Project no. IST-033576

XtreemOS

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

XtreemFS prototype month 18
D3.4.2

Due date of deliverable: 30-NOV-2007
Actual submission date: 30-NOV-2007

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP3.4

Responsible institution: BSC
Editor & and editor's address: Toni Cortes
Barcelona Supercomputing Center
Jordi Girona 29
08034 Barcelona
Spain

Version 1.0 / Last edited by Toni Cortes / 30-OCT-2007

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

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	25.09.07	Toni Cortes	BSC	Initial document structure and initial contents from work documents
0.2	25.10.07	Jan Stender, BjÃűrn Kolbek	ZIB	Architecture of Main components and code
0.3	25.10.07	Matthias Hess	NEC	Architecture of client layer
0.4	25.10.07	Michael Schoettner	UDUS	Architecture of OSS
1.0	30.10.07	Toni Cortes and Jan Stender	BSC and ZIB	Final editing
1.1	16.10.07	Toni Cortes	BSC	Updating reviewers comments
1.2	29.11.07	Florian Mueller	UDUS	Updating reviewer comments related to OSS

Reviewers:
Julita Corbalan (BSC) and Adolf Hohl (SAP)

Tasks related to this deliverable:

Task No.	Task description	Partners involved$^{\circ}$
T3.4.1	File Access Service	CNR*, BSC, ZIB
T3.4.3	Metadata Lookup Service	ZIB*
T3.4.5	Grid Object Management	UDUS*
T3.4.6	Access Layer for File Data and Grid Objects	NEC*, UDUS

$^{\circ}$This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

1 Introduction **3**

1.1 Document Structure . . . 3

2 Brief Description of the Prototype **3**

2.1 Current Functionality . . . 4

2.2 Main Architecture . . . 5

2.3 Architecture of the Main Components . . . 7

3 Installation and Configuration **20**

3.1 Checking out XtreemFS . . . 20

3.2 Requirements . . . 21

3.3 Building XtreemFS . . . 21

3.4 Installation . . . 21

3.5 Configuration. . . . 23

4 User Guide **25**

4.1 Mounting the File System . . . 25

4.2 Tools . . . 26

4.3 OSS Developers Guide . . . 27

5 Conclusion and Future Work **29**

5.1 Limitations of the Prototype . . . 29

5.2 XtreemFS and kDFS . . . 29

Executive Summary

This document presents the development state of XtreemFS, the XtreemOS Grid file system, as well as the Object Sharing Service (OSS), as it is in month 18 of the project. Even though many more features are currently in development or planned, we have decided to only mention those parts of our implementation that are fully functional.

We give an overview of the functionality currently supported in our prototype, as well as the main system architecture. The XtreemFS architecture is based on three services: Object Storage Devices (OSD) for storing file content, Metadata and Replica Catalogs (MRC) for maintaining file system metadata, and the access layer which allows user processes to access the file system. The OSS is an additional service that provides transaction-based sharing of volatile memory and will support memory-mapped files for XtreemFS in the near future.

The staged architecture of the different services including their stages and components are described in detail, together with the most important algorithms used internally. Finally, a user manual with instructions to install and use the file system, as well as a short developer guide for the OSS are provided.

1 Introduction

XtreemFS [2] is an object-based file system that has been specifically designed for Grid environments as a part of the XtreemOS operating system. In accordance with its object-based design [3, 5], it is composed of two different services: OSDs (Object Storage Devices) that store file content, and MRCs (Metadata and Replica Services) that store file system metadata. A client module provides the access layer to the file system. It performs file system operations on behalf of users by communicating with the file system services. The services are complemented by the Object Sharing Service (OSS), an additional component that provides transaction-based sharing of volatile memory objects and will support memory-mapped files for XtreemFS.

In a Grid, XtreemFS offers a global view to files. Files and directory trees are arranged into volumes. A volume can be mounted at a any Grid node, where any sufficiently authorized job can access and modify its files. In order to improve performance and efficiency, XtreemFS also offers striping of files across different OSDs. Moreover, replication of files will be supported in later XtreemFS releases, which will provide data safety and increase access performance.

In this deliverable, we present the main characteristics that appear in the current prototype of XtreemFS.

1.1 Document Structure

This document is structured as follows. In Section 2 we present description of the XtreemFS prototype and the OSS. We give an overview of the currently available functionality and the main architecture of these components. In Section 3, we present the procedures needed to download, install, and configure the XtreemFS. Section 4 presents the interface applications can use to access the file system, as well as a short developer guide for the OSS component. Finally, section 5 points out the topics on which we will work in the next few months.

2 Brief Description of the Prototype

In this section, we describe the current prototype of XtreemFS and the OSS. We first outline the functionality that is currently implemented and working.

Afterwards, we present a global view on the file system architecture to finally describe the architectural details of each component.

2.1 Current Functionality

Before starting with the description of the current functionality in XtreemFS, it is important to clarify that the current version only supports parts of the final functionality (e.g., no replica management is available) and that performance was not a key issue yet. The objective of the current release of XtreemFS was to offer a global file system that could be used transparently by applications. For the OSS part, the first major goal was to provide basic sharing functionality.

As already mentioned, volumes are the most high-level abstraction used to organize multiple files in XtreemFS. A volume contains a set of files organized in a tree structure (following POSIX semantics) which can be mounted in a Linux tree structure. Thus, the first set of functionality offered by XtreemFS is to create, delete, mount, and unmount these volumes.

The most important feature of XtreemFS is the global view it offers to applications. XtreemFS allows any application to access files that are physically located anywhere in the world in a transparent way. Files and directory trees in mounted volumes can be accessed by applications in a POSIX-compliant fashion, by using traditional Linux system calls.

In order to improve the access time of data, XtreemFS implements striping of files across multiple OSDs. Although in the future this striping will be on a per-file/replica basis, our current implementation only allows striping policies to be defined at volume level, i.e. all files in a volume have the same striping policy.

Finally, security issues have been taken into account by supporting SSL-secured communication channels.

As regards key functionality of XtreemFS which has not yet been implemented, we should mention that no replication and fault tolerance is yet available in the services.

The OSS component currently supports basic sharing and memory management functionality, including basic lock and replica management. Furthermore, the system call 'mmap' is intercepted to later support transparent memory-mapped files for XtreemFS. Next development steps include speculative transactions, scalable communication, and support for memory-mapped files.

