Project no. IST-033576

XtreemOS

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

XtreemFS and OSS Developer Guide
D3.4.5

Due date of deliverable: 30-MAY-2009
Actual submission date: 29-MAY-2009

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP3.4

Responsible institution: ZIB
Editor & and editor's address: Björn Kolbeck
Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Version 1.0 / Last edited by Jan Stender, Björn Kolbeck / 26-MAY-2009

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

Reviewers:
Mathijs den Burger (VUA), Roman Talyansky (SAP)

Tasks related to this deliverable:

Task No.	Task description	Partners involved[°]
T3.4.1	XtreemFS File Access	CNR[*], BSC, ZIB
T3.4.2	XtreemFS Metadata Server	ZIB[*]
T3.4.4	XtreemFS Pattern-Aware Data Access	BSC[*], CNR
T3.4.5	Object Sharing Service	UDUS[*]
T3.4.6	XtreemFS client	NEC[*], UDUS

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

Contents

Executive Summary

In this guide we describe the internals of the XtreemFS file system and the Object Sharing Service (OSS). This document is intended for developers as a reference documentation of the protocols and architectures as well as an introduction for those new to XtreemFS or OSS.

The section on XtreemFS describes the overall architecture, goals and features. Individual components are presented and their internal design is explained, the XtreemFS protocol is described and documented. Finally, the testing tools and methodology is described.

The following chapter describes the internals of OSS and its architecture. OSS' modules are described and explained in detail, and an overview of OSS' network protocol is given. The section on *internal interfaces* documents the inter-module function calls. The last section describes step by step how developers can implement their own consistency models.

Chapter 1

The XtreemFS Developer Guide

1.1 Introduction

XtreemFS [5] is an object-based [4, 10] file system designed for Grid environments. It is the main distributed file system in the XtreemOS operating system, which relies on XtreemFS for replicated and low-latency file storage between Grid machines.

From a user's perspective, XtreemFS offers a global view on files. Files and directory trees are arranged into volumes. A volume can be mounted at any Grid node where a sufficiently authorized job can access and modify files on the volume. Applications access directories and files on XtreemFS volumes through normal POSIX interfaces (`open`, `read`, etc.) and thus do not require re-compilation in order to work with XtreemFS. This stands in marked contrast with earlier Grid file systems such as GFarm [13], which often forced users to rewrite parts of their applications in order to access files across the Grid via special non-POSIX APIs or to adapt to a non-POSIX file system semantics.

From an administrator's perspective, an XtreemFS installation consists of file system clients running on each user's machine and network-based services for storing and retrieving file metadata and data. The former services are known as Metadata and Replica Services (MRCs), while the latter are called Object Storage Services (OSDs). These services are complemented by the Replica Management Service (RMS), which is responsible for creating replicas on demand in response to changing user access patterns as well as eliminating redundant replicas; and the Object Sharing Service (OSS), which provides transaction-based sharing of volatile memory objects and supports memory-mapped files for XtreemFS.

This deliverable is intended to serve as a developer guide. Its focus is on the current design and implementation of the XtreemFS client and servers, network protocols used between clients and servers, and test suites for XtreemFS.

1.1.1 Document Structure

The report is structured as follows. Sections 1.2.2, 1.2, 1.2.1, 1.2.3 describe the XtreemFS directory, metadata, and object store services. In section 1.3 we introduce the new XtreemFS client, which was designed from the ground up to take advantage of the new binary protocol and to remedy numerous performance and scalability problems in the previous revision of the client. Section 1.4 concerns the XtreemFS Replica Management Service. We conclude with a discussion of recent testing efforts in section 1.5. Finally, section 1.6 documents the new binary client-server and server-server protocol, a more efficient and easily-maintained replacement for the text-based protocol of previous releases.

1.2 XtreemFS Servers

All XtreemFS servers (DIR, MRC and OSD) are written in Java and employ an event-based staged design. In addition to the common architecture, they also share the basic libraries like RPC server and client or memory management.

In our design, a stage has one or more threads to do the work. Usually, a stage is used for processing which is blocking (e.g. I/O operations) or consumes larger amounts of CPU time (e.g. checking signatures). Each stage receives requests (events), processes them asynchronously and passes the result to a callback. Operations are the "glue" between the stages. For each client request or internal event, there is an Operation class which implements the logic of the call.

All servers also use a custom method of memory management. To avoid excessive data copying to and from the Java VM, we use direct ByteBuffers which represent raw memory on the heap. These direct ByteBuffers are not managed by the Java garbage collector and excessive allocation and freeing of them causes severe performance problems. To overcome this problem and to reduce overall memory consumption, we use a concurrent BufferPool to allocate ByteBuffers. In addition, we use a wrapper class (called ReusableBuffer) which implements reference counting. It also ensures that ReusableBuffers which have been returned to the pool cannot be used anymore.

The ReusableBuffers must be freed (i.e. returned to the BufferPool) after using them. Failing to do so will cause an error message to be printed on finalization which should help to detect memory leaks. Setting `BufferPool.recordStackTraces` to `true` will add a full stack trace of the allocation to the error message which is useful to locate memory leaks. This option is only for debugging and should not be used for production due to the performance penalty of recording stack traces on each allocation.

The ONC RPC server and client are used by all three servers as well. Both are implemented using Java's non-blocking network IO NIO and can be used with or without SSL.

The DIR and MRC also use an external key-value store called BabuDB (see http://babudb.googlecode.com) to persistently store information. How it is used and how the data is stored in BabuDB is described in the DIR and MRC sections, respectivly.

1.2.1 DIR - Directory Service

The Directory Service (DIR) is the central service registry of XtreemFS. All services register and regularly update their registration at the DIR. In addition, it keeps all address mappings which the services need to translate UUIDs to hostname and port. The directory service is also used by the MRCs and OSDs to synchronize their clocks.

Currently, the directory service is a single instance. In the future, this service will be replicated and and divided into a hierarchy of DIR services.

Persistent data is stored in BabuDB[1], a non-transactional key-value-store. The service and address mapping records are stored in their XDR representation. This means that the DIR database must be deleted or converted if data structures change.

1.2.2 MRC - Metadata and Replica Catalog

The Metadata and Replica Catalog (MRC) is responsible for the management of all metadata in an XtreemFS installation. Core tasks of the MRC are the management of volumes and directory trees, storage of file and directory metadata and access control enforcement.

[1] http://babudb.googlecode.com

Architecture

Aside from the ONC RPC server that listens for incoming client requests, the MRC architecture comprises two core components: the processing stage and the database backend. Each request received by the ONC RPC server is parsed and forwarded to the processing stage, which executes the respective file system logic. Any data that needs to be retrieved or modified during file system logic execution is stored in a database backend.

Processing Stage

The MRC interface consists of multiple so-called *operations*. Each operation relates to an implementation of the logic for the execution of a certain request. There are operations e.g. for opening files, reading directory content, creating volumes, and the like. Operations are named and parametrized similar to their corresponding POSIX calls. To circumvent locking issues in the underlying database, operation execution is serialized for each volume, i.e. no more than one thread may execute operations on a certain volume at the same time.

All operations have a similar composition. First, authorization checks are performed, in order to find out whether the user on behalf of whom the request was sent has sufficient permissions to execute the operation. In case of a positive result, the operation logic is executed. Operation logic execution may involve an arbitrary number of accesses to the underlying database backend. A `readdir` request will e.g. result in a database lookup for the content of a directory, a `setxattr` request will cause an extended attribute of a file to be added in the database.

A detailed description of the interface to the MRC including all operations is given in Sec. 1.6.4.

Database Backend

The database backend is accessed at record level, i.e. at a granularity of single key-value pairs. The creation of a new file could e.g. require several record modifications, since a file metadata object needs to be inserted in the database, a link to the parent directory needs to be established, time stamps of parent directories need to be updated, and so forth. Multiple such records can be combined in an insert group, which causes the insertion of a new set of records to take place in a single step, i.e. atomically.

The database backend implementation is decoupled from the remaining MRC code via an interface, which gives developers the opportunity to implement their own database bindings. The currently used implementation is based on BabuDB. A BabuDB instance may comprise multiple databases, which may in turn comprise multiple indices. Databases are identified by name strings, whereas indices of a database are serially numbered. Lookups and insertions are directed to single indices of a database; besides normal value lookups for keys, BabuDB supports queries for key prefixes, which provides the basis for an efficient lookup of consecutive key-value pairs.

A range of different indices are used to store XtreemFS metadata. How XtreemFS metadata is mapped to BabuDB indices will be described in the following.

Metadata for Volume Management Volume metadata is stored in a database named `V`. It is arranged in the following indices:

#	**Name**	**Description**
0	Volume ID Index	Maps a volume UUID to a volume metadata entity.
1	Volume Name Index	Maps a volume name to a volume ID.

