Project no. IST-033576

XtreemOS

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

The XtreemOS File System - Requirements and Reference Architecture
D3.4.1

Due date of deliverable: 30-NOV-2006
Actual submission date: 21-DEZ-2006

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

Type: Deliverable
WP number: WP3.4

Responsible institution: ZIB
Editor & and editor's address: Felix Hupfeld
Zuse Institute Berlin
Takustrasse 7
14195 Berlin
Germany

Version 1.1 / Last edited by Felix Hupfeld / 21-DEZ-2006

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
0.4	23.10.06			As sent out for review of document structure
1.0	23.11.06			Final submission to INRIA
1.1	21.12.06			Submission of harmonized version to INRIA

Contents

1 Introduction **4**

1.1 Document Structure 5

1.2 Features and Feature Lists 5

2 General Requirements **6**

2.1 Fault Tolerance and Scalability 7

2.2 Federation 8

2.3 POSIX Compliance 8

2.4 Requirements by Other WPs 10

2.5 Use Cases 12

3 Architecture **15**

3.1 XtreemFS and existing object-based file systems 15

3.2 Overview 16

3.3 General Definitions and Concepts 17

3.4 System Components 17

3.5 Architectural Artifacts 18

3.6 Interaction between Components 21

4 Object Storage Device (OSD) **28**

4.1 Objects and Object Storage Devices 28

4.2 Multi-Object Files 29

4.3 Transactional Files 30

4.4 Architecture 31

4.5 Feature List 32

4.6 Open Issues 37

4.7 Interface 38

4.8 Research Prospects 38

5 Replica Management Service (RMS) 40

5.1 Replication Policies 40

5.2 Architecture 41

5.3 Open Issues 47

5.4 Interface 48

5.5 Research Prospects 48

6 Metadata and Replica Catalogue (MRC) 50

6.1 Security 50

6.2 Architecture 52

6.3 Feature List 53

6.4 Testing 58

6.5 Open Issues 58

6.6 Interface 59

6.7 Research Prospects 59

7 Object Sharing Service (OSS) 61

7.1 Architecture 61

7.2 Feature List 65

7.3 Interface 68

7.4 Research Prospects 68

8 Access Layer 70

8.1 Introduction 70

8.2 Approach 70

8.3 Feature List 73

8.4 Implementation 80

8.5 Research Prospects 85

A Relations to other WPs **87**

A.1 Service Discovery - WP3.2 87

A.2 Publish/Subscribe - WP3.2 88

A.3 Remote Execution - WP3.3 88

A.4 Replica and Job Collocation - WP3.3 88

A.5 SSI - WP2.2 88

A.6 Checkpointing - WP2.2 89

A.7 VO Management - WP2.1 89

B Protocols **89**

B.1 HTTP Protocol for XtreemFS 90

C FAQ **91**

Executive Summary

This document describes XtreemFS, a federated object-based file system for Grid environments, and the Object Sharing Service (OSS), which facilitates for inter-process communication for applications.

The object-based file system architecture splits files into their pure content, the objects, which are stored on so called object storage devices (OSDs), and the file metadata, which is put on dedicated metadata servers. XtreemFS extends the architectural concept of object-based storage to Grid environments by replacing the centralised metadata servers with a federation of metadata servers in order to ensure independence of participating organisations while maintaining a global view of the system.

In order to achieve scalability and fault tolerance, XtreemFS will also feature replication and partitioning/striping for both file metadata and the file content. Dynamically created communication overlays will coordinate concurrent accesses and ensure the data's consistency in a scalable way. Data can be replicated across organisation boundaries, and therefore special attention needs to be paid to the latencies that the connecting wide area networks introduce and to failure cases like those of possible network partitioning.

Files can not only be placed manually to different stores, but an automatic system will monitor file access and resource conditions to automatically optimise data layout in the Grid. In addition, semantic naming and advanced query functions will allow users to find data in huge archives, with the aim of overcoming limitations for the organization of data of traditional hierarchical file systems.

The Object Sharing Service (OSS) will provide inter-process communication mechanisms via volatile memory, mapped files, dynamically allocated objects and grid pipes.

In the following chapters we review the requirements to the data management of XtreemOS, and describe the architecture of XtreemFS in general along with the design of its individual components. For each component, we also sketch a possible evolution of its implementation and name open issues and research prospects.

1 Introduction

This document is a draft of the architecture of XtreemFS and the Object Sharing Service. The architecture identifies *components*, their *interdependencies* and *interfaces*. Moreover, it incorporates use cases, requirements and issues.

1.1 Document Structure

Section 2 lists the general requirements and objectives of XtreemFS and the Object Sharing Service (OSS). It also includes comments on requirements to XtreemFS by other work packages (section 2.4) and a list of use cases (section 2.5) developed in cooperation with other work packages. The overall architecture and interaction of components is described in section 3. The following sections 4, 5, 6, 7 and 8 give details on the individual components, including a list of features and technologies for each component. Appendix A lists requirements to other workpackages. Appendix B gives an overview of technologies that the system will use.

1.2 Features and Feature Lists

The features listed in the respective sections are meant as a guideline to the implementation and research that will be conducted in this workpackage. Most aspects of our file system have not been implemented routinely. Thus, feature lists do not serve as a strict schedule but rather help to structure and synchronise our intended efforts within the workpackage. The features XtreemFS will expose to the environment are listed in section 2.

For the feature lists throughout the document the following template is used:

ID: Feature ID	
name	short name of the feature
type	the type of the feature can be one of • **basic** are features that are necessary to build a working prototype. • **advanced** are features that require more elaboration and are not absolutely necessary for a working prototype. • **research** features are similar to advanced but require additional research. • **optional** features add extra value to the system but are not part of the core functionality. They may be changed or not be implemented at all.
dependencies	a list of other major features on which the feature depends
description	a short description of the feature

2 General Requirements

Aiming to build a full-featured file system for XtreemOS as a software product, we have gathered features that are commonly required for file systems. This section states requirements that are derived from the project's description and goals, from file system literature, and from common features of other file systems. Likewise, input from other workpackages (incl. users) has been incorporated.

While XtreemOS is a research project, it is also expected to deliver software ready for production environments. Because data management is an integral part of the overall system, the reliability of XtreemFS is essential for the success of the entire project. Bearing that in mind, this document differentiates between basic and advanced features. *Basic features* are essential for a usable data management system and will be implemented during the first eighteen months. *Advanced features* are part of the research prospects of XtreemOS. They have not been routinely implemented before and aim to increase the attractiveness of the XtreemOS platform.

Other workpackages are more likely to depend on basic features than on advanced features. Therefore, basic features should be available earlier than advanced features in order to set the stage for progress with both interfaces and implementations. In turn, advanced features require less interaction with other workpackages and can therefore be studied with the required patience.

As a distributed system which encompasses systems from multiple organisations, it is important that:

- it is fault tolerant, so that outages do not affect the overall system and are automatically handled
- it is scalable, i.e. increased performance demands can be matched with proportionally adding more machines
- organisations are independent, i.e. can join and leave at will, and they can work without connectivity (referred to as federation)
- it is loosely coupled and asynchronous, so that failures and temporary performance issues do not spread over the whole system

Being a platform for distributed UNIX software, XtreemOS will need to develop a UNIX-like file system. Because of the laws of nature of distributed systems, which are especially relevant in a WAN-system like XtreemOS, trade-offs will have to be made for its UNIX compatibility. These trade-offs should be made visible or even accessible to client applications.

The main aspects of data management in XtreemOS are mainly user and group files and their sharing for collaboration purposes. Expecting to handle large numbers of files, it is recognised that the directory hierarchy and filenames of traditional file systems are not sufficient. In addition to hierarchical directory structures, the system hence requires support for extended meta-data. A semantic naming of files allows a database-like arrangement of the file system, including a retrieval of files by means of queries based on attributes.

To be attractive to a wide audience, the file system is supposed to run on commodity hardware. That means it does not assume the presence of unusual large amounts of main memory (>4 GB), RAID protected drives, etc.

In short, the system should

- exhibit UNIX-like (POSIX) behaviour where possible,
- support tunable trade-offs where necessary and possible,
- support extended metadata,
- support but not require hierarchical names (i.e. a directory structure),
- support private data, shared and collaboration data, and data archives,
- run on commodity hardware.

2.1 Fault Tolerance and Scalability

As a system running on standard hardware connected with WAN links, the system must tolerate hardware outages. Moreover, it must handle amounts of data and requests that exceed the capacity of single machines. This is achieved by distributing the system over multiple machines. Adding more machines to the system should not involve a disproportional increase in overhead.

2.2 Federation

XtreemFS will keep data of many institutions. While it allows a global view on all data available in the system, it must also guarantee the availability of data in presence of network disconnections or institutions leaving the federation of systems.

Institutions should be able to use their local system with their local data when they are disconnected or when parts of the overall system are disconnected.

2.3 POSIX Compliance

To support traditional UNIX applications, we have to be POSIX compliant. Because strict POSIX compliance severely restricts scalability of the file system, we allow XtreemOS-aware applications to fine tune guarantees in order to improve their performance.

This section aims to list all aspects of POSIX compatibility, and identifies their impact on the general architecture. Because of the lack of free availability of the POSIX standards, we refer to the Single UNIX Specification Version 3 [8] instead.

The specification basically deals with the following aspects:

- read/write consistency
- access control
- hierarchical file space
- file metadata
- extent locking

2.3.1 Read/Write Consistency

In POSIX read and write operations are seen as atomic. So read operations do either see the entire block written or nothing. However, this behaviour is only *intended* and not mandatory according to [8]. Future versions of the standard may see this as mandatory!
Tests on Linux systems showed that concurrent read and write operations on local filesystems (e.g. ext3 or ReiserFS) are *not* atomic. Tests for NFS will be conducted.

For subsequent write/reads [8] requires something that could be interpreted as strict consistency:

> "Writes can be serialized with respect to other reads and writes. If a read() of file data can be proven (by any means) to occur after a write() of the data, it must reflect that write(), even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from write() calls to subsequent read() calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics." [8]

2.3.2 Access Control

In short, POSIX defines an access control model based on users and group of users, and defines read/write/execute privileges for each filesystem entity (e.g. files or directories). Access control is evaluated recursively, i.e. the access rights of directories in the file's paths influence the access rights for the file, see [4], [20].

2.3.3 File Metadata

According to [8], delayed, periodic updates to file times are allowed (section 6.5). Moreover, some file systems have options to turn off `atime` updates to increase performance (`noatime` mount option) [15]. Other metadata requires online (immediate) updates, e.g. the filesize.

2.3.4 extent locking

POSIX includes facilities to lock byte-ranges of files. XtreemFS will support these mechanisms.

2.3.5 hierarchical file space

Like traditional filesystems XtreemFS will support a hierarchical file space (i.e. directory tree).

2.4 Requirements by Other WPs

2.4.1 R67

> "It must be possible to concurrently read from files which are open for write access. It must also be possible to concurrently write to files open for reading. [...]"

Concurrent read/write access is possible, and the system will provide real POSIX, actual UNIX, or application-defined consistency guarantees. Locking is not mandatory.

2.4.2 R71

> "It should be possible to check for file data and metadata changes. It should also be possible to subscribe to the information on file data and metadata changes, e.g. through registering a callback function."

Change notification mechanisms are supported via Feature MRC 4.4. However, most POSIX metadata (size, atime, etc.) will be updated lazily for performance reasons.

2.4.3 R72

> "File access time must be below 10 s to prevent time-out errors in applications. The data access time for interactive applications using a database must be below 150 ms. Furthermore, data access is very frequently. [...]"

Our system will provide means for partitioning and replication for load balancing. These features and the latencies will strongly depend on the network, switches, layout, cluster design, and the like.

2.4.4 R73

> "It must be possible to limit the number of replicas. Some applications need this possibility because of copyright, storage space, security, and performance constraints."

Replication policies will cover most of these aspects. Client-side built-in data encryption could also be useful for such scenarios.

2.4.5 R74

> "The versioning of data to allow incremental changes is required. [...]"

WP3.4 will investigate how copy-on-write techniques can be used to make versions of data available without sacrificing the overall architecture and system performance. We will try design in the possibility of this feature, but no statements can be made on when it might available, because it is a rather special and advanced feature.

2.4.6 R75

> "It must be possible to explicitly copy/move data between different resources. [...]"

WP 3.4 will provide access pattern-aware replication management. Behaviour can be controlled via fine-grained policies. Manual creation of replicas will be supported as well.

2.4.7 R76

> "The GOM layer must support an object based access/sharing. The GOM layer must support transactional consistency for object sharing. This includes restartable transaction combined with optimistic synchronization."

WP3.4 will provide object-based sharing combined with transactional consistency by the OSS service.

2.5 Use Cases

This section gathers use cases. They are not prioritised or guaranteed to be supported.

1. User rights management
   - Alice wants to store files in a file space and share them with Bob.
   - Bob can read and write files in this space.
   - In addition, Alice would like to give read permission to a group of users that does not include Bob.
2. Multiple file spaces
   - Alice works on project A, Bob works on project B. Both projects have nothing in common.
   - Alice does not want to see files of project B, while files of project A are not of interest for Bob.
   - Both Bob and Alice only want to be aware of files belonging to projects they are working on.
3. Efficient data retrieval
   - Alice is a chemist and works with approx. 35,000 files containing protein information.
   - Alice regularly searches for certain proteins based on additional information contained in the files.
   - Then she creates lists of matching protein files to feed into her programme.
   - She does this regularly with repeating queries.
4. Well-known interfaces
   - Bob has a huge amount of images on his Linux computer.
   - He wants to copy these files into the new XtreemFS repository.
   - Bob is always in a hurry and too lazy to learn how to use a fancy graphical UI programme for a new storage system. He wants to 'cp -a' his images.
5. Separate rights management for meta attributes

- Chuck is a student who has been assigned the task of making annotations to Alice' latest research results.
- Since Alice wants to keep him from accidentally destroying her data, she only grants him read access to the data while granting full access to the attributes.

6. Named grid pipes

   - Alice has a workflow where different processes produce data and some others consume it.
   - These processes may be executed at different nodes.
   - Producers should be able to send data to a named grid pipe and consumers should be able to read data from the pipe.

7. Job execution at different VOs

   - Alice belongs to two non-overlapping VOs.
   - Alice has a job that can be run in both VOs indistinctly.
   - This job uses a set of files distributed in the Grid.
   - The job should be able to access the files regardless of the VO in which it is executed.
   - The job should be able to locate the files in the same way regardless of the VO in which it is executed.

8. Exporting local files

   - Bob has a set of files in the home directory of his laptop.
   - Bob should be able to make the files globally available without copying them.
   - Jobs from any node should be able to access Bob's local files when they are exported.

9. "transactional files"/file snapshots/copy-on-write files - this use case is related to a functionality proposed in WP3.3

   - Alice has a very urgent job that can be run at different sites (or at different VOs).
   - The application execution service decides to queue the job in several queues.

- Once the job is started in one queue, it is cancelled from the remaining queues.
- As we cannot guarantee atomicity, an instance of the job may have been started before it is cancelled, and thus has potentially modified files.
- The application execution environment should be able to roll back all changes in files done by the instance being cancelled.
- This transactional behaviour should not be the default but on demand.

10. sequential processes relying on a file

    - Process A writes its results into a file.
    - Then A sends a message to process B indicating that it has written its output into the file.
    - Process A calls a method to enforce synchronisation (e.g. `fsync`).
    - Process B waits for the message and upon reception reads the file.
    - Process A writes to ODS1 and sends the message when writing to OSD1 was completed.
    - Process B reads from OSD2 (holding a replica of the file).

3 Architecture

File data in XtreemOS will be handled by XtreemFS, which is an object-based federated file system. This section describes the architecture of XtreemFS by reviewing the interactions between its components.

3.1 XtreemFS and existing object-based file systems

Object-based file systems [6] are the state of the art in commercial distributed file systems and are still a hot topic in research [19]. They are used in production deployements in both industry and research in sizes exceeding 10.000 clients [20]. Only highly specialized file systems like the Google File Systems (GFS) [7] are presumably used in larger installations. Object-based file system products and services are available from a number of a number of commercial vendors, including Lustre [1], Panasas [2], and IBM's GPFS [15].

Available object-based file system were designed with a the corporate data center in mind. They assume that both metadata servers and object storage devices are hosted in central facility on dedicated reliable hardware. Fault tolerance is usually achieved rather with hardware measures like RAID instead of software solutions.

In contrast, the operating environment of XtreemOS is characterized by a Grid with file system deployements at participating organizations. These deployements are likely to run on commodity hardware for economic reasons, they are connected by wide area networks, and participating organizations want to stay independent in that they can leave the Grid at anytime and their local availability of data does not rely on the Grid operation.