2.2 Main Architecture

The main components involved in our architecture are the Metadata and Replica Catalog (MRC), the Object Storage Device (OSD) the client/Access Layer, the Replica Management Service (RMS)[1], and the Object Sharing Service (OSS). In the following paragraphs, we describe the different components and how they work together.

Client/Access Layer. The XtreemFS access layer represents the interface between user processes and the file-system infrastructure. It manages the access to files and directories in XtreemFS for user processes as well as the access to Grid-specific file system features for users and administrators. It is the client side part of the distributed file system and as such has to interact with the services of XtreemFS: MRC and OSD. The access layer provides a POSIX interface to the users by using the FUSE [1] framework for file systems in user space. It translates calls from the POSIX API into corresponding interactions with OSDs and MRCs. An XtreemFS volume will be simply mounted like a normal file system, i.e. there is no need to modify or recompile application source code in order to be able to run applications with XtreemFS. In addition to the POSIX interface, the access layer will provide tools for creating and deleting XtreemFS volumes, checking file integrity, querying and changing the file striping policies, and other Grid-specific features.

OSD. OSDs are responsible for storing file content. The content of a single file is represented by one or more objects. With the aim of increasing the read/write throughput, OSDs and Clients support striping. Striping of a file can be performed by distributing the corresponding objects across several OSDs, in order to enable a client to read or write the file content in parallel. Future OSD implementations will support replication of files, for the purpose of improving availability and fault tolerance or reducing latency.

MRC. MRCs constitute our metadata service. For availability and performance reasons, there may be multiple MRCs running on different hosts. Each MRC has a local database in which it stores the file system metadata it accounts for. The MRC offers an interface for metadata access. It provides functionality such as creating a new file, retrieving information about a file or renaming a file.

[1]RMS will not appear anymore in this deliverable because its implementation will not start till month 18 of the project

OSS. The OSS enables applications to share objects between nodes. In the context of OSS, the notion *object* means a volatile memory region. The service resides in user space co-located with the applications and manages object exchange/synchronization almost transparently. Depending on the access pattern of an application and due to efficiency reasons OSS supports different consistency models. Furthermore, OSS intercepts 'mmap' calls to support memory-mapped files for XtreemFS in the near future.

To get a more general idea of how the different components work together, we can see Figure 1. The different processes running on the client side of XtreemFS access files normally via the regular Linux interface. The VFS redirects the operations related to XtreemFS volumes to the Access Layer via FUSE. In turn, the Access Layer interacts with the XtreemFS services to carry out the requested file system operations. Metadata-related requests, such as opening a file, are redirected to the MRC. File content-related requests, such as reading a file, are redirected to the OSDs on which the corresponding objects containing the data reside. The OSS is not included in the picture because in the current prototype this service is a standalone one.

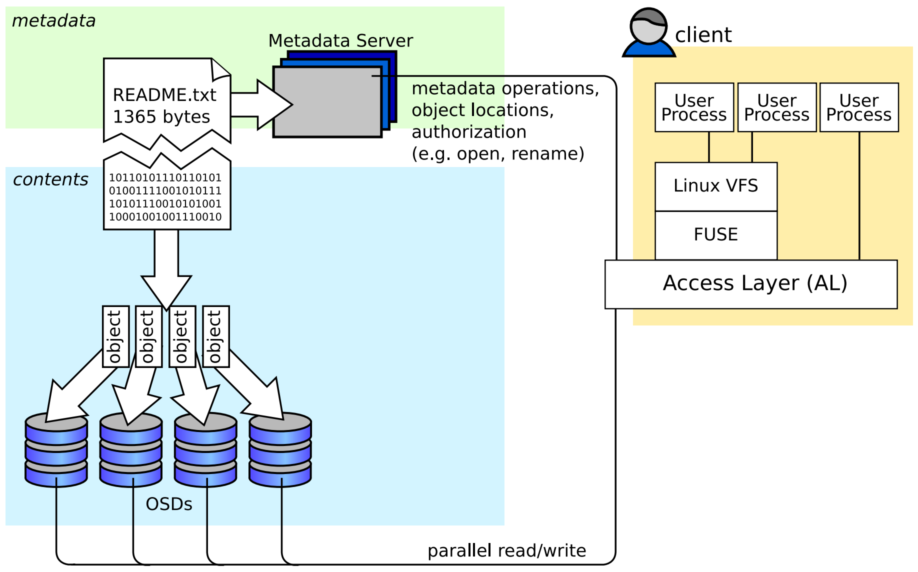

Figure 1: Relationship between XtreemFS components

Regarding the communication between all these components, we have decided to use the HTTP protocol, as it is well-established and fully suits the communication needs of our components.

2.3 Architecture of the Main Components

This section provides a detailed description of the architecture of the three components that currently constitute XtreemFS, as well as the OSS component.

2.3.1 Common Infrastructure for MRC and OSD

Both MRC and OSD rely on an event-driven architecture with stages according to the SEDA [6] approach. Stages are completely decoupled and communicate only via events which are sent to their queues. They do not share data structures and therefore do not require synchronization.

Each stage processes a specific part of a request (e.g. user authentication, disk access) and passes the request on when finished. This way, requests are split down to smaller units of work handled subsequently by different stages. Depending on the type (e.g. read object or write object), a request has to pass only a subset of the stages in a request type dependent order. This is modeled by a RequestHandler component, which encapsulates the workflow for all request types in a single class. It implements the order in which requests are sent to the individual stages.

For communicating with other servers and with clients, the MRC and OSD both use an HTTP server and client. Each is implemented as a separate stage.

Pinky is a single-threaded HTTP-Server. It uses non-blocking IO and can handle up to several thousand concurrent client requests with a single thread. It supports pipelining to allow maximum TCP throughput and is equipped with mechanisms to throttle clients to gracefully degrade client-throughput in case of high loads. Incoming HTTP-Requests are parsed by Pinky and passed on to the RequestHandler.

Speedy a single-threaded HTTP-Client, is Pinky's counterpart. It can be used to handle multiple connections to different servers and is able to use pipelining. Speedy automatically takes care of closing unused connections, reconnecting after timeout and periodic connection attempts if servers are unavailable. It is optimized to reach maximum performance when communicating with other services using Pinky.