Volume ID Index

key

Element	**# Bytes**	**Description**
`volumeID`	`var`	`the volume ID string`

value

Element	**# Bytes**	**Description**
`fileAccPolID`	2	`the file access policy ID for the volume`
`osdPolID`	2	`the OSD selection policy ID for the volume`
`offsVolName`	2	`the offset position of the 'volName' element, relative to the offset of the buffer's first byte`
`offsPolArgs`	2	`the offset position of the 'osdPolArgs' element, relative to the offset of the buffer's first byte`
`volID`	`var`	`the volume's UUID string`
`volName`	`var`	`the volume's name string`
`osdPolArgs`	`var`	`the volume's OSD selection policy argument string`

Volume Name Index

key

Element	# Bytes	Description
`volName`	`var`	`the volume name string`

value

Element	# Bytes	Description
`volId`	`var`	`the volume's UUID string`

Metadata for Files and Directories File system metadata of a volume is stored in a BabuDB database with a name equal to the volume's UUID. Various indices are used to manage metadata pertaining to files and directories, which will be described in the following tables. Indices have been designed with the following goals in mind:

- Lookups performed by frequently invoked operations should be as fast as possible, like metadata lookups for a given directory path.
- Database records that are frequently updated should include as little unchanged data as possible.
- Frequently performed database updates should be fast, i.e. involve as little index insertions as possible.
- Indices should contain as little redundancy as possible, in order to minimize database size and memory footprint.

With the aforementioned goals in mind, we decided to have a primary index for the primary metadata of files, which maps a key essentially consisting of a parent directory ID and a file name hash to a value that contains a metadata record. This way, BabuDB prefix lookups for parent directory IDs can be used to efficiently retrieve contents of a directory, while normal lookups can be used to retrieve metadata for a single file. Since POSIX requires support for hard links, i.e. different directory entries pointing to the same metadata, and some operations require a retrieval of file metadata by means of file IDs, we decided to maintain a secondary index that allows a retrieval of metadata by means of a file ID. Other indices are used to store extended attributes and access control lists.

#	Name	Description
0	File Index	Stores primary metadata for a directory entry. Values in the index may be of different kinds: • *frequently changed metadata* - encapsulates all metadata that is frequently modified, such as time stamps or file sizes • *rarely changed metadata* - encapsulates all metadata that is infrequently changed, such as file names, access modes, or ownership of a file • *replica location metadata* - encapsulates X-Location lists of files • *hard link targets* - in case additional hard links exist for one file, the value is a hard link target, i.e. a key in the File ID index. Lookups to file metadata will then be performed in two steps: first, a lookup in the File Index will be performed, in order to retrieve the hard link target; then, metadata will be looked up in the File ID Index.
1	XAttr Index	Contains any extended attributes of files and directories. This includes Softlink targets and default striping policies, since they are mapped to extended attributes.
2	ACL Index	Contains access control list entries of all files.
3	File ID Index	The file ID index is used to retrieve file metadata by means of its ID. If no hard links have been created to a file, the file ID will be mapped to a key in the file index, for which the metadata will have to be retrieved with a second lookup. Such a mapping is necessary for some operations that are based on file IDs instead of path names. If hard links have been created, the file ID will be directly mapped to the three different types of primary file metadata (i.e. rarely and frequently changed metadata, as well as replica locations). In this case, the file's entries in the file index point to the corresponding prefix key in the file ID index.
4	Last ID Index	Contains a single key-value pair that maps a static key to the last file ID that has been assigned to a file. The index ensures that new file IDs are assigned to newly created files or directories.

File Index

key

Element	# Bytes	Description
parentID	8	file ID of the parent directory
fileNameHash	4	a hash value of the file name
type	1	type of metadata (0=frequently changed metadata, 1=rarely changed metadata, 2=replica locations, 3=hard link targets)
collCount	2	counter that is incremented with each collision of file name hashes - will be omitted unless multiple file names in the directory have the same hash values

value, type = 0

Element	# Bytes	Description
fcMetadata	20/12/8	frequently changed metadata associated with the file (see 'fcMetadata' definition), 20 bytes for files & symlinks, 12 for directories, 8 for hard link targets

value, type = 1

Element	# Bytes	Description
rcMetadata	var	rarely changed metadata associated with the file (see 'rcMetadata' definition)

value, type = 2

Element	# Bytes	Description
xLocList	var/8	the replica location list associated with the file (see 'xLocList' definition), variable length for files & directories, 8 for hard link targets

XAttr Index

key

Element	# Bytes	Description
`fileID`	8	`the ID of the file to which the extended attribute has been assigned`
`ownerHash`	4	`a hash value of the attribute’s owner`
`attrNameHash`	4	`a hash value of the attribute name`
`collCount`	2	`counter that is incremented with each collision of (ownerHash, attrNameHash) pairs - will be omitted unless different attributes are hashed to the same such pair`

value

Element	# Bytes	Description
`offsKey`	2	`the offset position of the ’attrKey’ element, relative to the offset of the buffer’s first byte`
`offsValue`	2	`the offset position of the ’attrValue’ element, relative to the offset of the buffer’s first byte`
`attrOwner`	`var`	`the user ID of the attribute’s owner`
`attrKey`	`var`	`the attribute key`
`attrValue`	`var`	`the attribute value`

ACL Index

key

Element	# Bytes	Description
`fileID`	8	`the ID of the file to which the extended attribute has been assigned`
`entityName`	`var`	`the name of the entity associated with the ACL entry`

value

Element	# Bytes	Description
`rights`	2	`the access rights for the entity`

File ID Index

key

Element	# Bytes	Description
fileID	8	the ID of the file
type	1	type of metadata (0=frequently changed metadata, 1=rarely changed metadata, 2=replica locations, 3=hard link targets)

value, type = 0

Element	# Bytes	Description
fcMetadata	20/12	frequently changed metadata associated with the file (see 'fcMetadata' definition), 20 bytes for files & symlinks, 12 for directories

value, type = 1

Element	# Bytes	Description
rcMetadata	var	rarely changed metadata associated with the file (see 'rcMetadata' definition)

value, type = 2

Element	# Bytes	Description
xLocList	var	the replica location list associated with the file (see 'xLocList' definition)

value, type = 3

Element	# Bytes	Description
parentID	8	the ID of the parent directory in which the metadata for the file is stored
fileName	var	the file name in the parent directory

Last ID Index

key

Element	# Bytes	Description
`'*'`	1	`the only key in the table`

value

Element	# Bytes	Description
`lastFileID`	8	`the last ID that has been previously assigned to a file or directory`

Data types referenced in the index descriptions above are listed in the following:

frequentlyChangedMetadata

files

Element	# Bytes	Description
`atime`	4	`file access time stamp in seconds since 1970`
`ctime`	4	`file metadata change time stamp in seconds since 1970`
`mtime`	4	`file content modification time stamp in seconds since 1970`
`size`	8	`file size in bytes`

directories

Element	# Bytes	Description
`atime`	4	`file access time stamp in seconds since 1970`
`ctime`	4	`file metadata change time stamp in seconds since 1970`
`mtime`	4	`file content modification time stamp in seconds since 1970`

rarelyChangedMetadata

files

Element	# Bytes	Description
type	1	the type of the entry (0=file, 1=directory)
id	8	file ID
mode	4	POSIX access mode
linkCount	2	number of hard links to the file
w32attrs	8	Win32-specific attributes
epoch	4	current truncate epoch
issEpoch	4	last truncate epoch that has been issued
readOnly	1	a flag indicating whether the file is suitable for read-only replication
offsOwner	2	offset position of the 'owner' element, relative to the offset of the buffer's first byte
offsGroup	2	offset position of the 'group' element, relative to the offset of the buffer's first byte
fileName	var	name of the file
owner	var	user ID of the file's owner
group	var	group ID of the file's owner

directories

Element	# Bytes	Description
type	1	the type of the entry (0=file, 1=directory)
id	8	file ID
mode	4	POSIX access mode
linkCount	2	number of hard links to the file
w32attrs	8	Win32-specific attributes
offsOwner	2	offset position of the 'owner' element, relative to the offset of the buffer's first byte
offsGroup	2	offset position of the 'group' element, relative to the offset of the buffer's first byte
fileName	var	name of the directory
owner	var	user ID of the directory's owner
group	var	group ID of the directory's owner

xLocList

xLocList

Element	# Bytes	Description
version	4	version of the X-Locations list
replCount	4	number of replicas in the list
offsUpdPol	4	offset position of the 'updPol' element, relative to the offset of the buffer's first byte
offs1 ... offsN	4 ... 4	offset positions for all replicas, relative to the offset of the buffer's first byte
xLoc1 ... xLocN	var ... var	replicas in the X-Locations list
updPol	var	update policy string that describes how replica updates will be propagated

xLoc

Element	# Bytes	Description
offsOsdList	2	offset position of the 'osdList' element, relative to the offset of the buffer's first byte
strPol	var	striping policy associated with the replica
osdList	var	list of all OSDs for the replica

strPol

Element	# Bytes	Description
stripeSize	4	size of a single stripe (=object) in kB
width	4	number of OSDs for the striping
pattern	var	string containing the striping pattern

osdList

Element	# Bytes	Description
osdCount	2	number of OSDs in the list
offsOSD1 ... offsOSDn	2 ... 2	offset positions for all OSD UUIDs, relative to the offset of the buffer's first byte
osdUUID1 ... osdUUIDn	var ... var	UUIDs of all OSDs in the list

1.2.3 OSD - Object Storage Device

The Object Storage Device (OSD) is responsible for reading and writing objects from/to disk. In addition, it also implements the replication (which is transparent to clients). In this section, we first describe the stages and components of the OSD. We then describe the interaction between OSDs for striped files and for read-only replication.

