execution of a job needs to have a service that ensures the volume to be mounted before the

job needs it. This would have to be either a separate service which runs and mounts the volume as soon as the node is ready for execution, or an AEM service which mounts the volume before submitting the first job on the node for the VO.

From the requirements, and to make the usage of the statically mounted volumes, the following steps need to be taken:

- **VO creation:** the VO administrator should ensure the designated volumes (one for binaries, another one for data, possibly some other) are created. This could possibly happen automatically with the VO creation, thus making it part of a VO creation process. In Figure 14, the dash line between X-VOMS and MRC illustrates this interaction.
- **Job submission preparation:** the volume needs to be populated with the files needed for the job execution. This means that the VO administrator needs to copy the files commonly needed throughout the VO to the volumes. Additionally, each user that wants to submit a job would have to mount the volume on the client and copy the job-specific files there.
- **Internal node initialisation:** if the volume is to be mounted before any jobs are executed, then the services on the node have to mount the volume when the machine is added to the VO. The machine already having been added to the VO has to mount the volume upon boot-up.
- **Job execution:** alternatively, if the mounting occurs after the node initialisation, the node's services need to check whether the volume is mounted before executing the job, and if not, they should trigger the mounting.

5.1.2 Dynamic mounting

In the dynamic mounting scenario, as illustrated in Figure 15, the user requests a custom (i.e., a non-VO-default) volume to be mounted for the duration of the job. In the process of the job submission preparation, the user specifies the volume used as well as the mount point. We focus here just on attaching an existing volume to the node where the job is going to be executed. Since the resource management is considered as a task of the VO management a user has to canalize it this way. We can thus observe the following requirements for the scenario:

- **Mount point availability.** Currently, the job specification includes the paths to files that have to be known and valid in advance. This means that the mount point specified by the user has to exist on the node executing the job, and it needs to either be available or already host the required volume.