XtreemFS is designed with this operating environment environment in mind. In contrast to the centralized approach of existing object-based file systems, XtreemFS will consist of a federation of single XtreemFS installations at participating institutions while maintaing the view of a global file system. Federation will ensure that members can join and leave the Grid at anytime, and that local operation is not affected by remote outages.

In order to be able to ensure high-performance access to all data and to increase availability of data, XtreemFS will allow replication at both file metadata and file content level, a feature not available in existing object-based file systems. In addition, replication can be done manually, but a monitoring of data access will be put in place that is able to automatically

make replication decisions in order to optimize the layout and access of data in the Grid.

Also, XtreemOS will help its user to build high performance file system installations out of commodity hardware by parallelizing access to both metadata and file data. To this end, file data can be striped across storage devices, which allows parallel IO to a large amount of hard disks, and metadata can be partitioned across metadata servers, which makes large file volumes tractable with off-the-shelf hardware.

3.2 Overview

XtreemFS is a distributed file system structured according to the *object-based file system* approach [12][6]. Its core abstraction, the object, is the pure content of a file without its metadata.

The system consists of the following components:

- Object Storage Device (OSD) stores objects and implement a read-/write interface to them.
- Metadata and Replica Catalog (MRC). This component stores file metadata (extended and POSIX) and replica locations of files. It also make authorisation decisions according to access policies.
- Replica Management Service (RMS). This component will cooperate with other services to decide when and where replicas are created and when replicas should be removed from the system.
- The Access Layer consists of a client-side library and a POSIX compatible file system module. The library offers access to all XtreemFS features for XtreemOS aware applications. The file system module allows mounting of XtreemFS as part of the traditional UNIX file system layout.
- XtreemFS supports applications with an Object Sharing Service (OSS). It allows sharing of data residing in volatile memory with object granularity. In this context, objects are volatile memory regions containing dynamically allocated objects and/or memory-mapped file data.

3.3 General Definitions and Concepts

3.3.1 Physical Environment

Hosts are all XtreemOS machines in the Grid.

Storage Hosts are hosts that have disks attached that are available for storing file data and run OSD instances.

Server Hosts are hosts that have resources to run MRC instances.

3.3.2 Administrative Environment

An **organisation** is an administrative real-world body (like an institute, a university or a laboratory).

A **virtual organisation** (**VO**) consists of entities of multiple organisations that are connected for collaboration purposes, typically through resource sharing. An organisation can be member of multiple VOs. VO membership is usually implemented by accepting user credentials and enforcing associated policies. Organisations shall be able to join and leave VOs at any time. This definition is in agreement with the one from WP 3.5.

3.4 System Components

3.4.1 Object Storage Device (OSD)

The task of the Object Storage Devices 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.

3.4.2 MRC

The Metadata and Replica Catalog (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.

3.4.3 RMS

The Replica Management Service (RMS) is responsible for deciding when replicas have to be created and with what distribution among OSDs. This service is also responsible for removing replicas that are not needed or useful anymore.

3.4.4 Client

In this document 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 (MRC, OSD, RMS). This can be done transparently to the application through the traditional Linux file system interface. XtreemOS aware application can take advantage of the native XtreemFS interface through a library provided by the access layer.

3.5 Architectural Artifacts

3.5.1 Objects and Files

An **object** is a sparse array of bytes that is stored in an OSD, identified by a handle. The actual implementation of an OSD can choose how it wants to store the object (one file, multiple files, directories, etc.).

Files contain data and have metadata describing them. The data of files is stored in one or more objects. The metadata is kept separately in the Metadata Catalog (see MRC).

3.5.2 Multi-object Files

The data of a file may be stored in more than one object in the following cases:

- The file's data is cut into pieces which are distributed over several storage devices. Each piece is stored in a separate object. The file('s data) is **striped** into many objects.
- The data is redundantly stored (**replicated**) in several objects on different storage devices.

Both mechanisms can be combined. The file has then multiple copies ('instances') in the system, the replicas, each of which can be stored as one or multiple stripes in objects.

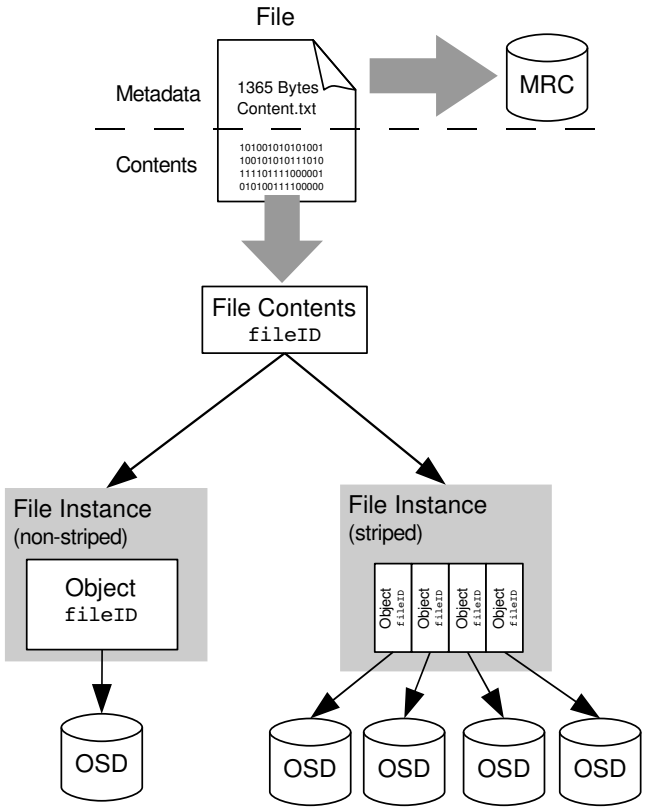


Figure 1: Relationship between Files, Objects and OSDs

In figure 1 the relationship between a file, its contents and the objects is shown. First the file is split in two parts, the metadata is stored on the MRC while the contents (raw data) is distributed among OSDs. Therefore, file instances are created, each holding one replica of the file's contents. These file instances can consist of one or several objects, depending on the striping pattern used. These objects are stored on OSDs.

3.5.3 Metadata

Any data maintained by the file system that describes an entity and is not part of the file data is considered as **metadata**. An entity in the file system

(e.g. file or directory) is associated with different kinds of metadata items, referred to as **meta attributes** or simply **attributes**. Certain attributes are required for POSIX-compliance, such as the size of the corresponding data or timestamps referring to last access, change of metadata or modification of content.

3.5.4 Additional Meta Attributes

Besides the required meta attributes, it should be possible to define new attributes with different scopes of visibility. Such attributes would be helpful in terms of arranging huge amounts of files in a grid file system. Instead of searching complex directory trees for data only by specifying file names, a user could describe and efficiently retrieve a set of files by specifying attribute constraints, e.g. by means of an SQL-like query language. Additional attributes could be added to files in different ways. Besides a user interface for manually setting user-defined meta attributes, there could be tools for an automatic extraction of meta information from file content. Non-POSIX metadata maintained by the file system could also be attached to files as additional meta attributes.

Some examples of metadata that could be held in the form of additional meta attributes are:

- personal annotations of single users, like "this file contains relevant information for me"
- information about replicas, replica locations and striping patterns of a file
- filetype-specific metadata, e.g. the dimensions of an image or the sampling rate of a sound file

3.5.5 Volumes

In order to structure the huge amounts of files in a cross-institutional file system, we introduce the concept of **volumes**. Volumes can be accessed by one or more users and contain a subset of the overall file space. Files can be shared between volumes.

Within a volume, files can be optionally arranged in the traditional way of a directory hierarchy, but can also be structured purely by their extended metadata.

Volumes can be mounted as a subtree of the normal UNIX file system.

3.5.6 File Access Capabilities

Access authorisation in the object-based architecture is split between the MRCs and OSDs. While MRCs make authorisation decisions according to their policies, OSDs have to enforce them later. The medium to convey this authorisation decision from an MRC to an OSD is the file access capability. When a client wants to access a file, it requests access from the metadata server, which hands out a capability to the client that securely captures its access control decision. The client presents this capability to a storage server, which is then informed about the MRC's decision, and grants the proper kind of file access.

File access capabilities are secured by encrypting them with a key that is shared between the MRC and OSDs. They contain:

- the `fileID` of the file that the client is allowed to access
- the operations that the client is allowed to perform
- some means for revocation (expiration time or object version)
- some means of client authentication (its IP address, or certificate subject for example).

3.6 Interaction between Components

In this section we describe the interaction between the XtreemFS components for basic file system operations. The file system is accessed by applications through the client interface. In turn, the client component invokes services provided by other data management components, i.e. MRC, OSD, RMS and OSS. In some cases it is also necessary that components interact directly with each other without involving the client, despite that behaviour should be avoided in general for scalability and performance reasons. The component diagram in Fig. 2 shows the different components and their interdependencies.

In the following we illustrate how components interact by showing details of the basic operations `read`, `write`, `create` and `delete`.

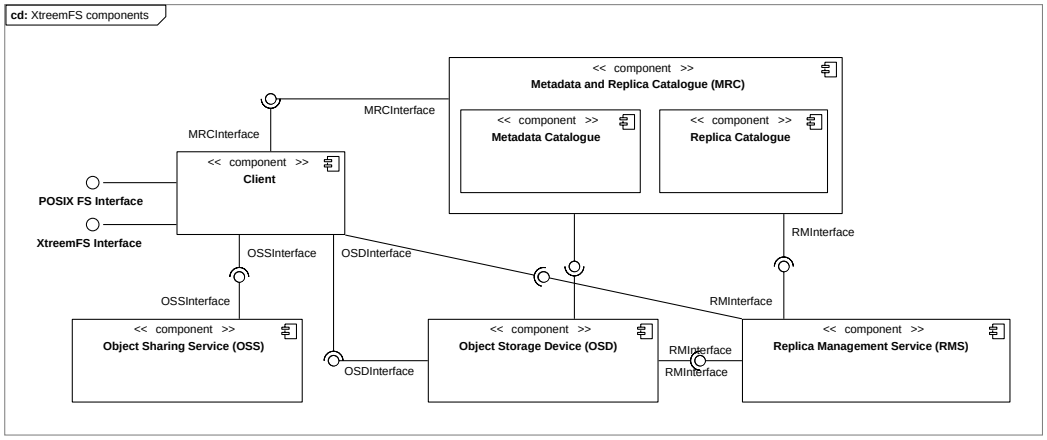


Figure 2: Components and their interdependencies

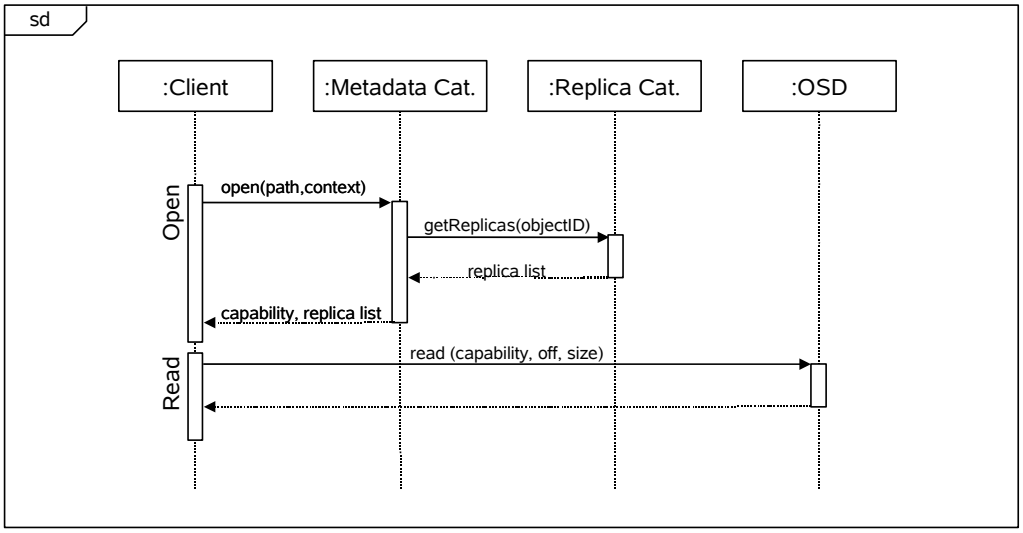


Figure 3: Interaction for reading a file with no replicas

3.6.1 Reading a File

Figure 3 shows a sequence diagram of the steps and actors involved when reading a file that is neither replicated nor striped. Some technical details are left out for the sake of simplicity.

A read operation consists of an open and a subsequent read request. The open operation at client side will contact the MRC, which retrieves a list of

replica locations of the file. In the present case it only receives one replica location because this file has not been replicated yet. In addition to the list, the MRC makes an access control decision according to the access policy and includes a file access capability in this response that allows this client to access the OSD directly. With this information, the client can read the object from the OSD by sending the offset and size of the request.

The second scenario for reading a file arises when the file is not replicated, but has been striped among different OSDs (Figure 4). In this case, the Metadata Catalog returns information for a single replica, but a list of OSDs and the policy used to distribute the data among the different OSDs. Using this information, the client can compute which offset of the file is stored on which OSD. Not that even a single read operation may have to interact with several OSDs.

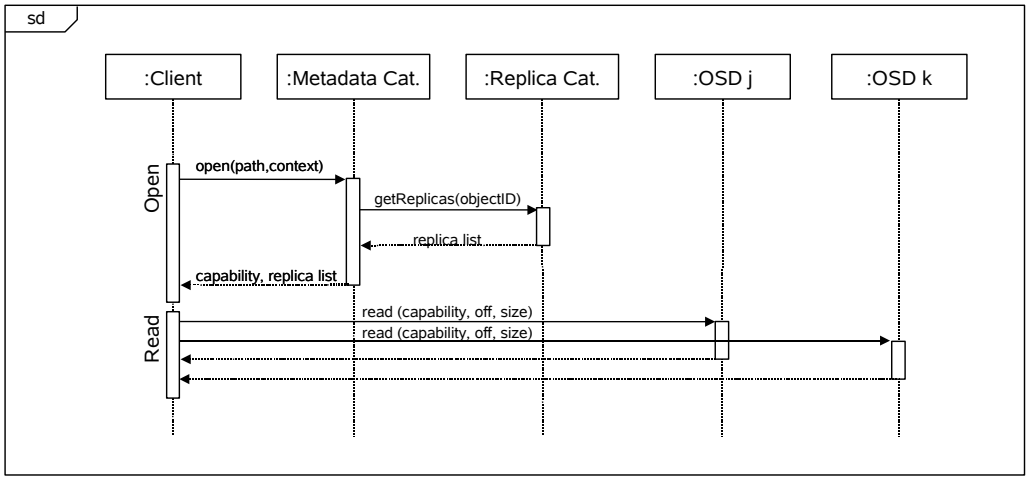


Figure 4: Interaction for reading a striped file

A third case arises when a file is replicated, but none of the replicas are striped (Fig. 5). As a response to the open request the client receives a list of OSDs that have a replica of the file. With this information the client can decide which replica is the best one to access and then act as if a single replica had been returned. A second option, the one presented in the figure, is that the client uses several of the replicas to perform parallel access to speed up a single client read operation. Once again, this implies more knowledge at the client side.

Combining these cases, we have a scenario where there are several replicas with some of them being striped.

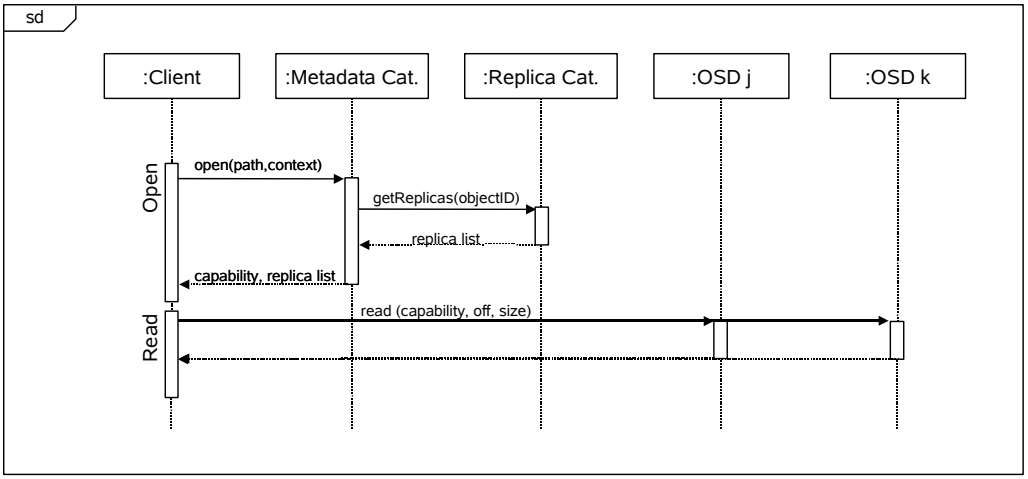


Figure 5: Interaction for reading a file with multiple replicas

3.6.2 Writing a File

A write operation is essentially equal to a read, but with the extra task of maintaining the other replicas up-to-date according to the consistency policy of the file. Figure 6 presents the interaction of the different actors when writing a file.