- **`StorageStage` and `StorageThread`**
  The StorageStage distributes the request onto a pool of StorageThreads. The allocation of requests is based on the fileID to ensure that all requests for a single file are handled by the same thread. This is necessary to avoid sharing of file metadata across multiple threads.

  The StorageThread implements the actual file I/O to access objects on disk. It uses a StorageLayout which is responsible for arranging the objects into files and directories in the underlying file system.

- `PreprocStage`
  Analyzes the incoming RPC requests and starts the matching Operation for the requests. It also parses the request arguments (RPC message) based on the Operation. In addition, it parses and validates the signed capability and ensures that the client is authorized to execute the operation. To enhance performance, the PreprocStage keeps a cache of validated Capabilities and XLocation lists. The PreprocStage also keeps a list of open files which is updated whenever a file (i.e. a file's object) is accessed. The list is regularly checked (approx. every mminute) for last access times and files which have timed out will be closed. This close event is sent to the other stages, to allow them to clean their caches. As POSIX requires that a file which is deleted while still opened can still be read or written to, the close event is also used to finally remove data of deleted files.

- `DeletionStage`
  This stage is removing the objects on disk for files which have been deleted. This is done directly when the unlink RPC is received or when the file is closed (see PreprocStage).

- `VivaldiStage`
  Implements the OSD's Vivaldi component and regularly updates its coordinates. See Sec. 1.4 for details on Vivaldi.

- `ReplicationStage`
  Fetches data from remote OSDs for files which are replicated. For more details about the read-only replication see Sec. 1.2.3.

- `CleanupThread`
  This is not a regular stage, but a background task to scan for orphaned files. If a file is deleted on the MRC but the client fails to delete the file at the OSD, we get so called zombies. To remove them, the OSD has to scan its file system from time to time and check the files at the MRC. How often and when these cleanup operations should be executed depends on the usage pattern of the system (e.g. client's often disconnecting during operations).

Striping

XtreemFS allows files to be striped (distributed) over several OSDs. To ensure correct POSIX semantics in this distributed case, OSDs need to exchange additional information on some write and read operations. We use additional UDP datagrams on write to disseminate file size update hints among OSDs. See [12] for a detailed description of the algorithms used in XtreemFS.

Read-only replication

The read-only replication allows users to replicate their immutable files with very low overhead. Users can set a file to read-only which means that it cannot be modified anymore. This allows users to add replicas on other OSDs which can either be a "full" or a "lazy" replica. For a "full" replica the OSD will automatically fetch all objects of that file. For a "lazy" replica the OSD only fetches the objects when a client tries to read them. Additional prefetching for "lazy" replicas will be added.

1.3 Client

The XtreemFS client connects applications to XtreemFS and acts as a gateway to the XtreemFS directory (DIR), metadata (MRC), and object store (OSD) servers. From a user's perspective, the client consists of a number of binary programs that reside on the user's machine. These programs and their functions are summarized in table 1.1.

Command line tool	Function
`xtfs_lsvol`	list volumes on an XtreemFS MRC server
`xtfs_mkvol`	create a volume on an XtreemFS MRC server
`xtfs_mount`	mount an XtreemFS volume
`xtfs_rmvol`	delete a volume on an XtreemFS MRC server
`xtfs_send`	send arbitrary RPCs to an XtreemFS server
`xtfs_stat`	print statistics on an XtreemFS file or directory

Table 1.1: XtreemFS client command line tools

1.3.1 Architecture

The client is structured as a network of message-processing *stages* connected by queues. These stages are similar to those in the XtreemFS servers, and are designed with the same intent: to increase concurrency while avoiding data races (see [14] and previous deliverables for an explanation of stages). Unlike the XtreemFS servers, which is implemented in Java and uses a custom-built set of classes for managing stages, the client is implemented in C++ and relies on a third party platform, Yield [2] for much of its low-level functionality, including concurrency control in the form of stages as well as platform-specific primitives such as files and sockets.

The stage architecture of the XtreemFS client is depicted in figure 1.1. Note that this particular configuration of stages is specific to `xtfs_mount`, which consists of a set of FUSE entry points and proxy stages for the various XtreemFS servers with which `xtfs_mount` communicates: the directory server (`DIRProxy`), the metadata server (`MRCProxy`), and one or more object stores (`OSDProxy`).

FUSE

`xtfs_mount` provides a file system interface to applications via FUSE [3], a library for implementing file systems in userspace. Applications make POSIX system calls such as `open` and `read` into the operating system kernel. A FUSE kernel module translates these calls into messages (i.e. Remote Procedure Calls), which are then passed to a FUSE file system via a pipe. The file system runs as a daemon in an infinite loop, reading and processing messages from the pipe and sending responses back into the kernel. The userspace part of the FUSE library handles most of the nitty gritty details

[2] http://yield.googlecode.com/
[3] http://fuse.sourceforge.net/

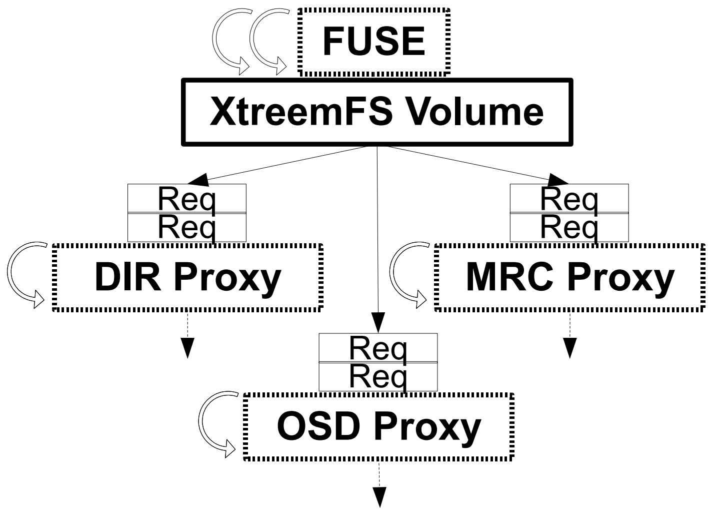


Figure 1.1: Client stages

of kernel-userspace communication, so that the file system implementator can concentrate on implementing file system logic. This is typically done by implementing a set of FUSE callbacks, each of which corresponds to a POSIX system call (and thus a message on the FUSE pipe as well). The FUSE library translates messages to calls on the callbacks supplied by the file system developer and translates return values from the callbacks back into messages for the FUSE kernel module. These FUSE callbacks are the main entry points into `xtfs_mount`.

From a FUSE callback such as `mkdir` the client makes a series of requests (messages) through the various stages shown in figure 1.1. The primary stages in the client are proxies for the servers the `xtfs_mount` instance is connected to: typically a single DIR server and a single MRC server and multiple OSD servers. In the case of `mkdir` the client would send a request to the `MRCProxy` to create the specified directory. Because the FUSE callbacks are synchronous the initial request from a callback to a stage must be synchronous, i.e. the sender must wait for the response before doing any further processing. However, this does not mean that the whole system is synchronous: only the FUSE callback blocks synchronously on a request, allowing the stages in the client to communicate asynchronously (and thus improve performance). Furthermore, since the FUSE callbacks may be multithreaded and reentrant a stage (such as e.g. the `MRCProxy`) can process requests from multiple FUSE callbacks simultaneously. In other words, the concurrency of the client is not limited by the FUSE front end and the number of threads processing FUSE messages.

1.3.2 Implementation

The XtreemFS client is implemented entirely in C++. Aside from the essential components listed above (FUSE callbacks, server proxies) `xtfs_mount` consists of a few support classes such as `Path` (which wraps XtreemFS `volumefile` global paths) and XtreemOS integration code such as a pluggable module for retrieving user credentials from an XtreemOS AMS server. Most of this code is shared between the XtreemFS command line tools listed in table 1.1. As mentioned previously, the XtreemFS client also relies heavily on Yield for many low-level classes, such as platform-specific file paths and sockets. The ONC-RPC [11] protocol implementation used to communicate with the XtreemFS servers is also a part of Yield.

Generated interfaces

The synchronous request-response messages exchanges between FUSE callbacks and stages such as the `MRCProxy` are hidden underneath a function call interface. The latter is generated from the same IDL interfaces used by the server. When one of the interface operations is called synchronously on the `MRCProxy` a request is created and filled with the function parameters; the request is sent to the `MRCProxy` stage, where it is processed asynchronously; and the caller blocks waiting for the response, which, when it is received, is unpacked and returned as a normal function return value. The extra level of abstraction allows the FUSE callback interface to be fully agnostic of message sending and receiving, and simply treat the MRCProxy as if it were making synchronous remote procedure calls. The FUSE callback for `mkdir` is shown in figure 1.2.

```
bool Volume::mkdir( const YIELD::Path& path, mode_t mode )
{
  mrc_proxy.mkdir( Path( this->name, path ), mode );
  return true;
}
```

Figure 1.2: FUSE callback for mkdir

Lines of code

With much of its low-level functionality in Yield and other libraries the code base for the XtreemFS client is quite minimal, with approximately 2800 lines of hand-written C++ and 4800 lines of C++ automatically generated from the XtreemFS IDL interfaces.