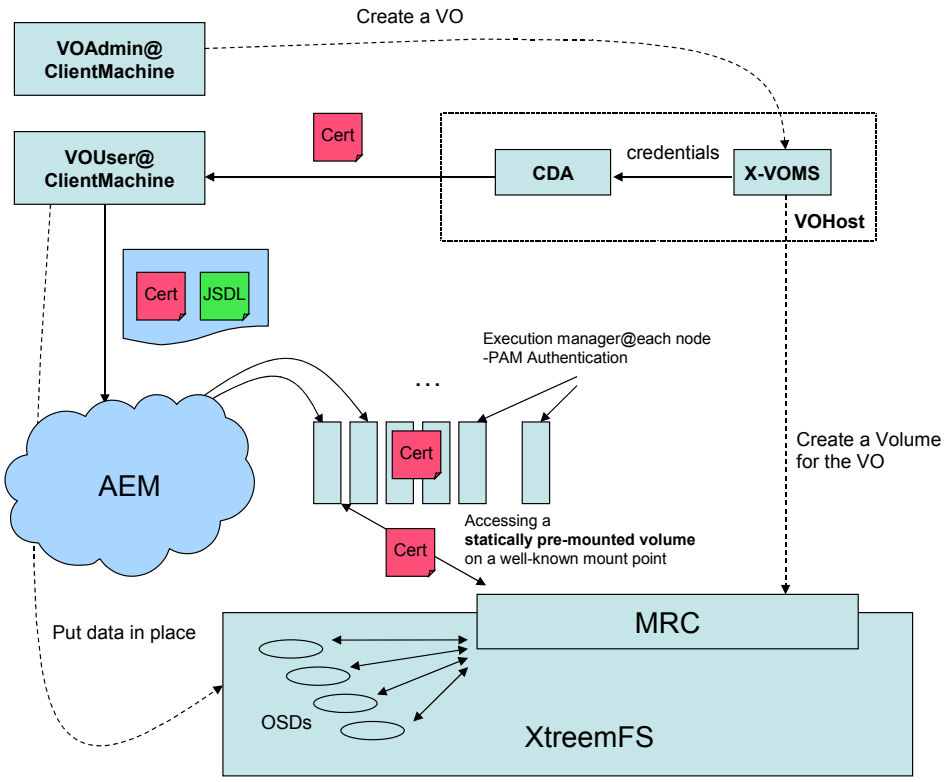

Figure 14: Static mounting with AEM job submission, with the volume information passed in the user's certificate, JSDL, or directly translated by the MRC based on the VO information embedded in the certificate. A few steps, illustrated by the dash lines, happen before the job can be submitted: a VO is created by the VO administrator on a VOHost; this triggers the creation of a volume for the VO on the XtreemFS; before a job can be submitted, the user should make sure the needed files (e.g. binaries) on the designated mounted volume on a well-known mount point.

- **User allowed to use the volume.** While in the static case the user is generally allowed the access to the volume, in the dynamic scenario this may not be the case. An attempt to mount a volume on behalf of a user not authorised for the volume usage will fail, and thus the whole job will fail.

The following steps need to be taken in their respective phases:

- **Job submission preparation:** the volume content preparation is up to the user submitting a job. The user thus needs to mount the volume on the client

machine and place the files required by the job into the volume. Next, the user needs to form up the job submission specification to include the volume information and the mount point in the way that the job's processes expect them. Note that the mount point of the volume and the mount status on the client machine is independent of the mounting points and their respective status on the target nodes.

- **Job execution:** a service designated for the job execution on the node (usually AEM's Execution Manager) needs to mount the volume in the specified mount point before executing the job, and unmount it after the job finishes.

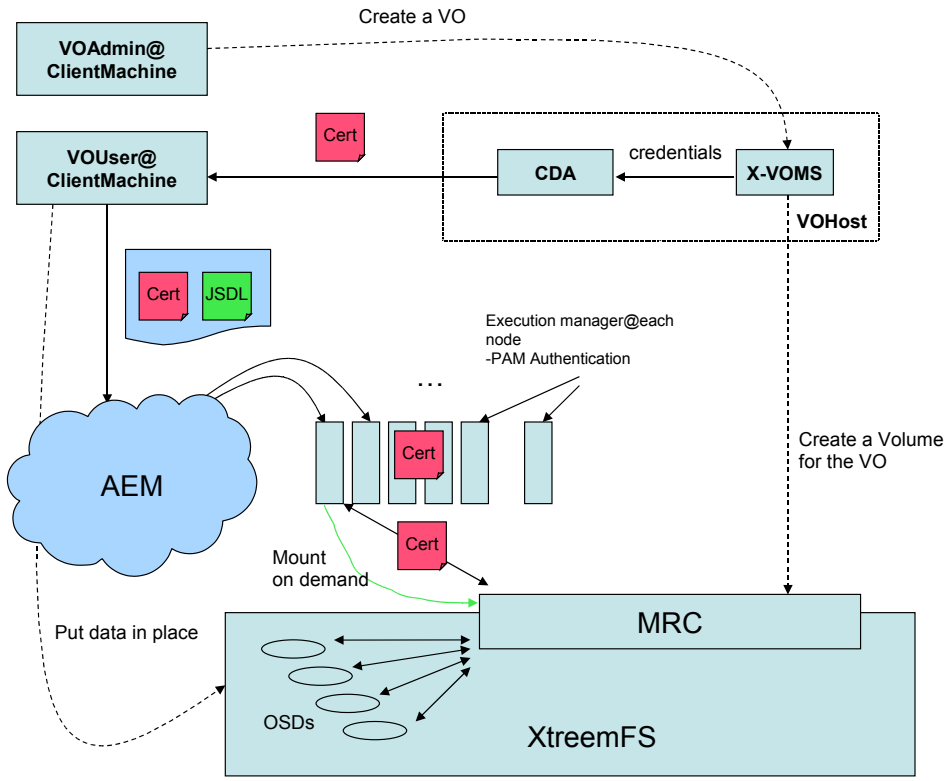

Figure 15: Dynamic mounting with AEM job submission, where the mounting occurs for the job on user's demand according to the job specifications in the JSDL. The green line illustrates a mounting action particular to a specific job submission.