The first step is to open the file. This step is the same as for reading. The difference is that either as part of the open, as part of the first operation, or in the middle, the client has to inform the OSD that it will be accessing a given object. This step is intended for the OSD to contact the Metadata Catalog and request all available replicas of the given file. These replicas will be used in the future to update modifications according to the consistency policy. This means that OSDs will be responsible for update propagation.

As an option, the OSD may contact the other OSDs to work on potential optimisations, e.g. to know which OSDs are really being used and only update those OSDs immediately, leaving the rest of replicas for an off-line update. However, these optimisations still need to be worked out.

Once the client performs the write operation, the OSD takes care of propagating the update. The client call will not be acknowledged until all required replicas (according to the consistency policy) have been updated.

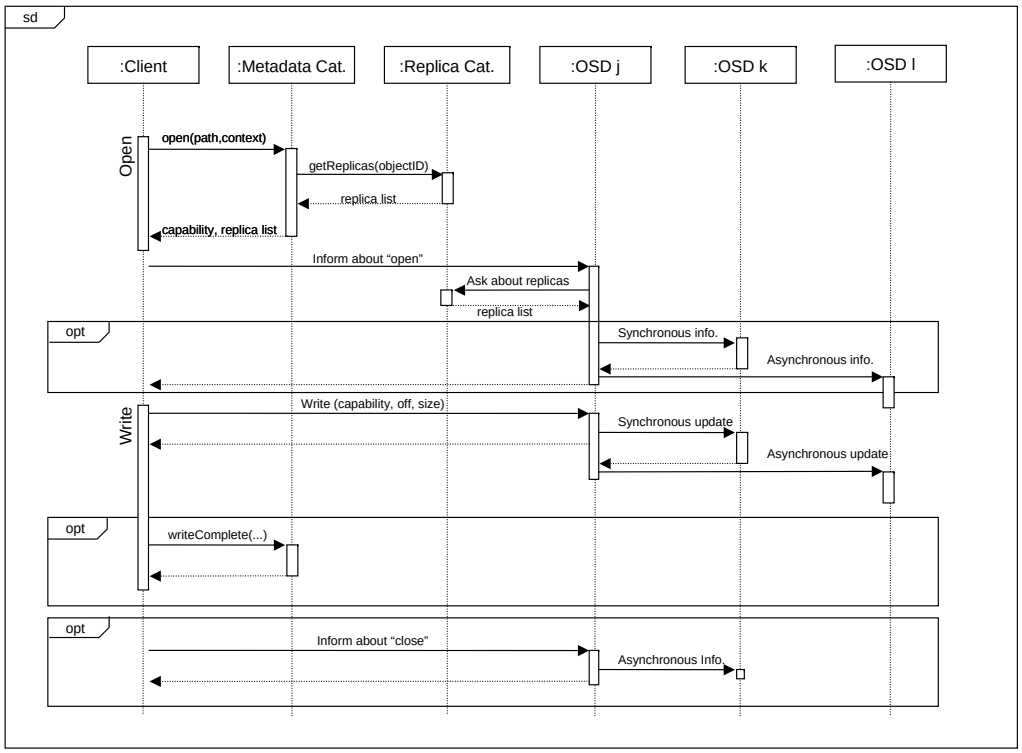


Figure 6: Interaction for writing a file

3.6.3 Creating a File

In order to create a file (see Figure 7) the client contacts the MRC for permission. The MRC will make use of its information about remaining OSD capacity and quota limitations in this step. If successful, it will add this location to its Replica Catalog. If necessary, it will also contact the OSD informing it that a new object is created. Once all these steps are done, the client will receive the same information as for an open and will act in the same way.

If the file was created in striped mode, nothing would change but the number of OSDs being contacted by the Metadata Catalog.

3.6.4 Deleting a File

Deleting a file is started by a client, but finally performed by the OSD (see Figure 8). The client accesses the Metadata Catalog to get permission to delete it. Once it has this information it contacts the OSD which will retrieve

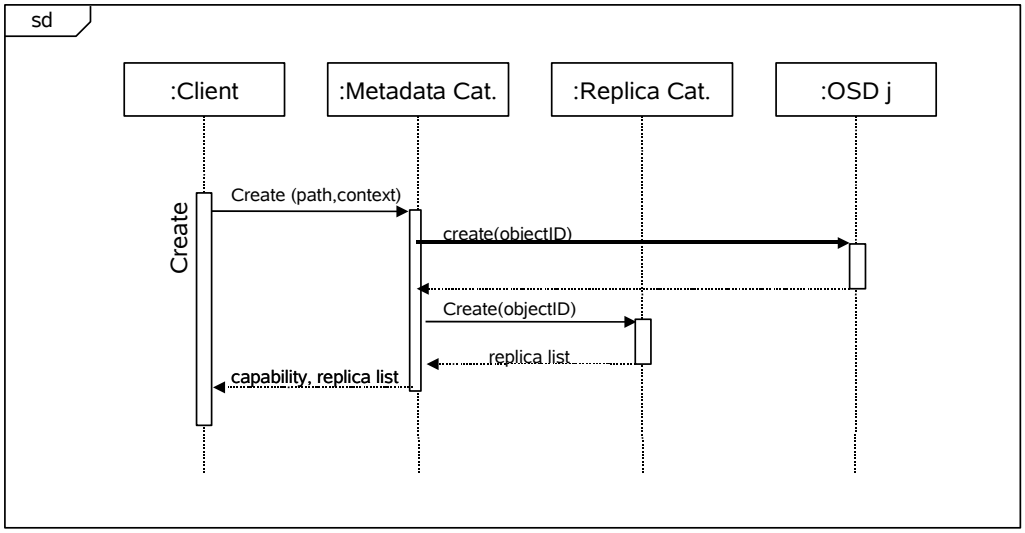


Figure 7: Interaction to create a file

a list of replicas and will take care that all of them are removed. In addition, it will also inform the MRC that this file is not existent anymore.

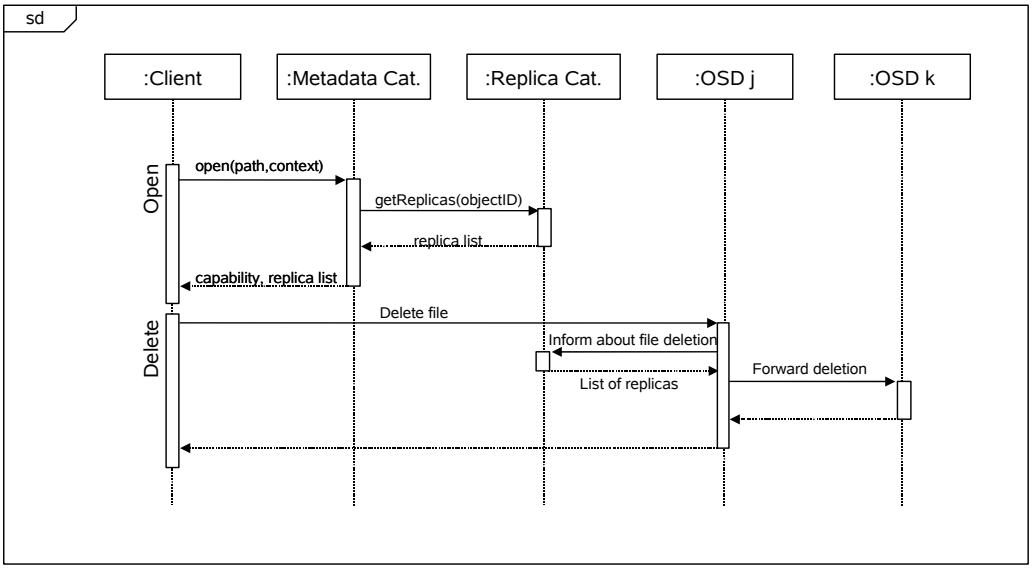


Figure 8: Interaction to delete a file

3.6.5 Creating and Deleting Replicas

Besides creating a new file, the system is also in charge of creating new replicas. This decision will usually not come from a client but from the RMS (see Figure 9). The RMS service will decide that a replica is needed and will contact a source OSD (or several soures if parallelism or striping is used) and the new OSD(s) where the replica will be placed. These two sets of OSDs cooperate to create the new replica and to copy the information (if needed). Finally, when the replica is created, the OSD that initiated the replication informs the Replica Catalog that the new replica is now available.

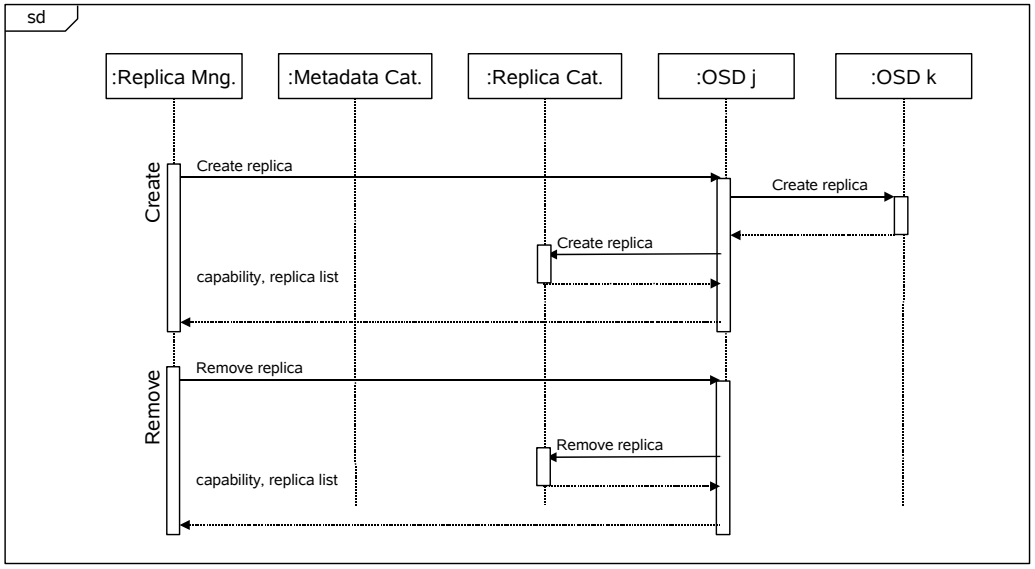


Figure 9: Interaction to create/delete a replica of a file

Finally, to delete a replica the same steps are done, but only the OSDs holding the obsolete replica are contacted.

4 Object Storage Device (OSD)

The OSD enables the creation of self-managed, shared and secure storage for network environments. Our storage architecture has been designed after the object-based storage paradigm, where the object is the fundamental unit of data storage and encapsulates the pure content of a file.

The task of an Object Storage Device is to provide functionality for data access in XtreemFS. Its main tasks consist in storing objects and offering an object-based storage interface to hide the complexity associated with underlying block-based storage mechanisms. Capabilities of the component include read/write access, concurrency control and communication with remote storage hosts.

The storage service runs on each OSD and handles access to file data. In addition, this service will also be in charge of implementing the "transactional files" described in Section 2.5. The idea of transactional files is that modifications will only be seen by the job making them until they are committed. If these changes are rolled back, they will be lost.

Furthermore, the OSD is an excellent place to perform monitoring that will be used by other services in the system. For instance, at this level issues such as access patterns, effective bandwidth and device load, among others, can be monitored. We should be aware that this monitoring (or part of it) can also be done in the access layer. The best place, if not a combination of both, still has to be studied.

The OSD service implements the functionality described by Task T3.4.1 (CNR) and Task T3.4.4 (BSC). An initial prototype is planned to be developed in Python. Later versions will be developed in Python/Java, with a partial reimplementation in C/C++ if necessary performance-wise.

4.1 Objects and Object Storage Devices

The *object* is the fundamental unit of data storage in the proposed service (see Section 3.5.1). Objects do not have any persistent information about their metadata. In order to support coordination of concurrent writes, open files might have a volatile knowledge about the whereabouts of their replicas, and track additional metadata like file size and access time in order to update the persistent metadata in the MRCs. Access control to objects is purely done by the MRCs, and communicated to OSDs via cryptographically secured capabilities.

An *Object Storage Device (OSD)* is essentially a container of objects, and its primary function is to reliably store and retrieve data from physical media. Like any conventional storage device, it manages the data as it is laid out into standard tracks and sectors. Each OSD provides an interface for accessing objects it contains. Furthermore, data is not accessible outside the OSD in block format, but only as an object. The OSD provides security enforcement for access to the objects it contains, but it does not provide security management, i.e. an OSD does not determine who is allowed to access an object. Indeed, as previously mentioned, objects do not have any persistent information about access capabilities. When an external entity (i.e. a client) wants to access an object, it requests permission from the MRC which is the unique manager of access capabilities for the objects. If the MRC returns a capability, the client presents it together with the request to the OSD.

OSDs raise the level of abstraction presented by a storage control unit from an array of blocks to a collection of objects. The object store provides fine-grained file-level security, improved scalability by localising space management, and improved management by allowing end-to-end management of semantically meaningful entities [3].

A file might be stored in more than one object on several OSDs. Such a scenario may occur in two different cases:
The first is when a single file is stored in several objects on many OSDs, i.e. when it is striped in pieces, with each one being stored in a separate object on a different OSD.
The second case consists of keeping several copies of the same file in different OSDs, either for fault-tolerance or for performance reasons. Both concepts can be combined, i.e. several replicas of a file exist and each replica of the file is striped. This could even include different striping sizes for the different replicas. These features will be analysed in the next section.

4.2 Multi-Object Files

In XtreemFS, striping is used in a more general sense. The idea is that pieces of file data can be distributed in a round-robin way among a set of OSDs. However, it does not mean that the objects placed in each OSD will have the same size. When a file or a file replica is created, the system decides on the striping policy to use. E.g. it may decide to place twice the number of bytes in one object than in the other objects. This decision will take into account the access patterns, the characteristics of the OSDs, the network to which they are connected, etc.

In addition, striping will be used on a per-replica basis. Thus, each replica may have a different striping policy. Such information will be part of the file metadata and will be kept by the MRC, as explained later.

Regarding replicas, it will be a task of the OSD to make sure that objects containing replicas of a file are being updated according to the consistency semantics defined for the file. Initially, OSDs do not keep information about other replicas. This information is only kept by the MRC. As soon as a client accesses a replica, it will obtain information about the current location of the replicas and will get in contact with all these replicas to start the needed coordination. Of course, in the case of strict sequential consistency, optimisations guarantee that most common operations such as reads or writes by a single client are as fast as if done locally. Optimisations will be mainly based on updating only OSDs being used and the rest will be updated off-line. On the other hand, multiple writes to many replicas will have a performance penalty in case of sequential consistency.

As a final consideration, although this kind of multi-object placement for files (striping and replication) is a very powerful mechanism, we need appropriate policies to limit its scope. For instance, we should be able to limit the places where a file's objects can be located. It makes no sense for a confidential file to be replicated to the disk of a user's laptop where the root account is not trusted. Legacy software may be another case that needs restrictions on where the object is really stored. A third case could be check-pointing of files. It may not be a good idea, in some cases, to place these objects on an OSD located at the same node where the application is running because if the node fails, there will be no way to migrate the application to a running node.

4.3 Transactional Files

Another concept that will be handled by OSDs is the transactional file. When a file is opened in transactional mode, all changes will be done in a copy at the OSD and changes will not be forwarded to other OSDs. If the file is finally committed (not necessarily at the close operation, but earlier) all changes will be updated following the consistency policy. If the file is rolled back, then all these changes will be lost.