1.4 RMS - Replica Management Service

One of the most important mechanisms in XtreemFS is the possibility to have several replicas of a file distributed over the Grid. This feature affords data-intensive applications achieving better performance as long as: there is no single access point for the data and mechanisms for parallel access can be exploited. Besides, replication also provides reliability and availability to the filesystem, which is of vital importance for a distributed environment.

However, the usage of Grid resources such as network (where data is transfered across) or storage (where data is stored) are finite, shared, and non-free. Furthermore, the synchronization of the replicas of any given file involves additional overheads, so that mechanisms that keep the tradeoff between the benefits and the extra costs are needed.

For aiming at all of these purposes, we are working on the implementation of the Replica Management Service. This service concerns about: selecting the best replicas for the applications, creating and deleting replicas automatically taking account of how and from where they are accessed and evaluating the maximum number of replicas of any given file.

1.4.1 Choosing the best replica

When a given client (or an OSD) has to access a file, the question is: which replica should it access? It should be able to detect which replica will provide better performance. The idea to solve this problem is to build a virtual 2D space and locate all replicas, OSDs, and clients in it. The distance between two different objects (i.e replica, OSD, or client) is an indicator of the distance (performance wise) of these two objects. Once a client wants to access a file, it just needs to compute the euclidian distance between itself and all replicas and choose the closer one.

Vivaldi Algorithm

Vivaldi is a light-weight algorithm developed by MIT [3] that allows assigning a position in a coordinate space to every node in a network, so the distance between the coordinates of two nodes predicts the real communication latency between them.

In order to generate a valid coordinate system, it is necessary to determine which space will be used and which formula will be used to calculate the distance between two given points. In our case, it is been proved that implementing a 2-D space, where the Euclidean distance between two coordinates accurately predicts the latency between their corresponding nodes, generates valid results with a really small error probability.

For the algorithm to work correctly, it is also necessary that the nodes of the system keep contacting themselves randomly and indefinitely to re-adjust their position, so any possible change in the network may be reflected. In each re-adjustment, a node contacts a different neighbor, gets its coordinates and modifies its own coordinates, so eventually the Euclidean distance is as similar as possible to the measured round trip time.

On the other hand, once a group of nodes have established a valid coordinate system, it is necessary to use some mechanism that helps to reduce the impact of introducing new nodes, so we avoid them to alter the already stabilized system. That is why Vivaldi keeps in every node, besides the coordinates, a local error that informs about how sure a node is about its position. This way, a node with a steadier position will have a smaller local error and will influence more the rest of nodes when they contact it to readjust their position (figure 1.3).

Once the system is adjusted, any node of the network can determine which nodes are the closest ones with a really simple calculation, in a very short period of time and without generating extra traffic.

Some already developed implementations of Vivaldi can be found in p2psim and in Chord. You might also be interested in Ledlie et al.'s work [9].

Vivaldi in XtreemFS

As we have different kinds of nodes in our architecture, not all of them work in the same way to integrate Vivaldi. While the clients usually execute during shorter periods of time, the servers are up and running , so the idea is to let the OSDs (at this moment they are the only servers that implement Vivaldi)

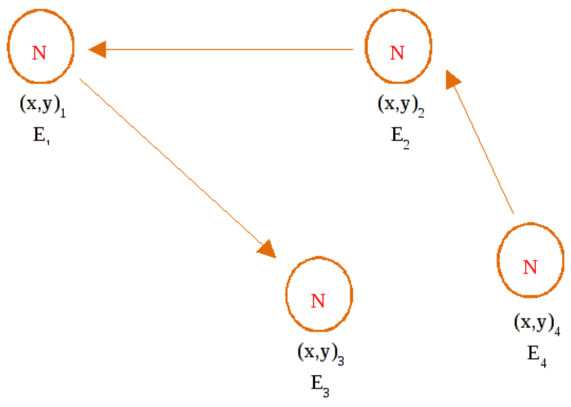


Figure 1.3: Nodes keep recalculating their position

establish a permanent coordinate system where a client can move through, to find its position.

Vivaldi in the OSDs

An OSD has an independent stage responsible of managing Vivaldi on its side and of providing to the rest of components a couple of valid coordinates that define the position of the node in the current coordinate system.

The stage keeps running indefinitely and periodically contacts a different OSD to ask it for its coordinates and its local error. With that data and the coordinates of the own OSD is possible to compute the Euclidean distance and to compare it with the real RTT measured against the contacted node.

The frequency an OSD readjusts its position is defined by the parameters MIN_TIMEOUT_RECALCULATE and MAX_TIMEOUT_RECALCULATE. Just after performing a readjustment, the stage typically calculates a random number included in the interval of time defined by those two parameters and sleeps during that number of seconds until the next iteration. This way we try to avoid generating traffic peaks where all the nodes send a request at the same time and to distribute the net use in time.

Larger periods will reduce the overhead in the network but will make the nodes to adjust more slowly to the possible changes in the environment, while smaller ones will require more traffic but will produce a more reactive system.

In each iteration, the introduced stage chooses a node to contact to from a list of available OSDs, which is filled with the information contained in the Directory Service. This list must be updated somehow so the stage can always notice a node going offline.

Vivaldi in clients

In our system, the clients usually execute during a much shorter period of time, so they have to be able to determine their position faster. This can be done because they do not influence the rest of the nodes and they just take some needed info from the already placed OSDs to locate themselves.

In Vivaldi, each node is responsible for its own coordinates and typically has to recalculate them at least a small number of times before they represent the real position in the coordinate system. Even if the set of OSDs is"adjusted", a client will need to recalculate its position (against one single node each time) several times before having an accurate approximation of its real location. Vivaldi requires that the nodes of the net generate traffic and communicate among themselves.

As in the case of the OSDs, a client also has the parameters MIN_TIMEOUT_-RECALCULATE and MAX_TIMEOUT_RECALCULATE that allow defining the recalculation period. Although the analogue parameters in the OSDs have the same names, they are different parameters and therefore they all must be defined in different files.

Finally, it is important to emphasize that after the first boot of the client, it keeps its coordinates and preserves them among executions, so it remains well located though it mounts and unmounts a lot of different volumes or opens and closes a lot of files. The coordinates are not reinitialized until the client node is rebooted.

Replica Selection with Vivaldi

Until this point we have introduced a mechanism able of establishing a coordinate system where all the nodes of a network have a pair of coordinates that allows them predicting the round trip time to the rest of neighbors. Now it is time to analyze how to take advantage of that information and to describe the current applications of Vivaldi in XtreemFS.

Sometimes during the execution of certain operations, the client has to choose which replica access, among several replicas stored in different nodes of the system. The ideal solution proposes to select always the replica that is stored

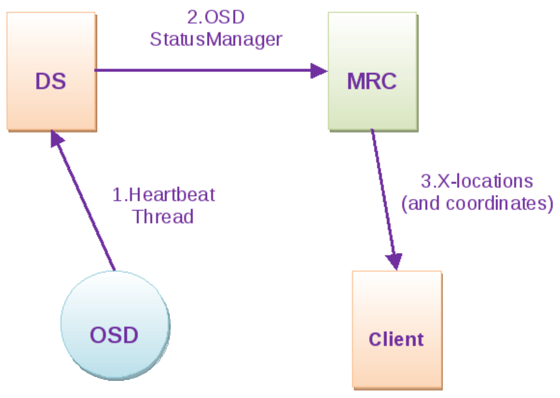


Figure 1.4: Collecting the coordinates

in the closest node, so the accesses can be made within the minimum time. But the problem is that most of the times measuring the RTT against every OSD, for each selection, is not computationally feasible.

Using the coordinates provided by Vivaldi, a client can calculate which replica is the closest one with a practically insignificant delay. At this point the only remaining problem seems to be how to gather all the coordinates so they can be available in the exact moment of the replica selection.

As mentioned earlier, in Vivaldi the coordinates are managed independently by the node they belong to and remain distributed among the different nodes of the network. In order to let the client take advantage of them, it is necessary to collect them in the MRC, so they can be included in every list of x-locations.

In figure 1.4 we show the following process:

1. HeartbeatThread is a component of the OSDs that periodically registers the OSD in the Directory Service. The information that is uploaded each time is determined by the function getServiceData, which is defined in the main constructor of the class OSDRequestDispatcher.

2. OSDStatusManager is a component of the MRC that regularly queries the Directory Service for available OSDs.

3. During an open operation, the MRC sends to a client a list of x-locations, so it can locate the different replicas associated to a cer-

tain file. The x-locations include the corresponding coordinates, so the client can use them to predict the closest replica.

1.4.2 Replica creation

Another important issue regarding replicas is to decide when and where to create a replica. For this functionality we have three different mechanisms. The first one is an explicit request from the user. In this scenario, the RMS will not take any action. The second one is a reactive replica creation. The system will detect that a replica is needed at a given location and will start a replica creation. Finally, in the third case, the system will predict the usage of a file in a location where no replicas are nearby and thus will try to create the replica before it is used. We call to this third mechanism proactive replica creation.

In both cases reactive and proactive, we plan to provide a mechanism able to study the access pattern of files and use it to decide if only a part of the file needs to be replicated (partial replication). This partial replicas will speedup the process of replication because only part of the data will need to be copied to the new location. Nevertheless if we miss-predict the parts of the replica that will be used, we will always be able to populate the missing parts on-demand (done directly by the OSDs).