The JSDL specification[6] is flexible enough to provide for the user the possibility to express a requirement of a volume to be mounted for the job. This can be done within the FileSystem element of the JSDL, providing the name of the VO volume needed to be mounted in the MountSource element, and, possibly, using MountPoint element to provide a fixed path where the volume has to be mounted in.

Here is an example of the JSDL catered for this purpose:

```
<jsdl:FileSystem name="Custom VO volume">
    <jsdl:Description>
        An example of expressing a requirement for the job
        to have the volume with the name/ID myVOvolume
        available from the node's local path
        /mnt/xtreemfs/myVOvolume. Here we assume that
        the volume can be expressed with:
        xtreemfs:volumeID
    </jsdl:Description>
    <jsdl:MountPoint>
        /mnt/xtreemfs/myVOvolume
    </jsdl:MountPoint>
    <jsdl:MountSource>
        xtreemfs:myVOvolume
    </jsdl:MountSource>
</jsdl:FileSystem>
```

5.2 Integration points

5.2.1 User Account Management

Like the conventional Linx/Unix based file systems, XtreemFS itself doesn't manage users. Instead, it relies on the user management facility provided by the XtreemOS VO Membership Service (X-VOMS), a component being developed by WP3.5 - security and VO management. In XtreemFS, a user is identified by a Global User Identifier (**GUID**). Two distinguished GUIDs represent two different users. These GUIDs, like its counterpart UIDs in Linux, provide a unique way to identify users across the entire system. Upon registration with a VO Hosting system (**VOHost**), a user is allocated a GUID, which is a 32-byte alphanumeric identifier in hexadecimal format, such as: db7749c4-1972-42d6-807c-6ef550c90e9b[7]. The GUIDs are globally unique even when multiple VOHost systems are present.

[6] http://www.gridforum.org/documents/GFD.56.pdf

[7] Note that the hyphens are not part of the GUID. They are included for presentation purpose.

It should be pointed out that it is possible for a user to register with a VOHost (thus, giving a GUID) without being associated with any VOs. This is supported in the current X-VOMS/VOHost implementation. A user can register with one VOHost multiple times or with multiple VOHosts, thus acquiring distinct GUIDs. However, in those cases, the user will be treated as different users, each identified by a GUID.

5.2.2 User Credential Management

In addition to GUIDs, Global Virtual Organization Identifiers (**GVIDs**) and Global Group Identifiers (**GGIDs**) are also useful credentials for XtreemFS. Similarly, these IDs are also managed by the X-VOMS. A GVID, generated when a VO is created, is associated with a user (i.e. the corresponding GUID), thus effectively making the user the owner or a normal user of the VO. Such an association can occur while the VO is created or after[8].

GGID, also generated by the X-VOMS upon the creation of a new group in a VO, is associated with a user when a user joins a group in a VO. The relationship between a VO group and a GGID is one-to-one. This is a VO model being supported in the M24 release. Putting it simply, a VO model is the type of attributes that a VO supports. For example, the EGEE VO membership service (VOMS) supports four types of vo attributes: role, group, subgroup, and capability. A VO creator can choose which attributes his VO supports.

In the future releases, GGIDs can be associated with other attributes under different VO models. For example, it is possible to associate a GGID with a role of a group within a VO. That is, a GGID is uniquely mapped to a VO role within a VO group of a VO. Depending on the VO model, furthermore, it is even possible to associate a user (i.e. GUID) with different GGIDs, one for each combination of attributes in various VO models.

We have described GUIDs in the previous section. This section will concentrate on GVIDs and GGIDs, especially on how they can be used, together with GUIDs, to manage the Access Control Lists (ACLs) in XtreemFS.

To support file sharing among users globally, XtreemFS uses the concept of Global Groups. A global group, identified by a Global Group Identifier (GGID), is uniquely defined across the entire XtreemFS. GGID mimics the GID concept in conventional Linux based file systems to allow users to share their files globally, regardless of the number of physical machines that files reside on and the administrative domains under which files are being managed.