4.4 Architecture

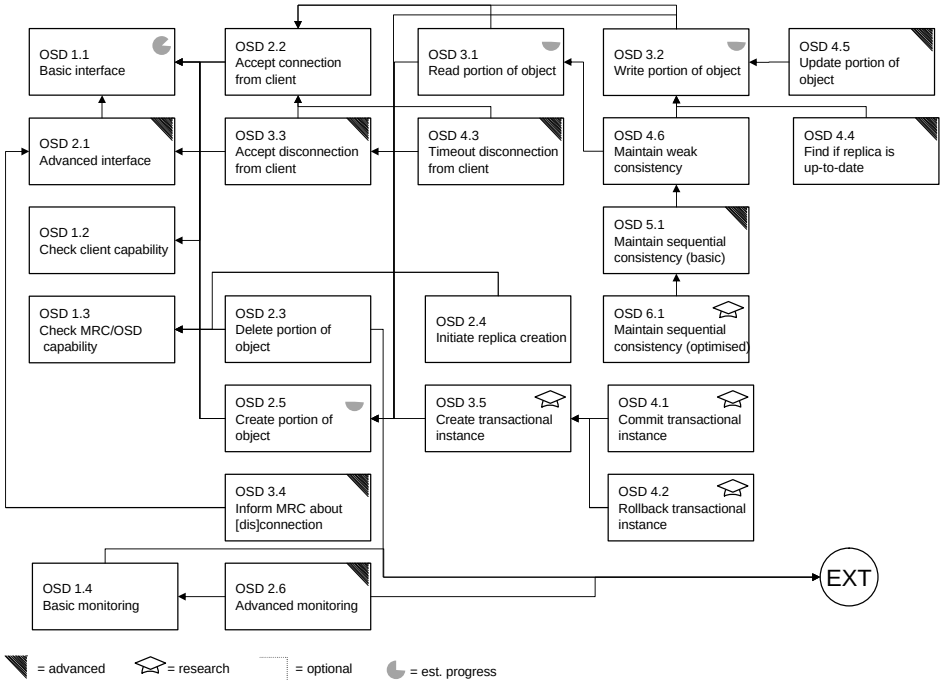
Three main entities compose the architecture of our module: the Storage Media, the Object and the Object Storage Device.

The Storage Media is any physical device storing data. The object is the fundamental unit of storage known in the system (see Section 3.5.1).

The Object Storage Device is an intelligent device that contains the storage media, a processor, RAM and a network interface that allows it to manage and store local objects, and autonomously serves requests from the network. Within the storage device, all objects are identified via their `objID`. In more detail, the object can be accessed by a simple interface based on the handle, the beginning of the range of bytes inside the object and the length of the byte range that is of interest.

In an object store environment, space is allocated and managed by the object-based storage architecture, not by higher level software such as a file system. Users of an OSD operate on data by performing operations such as creating an object, reading/writing at a logical position in the object or deleting the object. Since objects do not have persistent information about access capabilities, each request has to be accompanied with such information. For this reason, all operations carry a capability, and it is the responsibility of the OSD to validate capabilities carried by the user's request. It is evident that a client can handle an object through an OSD and work on it. Moreover, since no proxy functionality is performed, an OSD can access only an object it contains. Alternatively, it is possible to access an "external" object through the OSD holding it.

4.5 Feature List



ID: Feature ID	
name	the name of the feature
type	the type of the feature (one of 'basic', 'advanced', 'research', 'optional')
dependencies	other features on which the feature depends
description	a description of the feature

ID: OSD 1.1	
name	Basic interface
type	basic
dependencies	-
description	A basic interface with the features needed for the basic operations.

ID: OSD 2.1	
name	Advanced interface
type	advanced
dependencies	OSD 1.1
description	An interface containing, besides the basic functionality, all advanced functionality needed for the advanced and research tasks.

ID: OSD 1.2	
name	Check client capability
type	basic
dependencies	-
description	Every time a client want to connect, disconnect, read, or write an object, the OSD will need to check that it can do what it asks to do. Here we will take care of this checking.

ID: OSD 1.3	
name	Check MRC or RMS capability
type	basic
dependencies	-
description	Every time the RMS or MRC requests to create a replica or a portion of replica, as well as when it removes them, we need to guarantee that it has the privilege to do so. Here we will do this checking.

ID: OSD 2.2	
name	Accept connection from client
type	basic
dependencies	OSD 1.1, OSD 1.2, OSD 1.3
description	Although initially we might think that clients do not need to connect to the OSD to access objects, if we want OSDs to handle consistency with the rest of replicas, we will need to send the information we received from the MRC for them to be able to update the rest of OSDs. In addition it will help to have an updated version of connected clients to make optimisation in the replica update mechanism. The client connection will be associated with a lease.

ID: OSD 3.3	
name	Accept disconnection from client
type	advanced
dependencies	OSD 1.1, OSD 2.2
description	In the same way we need clients to connect, they also have to disconnect so we have a clear view of who is using a file to make future optimisations. This will actually be like returning the acquired lease.

ID: OSD 4.3	
name	Leases
type	advanced
dependencies	OSD 3.3, OSD 2.2
description	We ask clients to connect/disconnect, but we cannot guarantee that a client will disconnect. It can fail, become disconnected, etc. This is the reason we used leases. Leases have a timeout and after this timeout a new lease has to be requested to be able to access the protected object.

ID: OSD 3.1	
name	Read object
type	basic
dependencies	OSD 2.2, OSD 1.2
description	This is one of the basic operations for OSDs, read data from the devices and send it to the client.

ID: OSD 3.2	
name	Write object
type	basic
dependencies	OSD 2.2, OSD 1.2, OSD 2.5
description	This is the other basic operation for OSDs, receive data from the client and write it to the device.

ID: OSD 2.5	
name	Create object
type	basic
dependencies	OSD 1.1, OSD 1.2, OSD 1.3
description	In order to create new files and to create replicas we need to be able to create a new object in an OSD.

ID: OSD 2.4	
name	Initiate a replica creation
type	basic
dependencies	OSD 1.3, RMS 2.3, RMS 3.3
description	When the MRS decides to create a replica it will contact one or more OSDs to tell them that they need to copy their object (or part of it) to another OSD to create a new replica.

ID: OSD 2.3	
name	Delete object
type	basic
dependencies	OSD 1.1, OSD 1.3
description	In the same way we create objects, we also need to be able to remove them.

ID: OSD 3.4	
name	Inform MRC about connection/disconnection
type	advanced
dependencies	OSD 2.1
description	Although there will be mechanisms to detect failed OSDs, it is nice to know when an OSD leaves the system voluntarily. Here we will implement this mechanism that will contact the necessary services.

ID: OSD 4.4	
name	Find if replica is up-to-date
type	advanced
dependencies	OSD 3.2
description	As we will have optimised mechanisms to update replicas, it may happen that a replica that it is not currently being used is not up to date. If we want to use it, then we may need to update it for the user to find the correct data.

ID: OSD 4.5	
name	Update object
type	advanced
dependencies	OSD 3.2
description	After we have detected a given replica is not up-to-date, we may want to update it before allowing the user to use it. In this case, we will offer a mechanism for such an update. Nevertheless, we expect this to be a seldom used mechanisms because we plan to keep replicas updated as soon as possible (although not immediately for not used replicas)

ID: OSD 4.6	
name	Maintain weak consistency
type	basic
dependencies	OSD 3.1, OSD 3.2
description	In the first version we will offer some kind of weak consistency for files. (still to be decided)

ID: OSD 5.1	
name	Maintain sequential consistency (basic)
type	advanced
dependencies	OSD 4.6
description	In the second version of the file system we will provide a strict sequential consistency policy for files. This will be semantically correct, but not necessarily optimised performance wise.

ID: OSD 6.1	
name	Maintain sequential consistency (optimised)
type	research
dependencies	OSD 5.1
description	Once the basic version of the strict sequential consistency is implemented, we will work on ways to optimise it based on OSDs really being active. The goal will be to optimise the most common cases.

ID: OSD 3.5	
name	Create transactional instance
type	optional
dependencies	OSD 2.5
description	WP3.3 requested the idea of transaction files as described earlier. here we will create the necessary information to create such files.

ID: OSD 4.1	
name	Commit transactional instance
type	optional
dependencies	OSD 3.5
description	When a transactional file is committed, all changes are forwarded to all OSDs following the consistency policy of the file.

ID: OSD 4.2	
name	Rollback transactional instance
type	optional
dependencies	OSD 3.5, OSD 4.1
description	If the file is rolled back, all changes will be discarded.

ID: OSD 1.4	
name	Basic monitoring
type	basic
dependencies	RMS1.4
description	Monitoring the effective performance of the OSDs.

ID: OSD 2.6	
name	Advanced monitoring
type	advanced
dependencies	OSD 1.4, RMS 1.4
description	Monitoring the access patterns used in the objects.

4.6 Open Issues

- Disk space and quota handling.
- Update of POSIX metadata in the metadata server (times, file size). Who updates it (client or storage server), and when (periodically or on close)

- Protocol for keeping replicas in sync
- Protocol for coordinating concurrent reads/writes on multiple replicas
- Protocol for distributed extent lock coordination

4.7 Interface

The OSD provides the following simple interface for read/write operations:

- `Byte[] read(objID, size, offset)`
  returns `size` bytes read from the object identified by `objID` and starting at the specified `offset`.
- `int write(objID, size, data)`
  writes data of the object identified by `objID`, by appending it at the end of the object. It returns the exact offset/position (in bytes) at which data is started to be written.

For the communication between client and OSD, we intend to use the HTTP protocol. In fact, it seems sufficient to encapsulate all operations. If objects are accessed via HTTP, an object read is done via GET, a write via PUT, object deletion via DELETE, etc. The standard HTTP byte range header is used to name offset and length of the operation. The request URI has the format `/objID/capability`, where `capability` is the Base64 representation of the capability as defined in the architecture section. Byte ranges can be requested through the `Content-Range` header field, which is part of the HTTP 1.1 standard. This idea has to be refined, to provide a more detailed format for the HTTP request. In Appendix B more details are given.

Regarding monitoring, we need two sets of calls to *get()* and *reset()* monitoring information both for individual files and for the device. The information that we would get for a file would be its access pattern. On other hand, we would like to have global performance values such as the effective bandwidth for the device. These monitoring calls are not intended for end-users (although we could decide to make them available to end-users), but for other services in the system.

4.8 Research Prospects

In this section, we will mention some research topics and challenges. We could investigate one or more of them during the project. This is just a

preliminary list, and we will feel free to analyse some new topics, when they emerge (and if they are considered interesting) during the advancement of our work.

- **Efficient dynamic replica management**
  This file system will allow different levels of consistency for files and the efficient and scalable implementation of this consistency levels (especially the strict sequential consistency) will be a research challenge. The idea is to update used replicas in a synchronous way, and leave the rest of the replicas to a later update process.

- **Performance evaluation of a grid object-based storage system**
  Performance evaluation of the storage system is a fundamental point, because overall system performance will heavily depend on the file system performance; a find-grained monitoring of some suitable measures (i.e., disk I/O time) could be useful to evaluate the suitability of the architecture and it could lead to an improvement of the storage system.

- **Resource and file namespace design**
  Another aspect to be further studied is the Resource Namespace Service (RNS); the research effort, in this field, could provide namespace services for any addressable file by an easily accessible, hierarchically managed identifier.

- **File system grid services**
  We could analyse the way to make available our services as Grid Services, and to study suitable interfaces for them.

- **Intelligent storage data**
  It could be interesting to study some techniques aimed at a smart handling of objects in an object storage device; to achieve such a goal, we could employ data mining techniques for an efficient storing/retrieval of objects, based on their access pattern.

- **Definition of transactional files**
  This mechanism allows modifications to a file to be seen within a job, but not outside the job till they are committed. The research part of this feature is more in the definition of what is really needed and how to use it than on the implementation because the implementation can be easily done by duplicating files in the needed OSDs. This research should be conducted in close collaboration with WP3.3.

5 Replica Management Service (RMS)

The support of file replication has two important aspects. We need to maintain persistent meta-information about the files' replicas, including their locations. In addition, we need to make decisions on when or where to create new replicas or when to remove obsolete ones. These tasks are assigned to two services: the Metadata and Replica Catalogue (MRC) and the Replica Management Service (RMS).

The MRC, described in the next chapter, will take care of persistently storing meta-information about replicas, while the RMS, the service described in this chapter, will take care of autonomous creation and deletion of replicas. Therefore, this service will implement partially the functionality described in part of Task 3.4.2 (Replica Management service) (ZIB) and all the functionality described in T3.4.4 (Pattern Aware data Access) (BSC).

For deciding when to create new replicas we will take into account what jobs are being started and the files they need (in cooperation with the application execution manager), the files that are opened and where they are opened, the pattern of opened files (i.e. file A is always opened after file B is opened) etc. Regarding where and how these new replicas will be distributed, we will take into account access patterns learnt by previous usage as well as the characteristics of the available resources "near" the node or nodes using the replica. Nevertheless, these replicas will always follow the placement restriction supported by the system.

In addition, replicas that are not useful anymore, or replicas of files that use too much space will be removed. Also, if one of the replication policies (see next section) changes, replicas that do not fulfil these requirements will also be removed.

Initial prototype will be developed in Python. Later versions will be programmed in Python, although some parts may need a reimplementation in C/C++ for performance reasons.

5.1 Replication Policies

Users have a need that the system enforces policies on file location and replication to satisfy security needs and to comply with local regulations. Administrators must be able to implement these company policies, user needs, or laws into a replication policy. Thus, these policies must be fine-grained enough to cover a wide range of use-cases.

These policies could work on different levels like countries, real-world organisations, VOs or even racks in a data centre. Examples include regulations not to store EU customer information in countries outside the EU. Scientific users may want to make sure that novel results are not replicated to servers belonging to competitors.

An illustrative example could look like this:

```
<ReplicationPolicy>
    <rule>
        <entity>/volume_A</entity>
        <level>org</level>
        <filter>in (ZIB, CNR, BSC)</filter>
    </rule>
    <rule>
        <entity>/volume_A/customerData</entity>
        <level>cn</level>
        <filter>not Mordor</filter>
    </rule>
</ReplicationPolicy>
```

This policy would enforce that for everything on `volume_A` only OSDs are used that belong to ZIB, CNR or BSC. Anything in the directory `customerData` must not be replicated to the country of Mordor.

5.2 Architecture

5.2.1 Gathering Information

This service will heavily depend on the other services within the storage system as well as on services in the whole system in order to get the information it needs.

Every time a file is opened, either the MRC or the OSD that is finally selected (this decision still needs to be evaluated), has to contact the RMS to see if a better replica should be created for this access. This decision will depend on the size of the file (how long it would take to make the copy?), on the available OSDs (is there any better OSD to place this new replica?), etc. If it is the MRC that contacts the RMS, then we can delay the client till the replica has been created and then include the new replica in the list of available replicas. If it is the OSD, then we allow the client to start accessing

the file with the "bad" replica and then switch to the new replica, but this would require a mechanism to notify clients of new replicas (asynchronous notification).

Regarding the MRC, we also expect this service to keep track of opens so that we can predict future access from the previous ones. Things like "after file A is opened, file B will be opened by the same client" could be used to start replication of file B (if needed) after the open of file A. This kind of prediction can be easily done in a similar way to what was done long ago for Unix systems [10]. How this idea scales to the Grid is a research issue of the project.

Another service that could start a replication (and would allow much more efficient replica creation) is the execution management service. At some points, this service may know that a given application will be executed on a given node shortly and that this application uses a given set of files. With this information the RMS can decide whether the available replicas are good enough for the application or whether new replicas are needed. In the latter case, the creation of replicas is started even before the application starts running and thus we have it finished before the first access.

In addition, in the moment a replica has to be created, this service needs to decide on the best OSDs to place the replica and the best striping policy (if any) to be applied. To take this decision the RMS needs to have information on the effective performance of OSDs, their load, their predicted load and information on the access patterns usually occurring for that file. Such information will be gathered by OSDs (and potentially clients) and will be used by the RMS to take this kind of decisions.

5.2.2 Creating new Replicas

With all the information gathered from other services, the RMS will create a file replica. Actually the creation of a file replica will imply the following steps.

First, the RMS selects the OSDs that will hold objects of the new file replica. Second, it will decide which OSDs, already holding a replica or part of it, will be origin of the new replica. Once all involved OSDs are chosen, it will contact the source OSDs to ask them to start replicating the objects to the new OSDs. Once all objects are copied, the MRC has to be informed about the new replica and its striping policy (in case there is any).