Reactive replica creation with Vivaldi

In this scenario it is detected when replicas are currently needed in other parts of the Grid. Then, using the distance mechanisms we just described in Section 1.4.1, we will detect if clients request a replica from large distances. So in this case Vivaldi could be used to decide a better location for a replica and create it.

Proactive replica creation with Oraculo

We have implemented a service called Oraculo that carries out data mining on multi-order context models to analyze the file-access patterns and to compute future accesses. For the implementation of such multi-order context models, Oraculo keeps a trie (or prefix tree) as Kroeger et al. did [7, 8] for centralized environments, where they proved that such structures where effective.

Thus, each file access is modeled as a symbol (i.e. the file path) and it is recorded as a node of the prefix tree with an associated value that represents how many times the chain pattern from root to that node has ocurred.

Then, in order to interact with Oraculo, it provides a basic interface for:

1. Adding an event to the trie, given a sequence of the last events seen.
2. Getting a prediction from the trie, given a sequence of the last events seen.

So that when a file access is produced, it can be noticed to Oraculo which computes which parts of the trie must be modified, adding the new event to the corresponding branches or simply increasing the counter of the pattern if it is already present.

Notice that in order to keep the trie scalable Oraculo can prune it in two ways. First, keeping a predefined maximum number of nodes per branch. Thus, whenever a new event goes to a full branch all its nodes divide their corresponding value by two and nodes with a value lower than 1 are deleted from the trie. In the case that no node has been cleaned, the new event is not added. But, obviously the nodes in the branch keep the new value (after the division by two) so in the near future it will be eventually possible to add new events in that branch.

On the other hand, Oraculo also has a maximum of root-branches (also called partitions) to keep the horizontal scalability of the trie. Here we apply a LRU-based algorithm among the partitions taking account of their usage as well.

Finally, Oraculo can predict future file accesses by applying basic data-mining on the trie. It only needs to know some previous accesses to look for patterns based on them. Then, an OSD could eventually use this information to replicate data in advance.

Furthermore, we will propose a decoupled and off-line aggregation of the tries. Once in a while (still to be determined), OSDs could contact other OSDs (only a small subset) to exchange their trie information and build an aggregated one that has the information of all. This mechanism will allow all OSDs to have a more or less global view because what is learned by one OSD will be propagated though several aggregations. We have done some preliminary tests using this mechanism and seems to work very well with environments of many thousands of nodes.

Regarding the information of the access pattern of files, in most cases the information kept by a single OSD will be enough. Nevertheless, whenever we need the full information of the pattern, we can contact all OSDs that have a replica and aggregate the learned behavior. As we do not expect to keep many replicas of a file, this procedure seems reasonable and scalable.

Integration of Oraculo with OSDs

Unfortunately, the integration of Oraculo with OSDs could not be done yet because we are still evaluating it by using GridSim (a well-known Grid simulator). Once we get significant results with the simulations we will evaluate them and port Oraculo to the OSDs.

1.4.3 Replica deletion

On the one hand, if we want to be able to replicate files whenever needed but still maintain the maximum number for replicas per file, it would be interesting to keep the number of replicas a bit smaller than the maximum. This difference between the maximum and the real number of replicas would allow the system to create replicas whenever needed. On the other hand, if replicas are not used, it would also be nice to have them removed automatically to reduce disk usage in a given node and/or center.

To tackle these two issues we will implement a mechanism that automatically deletes the less important replicas. To know what replicas are less important we will use similar mechanisms than the ones used to create replicas. We will predict future usage using the same kind of tries. In addition we will perform some kind of preventive removal of replicas, which means that whenever a node decides to remove a replica it will inform other OSDs that have it to react accordingly.

1.4.4 Interaction with the Application Execution Management

The last mechanisms that we will implement to manage replicas consists of an interaction with the application execution management system (AEM). This interaction will be done in two steps.

Firstly, AEM analyzes the JSDL of the application and asks to XtreemFS for the locations (coordinates X,Y) of its files' references. Thus AEM computes an optimal coordinate around where the job should be executed. This coordinate is used for AEM to send a request to Resource Selection Service

(RSS) for a set of nodes close to it. Of course, RSS also considers other requirements, such as CPU or memory, to decide the resulting set of nodes.

In the second step, the AEM will inform XtreemFS on the final destination of a given job and the files it will use. With this information, the RMS will decide if new replicas need to be created to improve the I/O performance of this job. In addition, and in some cases, it might be that the RMS decides to advance this step from the information obtained in step 1. For instance, this may happen when the list is made of nodes that are close among themselves and one or two replicas could do the job.

Although this mechanism is very good in the sense that no prediction needs to be done, it has a couple of limitations. The first one is that the AEM might not know the files used by a job (it is not a requirement in the job description). The second one is that there might not be enough time from the moment XtreemFS receives the execution location of a job (and the files it uses) and the moment the job starts running. To solve these two cases we have proposed the previous prediction mechanisms (1.4.2).

1.5 Testing

Regular and extensive testing is of vital importance for any file system, in order to improve its reliability, scalability, code quality, stability, performance, etc. Therefore, a new activity inside the WP3.4 has been created, which is focused on all the aspects of testing. The goal of this task is to evaluate functionality and performance of XtreemFS, by exploiting some existing file system test suites.

Nowadays, from a software viewpoint, there are some available tools that are mainly designed for testing local file systems, but there are no ready-made tools available for testing Grid File Systems. Thus, in order to test XtreemFS, either some ad-hoc tools must be developed or some existing distributed testing tools must be extended.

Currently, there are two main guidelines leading the tests performed on XtreemFS: the first one is aimed at evaluating the POSIX compliance of XtreemFS and consequently its functionality; the second one addresses the performance evaluation by stressing XtreemFS with a set of tests and benchmarks (in the following we refer to such kind of tests as regression tests).

In the next subsections we will describe both such activities, the tools used and the results obtained.

1.5.1 Testing POSIX compliance of XtreemFS

One of the goals of XtreemFS is providing a POSIX-compliant filesystem. In order to evaluate the correctness of functionalities, we initially evaluated some available test suites aimed at the POSIX compliance testing. Firstly, we evaluated the "Open Posix Tests Suite" (https://sourceforge.net/projects/posixtest/); it resulted not very suitable for our tests, because it performs only AIO tests but nothing else related to the file system; moreover, we encountered some difficulties during the installation and in particular during the execution of the tests, and for such reasons we discarded it. A second tool that we evaluated was the NTFS-3G suite (http://www.ntfs-3g.org), that is a test suite available for the most important operating systems; it includes a POSIX filesystem test environment, the Pawel Jakub Dawidek's POSIX filesystem test suite, that immediately seemed more suitable for our purposes than the first one. For such a reason, we chose to exploit the PJD's POSIX filesystem test suite (PJD-fstest) and to run it over XtreemFS. The test suite is available on the Web under a BSD license and we got it from http://www.ntfs-3g.org/pjd-fstest.html. PJD-fstest performs almost 3700 regression tests that exhaustively check a wide amount of different scenarios for the following system calls:

- *chmod*: changes the permissions of a files or directories
- *chown*: changes the owner of files or directories
- *link*: creates hard link
- *mkdir*: creates directories
- *mkfifo*: creates fifo named pipes
- *open*: opens a file
- *rename*: renames files or directories
- *rmdir*: removes directories
- *symlink*: creates symbolic links
- *truncate*: truncates files
- *unlink*: removes files

For each system call, the suite contemplates the execution of a set of scripts. Each script performs a set of basic operations, like the creation of a directory, the change of its access rights, the change of its owner, etc., and for each operation it evaluates its return value. If such a value is different than that expected, an error is pointed out. Obviously, the scripts performing the tests for a particular system call are composed of operations targeted for the evaluation of the (hopefully correct) behaviour of that system call. Our work consists in the automatic execution of the scripts and in the evaluation of the failure events. Then, for each failure, we need to interpret the cause of the problem and reproduce manually the scenario (the sequence of operations) causing it. After this step (that some time hides some difficulties) the problem is pointed out in a bug tracker, in order to be scheduled for a solution.

How to execute the tests

After downloading the tarball of the PJD-fstest, we can simply extract its contents:

```
tar xvzf <package>
```

The most significant contents of the tarball are:

- *fstest.c*: the source code of the main program. It provides an implementation of all the syscalls commented before.
- *Makefile*: the Makefile that we can use to compile the program.
- *tests*: a directory that contains one subdirectory for every syscall. For each of this subdirectories, there is a set of scripts that conform the tests for the corresponding syscall.

Afterwards, we can compile the fstest.c file by executing ***make***.

To execute the tests, we have implemented a tool that basically automatizes all the process of updating, compiling, and installing XtreemFS, running a basic scenario with one Directory Service, one MRC and an OSD, and creating a volume and mounting it on a specific directory.

Once this scenario is up and running, we enter the mount-point where the volume is mounted and execute the tests by using the command ***prove*** as follows:

```
sudo prove -r <path_to_pjd_suite>/tests
```

The flag *-r* specifies that *prove* should traverse all the directories recursively (so all the tests are executed). Notice that we need root privileges so we also need to edit */etc/fuse.conf* and add the following line:

```
user_allow_other
```

This line allows non-root users to specify the option *allow_other* (or *allow_root*) among the mounting options of fuse.