[8]The former is being supported in the first XtreemOS release (i.e. May/June 2008, also refered as M24 release).

5.2.3 Credentials Used in Mounting

Before a user or a job can access the data in the volume, the volume needs to be mounted on the local file system. The rights to *mount a volume* differ from the rights to *access the contents/files of that volume*. For the job execution purposes, there are two options:

1. **User's credentials.** The user on whose behalf the job is running needs the rights to access the volume and to have it mounted on the node. The service performing the mounting also needs to have the mounting rights on the node. This case will be necessary in the dynamic mounting scenario.

2. **Node's credentials.** The machines providing resources to the VO have the machine certificates, and they also have an attribute certificate for each VO they are members of. This means that the service making static mounts to the VO can provide the machine certificate with attributes proving the membership in a VO to mount a VO-related volume.

Mounting a volume is typically managed by a privileged process on a node. Like other conventional network file systems, MRC can also control how and who can access XtreemFS volumes. This is the access control on volumes. On the other hand, accessing the contents on a volume is jointly determined by a user's credentials stored in the user's XOS-Cert and the file access control rights/lists managed by the MRC. Let us focus on the former case as this section is concerned with mounting volumes.

Privileged Accounts for Performing Mounting There are two sides of the story. First of all, the account which performs $mount$ needs to be a privileged one. Quoting from the Linux man page,

Appropriate privilege (Linux: the CAP_SYS_ADMIN capability) is required to mount and unmount filesystems.

Thus, the service, be it a stand-alone daemon service or an AEM service, performing $mount$ has to be run by such a privileged user. For static mounting, this may not be an issue as the mount operation is performed by the node beforehand, for example, during the system boot process. The operation has nothing to do with users who use the volume. As long as the volume is pre-mounted, any non-privileged user can access it.

However, this could be a problem for dynamic mounting. One solution is to have a prvileged daemon service awaiting for such an on-demand mounting request, for example, based on some specific privileges given in the user's XOS-Cert. Once the service authenticates the mounting request, it shall mount the

volume on behalf of the user. Subsequently, this volume can be made available to the user and others.

An alternative is to map (for example, by the PAM module) the user, to a privilege account so that the job can trigger $mount$ itselve on the nodes without contacting an separate privileged service.

Volume Mounting Controls on MRC On the other hand, the MRC, the receiving end of a mounting operation, also needs to have security control in place. Similarly, this can either rely on user or node credentials. For example, it is possible to have a ACL on MRC to control who, be it a user or a node, is allowed to mount a particular volume. There are various conventional ways of coping with such. For the approaches using PKI, there are utilities such as pam_mount, which can mount volumes for a user session and unmount it when a user logs out. Another approach, also based on PKI is $sshfs$, which is part of $FUSE$ and allows mounting using the SSH protocol. For further details, please refer to [9].

5.3 Discussion

5.3.1 How to obtain volume information?

Although volumes can be mounted in different ways, it still remains open how the volume information gets into the nodes. Essentially, this can occur via one of the following routes:

1. volume info is known to users so that they can specify it in the *JSDL*;
2. volume info is kept by X-VOMS/VOHost and disseminated via the *XOS-Cert*; or
3. volume info is *well-known among all nodes* (i.e. to the resource admins who own the nodes) so that the nodes know how to 'look up' the volume information locally, given a XOS-Cert.

Option (1) is most demanding as it requires users to keep track of volumes themselves. This is cumbersome, and does not scale well as resource owners (of the nodes) need to keep *all users* informed of their volmne setups. Option (2) has better scalability than (1) as resource owners only need to keep the VOHost system informed and users are kept transparent to the volume issue. Option (3) is most transparent to users. They are not required to be aware of the existence of XtreemFS volumes as the complexity of facilitating the interactions between jobs and XtreemFS volumes is totally hidden from the users.

It should be emphasised that all these choices are based on the fact that volumes are created per VO. Effectively, this demands the following settings:

- there is an interface between VOHost and MRC to allow creation of volumes for VOs.
- there is an agreed volume naming scheme among users and resource owners. One candidate of naming volumes is to use a VO name as a volume name in the static (and default) mounting setting.