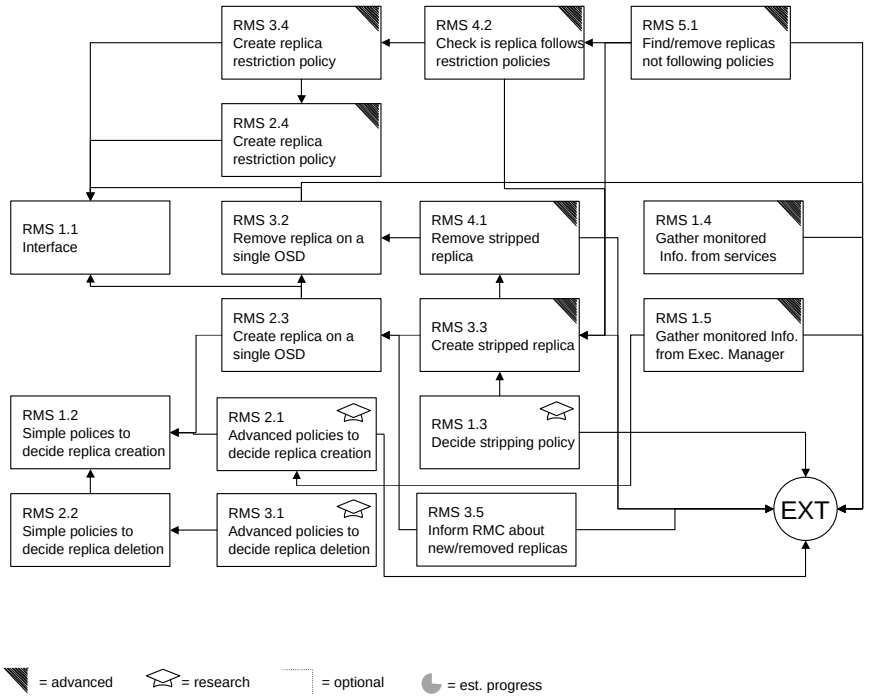
5.2.3 Removing "obsolete" Replicas

Regarding the cleaning of "obsolete" replicas, the RMS will take care of deciding when a replica has to be removed. This decision will take many parameters into account such as: free space (or lack of it) in a given OSD, a file (or the replica) is very seldom used, some close replicas were created at a given point of time when a lot of bandwidth was needed but this high bandwidth is not needed anymore, the space occupied by a file becomes too large etc.

In this case, the steps that the RMS will take are the following ones. First it will decide which replicas to remove and then it will inform the MRC that a given replica is not available anymore. The OSDs will be contacted by the RMS to remove these replicas from their storage.

It is important to notice that replicas can be removed at any time, even while they are being used (if this makes sense will be decided later when designing the policies) because when the client tries to access a deleted replica, the OSD will inform the client that the replica does not exist anymore. The client will either use other OSDs in the list it received when opening the file or contact the MRC once again to get a new list of replicas it can access.

5.2.4 Feature List



ID: RMS 1.1	
name	Interface
type	basic
dependencies	-
description	Interface needed to communicate with MRC and others services in order to get all the information it needs to take its decisions

ID: RMS 1.2	
name	Simple polices to decide replica creation
type	basic
dependencies	-
description	This feature will implement very basic replication polices that always replicate into close OSDs unless there is already a nearby copy.

ID: RMS 2.1	
name	Advanced policies to decide replica creation
type	research
dependencies	RMS 1.2, RMS 1.5, MRC 9.1, MRC 9.2, OSD 1.4, OSD 2.6
description	This feature will extend the simple polices in RMS 1.2 to include information such as file size, what is the access pattern that will be used, what is the real bandwidth needed by the application, hints from the execution management, prediction of files that will be accessed in the future by the application, etc.

ID: RMS 2.2	
name	Simple policies to decide replica deletion
type	basic
dependencies	RMS 1.2
description	These basic features will only take into account full OSDs.

ID: RMS 3.1	
name	Advanced policies to decide replica deletion
type	research
dependencies	RMS 2.2
description	This feature will extend the simple removal policies from RMS 2.2 with information such as the number of replicas for a given file, the actual amount of storage used, the utilisation of the replicas, etc.

ID: RMS 2.3	
name	Create replica on a single OSD
type	basic
dependencies	RMS 1.1, RMS 1.2, RMS 3.2, MRC 2.4
description	All the communication needed to ask OSDs to create a replica into another OSD

ID: RMS 3.2	
name	Remove replica from a single OSD
type	basic
dependencies	RMS 1.1, MRC 2.3
description	All the communication needed to ask OSDs to remove a replica

ID: RMS 1.3	
name	Decide striping policy
type	research
dependencies	OSD 1.4, OSD 2.6
description	Decide the best striping policy taking into account the access pattern of the file and the available OSD characteristics

ID: RMS 3.3	
name	Create striped replica
type	advanced
dependencies	RMS 1.3, MRC 2.4
description	All the communication needed to ask one (or several) OSD to create a replica into a set of OSDs

ID: RMS 4.1	
name	Remove stripped replica
type	advanced
dependencies	RMS 3.2, OSD 2.3
description	All the communication needed to remove a replica from several OSDs

ID: RMS 2.4	
name	Create replication policy
type	advanced
dependencies	RMS 1.1
description	Inclusion of new replication policies in the system

ID: RMS 3.4	
name	Remove replication policy
type	advanced
dependencies	RMS 1.1, RMS 2.4
description	Removal of replication policies from the system

ID: RMS 4.2	
name	Check if replica follows restriction policies
type	advanced
dependencies	RMS 2.3, RMS 3.3, RMS 3.4
description	Once a new replica restriction policy is inserted, we need to check whether some replicas fail to follow it, in which case these replicas will need to be removed

ID: RMS 5.1	
name	Find/remove replicas not following restriction policies
type	advanced
dependencies	RMS 2.3, RMS 3.3, RMS 4.2, MRC 1.1, MRC 1.2
description	This feature will be in contact with MRC to check whether all available replicas fulfil the new replication policies added.

ID: RMS 1.4	
name	Gather monitored information from other services
type	advanced
dependencies	MRC 9.1, MRC 9.2, OSD 1.4, OSD 2.6
description	This feature will be in charge of contacting the cooperating services and get their monitoring information to build the needed information to take decisions.

ID: RMS 1.5	
name	Receive hints from execution manager
type	advanced
dependencies	-
description	This feature will receive information from the execution manager about new jobs, where they will be executed, and the files they may use. With this information some replicas may be created.

ID: RMS 3.5	
name	Inform MRC about new/removed replicas
type	basic
dependencies	RMS 2.3, RMS 4.1, MRC 1.1, MRC 1.2
description	Every time a replica is created/removed, we need to inform the MRC about this change.

5.3 Open Issues

- Who should enforce replication policies such as "files on volume X cannot be replicated to OSDs outside the EU". The RMS can enforce rules when creating the replica, but the MRC has the complete list of replicas and when rules change, only the MRC knows which ones do not comply with the new rules.

5.4 Interface

The first interface we will need to work on is the one needed to specify restrictions on the location of replicas. A potential example was presented in section 5.1, but the real interface will be defined during development.

In order to be able to offer users a specific level of fault tolerance, we will allow users to set the number of replicas that a file must have to avoid failure problems. This will be specified extending the same interface offered to change general characteristics of files.

Finally, we will need an interface for the application execution manager to inform the RMS that a given job will be executed at a given location and the files it will use. Once again this interface needs to be studied (in cooperation with WP3.3).

5.5 Research Prospects

- **Information management**
  This service depends a lot on having huge amounts of information from all potential OSDs and files. Gathering all this information and being able to use it is a research challenge. To approach this problem, we plan on implementing clustering and pseudo-general policies to take the decisions.

  A second research challenge regarding the information management is to decide what information is really usefull and what is not necessary and thus should not be taken into account.

- **Replication policies: where and how**
  Deciding where is the best location of a replica and if this replica should be striped or not is another research challenge. To solve this problem we plan on using autonomic approaches by modelling the possibilities and chosing the best possible (or near optimal) solution given the gathered information from the system and files.

- **Replication policies: when**
  Besides the place to create a replica, another vital issue is the time to create it. To approach this item, we will use traditional mechanisms such as cooperating with the aplication execution environment (but in a more tightly way than in the past). In addition, we will try to predict which files will be used by running jobs before the file is really used

and then replicate it in advance. This prediction will be based on the job and a history of open files (not necessarily within a job).

- **Replica deletion**
  If replicas are not removed, they will tend to fill the entire available storage. Deciding which replicas and when they need to be removed is also another research topic (that has not been addressed enough in the past). We will work on polices that take into account space availability, space used by individual files, file and replica usage, etc.

6 Metadata and Replica Catalogue (MRC)

While OSDs keep pure file data (objects) without metadata and handle all direct data access to them, the MRC is responsible for storage, retrieval and querying of all file metadata and locations of replicas. Therefore, this service implements the functionality described in Task T3.4.2 (Replica Management Service, ZIB) and Task T3.4.4 (Metadata Lookup Service, ZIB).

As 'metadata', we consider all non-file-content data in the file system, i.e. data necessary to describe the file system itself. Amongst others, this includes locations at which the actual file content data is stored, as well as file attributes, such as file size or access rights. The MRC stores this metadata. To manipulate and query metadata, it provides dedicated functionality in the form of a service interface which is accessed by the client side in connection with file system operations. The main task of the MRC is the arrangement of distributed file system metadata with respect to a fast and efficient file access and retrieval.

The MRC will act logically as one service in a non-partitioned network, but will be composed of partitioned and replicated service instances on many hosts in order to improve availability and performance.

An initial prototype was developed in Python. Java will be used for the final version. Some parts might be implemented in C or C++ if necessary for increased performance. The current prototype uses Java and XMLRPC [21].

6.1 Security

6.1.1 Access Control Mechanisms

Objects stored in XtreemFS will be accessed by different users and user groups. We assume that each user and user group respectively has a unique identifier. In order to keep unauthorised users from reading or manipulating data, some kind of user rights management is necessary.

Access control is required to support data privacy, sharing, and collaboration. As defined in POSIX, file systems usually support access control on a per-file and per-directory basis.

The MRC should support a variety of different access control policies based on different entities of the file system. This enables us to develop the MRC independent from any decision on a concrete policy. Moreover, different policies can be used on different volumes as defined by system administrators.

In general a policy must satisfy the following requirements to be usable by a MRC:

1. It must be able to grant or deny access to a file system volume, directory, file or metadata based on the current grid user ID and VO
2. It must be translatable into POSIX permissions (owner, group, other `rwx`) based on the current context (grid user ID and VO)
3. In addition, policies can support translation from some or all POSIX permission changes into their mechanism to support `setattr` functions (e.g. `chmod` command).

Those policies can be implemented using a plug-in mechanism providing some basic interface:

```
boolean checkPermission(objID, userID, VO, accessMode)
int     convertToPosixACL(objID, userID, VO)
boolean setByPosixACL(objID, userID, VO, posixACL)
```

Volume ACLs We consider it more beneficial to have access rights on a per-volume basis, while supporting the easy creation of volumes as the unit for collaboration. We allow users to have private volumes for their private data, public volumes for archive data with extended access rights for archive administrators, and collaboration volumes, which may link to private data and give a group of users access to common files.

In order to enable an user to specify who is allowed to access objects in the file system, each volume is associated with an access control list (ACL). Since there is no hierarchy of volumes, an ACL-based access control model at volume level can minimise complexity and overhead while ensuring a reasonable granularity of the file access control model.

An ACL comprises both access rights for the metadata as well as for the data object itself. Distinguishing between these two kinds of rights gives a user the possibility to allow another user to modify meta attributes of an object while keeping that user from reading or making changes to the actual data.

6.1.2 Encryption

Encryption of file contents can only be part of the client libraries. Support for encryption of metadata is not planned.

6.2 Architecture

6.2.1 Data Model

The MRC implements a data model for organising file data by both hierarchical directory structures and extended metadata. The usage of directories or extended metadata is optional, the user can chose either or both to support retrieving data.

The core abstraction for controlling access to file metadata and file data is the volume. Volumes are associated with a set of users or user groups along with a definition of the kind of operations that they are allowed to perform. Volumes can be mounted as subtrees of the directory hierarchy of any host in the system.

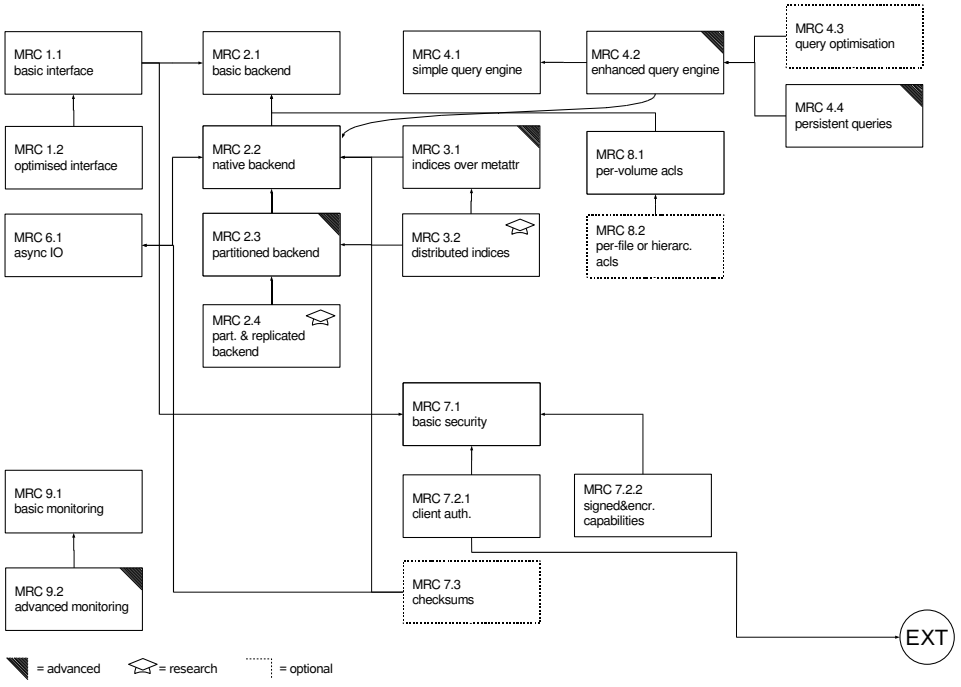
Files can be copied between volumes, and links to files in other volumes can be created.

6.2.2 Internal Architecture

Both system and user-defined attributes of a physical object in XtreemFS are encapsulated in a meta object. For each physical object on a storage device, exactly one meta object is maintained by the metadata service.

Meta objects belong to one volume. In addition, links to a meta object can be created on other volumes and/or other directories. How this will be implemented (soft links, internal soft links, hard links, aliases) is not clear at this time.

6.3 Feature List



ID: MRC 1.1	
name	basic MRC interface
type	basic
dependencies	MRC 7.1
description	A basic API is implemented in XMLRPC over HTTP. This is extremely useful during the software development process as it is easy to debug. This implementation relies on an existing J2EE container like Tomcat.

ID: MRC 1.2	
name	optimised MRC interface
type	basic
dependencies	MRC 1.1, MRC 6.1
description	Since XML-based protocols, especially untyped ones like XMLRPC, are not very performant w.r.t. parsing and message size, a different protocol might be used for production use. This could be something like Sun RPC or a native protocol. This feature will be developed together with MRC 6.1 since both have to be tightly integrated in the MRC to ensure maximum performance.

ID: MRC 2.1	
name	basic MRC backend
type	basic
dependencies	-
description	Storage of MRC metadata and directory hierarchy in a MySQL database, no additional indices over metadata are possible. Used for rapid prototyping, however not suitable for testing advanced techniques like replication, partitioning or indices.

ID: MRC 2.2	
name	native MRC backend
type	basic
dependencies	MRC 2.1, MRC 6.1
description	Implementation of a native backend for data storage is absolutely necessary since the data model used by the MRC is not optimal for relational databases. The implementation of indices requires additional data structures. This task is also intended to provide a base for advanced techniques like data replication.

ID: MRC 2.3	
name	partitioned MRC backend
type	advanced
dependencies	MRC 2.2
description	Partitioning of data (into slices) and distribution among several MRCs. The model used for partitioning must be able to provide means to avoid hotspots especially in directories.

ID: MRC 2.4	
name	partitioned and replicated MRC backend
type	research
dependencies	MRC 2.3
description	Replication of slices to increase performance and resilience. Research into applicable consistency models and the evaluation of available algorithms. Performance comparisons.

ID: MRC 3.1	
name	indices over metadata
type	advanced
dependencies	MRC 2.2
description	Creation and maintenance of indices over any (user defined) metadata attribute to increase performance of search operations.

ID: MRC 3.2	
name	index distribution
type	research
dependencies	MRC 3.1
description	Research into optimal distribution of indices, realisation of global indices. Consistency models for distributed and/or replicated indices. Replication and partitioning will add to parallelisation of search requests.