Request Pipelining. Pinky can receive requests one-by-one or pipelined. When sending requests one-by-one the client waits for the response before sending the next request. With HTTP pipelining, the clients can send more requests while waiting for responses. The one-by-one approach has significant performance problems, especially when the latency between client and server is high.

2.3.2 MRC

Figure 2 shows the request flow in the MRC. The stages including their components and functionality are described in more detail in the following sections.

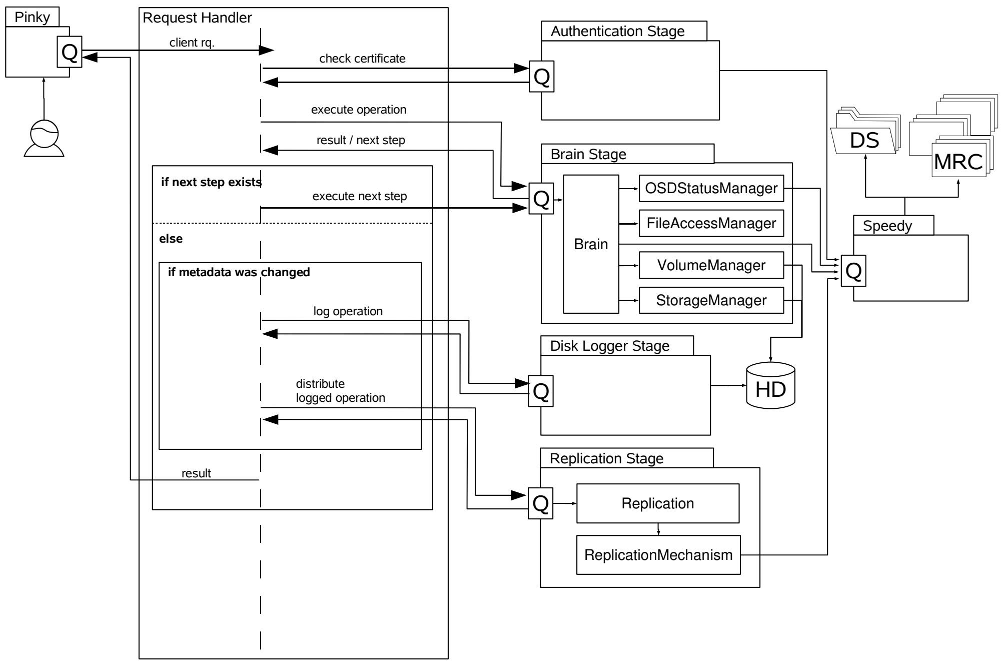

Figure 2: MRC design

Stages. The internal logic of the MRC is implemented in four stages.

Authentication Stage checks the authenticity of the user. It performs a validity check of user certificates and extracts the global user and group IDs.

Brain Stage is the core stage of the MRC which implements all metadata related file system logic such as file name parsing, authorization and operations like `stat` or `unlink`. For each volume it stores the directory tree, the file metadata and the access control lists. Internally it is composed of several components.

Brain. Contains the specific logic for each operation. It decomposes the different calls into invocations of the other Brain Stage components.

File Access Manager. Checks whether the user is sufficiently authorized for the requested operation. If not, it returns an error which results in a corresponding notification to the client.

Volume Manager. The Volume Manager is responsible for managing all volumes which have been created on the local MRC.

Storage Manager. For each volume held by the local Volume Manager, the Storage Manager provides access to the respective data. It translates requests for metadata modification and retrieval into calls to the database backend.

OSD Status Manager. Regularly polls the Directory Service for the set of OSDs that can be assigned to a newly created file.

Disk Logger Stage persistently logs operations that change file system metadata. This allows the backend to work with a volatile in-memory representation of the database, while recording all changes in the operations log. When the log becomes too large, a checkpoint of the database is created from the current in-memory representation, and the log is truncated. Thus, the MRC can recover from crashes by restoring the state of the database from the last checkpoint plus the current operations log.

Replication Stage replicates individual volumes in the MRC backend and ensures replica consistency.

The Replication Stage internally relies on the following components:

Replication. Sends operations to remote MRCs which hold replicas of the corresponding volumes through the Speedy stage.

ReplicationMechanism. A specific (exchangeable) mechanism for replication. It determines how operations are propagated to remote replicas. Currently, only master/slave replication without automatic failover is implemented.

Remarks on Request Processing. Pinky can receive requests one-by-one or pipelined. With HTTP pipelining, the clients can send more requests while waiting for responses. Since the MRC does not guarantee that pipelined requests are processed in the same order as they are sent, different requests should only be added to the same pipeline if they are causally independent. Thus, certain sequences of requests such as "createVolume" with a subsequent "createFile" on the newly created volume should not be processed in a pipelined fashion, as the second request could possibly be executed in the MRC prior to the first one, which would lead to an error.

2.3.3 OSD

The OSD has four stages. Incoming requests are first handled by the Parser stage which parses information sent together with the request, such as the capability or the X-Locations lists. The validity of capabilities and the authenticity of users is checked by the Authentication stage. The Storage stage handles in-memory and persistent storage of objects. The processing of UDP packets related to striping is handled by the UDPCom stage, striping-related RPCs are handled by the Speedy stage. The internal architecture is depicted in Figure 3.

Stages. The main stages of the OSD will be described in the following paragraphs.

Parser Stage is responsible for parsing information included in the request headers. It is implemented in a separate stage to increase performance by using an extra thread for parsing.

Authentication Stage checks signed capabilities. To achieve this, the stage keeps a cache of MRC Keys and fetches them if necessary from some authoritative service (e.g. the Directory Service).