Results

At the moment, XtreemFS does not support fifos, so mkfifo tests are being ignored.

The tests corresponding to the rest of the syscalls were executed completely and the only errors encountered were due to the lack of implementation of sticky bits on files and directories.

We plan to work on it in next XtreemFS versions, although it is not a very important feature and currently we have to spend our efforts on other more significant issues.

1.5.2 Regression Tests

We use a set of regular file system test tools and custom made tests to automatically check the XtreemFS development version (the svn trunk) every night. This test environment can also be used to manually run these tests.

The main test script is `trunk/bin/xtfs_test` and can be executed to run all tests automatically or to start/stop a test environment.

To start a test environment with all XtreemFS servers and clients, run

```
 > trunk/bin/xtfs_test --start
```

This script will put all data and logfiles in the current working directory.

After setting up the test environement, the tests in `trunk/tests/` can be executed individually by calling the test scripts in the mounted XtreemFS volume.

```
 > python trunk/tests/01_simple_metadata.sh
```

To shutdown all servers and unmount the clients after testing, execute

```
> trunk/bin/xtfs_test --stop
```

To clean up all data and logfiles use

```
> trunk/bin/xtfs_test --clean
```

If you want to run all tests automatically, run the test script in auto mode

```
> trunk/bin/xtfs_test --autotest
```

1.6 Protocol and Interactions

XtreemFS uses ONC RPC[11] for executing remote operations. Interfaces and records are defined in a subset of CORBA IDL. Yidl[4] is used to generate the code for the interfaces and records in C++ and Java.

1.6.1 Constants

Globally shared constants are defined in `interfaces/constants.idl`.

`ACCESS_CONTROL_POLICY_NULL` don't use any access policy (on the MRC). This will allow all users to do everything on the volume.

`ACCESS_CONTROL_POLICY_POSIX` use standard POSIX permissions (user, group, others) on the volume.

`ACCESS_CONTROL_POLICY_VOLUME` similar to POSIX permissions but the permission for the root (/) is used for the entire volume.

`ACCESS_CONTROL_POLICY_DEFAULT` the policy to use in e.g. mkvol if nothing is specified.

`ONCRPC_SCHEME` scheme for URLs.

`ONCRPCS_SCHEME` scheme for URLs when using SSL.

`ONCRPC_AUTH_FLAVOR` constant to use for ONC RPC auth_flavor to indicate XtreemFS auth. If present, a UserCredentials record is sent in auth_opaque

[4]http://code.google.com/p/yidl/

`OSD_SELECTION_POLICY_SIMPLE` only OSDs which are alive and which have more than 2GB free space are used.

`OSD_SELECTION_POLICY_DEFAULT` the policy to use in e.g. mkvol if nothing is specified.

`REPL_UPDATE_PC_NONE` no replication is used

`REPL_UPDATE_PC_RONLY` read-only replication

`SERVICE_TYPE_MRC` for DIR service registry, service is an MRC

`SERVICE_TYPE_OSD` for DIR service registry, service is an OSD

`SERVICE_TYPE_VOLUME` for DIR service registry, service is a volume

`STRIPING_POLICY_RAID0` RAID0 (striping)

`STRIPING_POLICY_DEFAULT` the policy to use in e.g. mkvol if nothing is specified.

`STRIPING_POLICY_STRIPE_SIZE_DEFAULT` default stripe size in KB to use if nothing is specified.

`STRIPING_POLICY_WIDTH_DEFAULT` default striping width (number of OSDs) to use if nothing is specified.

`SYSTEM_V_FCNTL_H_O_`. . . POSIX constants

1.6.2 Types

Globally Shared Types

Globally shared data structures are defined in `interfaces/types.idl`.

`struct UserCredentials`

User information sent in the ONC RPC `opaque_auth` body if XtreemFS authentication is used. How the userID and groupIDs look like depends on the policy used in the client which translates the local uid/gid.

`user_id`	globally unique userID
`group_ids`	list of globally unique groupIDs (must contain at least one entry)
`password`	admin password (in cleartext) required for some operations (e.g. mkvol)

`struct VivaldiCoordinates`

Structure used to exchange Vivaldi coordinates between components, also used in UDP packets for measuring latency between XtreemFS clients and OSDs.

`x_coordinate`	x coordinate
`y_coordinate`	y coordinate
`local_error`	confidence in correctness of x/y coordinates

Types Shared between MRC and OSD

Types that are mainly shared between MRC and OSD are defined in `interfaces/mrc_osd_types.idl`.

`struct NewFileSize`

Sent by the OSD in response to a file modification operation if the file size has changed. A client may cache these updates and send them to the MRC when renewing a capability, on fsync/flush and close. The client needs only to send the most recent record it received from the OSD for a given file. Most recent means that: `(size_in_bytes' > size_in_bytes AND truncate_epoch' == truncate_epoch) OR (truncate_epoch' > truncate_epoch)`

The client should update its local file size cache with the NewFileSize records received from the OSD. The client should use the *locally cached* file size on stat rather than the result from the MRC to ensure that local processes see their own modifications.

`size_in_bytes`	the new file size in bytes
`truncate_epoch`	truncate epoch in which this operation was executed (used by the MRC for ordering updates)

`struct OSDtoMRCData`

Data sent by the OSD to the client which is expected to pass it on to the MRC. When the data should be passed to the MRC depends on the `caching_policy`. This feature is currently not used.

`caching_policy`	describes how the client is allowed to cache the data (when to send it to the MRC)
`data`	opaque data

`struct OSDWriteResponse`

Record containing file size updates and/or OSDtoMRCData. Returned from all data-modifying operations.

`new_file_size`	contains no record or at most one record if the file size changed
`opaque_data`	contains 0 or more records

`struct StripingPolicy`

Describes how a replica (one copy of the file) is split into objects.

`policy`	describes the scheme to use for distributing the objects among the OSDs, e.g. RAID0 for simple round robin striping.
`stripe_size`	the size of the objects in kilobytes, must be >= 4
`width`	the number of OSDs to use for striping, must be >= 1.

`struct Replica`

Describes a single copy of a file.

`striping_policy`	the striping policy to use for this replica.
`replication_flags`	value depends on the replication policy, e.g. to indicate a full or lazy replica.
`osd_uuids`	ordered (!) list of OSDs holding objects of the file.

`struct XLocSet`

Describes a complete file together with all replicas (copies) and how they are kept consistent.

`replicas`	list of the file's replicas (i.e. list of Replica structs)
`version`	incremented by the MRC on each modification of the list. Used by the OSD to reject clients working with outdated lists.
`repUpdatePolicy`	the policy used for keeping replicas in sync.
`read_only_file_size`	the size of the file in bytes, used only for read-only replication.

`struct XCap`

Security token which is issued by the MRC and authorizes a client to execute operations on a file at the OSDs.

`file_id`	file for which the capability can be used
`access_mode`	POSIX access mode for which client is authorized (e.g. read only, delete, write, truncate).
`expires_s`	absolute timestamp when the capability becomes invalid (seconds since epoch).
`client_identity`	the client identity set by the MRC, currently the client's IP address.
`truncate_epoch`	the file's current truncate epoch.
`server_signature`	the MRC's signature for the capability which is used by the OSD to validate the XCap. Signature is created using shared secret specified in the MRC and OSD configuration.

`struct FileCredentials`

A record containing the XLocSet and XCap for a file. Required for most OSD operations.

`xlocs`	the XLocSet
`xcap`	the capability

`sequence<FileCredentials> FileCredentialsSet`

Used by the MRC to return no or at most one FileCredentials record.

Exceptions

Exceptions that may be thrown in connection with an RPC are defined in `interfaces/exceptions.idl`.

`exception ProtocolException`

Thrown on ONC RPC errors (e.g. GARBAGE_ARGS)

`accept_stat`	ONC RPC `accept_stat` value
`error_code`	POSIX errno, if available
`stack_trace`	optional, for debugging only

`exception errnoException`

Thrown by the MRC to indicate a POSIX error.

`error_code`	POSIX errno, if available
`error_message`	optional text message
`stack_trace`	optional, for debugging only

`exception RedirectException`

Thrown by the DIR,MRC and OSD to redirect the client to another service. Use e.g. for master slave replication to direct the client to the current master.

`to_uuid`	service to contact

`exception ConcurrentModificationException`

Thrown by the DIR if a record was modified by another service on the meantime.

`stack_trace`	optional, for debugging only

`exception InvalidArgumentException`

Thrown by the DIR if an input value is not acceptable.

`error_message`	error message describing the correct values.

1.6.3 Directory Service Interface

The Directory Service interface is defined in `interfaces/dir_interface.idl`.

`struct AddressMapping`

Maps a service UUID to protocol, hostname/IP and port. A service can have multiple mappings for different networks (e.g. inside a cluster with private IP addresses). At the moment only “*” is supported for `match_network` which indicates a match for all networks.

`uuid`	the service UUID
`version`	the record's version, used by the DIR to detect concurrent modifications
`protocol`	the protocol used by the service
`address`	resolvable hostname or IP address in text form
`port`	port on which the service listens
`match_network`	for future use, must be `*`
`ttl_s`	time to live in seconds, indicates how long this record can be cached before it is re-fetched from the DIR

`sequence<AddressMapping> AddressMappingSet`

Future releases of XtreemFS will support multi-network setups to ease the usage of XtreemFS in shared public/private network environments often found in clusters.