ID: MRC 4.1	
name	simple query engine
type	basic
dependencies	MRC 2.1
description	Search over metadata attributes w/o indices or query optimisation. For the basic SQL backend this is just a mapping to corresponding SQL statements. With the native backend this will be replaced by a native query engine.

ID: MRC 4.2	
name	enhanced query engine
type	advanced
dependencies	MRC 2.3, MRC 4.1
description	Search over metadata attributes w/ usage of (global) indices and partitioned MRC. Could also include parallelisation on replicated indices.

ID: MRC 4.3	
name	optimised query engine
type	optional (research)
dependencies	MRC 2.3, MRC 4.2
description	Query optimisation, query caching... (open to future ideas)

ID: MRC 4.4	
name	persistent queries
type	advanced
dependencies	MRC 4.2, WP 3.2 pub/sub
description	Change notification via pub/sub on persistent queries implemented in the MRC.

ID: MRC 6.1	
name	embedded asynchronous socket IO
type	basic
dependencies	MRC 2.2
description	Evaluation of server models, especially alternatives to server/worker model. Identified as basic because it is a fundamental part of the software. However, research into alternatives to server/worker model is ongoing and has to be evaluated. (So, this item is basic in terms of software development but requires research – at least evaluation of proposed models).

ID: MRC 7.1	
name	basic security
type	basic
dependencies	-
description	In a first implementation clients are not authenticated and capabilities are sent as plain text. This module should be exchangeable with 7.2.x any time of the development process to ease debugging!

ID: MRC 7.2.1	
name	client authentication
type	basic
dependencies	MRC 7.1, WP 3.5
description	Authentication of client hosts through mechanisms provided by WP 3.5. Subject to interfaces provided by WP 3.5.

ID: MRC 7.2.2	
name	signed and/or encrypted capabilities
type	basic
dependencies	MRC 7.1
description	Implement a protocol to sign and encrypt capabilities. Should closely resemble existing and **well studied** security protocols, e.g. kerberos.

ID: MRC 7.3	
name	checksums/advanced integrity checks
type	optional
dependencies	-
description	Additional checksums to increase integrity of on-disk data and data transmissions. Experiments show that checksums provided by TCP and/or UDP are not sufficient to protect transmission of file contents (see [5]).

ID: MRC 8.1	
name	volume ACLs
type	basic
dependencies	-
description	A plug-in architecture for Access Control is implemented in the MRC (see 6.1.1). A basic module defines user/group-based Access Control Lists per volume.

ID: MRC 8.2	
name	hierarchical ACLs
type	optional
dependencies	MRC 8.1
description	Finally, other mechanisms can be developed. This will be the result of discussions/requirements from WP 3.5.

ID: MRC 9.1	
name	basic MRC monitoring
type	basic
dependencies	-
description	Monitoring of the MRC. Will be specified during the software development process as needed. Basic information for hotspot detection, load, disk and memory usage will be provided.

ID: MRC 9.2	
name	advanced MRC monitoring
type	advanced
dependencies	MRC 9.1
description	More elaborate/uncommon functions go into this feature.

ID: MRC 10.1	
name	MRC management console
type	optional
dependencies	MRC 1.1, MRC 9.1 (and many more)
description	GUI-based management and monitoring tool. Could be completed by visual access policy and replication policy editors. Might also include a web-based interface for users to allow file access regardless of location, system or installed client libraries.

6.4 Testing

On module level we try to employ white-box testing to allow for regression tests. These tests are done on a regular basis and have to be documented. Similar to agile methods, testing should be part of the development process from day one. For some modules unit testing is not feasible, e.g. the interface modules. These should be tested as part of integration test of the MRC for each build.

6.5 Open Issues

1. How does the MRC get information about the actual filesize after a write operation has completed?
   Possible solution: The client notifies the MRC when the file is closed.
   Alternative solution: The MRC includes a notification request in the capability sent to the OSD, who in turn informs the MRC upon completion of an operation (cf. Section 4.6).

2. How are `fsync` and `fflush` handled regarding the filesize attribute in the MRC?

3. Is a fast update of `atime` necessary, are there use cases for applications relying on correct `atimes`?

4. How to enable the revocation of capabilities, e.g. when permissions change? Possible approaches include: expiration of a capability, versioning of files/volumes and capabilities or lists of issued capabilities in the MRC.

5. How to handle a/c/mtime with clients and servers in different timezones?

6.6 Interface

The final version of the MRC interface is still in development. This section outlines the basic functionality provided by the MRC which covers

- traditional FS operations like `mkdir`, `readdir`, `rename`
- operations for meta-attribute handling like `getAttributes`
- query operations to explore the filesystem based on the meta-attributes rather than through the hierarchy (although combinations are possible, too)
- operations for clients to acquire/renew capabilities necessary for authentication with OSDs
- operations for servers to find out about replica locations (e.g. for job schedulers)

The MRC Services are available through standard XMLRPC via HTTP. Interoperability between different implementations (and different programming languages) was tested.

6.7 Research Prospects

- **Implementation of a replicated database (MRC backend)**
  Resilience will be an important aspect of XtreemFS. We plan to tackle this problem by means of data replication. With respect to this, it will be important to evaluate different approaches to provide for consistency of replicas.

- **Partitioning of tables (in the MRC backend)**
  Partitioning is a means to reduce the load on individual MRC servers by splitting a table into several parts which are then distributed among MRC servers.

- **Federation of MRCs**
  The XtreemFS offers a global view on all available storage resources. Since the system is built of several distributed MRC installations a federation of these is needed. However, autonomous operation of a site's part of the file system is required allowing only a loose coupling of the individual sites. Finding out to what extent it is possible to decouple system components while preserving a global view to the system is a research issue.

- **Global queries and global indices**
  Users may want to search for files in a number of different volumes on different sites or even on the entire system. This requires global indices to speed up global search operations.

- **Query engine optimisations**
  A query engine will be needed to evaluate queries. Optimisations like caching and pre-fetching might be reasonable to speed up query processing.

7 Object Sharing Service (OSS)

This OSS service implements the functionality described by task T 3.4.5 Grid Object Management. Furthermore, grid pipes will be implemented.

The object sharing service (OSS) runs on each client machine allowing sharing of objects residing in volatile memory. An object in this context is a replicated volatile memory region, dynamically allocated by an application or (a part of) a memory-mapped file. Larger memory regions store groups of objects. Grid pipes are offered, too; built on a shared memory page that may be accessed by several writers and readers. Sharing of objects is possible within a VO, only.

Objects may contain scalars, references, and code. OSS handles the concurrent read and write access to objects and is responsible for maintaining consistency of replicated objects. Persistence and security for objects (i.e. files) is provided by XtreemFS.

7.1 Architecture

The OSS service manages volatile shared memory for dynamically allocated objects, memory-mapped files, and grid pipes. All components will be designed to be scalable and fault tolerant to deal with the dynamic behaviour of the grid.

7.1.1 Terminology

False Sharing This is a well-known performance penalty causing page thrashing when using page-based consistency units. If two objects reside on a memory page and each object is accessed by a different node, at least one node writing, the page is exchanged all the time between the involved nodes. It is important to note that this is a time dependent phenomenon and during a time interval none of the two nodes accesses both objects. Obviously, the problem could be solved by moving one object two another memory page or reducing the size of the consistency unit to object or variable level. Of course false sharing can also occur if more objects are located on a 4 KB page and more nodes are accessing such a page.

True Sharing Occurs if two or more objects are stored on a memory page that are all accessed by a node during a time interval. As these objects

are anyway accessed during a time interval by a node it is a good idea to store them together on a memory page. When the first object is accessed the others will be sent to the accessing node, too. Thus only the first object access is expensive. When using a fine grained consistency unit, e.g. object level, each object must be fetched over the network. There may be situations with true and false sharing of objects on the same memory page.

But of course also true sharing is a time dependent effect and may turn into false sharing and vice versa. Important to note is that resolving false sharing means also to prevent a later true sharing. Finally, true sharing does not always imply good scalability. If a parallel algorithm is not partitioned well there may be many conflicts on true sharing pages. But in these case conflicts will occur independent of the chosen consistency unit size and a refinement of the algorithm is necessary.

7.1.2 Heap Management

As long as no references are shared relevant logical memory blocks may be located at different addresses on different nodes. This is sufficient for memory-mapped files, grid pipes, and shared memory blocks storing scalars only, e.g. matrices storing floats.

But when sharing full-fledged object structures including references (e.g. scene graphs), objects need to be stored at the same address on each participating node. Furthermore, when objects are allocated dynamically a scalable distributed heap management is mandatory. Small objects will be allocated using a two-level memory allocation scheme. On the first level a larger chunk of memory is allocated in an hierarchical fashion. Subsequent allocations for small objects are served by memory regions within this larger chunk. Further memory allocation strategies will be studied.

For 32-Bit machines it may be necessary to support fragmented shared address spaces for large object structures.

7.1.3 Object access detection

Accesses to objects need to be detected to request objects if not present locally and to implement consistency protocols. To be generic and transparent a page-based approach seems to be the best choice as a basic memory access detection scheme. The first prototype will be implemented at the user level

but some functions may move into the kernel depending on the synergies with WP2.2.

When several objects (instances of object-oriented languages typically consume 32-64 byte) are stored on a 4 KB memory page the access detection may be to coarse. We plan to allow allocating one object per single logical memory page what is feasible for 64-Bit address spaces. To avoid wasting of physical memory several objects may be stored together on a single physical memory page. Thus we can detect memory access on an object-based level. The granularity of access detection may be adapted during runtime. OSS can enable access to all/some/one logical object(s) residing on an affected physical page.

7.1.4 Object access monitoring

OSS will also implement object access monitoring to control false sharing and object replicas. The latter are used to speed up object access and to provide a base for fault tolerance. Writes to object data items are propagated to replicas as invalidate or update messages.

Depending on the monitored object access pattern history and/or programmer hints the OSS may control where and how many replicas of an object exist. Typically, not all replicas will be updated each time but those are preferred that are stored on clients that have accessed them frequently in the past. Furthermore, access information may also be used for prefetching of shared objects to hide network latency.

Another important task of the access monitoring facility is to provide the necessary information to allow controlling the consistency unit size. This is necessary to cope with the problem of false sharing causing expensive page thrashing. We plan to use the monitored access information to detect false sharing. The situation will be dynamically resolved by changing the consistency unit size from page-level to object-level during runtime for the affected objects.

The advanced Java support runtime (e.g. used by Wissenheim in WP4.2) will allow to concurrently relocate objects during runtime. Because false sharing is a time dependent penalty OSS will later switch back to a coarser consistency granularity to allow true sharing. The latter speeds up memory access and hides network latency.

7.1.5 Object consistency management

A lot of consistency protocols have been proposed for replica management in shared memory environments [14]. OSS will be configurable allowing to plug-in new consistency protocols as needed. Furthermore, OSS will support multi consistency for shared objects, so an object replica may be seen by different clients under different consistency constraints.

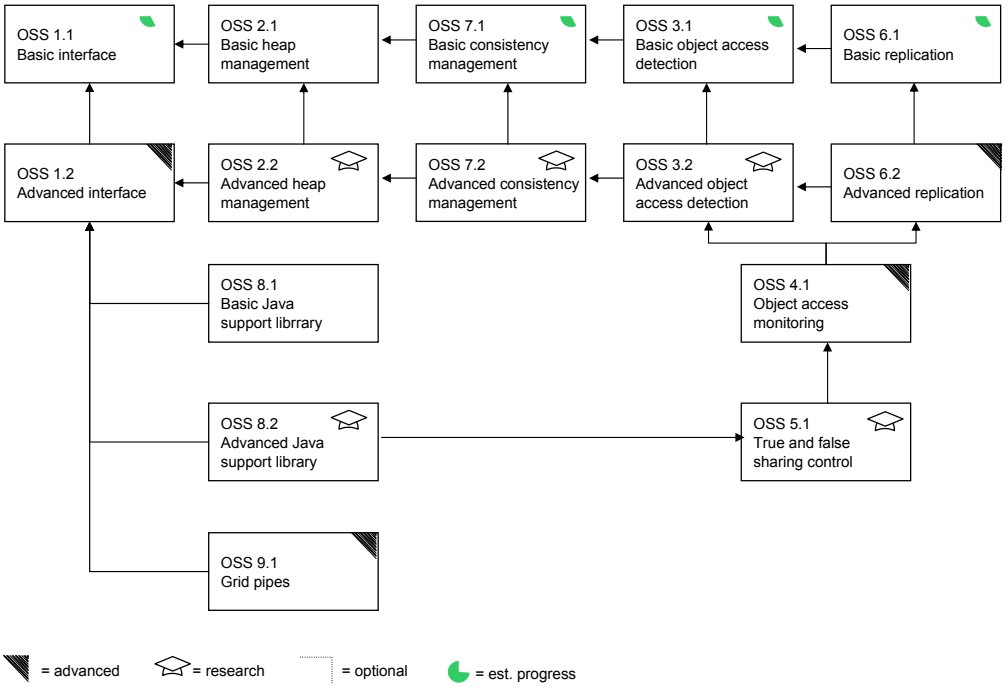
We plan to provide three reference implementations: strict, weak, and transactional consistency. Strict consistency is recommended for single shared objects only and for testing purposes. Weak consistency is described in the literature, e.g. [11].

Transactional consistency bundles object access in transactions thus reducing the number of messages. Basic transaction support will rely on locks. Transactions doing speculative object accesses will provide more efficiency by hiding network latency. These transactions need to be restartable in case of a conflict. The programmer needs to annotate where transactions (TAs) start by BOT and where they end by EOT. TAs may also be aborted voluntarily to avoid network communication if a condition is not true.

The restart property necessary for speculative TAs requires that OSS is able to reset TAs to their initial state before BOT. Shadow copies will be used for resetting memory. Device and file access may be considered as an advanced feature. We expect synergies with the checkpointing and restart mechanisms developed within and WP2.1 & WP2.2. Another advanced feature is to allow conflicting transactions to commit under the control of the programmer. This will increase the transaction throughput.

The main part of OSS will be developed as a shared library in C. Later versions will include a Java library to allow Java applications (e.g. Wissenheim, see WP4.2) to take advantage of the OSS service. Advanced OSS features implemented in this Java library will be considered, too.

7.2 Feature List



ID: OSS 1.1	
name	Basic interface
type	basic
dependencies	-
description	Basic interface offered to client applications and to communicate with the XtreemFS client.

ID: OSS 1.2	
name	Interface
type	advanced
dependencies	-
description	Final interface offered to client applications and to communicate with the XtreemFS client.

ID: OSS 2.1	
name	Basic heap management
type	basic
dependencies	-
description	This feature will implement very basic heap management allowing to allocate memory blocks at same addresses.

ID: OSS 2.2	
name	Advanced heap management
type	research
dependencies	-
description	This feature will implement a scalable heap management (for 64-Bit address spaces) allowing dynamic allocation of small and large objects.

ID: OSS 3.1	
name	Basic object access detection
type	basic
dependencies	-
description	This feature will implement a basic page-based access detection scheme.

ID: OSS 3.2	
name	Advanced object access detection
type	research
dependencies	OSS 2.2
description	This feature will allow to control the access detection on an per object granularity that may be adapted during runtime.

ID: OSS 4.1	
name	Object access monitoring
type	advanced
dependencies	OSS 3.1, OSS 3.2
description	This feature will monitor and record access patterns to be used as an input for replica management and true and false sharing control.

ID: OSS 5.1	
name	True and false sharing control
type	research
dependencies	OSS 4.1
description	This feature will be used to detect and resolve false sharing.

ID: OSS 6.1	
name	Basic replication
type	basic
dependencies	-
description	This feature will implement a basic replication scheme for shared objects.

ID: OSS 6.2	
name	Advanced replication
type	advanced
dependencies	-
description	This feature will implement an advanced replication scheme that will integrate fault tolerance strategies.

ID: OSS 7.1	
name	Basic consistency management
type	basic
dependencies	-
description	This feature will implement a basic consistency management including strict consistency and transactions with locks.

ID: OSS 7.2	
name	Advanced consistency management
type	research
dependencies	-
description	This feature will implement an advanced consistency management including speculative transaction support and multi consistency.

ID: OSS 8.1	
name	Basic Java support library
type	basic
dependencies	-
description	This feature will implement a basic Java native library to support shared memory Java applications (e.g. Wissenheim in WP4.2).

ID: OSS 8.2	
name	Advanced Java support library
type	research
dependencies	-
description	This feature will implement an advanced Java native library including garbage collection and a concurrent object relocation facility. The latter will allow a more flexible true and false sharing control.

ID: OSS 9.1	
name	Grid pipes
type	advanced
dependencies	-
description	This feature will implement grid pipes that emulate traditional named pipes in UNIX.