Storage Stage is responsible for storing and managing objects. For the sake of performance, the Storage Stage relies on caching. The Cache is used for fast access to objects which are still in memory. If an object

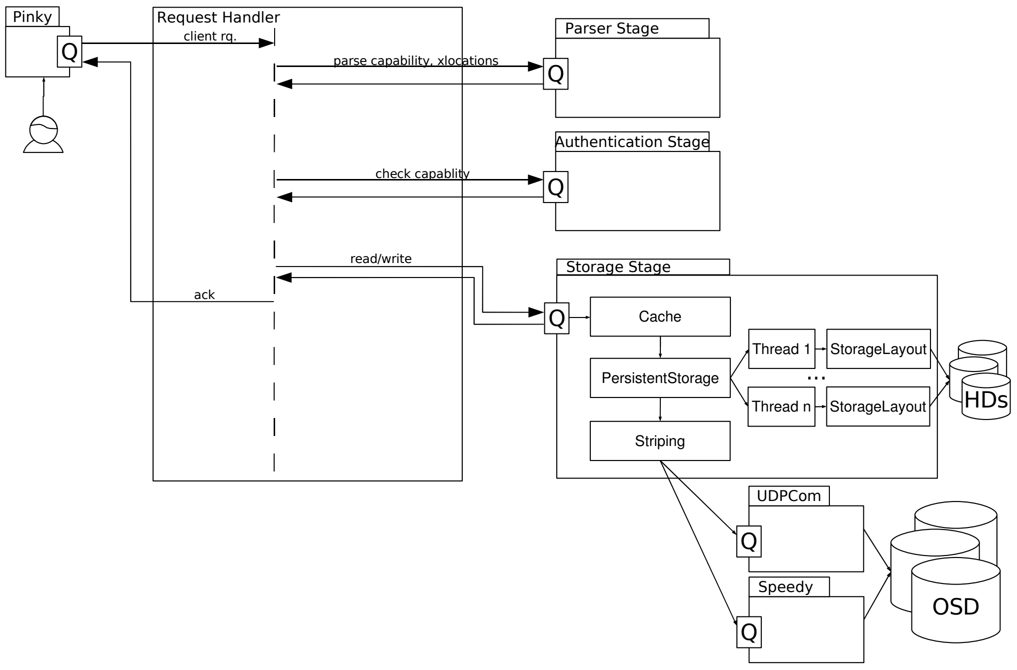

Figure 3: OSD Design

is not available in-memory, the Cache will instruct the PersistentStorage component to load the object from disk. Both Cache and PersistentStorage will also take care of writing objects to disk and managing the different versions of an object to ensure copy-on-write semantics.

As part of the PersistentStorage component, the StorageLayout component is used to map objects to the underlying storage devices. For efficient on-disk data access, the current implementation of the storage layout relies on an arrangement of objects on the OSD's local file system. Rather than storing all objects in a single file, we exploit the local file system's capability to organize files in a directory hierarchy. A file is represented by a physical directory, and an object is represented by a physical file. Directories representing single files, in turn, are arranged in a directory hierarchy, by means of hash prefixes of their file IDs. Such an arrangement ensures that single objects can be retrieved efficiently, as the amount of files contained by a single directory does not become too large.

The Striping component implements the logic needed to deal with striped files. Striping in a distributed file system has to take into account some special considerations. When a file is read or written, the system must compute the OSD on which corresponding stripe resides. Furthermore, I/O operations like truncate are more complex. Decreasing or increasing the size of a file may require communication with remote OSDs (i.e. the ones where the affected stripes reside on), since the difference between the original and the new size can involve several stripes. Moreover, when a file has to be deleted, the system has to contact each OSD holding a stripe of the file. A detailed description of the striping protocols can be found at the end of this section.

Important Data Structures. Data structures should not be shared among stages. Information should be attached to the requests instead (reference confinement must be guaranteed!).

OpenFileTable. This table contains a list of currently opened files, time of last access and a flag if they will be deleted on close. Keeping track of the *open state* of files is necessary for POSIX compliance, since files opened by a process need remain accessible until they are closed even if they are concurrently deleted by a different process.

In-memory cache. The cache stage has an in-memory cache of objects to increase performance for read operations. Accesses to objects are first attempted to be served from the cache. In case of a cache miss, the PersistentStorage component retrieves the corresponding object from disk.

Protocols for Striping. We defined a set of protocols for reading, writing, truncating and deleting striped files. The protocols are optimized in a way that operations occurring frequently, like read and write, can normally be handled fast, whereas truncate and delete have to be coordinated among all OSDs holding stripes.

For each file, each OSD keeps persistently the largest object number of the objects stored locally ($localMax$). In addition, each OSD keeps the global maximum object number it currently knows ($globalMax$). $globalMax$ is part of the "open" state and does not need to be persistently stored.

Truncate involves the MRC, a client, the head OSD for the file (OSD0), and all other OSDs (OSD1..N).

The pseudocode for the MRC:

```
UPON truncate($fileId$)
  $file[fileId].issuedEpoch++$
  return capability(TRUNCATE, $fileId$, $file[fileId].issuedEpoch$)
END

UPON X-New-Filesize($fileId$, $fileSize$, $epoch$)
  IF $epoch > file[fileId].epoch \lor$
       $(epoch = file[fileId].epoch \land fileSize > file[fileId].size)$
  THEN
    -- accept any file size in a later epoch or any larger file size in the current epoch
    $file[fileId].epoch := epoch$
    $file[fileId].size := fileSize$
  END IF
END
```

The pseudocode for the head OSD:

```
UPON truncate($fileId$, $fileSize$, $capability$)
  $file[fileId].epoch := capability.epoch$
  truncate_local($fileId$, $fileSize$)
  FOR $osd$ in $OSD1..N$ DO
    relay truncate($fileId$, $fileSize$, $capability.epoch$)
  DONE
  return X-NEW-FILESIZE $fileSize$, $capability.epoch$
END
```

The pseudocode for other OSDs:

```
UPON relayed_truncate($fileId$, $fileSize$, $capability$)
  $file[fileId].epoch := capability.epoch$
  truncate_local($fileId$, $fileSize$)
END
```

The pseudocode for the client:

```
BEGIN truncate(fileId)
  $capability$ := MRC.truncate($fileId$)
  X-NEW-FILE-SIZE := headOSD.truncate($fileId$, $fileSize$, $capability$)
  MRC.updateFilesize(X-NEW-FILE-SIZE)
END
```

Delete is performed in a fully synchronous fashion. The head OSD relays the request to all other OSDs. All OSDs will either delete the file (and all on disk objects) immediately, or mark the file for an "on-close" deletion.