5.3.2 Optimizing Static Mounting

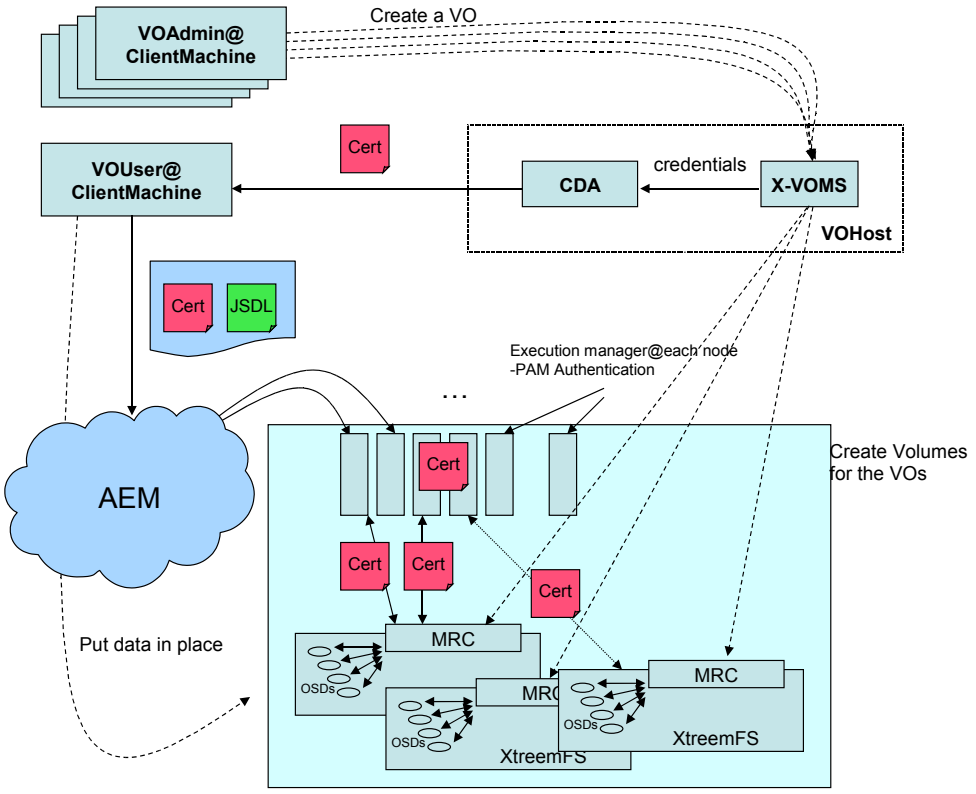

Figure 16: When we have multiple XtreemFS providers and VOs to host on the nodes, we can consider pre-mounting (solid lines) or lazy mounting upon need (dotted line).

In the process of mount-use-unmount of the volumes, we need to consider the performance of XtreemFS interactions when we have multiple storage providers. Figure 16 illustrates this case where, on the nodes, we host multiple VOs, each of which might have one or more XtreemFS provider. In this case, keeping all the

possible volumes mounted statically all the time may introduce a large overhead into the system.

Alternatively, we can consider "lazy" mounting procedure even for **the static case**. In this procedure, we do not actually mount the volume until the first job requires the data from the volume. However, while lowering the overhead, the procedure would complicate the system, requiring it to keep accounts on the accesses and periodically unmount those volumes that have not been used for a certain time duration.

5.3.3 Lifecycle of VOs and Volumes

So far, the discussion has been based on one volume per VO. This strategy gives the benefit of managing files belonging to different VOs easily. However, in some cases (e.g. short-lived VOs or sharing among VOs), the one volume per VO strategy may incur unnecessary overheads or even make the mount-unmount operations overly expensive.

An alternative solution would be to allow all VOs belonging to a VOHost to share one common volume. In this strategy, files belonging to all VOs can be found on a well-known mount point on a single volume. Thus, the mount point and path to the volume remain the same throughout the lifetime of the volume, regardless of

- whether it is static or dynamic mounting and
- however many VOs are created dynamically.