7.3 Interface

OSS will expose its basic functionality through the XtreemFS interface. This will include a function like `mmap()` to support memory-mapped files. Through the access layer OSS will be able to load persistent data (OSD), to memory-map files, and to do an authorisation for file/object access (MRC).

All advanced OSS functions, e.g. sharing of dynamically allocated objects, transactions, and grid pipes will be exposed by the OSSInterface.

7.4 Research Prospects

We expect fruitful research prospects from the consistency topic with respect to scalability and fault tolerance, especially when relying on speculative transactions. We are also interested in strategies dynamically controlling

true and false sharing based on monitored access patterns. Finally, a 64-Bit heap management for a larger scale is another challenging research topic.

8 Access Layer

8.1 Introduction

The access layer serves as interface between user processes and the file system infrastructure. It manages the access to files and directories in the XtreemFS for user processes as well as the access to grid specific filesystem features for users and administrators.

All accesses of user processes to file system entities have to go through this layer. As such the access layer implements the client-side part of the distributed file system. It knows how to do all file operations by communicating with the XtreemFS services like MRC, OSDs, and uses XtreemOS and XtreemFS level mechanisms to enforce grid policies for file access.

The implementation of the access layer will follow the user requirements defined in workpackage WP4.2 and the use cases described in section 2.5. The functionality of the access layer will be mapped as closely as possible and reasonable to the POSIX standard (IEEE 1003.1, 2004) and in addition implemented as an independent library that can be used by applications. The former case has the inherent advantage of being able to serve legacy applications.

In the following we describe the approach chosen for the access layer. The feature list classifies the expected features, it is followed by an implementation section listing more details. The section closes with research ideas related to the access layer.

8.2 Approach

As stated in the section above, the access layer can be layed out in different fashions. A schematic view is given in figure 10. As a first step we choose the POSIX interface approach and implement a prototype using a user space file system called FUSE[1]. FUSE is a collection of a library and a kernel module that allows the implementation of a file system in user space.

The access layer itself is not a single piece of software but is distributed over several components. The MRC and OSD have some aspects of the access control implemented (cf. section 6).

[1]Filesystem in Userspace: http://fuse.sf.net

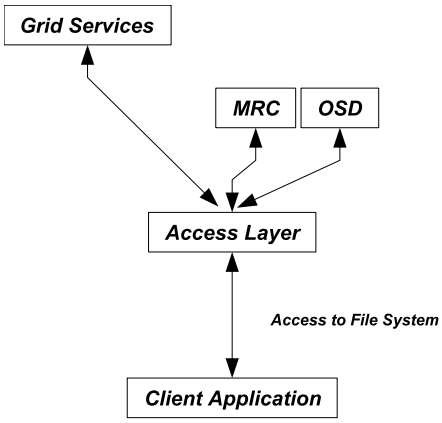


Figure 10: Schematic overview of access layer

8.2.1 Direct Access to XtreemFS Components

The simplest way to provide access to files and directories living in the XtreemFS is through the client interfaces delivered by the components MRC, OSD, RMS and OSS. As described in section 3.6 the client will need to interact with the MRC as well as the RMS for opening or creating a file, then transfer file data directly to/from the OSDs. These calls and data transfers will be encapsulated into a user library linkable by applications. The library will provide access to files through an appropriate API as well as access to lower level functionality of the XtreemFS services in order to gain finer control over the filesystem behavior.

The direct access to XtreemFS functionality through the library API allows full control to any capabilities provided by the filesystem services. No restrictions to the API exist, therefore this can be chosen to fit the XtreemFS design optimally. The drawback is that applications will need to be adapted to this particular API and be less portable to other filesystem access methods.

8.2.2 POSIX with FUSE

In order to easy portability and support for legacy applications, we will provide a POSIX compliant access layer in addition to the direct library based API. As a first approach to POSIX compliance we will implement a FUSE-based filesystem. FUSE provides a simple library API for implementing most of the filesystem functionality in user space. The user space part interacts with the FUSE Linux kernel module, which is integrated into the mainstream

Linux kernels. The kernel module integrates the filesystem functionality into the Linux VFS (virtual filesystem) layer, making it look similar to any native Linux filesystem. With the filesystem functionality being in userspace, it can be coded in any programming language supported by the FUSE API and is not limited to C, as a normal kernel module filesystem client would be.

Local to Global Identity Mapping

Grid filesystem entities have a global scope and are therefore assigned to grid identities like global user IDs or VO IDs. When accessed by processes which carry temporary local identities (local user and group IDs), the access control mechanisms must convert local to global identities and enforce the file access policies defined also in global context.

The file system components MRC and OSD must be able to provide user ID and VO information for each filesystem object (file, directory, object). With this information, the access layer can translate the grid access control mechanisms to local ones. The way in which data is stored in XtreemFS is completely transparent to the client application.

Sharing files between members of a VO is one of the requirements for the file system. The local IDs should therefore reflect the relationship between users. The same VO should get the same local group ID.

The conversion between local and global identities is expected to be done by a **mapper** grid service. This service should run on every node and will probably be implemented within workpackage WP2.1.

Client Application View

An application that runs on a node and uses XtreemFS with a POSIX-like interface cannot distinguish between local and grid files. It must therefore be possible for an application to `fstat` a file in the grid file system and retrieve information about the local user id, group id and permissions. This information should be meaningful in the sense that, for instance, the user id corresponds to a grid ID and the local group id corresponds to the user's VO. The application can then decide what it is allowed to do with that file and whether its contents is accessible, even if the user does not own it. The access restrictions are enforced by the local file system layer.

8.2.3 Advanced POSIX Interface

In a more advanced phase of the project we will investigate the suitability of the FUSE approach, its performance and limitations. We might consider the implementation of the client filesystem completely in kernel space as a module.

8.3 Feature List

This feature list describes the necessary and desirable items that the access layer should incorporate. It lies in the nature of an access layer that it accesses features that are implemented elsewhere. As such the access layer is dependent on the features of the MRCs, the OSDs and the OSSs that are visible to the outside world.

8.3.1 Basic POSIX I/O

Features and capabilities for file I/O are as defined by POSIX. For most features only a brief description is given. More detailed information can be found in the POSIX standard. A prerequisite for using the XtreemFS in a POSIX like fashion is of course the ability to mount and unmount the XtreemFS in the local file system hierarchy. If the OSS can be accessed through the file system, some of the POSIX I/O functions can be used.

ID: AL 1.1	
name	`create`
type	basic
dependencies	-
description	Create a file with certain permissions
ID: AL 1.2	
name	`stat`
type	basic
dependencies	-
description	Get information on a file, like user id or file protections.
ID: AL 1.3	
name	`open`
type	basic
dependencies	-
description	Open a file in specified access modes.

ID: AL 1.4	
name	`write`
type	basic
dependencies	-
description	Write memory content to a file.

ID: AL 1.5	
name	`read`
type	basic
dependencies	-
description	Read file content to a memory location.

ID: AL 1.6	
name	`lseek`
type	basic
dependencies	-
description	Set file position pointer of a file.

ID: AL 1.7	
name	`close`
type	basic
dependencies	-
description	Close an open file.

ID: AL 1.8	
name	`mkdir`
type	basic
dependencies	-
description	Create a directory.

ID: AL 1.9	
name	`opendir`
type	basic
dependencies	-
description	Open a directory.

ID: AL 1.10	
name	`readdir`
type	basic
dependencies	-
description	Get next entry in an already open directory.

ID: AL 1.11	
name	`closedir`
type	basic
dependencies	-
description	Close directory.

ID: AL 1.12	
name	`rmdir`
type	basic
dependencies	-
description	Delete a directory.

ID: AL 1.13	
name	`chmod`
type	basic
dependencies	-
description	Change permissions of a file.

ID: AL 1.14	
name	`chown`
type	basic
dependencies	-
description	Change ownership of a file.

ID: AL 1.15	
name	`acces`
type	basic
dependencies	-
description	Check permissions of a file.

ID: AL 1.16	
name	`utime`
type	basic
dependencies	-
description	Change access and modification time of a file.

These basic I/O features need capabilities to map from the POSIX view of users to the grid view. These capabilities are provided elsewhere and the interface to that provider will be given here. In the POSIX view these features are not exported to a user's application but are needed internally in the access layer.

ID: AL 1.17	
name	getGridID
type	basic
dependencies	-
description	Requests the grid identity of a user identified by a local uid.
ID: AL 1.18	
name	getVOID
type	basic
dependencies	-
description	Get VO identitf from a local gid.
ID: AL 1.19	
name	getLocalUID
type	basic
dependencies	-
description	Get local uid from user's grid identity.
ID: AL 1.20	
name	getLocalGID
type	basic
dependencies	-
description	Get local gid from user's grid identity and VO.
ID: AL 1.21	
name	testAccess
type	basic
dependencies	-
description	Tests whether the policy governing a file or directory would allow a specific POSIX access mode, like writing to a file.

8.3.2 Advanced POSIX Interface

The features in this section will not be part of the basic version.

ID: AL 2.1	
name	`mmap`
type	advanced
dependencies	-
description	Map a contiguous part of a file into memory with given access protection.

ID: AL 2.2	
name	`mprotect`
type	advanced
dependencies	-
description	Set protection for memory mapped file area.

ID: AL 2.3	
name	`msync`
type	advanced
dependencies	-
description	Write back mapped part of a file to the file system.

ID: AL 2.4	
name	`munmap`
type	advanced
dependencies	-
description	Delete the mapping of a file.

ID: AL 2.5	
name	`lockf`
type	advanced
dependencies	-
description	Set, test or remove a lock of portions of a file.

ID: AL 2.6	
name	`aio_read`
type	advanced
dependencies	-
description	Asynchronous read from a file.

ID: AL 2.7	
name	`aio_write`
type	advanced
dependencies	-
description	Asynchronous write to a file.

ID: AL 2.8	
name	`aio_fsync`
type	advanced
dependencies	-
description	Synchronize all outstanding operations.

ID: AL 2.9	
name	`aio_suspend`
type	advanced
dependencies	-
description	Suspend execution of a process.

ID: AL 2.10	
name	`aio_cancel`
type	advanced
dependencies	-
description	Cancel an outstanding I/O operation.

8.3.3 OSS Interface

The OSS is a general service for sharing objects living in volatile memory between nodes. Virtual files in the file system could be the easiest way to integrate the OSS with XtreemFS. For such virtual files, the basic POSIX I/O features for files (cf. 8.3.1) can be used and the OSS can be accessed through the POSIX file interface.

In a future version the use of `mmap` and `munmap` for shared objects is planned. The shared objects will then be part of a processes memory.

The OSS also provides grid pipes. These can – like the distributed shared memory files – be mapped as special files in XtreemFS. The MRC must therefore know about the special files for shared memory and grid pipes as these are mapped into the directory structure maintained by the MRC. For these purposes it does not matter whether the MRC has internally a flat or hierarchical structure.

The access to pipes would also be covered by the basic file interface. Reading from and writing to pipes is done with the usual commands.

8.3.4 Management and Monitoring of XtreemFS

The access layer is not only intended to access files but also to provide general access to the entire system. This includes monitoring and management of the

file system. While simple operations like deleting a file or copying a file can be done with already existing system tools such as `rm` or `cp` in the POSIX view in a transparent way, more sophistication is needed to incorporate more aspects of the file system. For the direct approach the access layer must provide the following capabilities.

ID: AL 3.1	
name	getMetadata
type	advanced
dependencies	-
description	Get all or parts of the meta data for a file.
ID: AL 3.2	
name	addMetadata
type	advanced
dependencies	-
description	Add a meta data entry.
ID: AL 3.3	
name	clearMetadata
type	advanced
dependencies	-
description	Clear all user defined meta data.
ID: AL 3.4	
name	getOSDInfo
type	advanced
dependencies	-
description	Get information on which OSDs hold the files data.
ID: AL 3.5	
name	getStripingInfo
type	advanced
dependencies	-
description	Get striping information of a file.
ID: AL 3.6	
name	setStripingPattern
type	advanced
dependencies	-
description	Hint on a striping pattern that suits the data best.

ID: AL 3.7	
name	getNumOfReplica
type	advanced
dependencies	-
description	Get the number of replica of a file.
ID: AL 3.8	
name	getReplicaLocation
type	advanced
dependencies	-
description	Get the URI of a replica of a file.
ID: AL 3.9	
name	setNumOfReplica
type	advanced
dependencies	-
description	Hint on the number of possible replica of a file.
ID: AL 3.10	
name	getPolicy
type	advanced
dependencies	-
description	Get the policy that is governing a file.

8.4 Implementation

The different file system components MRC and OSD need intensive testing in order to provide high performance and reliability. To help the development of these components we plan to have an early prototype of the access layer based on FUSE. This prototype also helps to identify possible conflicts and problems as early as possible that were not thought of in the planning phase. Refinement of the requirements and further implementation details will also be derived on the experiences gained with the prototype.

8.4.1 FUSE Prototype

Based on the general idea that the user of the filesystem should be able to use it as if it was a local file system, the architecture must provide features to mount the filesystem in the local filesystem hierarchy. One such means is FUSE, a file system in user space that will be employed here for prototyping. FUSE allows file systems to be implemented in the user space of different

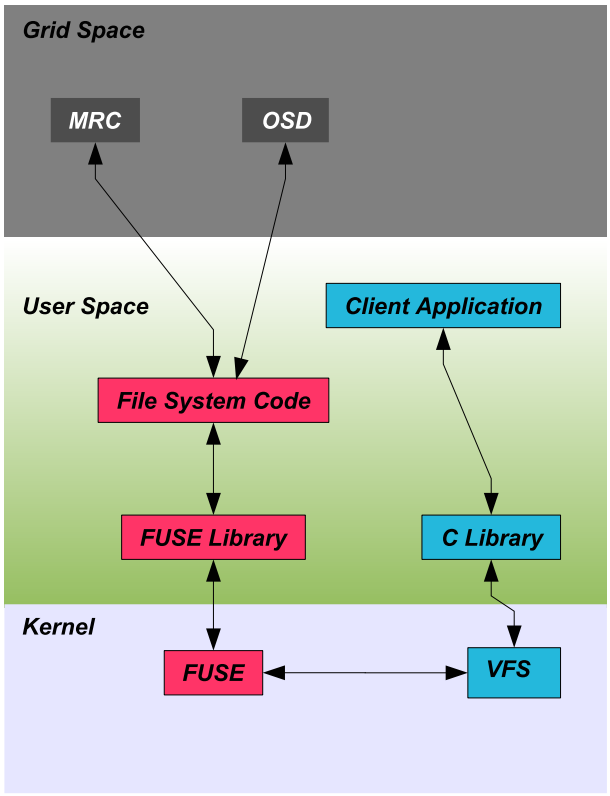


Figure 11: File system architecture. This figure shows how the file system is split into components that are located in the kernel, the nodes user space and the grid.

operating systems (right now Linux and FreeBSD) and thus avoiding the hassle of dealing with kernel level implementations. Figure 11 shows different components of the architecture and their location in the kernel space, the user space and in the grid itself.

FUSE provides a kernel module that interacts with the kernel's virtual file system layer (VFS). Since kernel version 2.6.14 the kernel module of FUSE is contained in the official kernel tree. The file system code uses a FUSE library which in turn communicates with the kernel over a device called `/dev/fuse`. The filesystem code itself will take advantage of the services offered by the MRC and OSD services.

The client application that uses the file system will not know if it is using the XtreemFS. It will access the files in the XtreemFS through the VFS layer of Linux using a POSIX interface.

8.4.2 POSIX I/O

As stated in the description of the architecture, a client application is not necessarily aware of the underlying XtreemFS. It treats files as if they were on a local file system. This approach requires a common interface between all different file systems. POSIX I/O is a standard for accessing I/O and the file system is required to adhere to that standard as strictly as possible and reasonable.

POSIX conformance can be achieved through the use of FUSE. The user space code of the file system can register the functions presented in listing 1 with the FUSE library. These functions represent some parts of the POSIX I/O interface for file systems (not for general I/O).

Listing 1 shows a summary of different file system operations supported by FUSE and the corresponding C language API. The POSIX I/O interface to applications is provided by the C library and the VFS of the Linux kernel. The corresponding user space code of the file system must have a POSIX compliant semantic.