Write has to consider cases in which the file size is changed.

```
BEGIN write($objId$)
  write object locally
  IF $objId > globalMax$ THEN
    udpBroadcast($objId$) as new globalMax to all OSDs
    send X-NEW-FILESIZE to client
  ELSE IF $objId = globalMax \wedge$ object is extended THEN
    send X-NEW-FILESIZE to client
  END IF
END
```

Read requires a special treatment of border cases in which objects do not or only partially exist on the local OSD.

```
BEGIN read($objId$)
  IF object exists locally THEN
    IF object is not full $\wedge$ $objId < globalMax$ THEN
      send object + padding
    ELSE IF object is not full $\wedge$ $objId = globalMax$ THEN
      -- not sure if still is last object
      IF read past object THEN
         retrieve $localMaxOSD1..N$ from all OSDs
         $globalMax$ := max($localMaxOSD1..N$)
         IF $objId = globalMax$ THEN
           send partial object
         ELSE
           send padded object
         END IF
      ELSE
        send requested object part
      END IF
    ELSE
```

```
        -- Object is full
      send object
    END IF
  ELSE
    -- check if it is a "hole" or an EOF
    IF $objId > localMax$ THEN
      IF $objId > globalMax$ THEN
        -- not sure if OSD missed a broadcast
        -- coordinated operation for update
        retrieve $localMaxOSD1..N$ from all OSDs
        $globalMax$ := max($localMaxOSD1..N$)
      END IF
      IF $objId > globalMax$ THEN
        send EOF (empty response)
      ELSE
        send zero padding object
      END IF
    ELSE
      -- "hole", i.e. padding object
      send zero padding object
    END IF
  END IF
END
```

2.3.4 Client layer

The XtreemFS component called *client* is the mediator level between an application and the file system itself. The focus lies currently on a POSIX compliant interface in order to support applications that are not specifically written for XtreemFS.

The client layer is implemented as a file system based on FUSE [1].

Stages The client is built up from different stages. Each stage employs one or more threads to handle specific requests. One stage can generate multiple other requests for the following stages, so called child requests. The request initiating thread can go on with other work or wait until the request is finished. In any case the request itself is handled by a different thread. Once the work is finished, the work handling thread calls a callback function that finalizes the request and eventually wakes up any thread that is waiting for the request to be finished. A simplified sequence diagram of handling a request is presented in fig. 4.

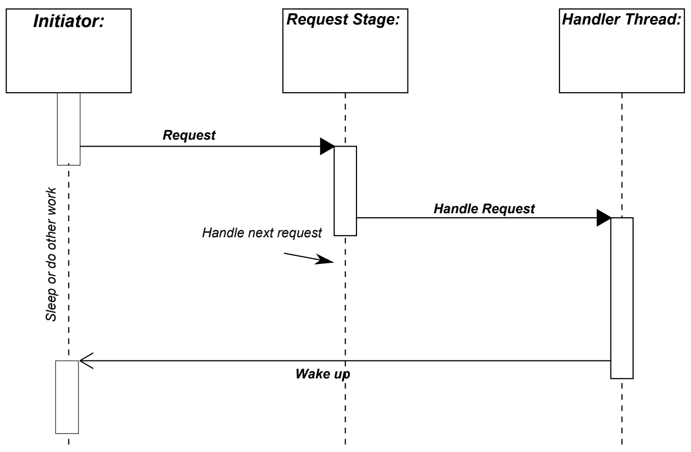

Figure 4: Sequence diagram for the clients stages.

In fig. 5 we present an overview of how an I/O operation is divided into operations on file objects and ultimately into operations on stripe objects. An application that uses XtreemFS and interacts with it via the FUSE interface does I/O operations based on bytes. Each request – read or write – specifies an offset (in bytes) and a number of bytes starting from that offset. As XtreemFS is an object-based file system, these requests must first be translated into requests that are based on file objects. XtreemFS also allows striping over different OSDs. The next step therefore is to associate each file object with the corresponding OSD and transferring the objects to and from the OSD. In future versions of XtreemFS there will be also RAID policies implemented to allow some redundancy of the data. The redundant information (like parity calculations) will also be considered an object that must be stored onto an OSD. Currently only RAID level 0 is implemented in XtreemFS so there will not be any additional data.

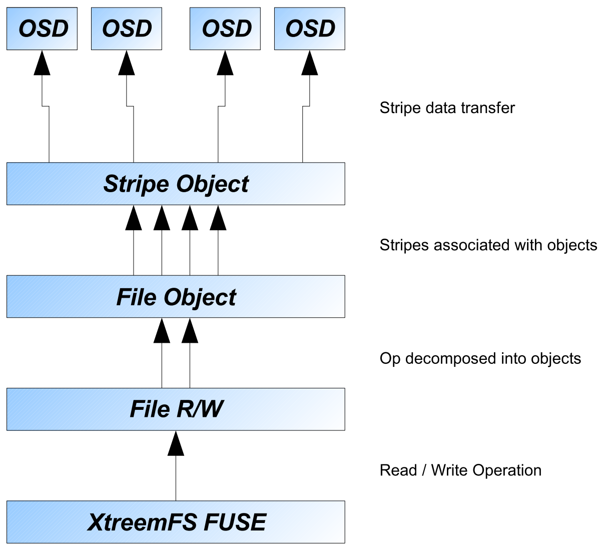

Figure 6: Interdependencies of the I/O stages in the client

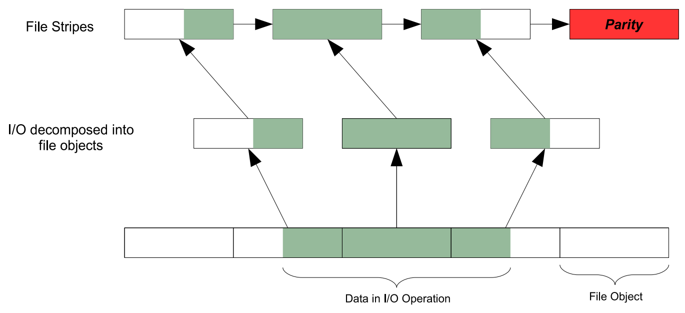

Figure 5: I/O operations are first divided into file objects and then associated with stripe objects that correspond to the RAID level of a file.

I/O requests are handled by different stages that take care of a specific aspect of the operation. In fig. 6 the control flow for an I/O operation is sketched.

File read/write stage. File requests like read or write operations are translated into requests on object level.

File object stage. File object requests are translated into stripe object requests according to the striping policy (or RAID level in particular).

Stripe object stage. In this stage the actual communication with the right OSDs is executed. For each stripe the right OSD is determined and requests are prepared and submitted to them. The answers from the OSDs are analyzed. OSDs might answer with failure of the operation, a request to redirect to another OSD or with success. If the operation was a success, the client knows that the data have been transferred to the OSD successfully (in a write operation) or that a read operation succeeded.