`struct Service`

Information on a service registered at the DIR.

`uuid`	the service UUID
`version`	the record's version, used by the DIR to detect concurrent modifications
`type`	service type (see 1.6.1)
`name`	human readable name of the service; for volumes: the unique volume name
`last_updated_s`	timestamp of the last time (in seconds since epoch) the service updated its entry at the DIR. Used as a coarse-grained heartbeat-signal.
`data`	a map of additional data which depends on the service (e.g. MRC of a volume or free space of an OSD)

`void xtreemfs_address_mappings_get( string uuid, out AddressMappingSet address_mappings )`

Get an address mapping for the service specified by `uuid`.

`uuid`	the service UUID
`out address_mappings`	empty, if no mapping exists, one (or more) records otherwise

void xtreemfs_address_mappings_remove(string uuid)

Remove an address mapping from the DIR.

`uuid`	the service UUID

uint64_t xtreemfs_address_mappings_set(AddressMappingSet address_mappings)

Updates the address mappings for a service.

`address_mappings`	the new mappings. The UUID in all records must be the same. The version must be 0 for a new mapping or the version obtained with the last read from the DIR.
returns	the new version of the mapping
throws	ConcurrentModificationException if the record was updated (version incremented by DIR) between reading and updating the record.

void xtreemfs_checkpoint()

Forces the DIR to create a BabuDB checkpoint. This operation does not block, the checkpoint is created asynchronously. The admin password must be sent via the XtreemFS authentication.

uint64_t xtreemfs_global_time_s_get()

Returns the current system time on the DIR in seconds since epoch. Used to synchronize MRCs and OSDs to the global XtreemFS system time. The DIR system should be synchronized with a precise clock using e.g. ntp.

returns	system time in seconds since UNIX epoch.

void xtreemfs_service_get_by_type(uint16_t type, out ServiceSet services)

Get all services of a specific type registered at the DIR.

`type`	the service type to return
`out services`	all matching services

void xtreemfs_service_get_by_uuid(string uuid, out ServiceSet services)

Get the service information for a service with a specific UUID.

`uuid`	the service uuid
`out services`	one record, if the service is registered, empty list otherwise

`void xtreemfs_service_get_by_name( string name, out ServiceSet services )`

Get the service information for a service with a specific name.

`name`	the service's name
`out services`	one record, if the service is registered, empty list otherwise

`uint64_t xtreemfs_service_register( Service service )`

Update a service registration at the DIR. Updates the `last_update_s` field of the service.

`service`	the service's data. The UUID must be the service's UUID, the version must be 0 for a new service or the version obtained with the last read from the DIR.
returns	the new version of the mapping
throws	ConcurrentModificationException if the record was updated (version incremented by DIR) between reading and updating the record.

`void xtreemfs_service_deregister( string uuid )`

Removes the service registry entry for the service from the DIR.

`uuid`	the service uuid

`void xtreemfs_service_offline( string uuid )`

Sets the `last_update_s` field to 0 which indicates that the service was taken offline.

`uuid`	the service uuid

`void xtreemfs_shutdown()`

Shuts down the DIR service, does not force a checkpoint of the database. The admin password must be sent via the XtreemFS authentication.

1.6.4 Metadata and Replica Catalog Interface

The MRC interface is defined in `interfaces/mrc_interface.idl`.

`struct Stat`

Contains information about a file, directory or symbolic link that is sent to the client in response to a `getattr` request.

`mode`	the file's current access mode
`nlink`	the number of hard links to the file
`uid`	the numeric UID of the file's owner (just for compatibility reasons, will not be filled)
`gid`	the numeric GID of the file's owner (just for compatibility reasons, will not be filled)
`unused_dev`	(just for compatibility reasons, will not be filled)
`size`	the current file size in bytes
`atime_ns`	the file's atime in nanos
`mtime_ns`	the file's mtime in nanos
`ctime_ns`	the file's ctime in nanos
`user_id`	the XtreemFS user ID string of the file's owner
`group_id`	the XtreemFS group ID string of the file's owner
`file_id`	the XtreemFS file ID
`link_target`	the target path for symbolic links
`truncate_epoch`	the file's current truncate epoch
`attributes`	a set of Win32 specific file attributes

`struct DirectoryEntry`

Contains information about a directory entry that is sent to the client in response to a `readdir` request.

`name`	the name of the directory entry
`stbuf`	a buffer of type `struct Stat` that contains information about the file

`struct StatVFS`

Contains information about a mounted XtreemFS volume, which is sent to the client in response to a `statvfs` request.

`bsize`	the file system's block size (1024)
`bfree`	the number of free blocks
`fsid`	the file system ID (volume ID)
`namelen`	maximum file name length (1024)

`struct Volume`

Contains information about a volume.

`name`	the volume name
`mode`	the access mode for the volume's parent directory
`osd_selection_policy`	the ID of the OSD selection policy for the volume
`default_striping_policy`	the ID of the default striping policy for the volume
`id`	the volume UUID
`owner_user_id`	the XtreemFS user ID of the volume's owner
`owner_group_id`	the XtreemFS group ID of the volume's owner

`const DEFAULT_ONCRPC_PORT`

Constant defining the default MRC ONC RPC port.

`const DEFAULT_ONCRPCS_PORT`

Constant defining the default MRC ONC RPC port for SSL.

`const DEFAULT_HTTP_PORT`

Constant defining the default MRC HTTP port.

`exception MRCException`

Thrown by all MRC operations.

`boolean access( string path, uint32_t mode )`

Checks access to a file or directory. Responds with `true` if access is granted, `false`, otherwise.

`path`	the path to the file or directory
`mode`	the access flags to check

`void chmod( string path, uint32_t mode )`

Changes the access mode of a file or directory.

`path`	the path to the file or directory
`mode`	the new access mode

`void chown( string path, string user_id, string group_id )`

Changes the owner of a file or directory.

`path`	the path to the file or directory
`user_id`	the new owner ID
`group_id`	the new owning group ID

`void create( string path, string user_id, string group_id )`

Creates a new file.

`path`	the path to the new file
`mode`	the initial access mode for the new file

`void ftruncate( XCap write_xcap, out XCap truncate_xcap )`

Issues a new truncate capability for an open file.

`write_xcap`	a valid Capability with write permissions to the file
`out truncate_xcap`	a new capability with write and truncate permissions, which has to be used for subsequent operations

`void getattr( string path, out Stat stbuf )`

Returns information on a file or directory.

`path`	the path to the file or directory
`out stbuf`	a buffer containing information on the file or directory

`void getxattr( string path, string name, out string value )`

Returns the value of an extended attribute of a file or directory.

`path`	the path to the file or directory
`name`	the name of the attribute
`out value`	the attribute value

`void link( string target_path, string link_path )`

Creates a new hard link to an existing file.

`target_path`	the path to the existing file
`link_path`	the path defining where the new hard link shall be created

`void listxattr( string path, out StringSet names )`

Returns the set of extended attributes assigned to a file or directory.

`path`	the path to the file or directory
`names`	the list of attribute names assigned to the file or directory

`mkdir( string path, uint32_t mode )`

Creates a new directory.

`path`	the path to the new directory
`mode`	the initial access mode for the new directory

`open( string path, uint32_t flags, uint32_t mode, out FileCredentials file_credentials )`

Opens a file by performing an access check and issuing a new Capability for OSD access in case of success.

`path`	the path to the file
`flags`	a set of flags specifying the kind of access that is requested
`mode`	initial access mode for a newly created file in case `flags` contains `O_CREAT`
`out file_credentials`	a set of file credentials containing the file's X-Locations list and the newly issued Capability

`readdir( string path, out DirectoryEntrySet directory_entries )`

Lists the content of a directory, including all file metadata.

`path`	the path to the directory
`out directory_entries`	a list containing all nested directory entries for the given directory

`void removexattr( string path, string name )`

Removes an extended attribute from a file or directory.

`path`	the path to the file or directory
`name`	the name of the attribute to remove

`void rename( string source_path, string target_path, out FileCredentialsSet file_credentials )`

Renames a path.

`source_path`	the former path to the file or directory
`target_path`	the new path to the file or directory
`out file_credentials`	contains an X-Locations list and deletion Capability in case the target path was overwritten

`rmdir( string path )`

Removes an empty directory.

`path`	the path to the directory

`void setattr( string path, Stat stbuf )`

Sets metadata of a file or directory. Currently, this call is only used to set the file's Win32 attributes.

`path`	the path to the file or directory
`stbuf`	a buffer containing the metadata to set

`void setxattr( string path, string name, string value, int flags )`

Sets an extended attribute of a file or directory.

`path`	the path to the file or directory
`name`	the name of the attribute
`value`	the new value for the attribute
`flags`	a set of system flags associated with the attribute (currently ignored)

`void statvfs( string volume_name, out StatVFS stbuf )`

Returns information about a volume.

`volume_name`	the volume name
`stbuf`	a buffer containing information about the volume

`void symlink( string target_path, string link_path )`

Creates a symbolic link.

`target_path`	the target path
`link_path`	the path for the new symbolic link

`void unlink( string path, out FileCredentialsSet file_credentials )`

Unlinks a file from a directory. If no more links to the file exist, file metadata will be deleted.