For example, a VOHost system hosts three VOs: VO_1, VO_2, and VO_3. The named volume on a well-known mount point, say $/mnt/xtreemfs/VO\_FOR\_VOHOST1$, for the VOHost is allocated when the VOHost establishes connections with the MRC. The path to each VO directory becomes:

- $/mnt/xtreemfs/VO\_FOR\_VOHOST1/VO1$,
- $/mnt/xtreemfs/VO\_FOR\_VOHOST1/VO2$ and
- $/mnt/xtreemfs/VO\_FOR\_VOHOST1/VO3$.

This strategy may be able to reduce the complexity of managing volumes and could be suitable for hiding away the complexity of managing VOs from XtreemFS. However, because files belonging to all VOs are mounted, it still remains unclear whether the overheads are acceptable.

6 Conclusion

In this document we addressed the aspects which are of increasing interest at that point in time since the XtreemOS project more and more will be integrating the different components. The goal is to provide different variants of possible solutions and support an efficient decision process within XtreemOS.

The first question we picked up is how access control within XtreemFS can work. Here we reviewed the literature and pointed out the applicability of these approaches to XtreemFS. This work is actually two dimensional, since on one dimension different mechanisms can be used and on the second dimension different cryptographic mechanisms are proposed for the application within XtreemFS. Since crypto algorithms can usually be exchanged very quickly due to their similar interfaces we put into account if implementations exist and are compatible to in terms of the software license.

The third part of the document is focussing on the integration of XtreemFS together with XtreemOS. Here we focus on what has to be prepared to have everything in place once a job starts its execution. Starting from the current state of implementation we sketched what kind of information has to be provided in order to make the binaries and the job data available in order to execute the job. Again we provide different alternatives in order to prepare an efficient decision process within XtreemOS.

References

[1] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Mike Burrows, Timothy Mann, and Chandramohan A. Thekkath. Block-level security for network-attached disks. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 159–174, Berkeley, CA, USA, 2003. USENIX Association.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, pages 1–15, 1996.

[3] CERTICOM. Ecc - the eliptic curve cryptosystem. Certicom Website, 1998.

[4] XtreemOS Consortium. *D3.4.1 - The XtreemOS File System - Requirements and Reference Architecture*, November 2006.

[5] XtreemOS Consortium. *D3.4.2 - XtreemFS prototype*, November 2007.

[6] Michael Factor, David Nagle, Dalit Naor, Erik Riedel, and Julian Satran. The osd security protocol. In *SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)*, pages 29–39, Washington, DC, USA, 2005. IEEE Computer Society.

[7] SNIA Technical Working Groups. Storage management initiative specification. SNIA Website, 2007.

[8] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. *Lecture Notes in Computer Science*, 1423:267–288, 1998.

[9] johns. pam_mount and sshfs with password authentication. Debian administration web site, 2008.

[10] Lamport L. Constructing digital signatures from a one-way function. In *SRI Intl. CSL-98*, 1979.

[11] Andrew W. Leung and Ethan L. Miller. Scalable security for large, high performance storage systems. In *StorageSS '06: Proceedings of the second ACM workshop on Storage security and survivability*, pages 29–40, New York, NY, USA, 2006. ACM.

[12] Andrew W. Leung, Ethan L. Miller, and Stephanie Jones. Scalable security for petascale parallel file systems. In Becky Verastegui, editor, *SC*, page 16. ACM Press, 2007.

[13] Ethan Miller, Darrell Long, William Freeman, and Benjamin Reed. Strong security for network-attached storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 1, Berkeley, CA, USA, 2002. USENIX Association.

[14] Christopher Olson and Ethan L. Miller. Secure capabilities for a petabyte-scale object-based distributed file system. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 64–73, New York, NY, USA, 2005. ACM.

[15] Certicom Research. SEC1: Elliptic Curve Cryptography. Certicom Website, 2000.

[16] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.