Listing 1: FUSE basic operations

```
/*
    FUSE: Filesystem in Userspace
    Copyright (C) 2001-2006  Miklos Szeredi <miklos@szeredi.hu>

    This program can be distributed under the terms of the GNU LGPL.
    See the file COPYING.LIB.
*/

struct fuse_operations {
    int (*getattr) (const char *, struct stat *);
    int (*readlink) (const char *, char *, size_t);
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
    int (*mknod) (const char *, mode_t, dev_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*symlink) (const char *, const char *);
    int (*rename) (const char *, const char *);
    int (*link) (const char *, const char *);
    int (*chmod) (const char *, mode_t);
    int (*chown) (const char *, uid_t, gid_t);
    int (*truncate) (const char *, off_t);
    int (*utime) (const char *, struct utimbuf *);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,
                  struct fuse_file_info *);
    int (*statfs) (const char *, struct statvfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*setxattr) (const char *, const char *, const char *, size_t, int);
    int (*getxattr) (const char *, const char *, char *, size_t);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
    int (*opendir) (const char *, struct fuse_file_info *);
    int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t,
                    struct fuse_file_info *);
    int (*releasedir) (const char *, struct fuse_file_info *);
    int (*fsyncdir) (const char *, int, struct fuse_file_info *);

    void *(*init) (void);
    void (*destroy) (void *);
```

```
    int (*access) (const char *, int);

    int (*create) (const char *, mode_t, struct fuse_file_info *);
    int (*ftruncate) (const char *, off_t, struct fuse_file_info *);
    int (*fgetattr) (const char *, struct stat *, struct fuse_file_info *);
};
```

8.4.3 Access control

Processes that run under Linux in XtreemOS are ordinary UNIX processes with an associated local user ID (UID) and local group ID (GID) that identify the user that runs the process. The process in XtreemOS also carries a global user ID and an identifier for the VO that the process belongs to. The most common way for processes to access files or directories in a file system is by using a POSIX compliant interface. The access to file system entities like files and directories is controlled by the permission bits read (`r`), write (`w`) and execute (`x`) and eventually access control lists (ACLs).

On the other side, a user in the grid is represented by a globally unique identifier and executes jobs in the context of a specific VO. A job may consist of several UNIX processes running on different nodes. The job's processes execute under temporary local UIDs and GIDs which are assigned by a part of the grid's job management service.

Considering the case where a process accesses grid files via a POSIX-like interface, the process itself only sees the local world of the node where it is running. The process does not have a notion of policies, user certificates or VOs but works with the POSIX view of the file system, that contains permission bits, local uids and gids and access control lists, eventually. Therefore access mechanisms from the XtreemFS (like, for instance, user identities and VOs and policies for access restriction) must be mapped onto mechanism that are defined in the POSIX I/O.

A grid file should at any moment be associated with a global user ID (specifying the file's owner) and the owner VO ID. The VO ID doesn't need to be associated with the file, this can be inherited for example from upper layers like the namespace to which a file belongs to. Both global user ID and VO ID are entities permanently carried by each grid process, they are required for accounting and policy enforcement purposes.

When using the POSIX interface a process that creates a new file can associate only POSIX permissions, including local UID and GID, with that file. It is the task of the file system to translate these into something globally meaningful for the XtreemFS, respecting for example policies on global user

and VO identities. The local file ownership and permission will need to be correlated with the global UID and VO ID carried by the process.

The mapping of local UIDs and GIDs to global user IDs and VO IDs is an economic way to enforce UNIX like access control to global grid files on grid nodes. It uses native UNIX mechanisms for access permission checking and enforcement which are implemented in the kernel. The mapping gives the global identities a simple meaning in local node context: that of user and group IDs.

The mapping from local IDs to grid identities is temporary and specific to each grid node. Once a grid process is running on a node, the mapping of its global identities exists on that node and the local UID/GID can be associated with the grid user ID and VO ID. The process will see files belonging to its grid user and the POSIX interface will take care of presenting these files as if they were owned by the corresponding local UID/GID. Therefore at a given time a file can seem to belong to different local UID/GIDs on different grid nodes. This leads to the corner case of a process attempting to access a file that doesn't belong to itself (its global UID or VO). While the process has a local UID/GID mapping associated with its global UID and VO ID, the file doesn't. A temporary mapping for the global UID and VO ID associated with the file must be created on the fly and persist as long as it is needed, i.e. as long as the file is opened.

This shows the necessity for a service that manages the mapping from grid identities to local IDs on every node. This service is part of WP 2.1.

Beyond the basic file access control and identity mapping described above, the access layer will need to incorporate more advanced mechanisms to enforce arbitrary file-related policies. These are considered advanced features and will be implemented after M18.

8.4.4 Non POSIX

Of course the POSIX like access is not the only possibility to access the grid file system. The XtreemFS can also be accessed by a specifically designed API that may be implemented in a shared library. As presented in figure 11 the POSIX way is through the Linux kernel. But the user space part of the file system code might be accessible via a shared library that is placed between the client application and the user space code. This approach has the disadvantage that any kind of access control has to be implemented in this library and that the library cannot take advantage of the kernel support

for access control. Moreover, application I/O must be reimplemented to meet the requirements of the new API.

8.4.5 Access to Grid-specific Filesystem Features

The functionality of XtreemFS goes far beyond simple file access. Replicas are created or removed, files can be striped in different ways – even over multiple OSDs, and file metadata could be very complex, too. A grid-aware application should be able to take advantage of these features and to give hints to the file system how to store data best, because the application has the best knowledge about the structure of data. This information can, for instance, include information about possible striping patterns or information on how many replicas should be created. In turn, applications could retrieve information about VOs which the files belong to and different, more fine-grained access policies. These policies can include the role of a user in a VO.

In an advanced version, the access layer should allow applications to communicate with the different components of the file system directly. It is not yet clear how this can be done. But one approach might be through calls of `ioctl` in order not to introduce too many fundamental changes in existing codes.

8.4.6 Open Issues

- Handling of special files by MRC
- Dealing with failures of nodes with volatile shared objects

8.5 Research Prospects

One research topic is the meaningful extension of the POSIX-like interface in order to take advantage of the XtreemFS features. These extensions allow applications to better adapt to the grid infrastructure. A monitoring interface could help them avoiding performance bottlenecks, because the application usually knows its data access pattern best.

Another research issue is the integration of distributed filesystem concepts, ideas for replica management and distributed shared memory into a common interface and how far this integration can be done with conventional interfaces (like POSIX) The OSS component emerged from object-oriented

programming methods. Building an object-oriented filesystem interface optimally fitting the OSS needs is another potential research topic.

APPENDIX

A Relations to other Workpackages

A.1 Service Discovery - WP3.2

MRC requirements for Service Discovery/Directory Service. In our design of the XtreemOS file system, the directory service is a central service which ties its components together. File system clients use it to find metadata servers for the volumes they want to access, and metadata servers use it to find out extended static and dynamic information about object storage devices. All changes of information in the directory service is subject to the publish/subscribe system, so that clients can monitor services.

Thus, we will register all our service instances. Those registrations are at least (name, access point) pairs, but usually contain extended static information:

- organisation, country, VOs (these must be validated through mechanisms from WP3.5)

and dynamic information:

- availability (e.g. online, offline, dead)
- load information
- service-specific information (e.g. current disk capacity)

Result sets of query requests need to be sorted by dynamic metrics, for example bandwidth/latency/... between a given address and the server (as described in Annex.1). Queries over the list of servers should be possible (e.g. select specific server types). Updates to the volume name → MRC list must be atomic and should be synchronised with the list of available volume names. Status changes of servers (e.g. dead) could be disseminated via WP 3.2 pub/sub service.

A.2 Publish/Subscribe - WP3.2

Within the scope of WP3.2, a scalable pub/sub system will be implemented. Possible applications for pub/sub in our workpackage are:

- disseminating application/network/host monitoring information
- persistent queries in the Metadata Catalog
- notification on OSD join/leave/crash may need reliable transport

A.3 Remote Execution - WP3.3

We would like WP3.3 to offer us support for remote control of demons. This could be done by having all demons behave as processes of a single job and thus be able to treat all of them with job functionality to control them, monitor them, signal them, etc. WP3.3 will offer a mechanism to create this kind of jobs.

A.4 Replica and Job Collocation - WP3.3

It is important for efficiency reasons that replicas of files are located near (performance wise) to where the jobs using them will be executed. This can be done in two different ways. The first option is to try to place the job near the files. To approach this solution we will work on mechanisms to inform the execution management system of how near (performance wise) the files used by a job are to the set of resources being evaluated. On the other hand, once the resources have been assigned to a job, and before it is executed, this service will inform the file system to allow the creation of replicas nearby if needed.

A.5 SSI - WP2.2

The SSI functionality of WP2.2 includes a basic DSM for clusters. It is not planned to put much efforts on the existing Kerrighed DSM within WP2.2 but it seems natural that there may be synergies with the OSS service of WP3.4.

Furthermore, the container concept of Kerrighed - a data sharing facility - might be a good candidate to be extracted from Kerrighed and pushed into

the Linux kernel mainstream. Pushing Kerrighed as a single big kernel extension into the Linux community seems too be to ambitious. If the containers would be extracted within WP2.2 further synergies with OSS may show up.

A.6 Checkpointing - WP2.2

The OSS service needs to support fault tolerance, e.g. by ensuring that always enough replicas of an object exist. In the world of reliable DSM systems checkpointing has been interweaved into consistency protocols.

Checkpointing requires resetting a node to a previously saved state. OSS will allow speculative memory access typically bundled within transactions that may also be aborted and thus affected operations must be reset, too.

Although these resetting actions have different origins (crashes and aborts) both would prefer a fast recovery with a low overhead during fault free execution. Definitely, a discussion with WP2.2 is reasonable to identify potential synergies.

A.7 VO Management - WP2.1

The access layer needs to dynamically translate file ownership and access policies from their global scope defined on the grid, i.e. related to global user ID, virtual organisation (VO) ID, other global entities, to local, node specific permissions and access rights. The mapping between global and local entities is expected to be a service provided as output of workpackage WP2.1. The concrete mechanisms and APIs need to be adapted to the XtreemFS access layer needs in close collaboration with WP2.1. This will be eased by the fact that some WP3.4 members are working on WP2.1, too.

B Protocols

The standard protocol for storage systems iSCSI [9] of T10, with its OSD extension for object-based storage systems. The important design considerations of iSCSI are [13]:

1) It uses TCP instead of a custom reliable transport protocol built on UDP

2) It mixes command/control and data in one connection

3) It allows parallel connections

4) It is stateless

5) It uses a binary representation with fixed size field lengths in order to facilitate in-hardware parsing

The disadvantages of iSCSI are:

- It is a complex protocol and there are not many open source implementations around (reusability of code)
- Being a binary protocol, it is harder to debug

In order to not hinder ourselves by protocol stack implementations, we chose to use HTTP as our base protocol. It shares all design features of iSCSI but is ASCII-based, which makes it easier to debug, and hardware implementations are not in our scope anyway. Also, it is very extensible, and there are many tools and implementations available. If iSCSI support becomes desirable at one point, for example to communicate with commercial object stores, we have similar protocol semantics and therefore supporting it should be doable.

A marshalling protocol for structured data/RPCs is still to be found, candidates are XMLRPC, SOAP, JSON, Sun XDR ([16][17][18]).

B.1 Usage and Extensions of the HTTP Protocol for XtreemFS

Communication with the MRC is done through XMLRPC, all necessary information is contained in the XMLRPC request. No extension to the HTTP protocol is necessary.

For the OSD the HTTP protocol itself is sufficient to encapsulate all operations. Byte ranges can be requested through the `Content-Range` header field, which is part of the HTTP 1.1 standard. For the capabilities and information on replicas, we introduce non-standard header fields

`x-capability: "<capability string>"`

and

`x-replicas: "<list of OSDs holding replicas>"`.

OSD operations can now easily be translated into HTTP methods. The `objID` is transmitted as the URI.

- `read` is translated into a `GET` request.
- `write` is translated into a `PUT` request.
- `delete` is translated into a `DELETE` request.
- `create` is not translated but implied by a `PUT` for a non-existing object.

An example request to the OSD could look like this:

```
GET /123456789 HTTP/1.1
Host: osd1.xtreemos.org
Content-Range: bytes 1000-2999/*
x-capability: "74f876e8a8340d6e0b06585f9834e0659a"
x-replicas: "osd1.xtreemos.org:8080 osd2.xtreemos.org:8084
             osd5.xtreemos.org:8080"
```

The answer could look like this:

```
HTTP/1.1 200 OK
Host: osd1.xtreemos.org
Content-Range: bytes 1000-2999/*
Content-Type: application/octet-stream
Content-Length: 2000
Cache-Control: no-cache

Here starts the filecontent...
```

C FAQ

Does the client do caching?

No, caching is not done in the client. This is required in order to be able to offer POSIX/Linux semantics in presence of concurrent access from multiple clients. Therefore, we don't need any special deviating semantic definitions as for example CODA's session semantics.

Can I export a directory on my local machine to XtreemFS?

No. But you can setup an XtreemFS volume on your machine which becomes available on the grid when you are connected to the internet. Technically speaking you set up an MRC and an OSD on your machine creating a new XtreemFS volume.

Is XtreemFS similar to NFS?

Yes and No! From the user's perspective it is similar. You can mount XtreemFS volumes through the FUSE Interface in your directory tree. Just like an NFS server. Volumes can also be automounted when you need them. From the technical perspective XtreemFS is completely different from NFS. First of all, it is distributed, there is no central server as in NFS. In addition, various XtreemFS installations export their volumes and are tied together in a federation. The user, however, does not notice that and how XtreemFS is distributed.

Do I have to make backups of XtreemFS volumes?

Not really. You can choose the replication policy of volumes or files. So you can decide how many copies you need. Of course, you can still make backups of XtreemFS volumes using standard software.

Do I have to change my applications to work with XtreemFS?

No. Legacy applications can use the traditional Linux file system interface without restrictions. To benefit from advanced features like replication policies or user defined metadata your application can be linked against the XtreemFS client lib.

A single MRC must be a bottleneck?

Yes. But you usually have a federation of MRCs, and those MRCs are replicated and partitioned. This does not only enhance performance but increases also the availability.

How can a file be striped?

The user can choose from a range of striping patterns depending on his or her needs, e.g. RAID 0 for performance or RAID 5 to save disk space.

But I don't want my files to be stored at host X or in country Y!

Users can specify replication policies to tell XtreemFS where to replicate to, and where not.

Will you support ACLs?

Yes, it is planned to support them.

What is the difference between XtreemFS and a RDBMS?

XtreemFS is not an RDBMS. However, it is a database and a filesystem. You can store arbitrary attributes per file and do queries over attributes. But you do not have tables or SQL statements.

Can a file belong to multiple VOs?

Yes, a file can be accessed from users of multiple VOs, if access policies permit.

Can XtreemFS notify me/my application if a file changes?

Yes. XtreemFS supports persistent queries, i.e. you send a query to an MRC which it will keep. Once a file matching your query changes, you get a notification via WP3.2's pub/sub service.

Does XtreemFS provide POSIX/UNIX file system semantics?

Yes. We try to be as close as possible to Linux file system semantics. This is, however, not always what you expect when reading the POSIX standard.

References

[1] Lustre: A Scalable, High-Performance File System.

[2] Panasas ActiveScale File System (PanFS).

[3] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an Object Store. In *MSST '03: Proceedings of 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, 2003.

[4] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient Metadata Management in Large Distributed Storage Systems. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 290, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[6] Micahel Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, 2005.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[8] The Open Group. The Single Unix Specification, Version 3.

[9] IBM, Hewlett-Packard. RFC 3720 - Internet Small Computer Systems Interface (iSCSI), 2004.

[10] T. M. Kroeger and D. D. E. Long. Predicting the future file-system actions from prior events. In *Proceedings of the Annual Technical Conference*, 1996.

[11] F.A. Briggs M. Dubois, C. Scheurich. Memory access buffering in multiprocessors. In *Proceedings of the Symposium on Computer Architecture*, 1986.

[12] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 8:84–90, 2003.

[13] Kalman Z. Meth and Julian Satran. Design of the iscsi protocol. In *IEEE Symposium on Mass Storage Systems*, pages 116–122, 2003.

[14] D. Mosberger. Memory consistency models. In *Proceedings of the ACM SIGOPS review, 27(1)*, 1993.

[15] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02: Proceedings of the 1st USENIX*

Conference on File and Storage Technologies, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[16] Sun Microsystems, Inc. RFC 1050 - RPC: Remote Procedure Call Protocol specification, 1988.

[17] Sun Microsystems, Inc. RFC 1831 - RPC: Remote Procedure Call Protocol Specification Version 2, 1995.

[18] Sun Microsystems, Inc. RFC 1832 - XDR: External Data Representation Standard, 1995.

[19] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, volume 7. USENIX, November 2006.

[20] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Dave Winer. XML-RPC Specification.