2.3.5 OSS

Built as an event-driven modular architecture, OSS separates the mechanisms that realize basic consistency from an extensible set of policy-driven consistency models. Furthermore, the service is independent of the underlying transport protocol which is used for inter-node communication. Thus OSS consists of several modules for memory and page management, communication, and consistency protocols. We will describe their responsibilities and dependencies in the following paragraph, also shown in figure 7.

Modules

Communication module. The communication module manages the entire communication of the nodes. It is used by the heap, page and consistency modules to exchange data over the network. The module transfers information as messages which are encapsulated in Protocol Data Units (PDU). PDUs are sent over the network via the underlying transport protocol. Received messages are enqueued into a FIFO message queue. However, the queue allows reordering of PDUs. This is sometimes necessary if an affected memory page is locked locally by the application, but the message handler must remain responsive. The current implementation uses a TCP implementation that will be replaced by a UDP-based overlay multicast implementation.

Page module. The page module manages the exchange and invalidation of memory pages. A process is able to request memory pages from other nodes or reversely to update memory pages at any other node with its own page content. The process can also invalidate replicas of memory pages in the grid. Before serving external requests, the module generates an event to the appropriate consistency module to check whether consistency constraints are fulfilled. Compression of pages and/or exchanging differences are planned.

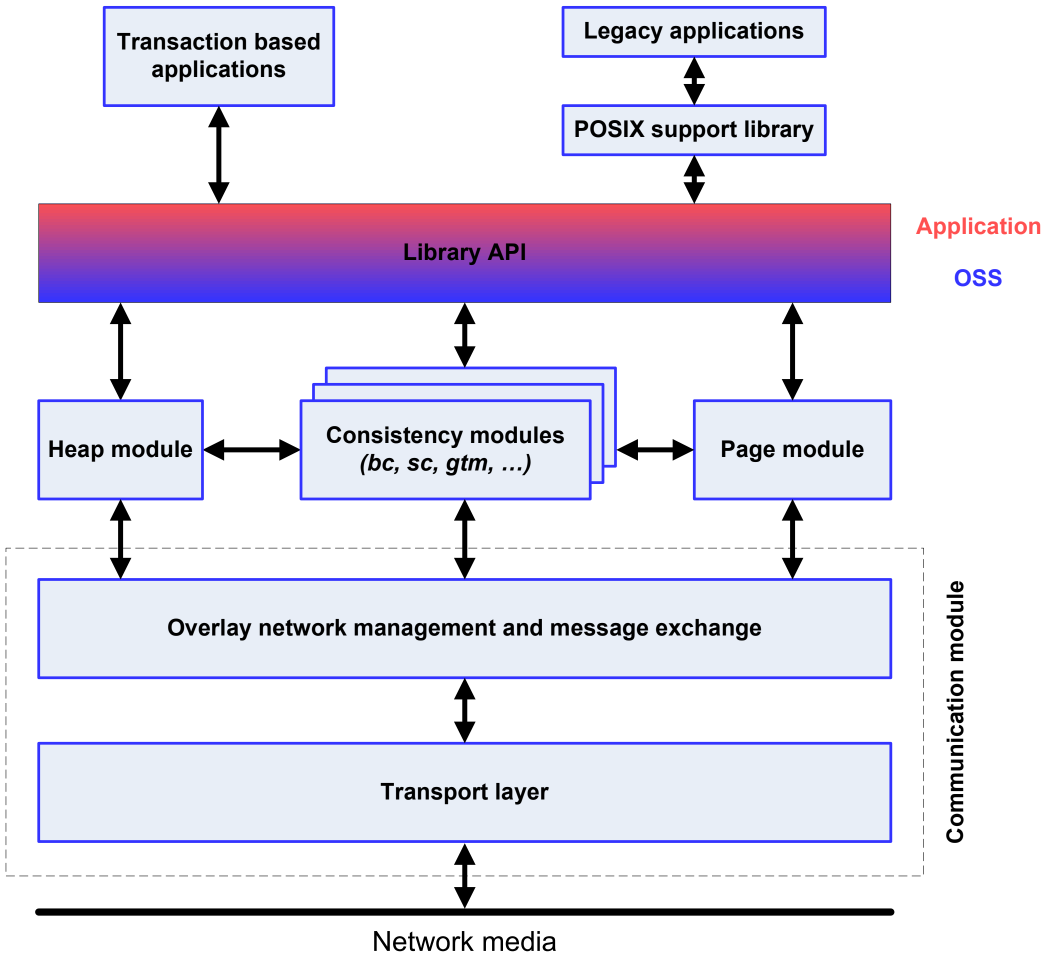

Figure 7: OSS architecture

Heap module. The heap module manages shared objects on behalf of the applications. It exports several functions for allocation and deallocation of memory analogous to the standard C library functions (e.g. *malloc* or *mmap*). Every allocated memory block is linked with a consistency model. A hierarchical scalable 64-Bit memory management is under development.

Consistency modules. A consistency module implements the rules of a consistency model that defines when write accesses become visible for other nodes. The current prototype offers *strong consistency*. This model guarantees that all processes sharing an object see write accesses immediately. Obviously, this model will not scale well and heavily depends on the programmer to carefully allocate data to avoid page thrashing. Nevertheless, there are legacy programs requiring such a model and are (fortunately) designed to minimize conflicts.

The *basic consistency* modul implements basic mechanisms, e.g. lock management, but does not define a concrete model. The goal is to provide basic operations that can be re-used to speed-up the development of future consistency models. Furthermore, this module routes memory access events to the appropriate consistency modules.

Currently, speculative transactions are being developed providing a sound basis for scalable and efficient data sharing.

POSIX support library. This is the module providing the interception facility to hook application calls to e.g. 'mmap' (to support memory-mapped files) and to other memory relevant functions, e.g. 'malloc' (for legacy POSIX applications). Our current implementation instructs the dynamic linker of the GNU/Linux system at application load time to additionally load the legacy support library for the object sharing service. The library exports a subset of the C library interface used by virtually all applications under GNU/Linux. Therefore, the application's memory allocation functions are linked against the legacy support library, which implements the functions by means of the object sharing service. The Linux kernel needs not be modified in order to support legacy applications.

3 Installation and Configuration

3.1 Checking out XtreemFS

In order to obtain XtreemFS, execute the command

```
%> svn checkout svn+ssh://<user>@scm.gforge.inria.fr/svn\
 > /xtreemos/WP3.4/branches/internal_release_2
```

`<user>` represents your INRIA SVN user name.

XtreemFS Directory Structure. The directory tree obtained from the checkout is structured as follows:

`AL`	contains the access layer (client) source code
`bin`	contains shell scripts needed to work with XtreemFS (e.g. start XtreemFS services, create volumes, mount an XtreemFS directory, . . .)
`config`	contains default configuration files for all XtreemFS services
`docs`	contains XtreemFS documentation files
`java`	contains the Java source code for all XtreemFS services
`OSS`	contains source code for the Object Sharing Service (OSS)

3.2 Requirements

For building and running XtreemFS, some third-party modules are required which are not included in the XtreemFS release:

- gmake 3.8.1
- gcc 4.1.2
- Java Development Kit 1.6
- Apache Ant 1.6.5
- FUSE 2.6
- libxml2-dev 2.6.26
- openssl-dev 0.9.8

Before building XtreemFS, make sure that `JAVA_HOME` and `ANT_HOME` are set. `JAVA_HOME` has to point to a JDK 1.6 installation, and `ANT_HOME` has to point to an Ant 1.6.5 installation.

3.3 Building XtreemFS

Go to the top level directory and execute:

```
%> make
```

3.4 Installation

Loading the FUSE Module. Before running XtreemFS, please make sure that the FUSE module has been added to the kernel. In order to ensure this, execute the following statement as root:

```
# modprobe fuse
```

Automatic XtreemFS Setup. The fastest way to completely set up XtreemFS on the local machine is to simply execute

```
%> bin/basicAL_tests
```

It will take about 15 seconds to set up a running system consisting of a Directory Service, an OSD and an MRC. The shell script creates a temporary directory in which all kinds of data and log output will be stored. A newly-created volume called `x1` will automatically be mounted to a subdirectory of the temporary XtreemFS directory; see console output for further details.

As long as the prompt (>) appears, the system is ready for use. In order to test the basic functionality of XtreemFS, you can enter:

```
> test
```

Note that any other command will shut down all XtreemFS services and unmount the XtreemFS directory. If you want to manually work on the mounted directory, you have to use a different console.

Manual XtreemFS setup. As an alternative to setting up XtreemFS in one step, the different services can also be set up manually. For this purpose, use `bin/xtreemfs_start`. Note that you have to set a Directory Service before setting up (at least one) MRC and (at least one) OSD. See `bin/xtreemfs_start --help` for usage details.

Example:

```
%> bin/xtreemfs_start ds -c config/dirconfig.properties
%> bin/xtreemfs_start mrc -c config/mrcconfig.properties
%> bin/xtreemfs_start osd -c config/osdconfig.properties
```

Once a Directory Service and at least one OSD and MRC are running, XtreemFS is operational.

XtreemFS relies on the concept of volumes. A volume can be mounted to a mount point in the local file system. In order to create a new volume, execute `bin/mkvol`. See Section 4.2.1 for usage details.

Example:

```
%> bin/mkvol http://localhost:32636/MyVolume
```

After having created a volume, you can mount it by executing `AL/src/xtreemfs`. See Section 4.1 for usage details.

Example:

```
%> bin/xtreemfs -o volume_url=http://localhost:32636/MyVolume,\
 > direct_io ~/xtreemfs-mounted
```

3.5 Configuration.

Sample configuration files are included in the distribution in the `config/` directory. Configuration files use a simple `key = value` format.

3.5.1 Directory Service Configuration File.

```
#an integer value between 0 (errors only) and 4 (full debug)
debug_level = 0

#TCP port the server listens on, default port is 32638
listen_port = 32638

# absolute path to the directory in which to store the database
database_dir = /var/run/xtreemfs/dir
```

3.5.2 MRC Configuration File

```
# an integer value between 0 (errors only) and 4 (full debug)
debug_level = 0

# TCP port the server listens on, default port is 32638
listen_port = 32636

# absolute path to the directory in which to store the database
database_dir = /var/run/xtreemfs/mrc

# absolute path to the file which is used as operations log
append_log = /var/run/xtreemfs/mrc/dblog

# interval in seconds between OSD checks
osd_check_interval = 300
```

```
# hostname or IP address of directory service to use
dir_service.host = localhost

# TCP port number of directory service to use
dir_service.port = 32638

# flag indicating whether POSIX access time stamps are set
# each time the files are read or written
no_atime = true

# interval between two local clock updates (time granularity, in ms).
# Should be set to 50.
local_clock_renewal = 50

# interval between two remote clock updates (in ms).
# Should be set to 60000.
remote_time_sync = 60000

# defines whether SSL handshakes between clients and the MRC
# are mandatory
# if use_ssl = false, no client authentication will take place
use_ssl = false

# file containing server credentials for SSL handshakes
ssl_server_creds = /tmp/server_creds

# file containing trusted certificates for SSL handshakes
ssl_trusted_certs = /tmp/trusted_certs
```

3.5.3 OSD Configuration File

```
# an integer value between 0 (errors only) and 4 (full debug)
debug_level = 0

# TCP port the server listens on, default port is 32638
listen_port = 32640

# absolute path to the directory in which to store objects
object_dir = /var/run/xtreemfs/osd/objects
```

```
# hostname or IP address of directory service to use
dir_service.host = localhost

# TCP port number of directory service to use
dir_service.port = 32638

# interval between two local clock updates (time granularity, in ms).
# Should be set to 50.
local_clock_renewal = 50

# interval between two remote clock updates (in ms).
# Should be set to 60000.
remote_time_sync = 60000
```

4 User Guide

In this section we give a brief outline on how to use the Xtreem File System. As stated earlier, the main use for applications is through the normal POSIX File API which is described in [4]. So we focus on some aspects that are not related to this API.

4.1 Mounting the File System

The file system itself is a user-space implementation based on FUSE. Such kind of file systems can be mounted with one call and several standard FUSE options. This call starts the user-space part of the file system. For the sake of brevity we will focus in this section on the relevant and additional options.

The XtreemFS will be mounted by the call

```
xtreemfs -o <xtreemfs opts>,direct_io,<fuse opts> <mount point>
```

The option `direct_io` is necessary for proper operation when multiple clients access the same file. Otherwise data corruption may occur.

Option	Effect
`-o volume_url=<volume url>`	Specify the URL of the volume that is to be mounted. This URL is composed of the MRC URL and the volume on that MRC, ie. the volume url `http://demo.mrc.tld/vol1` would specify the volume `vol1` on the MRC `demo.mrc.tld`.
`-o logfile=<log file>`	Write logs to specified file
`-o debug=<dbg lvl>`	Set debugging level
`-o logging=<enabled>`	Allow tracing of program execution.
`-o mem_trace=<enabled>`	Trace memory allocation and usage for debugging purposes.
`-o monitoring=<enabled>`	Allow monitoring of client. Option is available but has no effect right now.
`-o ssl_cert=<SSL cert file>`	Use the specified certificate to identify the client host. Option is available but has no effect right now.
`-o stack_size=<size>`	Set stack size for each thread of the client.

Table 1: Table with available mount options of the XtreemFS client

Because XtreemFS is a distributed file system, the filesystem can be mounted on several clients. These clients can access the same volume and the volume is uniquely identified by its volume url.

4.2 Tools

XtreemFS has the notion of volumes which is not covered by a POSIX like standard. So the additional tools are mainly for volume handling.

4.2.1 Volumes

There are three tools that deal with volumes: `mkvol`, `lsvol` and `rmvol`. Anyone familiar with UNIX command line interfaces can guess their purpose:

`mkvol` This tool is used to create a new volume on a given MRC. The syntax of this command is

```
mkvol [-a <access policy>] [-p <striping policy>] <vol url>
```

`<access policy>`	Specifies the policy how access to the volume is controlled: 1 Access is allowed for everyone (no access control at all) 2 Access is controlled in a POSIX like fashion.
`<striping policy>`	This parameter has the generic form `<name>, <size>, <width>`. For instance, the policy string `RAID0,32,1` specifies a RAID0 policy across one OSD with a stripe size of 32 kB. Right now, the only supported policy is RAID0.
`<vol url>`	This is the location of the volume to be created as an URL.

The `mkvol` can be executed on any client and must not necessarily be executed on the MRC itself. Permissions to create new volumes will be checked for the user who executes this command.

`lsvol` In order to list all the volumes that are available on a specific MRC this tool can be used. Its calling syntax is simple

```
lsvol <mrc url>
```

This will currently list the volumes names and their internal identification. Future versions will allow more fine grained information like available replica, available space etc.

`rmvol` If a volume is no longer used, this tool can be used to delete it from the MRC permanently. Afterwards the volume is no longer available and all data that might have existed on that volume before are lost. The calling syntax is similar to `lsvol`:

```
rmvol <volume url>
```

4.3 OSS Developers Guide

The prototype is built as a shared library which is linked to the applications and manages all operations regarding shared memory nearly transparent in the background. Applications are linked to the library by specifying the flag "`-loss`" to the linker.

4.3.1 Requirements

The following requirements must be fulfilled:

- x86 Processor (currently only IA-32 mode supported)
- Linux Kernel 2.6 or newer
- GNU C-Compiler 4.1.2 or newer
- libreadline5-devel 5.2 or newer
- libglibc-devel 2.6.1 or newer
- libglib2-devel 2.14.1 or newer

4.3.2 Building the Library

At the top level of the OSS directory run the build script by typing

```
%> make
```

After successful build, the library resides in the subdirectory `build`. Before using the shared library the developer has to register it to the system. In future versions the Makefile will allow to install and uninstall the library automatically. For the prototype we prefer to extend the library path without copying it to the system's library directory. This is done by typing

```
%> export LD_LIBRARY_PATH=<path to osslib>:$LD_LIBRARY_PATH
```

4.3.3 Application Development

When developing applications using OSS, the developer explicitly defines the memory regions for shared objects. A call to the function `hm_alloc()` returns a new shared memory region which is bound to a certain consistency model. Afterwards, applications on other nodes can access these memory regions. As a parameter to `hm_alloc()`, the developer can choose between the following two models: the basic (`CONSISTENCY_BC`) and the strict (`CONSISTENCY_SC`) consistency model. On every access to a shared object, OSS will check the consistency constraints.

Nevertheless the application can also manually deal with shared objects by using the low-level *request*, *update* and *invalidate* functions.

5 Conclusion and Future Work

In this deliverable, we have presented the architecture and the functionality available at month 18 for both XtreemFS and OSS. If we check the current prototype with the list of requirement in D4.2.3, we could mention that around 35% of the requirements have already been fullfiled.

Regarding the future work in XtreemFS, we plan to include replica management within the next few months. After replicas are implemented, the file system will be tuned to improve its performance and advanced functionality will start to be developed.

The prototype of the OSS component is able to share objects across multiple nodes in the grid using a strong consistency. The event-driven architecture is designed to support different consistency models. Obviously, strong consistency models will not scale well. One of the major goals of OSS is to implement a *Grid Transactional Memory* (GTM). This GTM uses speculative transactions each bundling a set of write operations thus reducing the synchronization frequency. Further transaction-based optimizations to hide network latency and to reduce the number of nodes involved during a transaction commit have been designed. The implementation of transactions is one of the next steps in the development roadmap for OSS. Furthermore, we will align OSS within the next months with the XtreemFS client to support memory-mapped files (see also 5.1.1).

5.1 Limitations of the Prototype

FUSE does not support mmap in connection with direct I/O. In order to get applications running on XtreemFS that rely on mmap, volumes currently have to be mounted without using the FUSE option `-o direct_io`. However, this might lead to inconsistencies if different clients concurrently work on the same file, as requests might be serviced from the local page cache.

5.2 XtreemFS and kDFS

Currently, within the XtreemOS project, two file systems are being developed with very different objectives. On the first hand, we have XtreemFS (presented here) that aims at giving a global Grid view of files from any node in the Grid. On the other hand, kDFS's objective is to build a cluster file system for all then nodes in a cluster running Linux-SSI and its main objective is performance.

In the future, we plan to allow files from kDFS to be accessed via XtreemFS and allow all performance optimization is the files ain kDFS are accessed form the nodes in the same cluster as the file system.

References

[1] FUSE Project Web Site. `http://fuse.sourceforge.net`.

[2] XtreemFS Consortium. D3.4.1: The XtreemOS File System - Requirements and Reference Architecture, 2006.

[3] Michael 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.

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

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

[6] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.