`path`	the path to the file
`out file_credentials`	a set of file credentials containing a deletion Capability and X-Locations list in case file metadata needs to be deleted.

`void utimens( string path, uint64_t atime_ns, uint64_t mtime_ns, uint64_t ctime_ns )`

Sets the POSIX time stamps of a file or directory.

`path`	the path to the file or directory
`atime_ns`	the new access time stamp in nanos (will be ignored if set to 0)
`mtime_ns`	the new modification time stamp in nanos (will be ignored if set to 0)
`ctime_ns`	the new change time stamp in nanos (will be ignored if set to 0)

`void utimens( string path, uint64_t atime_ns, uint64_t mtime_ns, uint64_t ctime_ns )`

Sets the POSIX time stamps of a file or directory.

`path`	the path to the file or directory
`atime_ns`	the new access time stamp in nanos (will be ignored if set to 0)
`mtime_ns`	the new modification time stamp in nanos (will be ignored if set to 0)
`ctime_ns`	the new change time stamp in nanos (will be ignored if set to 0)

`xtreemfs_checkpoint()`

Enforces the creation of a new database checkpoint. The call blocks until checkpoint creation has completed.

`void xtreemfs_check_file_exists( string volume_id, StringSet file_ids, out string bitmap )`

Checks for a set of file IDs whether the given files exist in the given volume. This call is necessary for cleanup purposes.

`volume_id`	the volume's UUID
`file_ids`	a list of file IDs to check
`out bitmap`	a string containing '1's for each file in `file_ids` that exists, and '0's for each file that does not exist, in the same order as the file IDs given in `file_ids`

`void xtreemfs_dump_database( string dump_file )`

Creates an XML dump of the MRC database on the server.

`dump_file`	the path at which to store XML dump file on the MRC's local file system

`void xtreemfs_get_suitable_osds( string file_id, out StringSet osd_uuids )`

Returns a list of suitable OSDs for the given file. The call can be used to find OSDs that are suitable for new replicas of a file.

`file_id`	the file ID
`out osd_uuids`	a list of OSD UUIDs that can be used for the file

`void xtreemfs_lsvol( out VolumeSet volumes )`

Returns a list of all volumes stored on the MRC.

`out volumes`	a list containing information on each volume stored on the MRC

`void xtreemfs_mkvol( Volume volume )`

Creates a new volume.

`volume`	information about the volume to create

`void xtreemfs_renew_capability( in XCap old_xcap, out XCap renewed_xcap )`

Extends the validity of a capability. The capability must be valid in order to be renewed.

`old_xcap`	the capability to be renewed
`renewed_xcap`	a new capability with the same properties except for an extended validity period

`void xtreemfs_replica_add( string file_id, Replica new_replica )`

Adds a new replica to a file. The file's read-only flag must be set to `true`.

`file_id`	the file ID
`new_replica`	the new replica to be added

`void xtreemfs_replica_list( string file_id, out ReplicaSet replicas )`

Returns the list of replicas of a file. Information on each of the replicas in the list includes the striping policy and X-Loc list.

`file_id`	the file ID
`out replicas`	the list of replicas

`xtreemfs_replica_remove( string file_id, string osd_uuid, out XCap delete_xcap )`

Removes a replica from a file.

`file_id`	the file ID
`osd_uuid`	the UUID of the head OSD
`delete_xcap`	a capability for deleting the data associated with the replica on the OSD

`xtreemfs_restore_database( string dump_file )`

Restores the MRC database from an XML dump. When the operation is invoked, no volumes may exist in the current database.

`dump_file`	the path to the XML dump file on the MRC's local file system

`xtreemfs_restore_file( string file_path, string file_id, uint64_t file_size, string osd_uuid, int32_t stripe_size )`

Restores a file from the given metadata.

`dump_file_path`	the path associated with the restored file
`file_id`	the ID associated with the restored file
`file_size`	the size associated with the restored file
`osd_uuid`	the OSD on which the file content is stored
`stripe_size`	the stripe size associated with the restored file

`void xtreemfs_rmvol( string volume_name )`

Deletes a volume, including the metadata of all nested files and directories.

`volume_name`	the name of the volume to delete

`void xtreemfs_shutdown( )`

Gracefully terminates the MRC with all its sub-components.

`void xtreemfs_update_file_size( XCap xcap, OSDWriteResponse osd_write_respons`

Updates the size of a file in response to an OSD write operation.

`osd_write_response`	the response from the OSD, which may contain a new file size

1.6.5 Object Storage Device Interface

The OSD interface is defined in `interfaces/osd_interface.idl`.

`struct InternalGmax`

Sent by OSDs to determine the actual file size of a striped file.

`epoch`	the file's latest truncate epoch the OSD knows
`last_object_id`	last object number known by the OSD
`file_size`	locally known file size in bytes

`struct ObjectData`

Sent by OSDs to determine the actual file size of a striped file.

`data`	object data (file content)
`checksum`	checksum for data
`zero_padding`	zeros to append to data (padding objects, POSIX sparse file semantics)
`invalid_checksum_on_osd`	true if the OSD detected corrupted on-disk data

`exception OSDException`

Thrown by all OSD operations.

`error_code`	see class `org.xtreemfs.osd.ErrorCodes` for a list of error codes
`error_message`	optional, human readable error message
`stack_trace`	optional, for debugging only

`void read( FileCredentials file_credentials, string file_id, uint64_t object_number, uint64_t object_version, uint32_t offset, uint32_t length, out ObjectData object_data )`

Reads on object.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
`object_number`	the object requested (first object is 0)
`object_version`	for future use
`offset`	offset within object
`length`	number of bytes to read (offset+length must be < object size)
`out object_data`	the object data read from disk. If less data (data + zero padding) is returned, this indicates an EOF.

`void truncate( FileCredentials file_credentials, string file_id, uint64_t new_file_size, out OSDWriteResponse osd_write_response )`

Truncates a file to the specified length. The client must have a capability valid for truncating. For files which are striped over more than one OSD, this operation must be executed at the *head OSD* which is the first OSD in the replica's OSD list.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
`new_file_size`	the new size of the file in bytes to which the file is truncated
`out osd_write_response`	information which should be passed to the MRC.

`void unlink( FileCredentials file_credentials, string file_id )`

Deletes all objects of the file. If the file is currently open (in use) the objects will be deleted on close. This operation returns immediately, the objects are deleted by the OSD asynchronously. The client must have a capability valid for deleting. For files which are striped over more than one OSD, this operation must be executed at the *head OSD* which is the first OSD in the replica's OSD list.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id

`void write( FileCredentials file_credentials, string file_id, uint64_t object_number, uint64_t object_version, uint32_t offset, uint64_t lease_timeout, ObjectData object_data, out OSDWriteResponse osd_write_respons`

Writes an object.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
`object_number`	the object to write (first object is 0)
`object_version`	for future use
`offset`	offset within object
`lease_timeout`	for future use (timestamp of client lease timeout in seconds since epoch)
`object_data`	the object data to write into the object; `zero_padding` is ignored.

`ObjectData xtreemfs_check_object( FileCredentials file_credentials, string file_id, uint64_t object_number, uint64_t object_version )`

Similar to read. The OSD reads the object and validates the checksum but doesn't send the actual data. Used by the file system scrubber to check data integrity.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
`object_number`	the object requested (first object is 0)
`object_version`	for future use
returns	the size of the object in `zero_padding` and wether the data is corrupted

`InternalGmax xtreemfs_internal_get_gmax( FileCredentials file_credentials, string file_id )`

Returns the locally know truncate epoch, number of objects and file size for a file. Used by an OSD to determine the file size for a file which is striped over more than one OSD.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
returns	OSD-local file size information

`uint64_t xtreemfs_internal_get_file_size( FileCredentials file_credentials, string file_id )`

Returns the actual file size on the OSD(s). For files which are striped over more than one OSD, this operation must be executed at the *head OSD* which is the first OSD in the replica's OSD list. Used by file system scrubber tools. Should be used to update file size before marking a file read-only.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
returns	the actual file size of the file

`void xtreemfs_internal_truncate( FileCredentials file_credentials, string file_id, uint64_t new_file_size, out OSDWriteResponse osd_write_response )`

Used by the head OSD to truncate the file on all OSDs for a file which is striped across more than one OSD. May only be used by OSDs.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
`new_file_size`	the new size of the file in bytes to which the file is truncated
`out osd_write_response`	information which should be passed to the MRC.

`InternalReadLocalResponse xtreemfs_internal_read_local( FileCredentials file_credentials, string file_id, uint64_t object_number, uint64_t object_version, uint64_t offset, uint64_t length )`

Used by OSDs to fetch objects from other OSDs for replicated files. May only be used by OSDs. This method does not follow POSIX semantics by add padding data but sends the raw object data from disk.

`file_credentials`	XLocSet and Capability for the file
`file_id`	the file's file Id
`object_number`	the object requested (first object is 0)
`object_version`	for future use
`offset`	offset within object
`length`	number of bytes to read
returns	raw object data from disk

`void xtreemfs_cleanup_start(boolean remove_zombies,`
`boolean remove_unavail_volume, boolean lost_and_found )`

Starts the cleanup process on an OSD. The cleanup process will check for all files on the OSD's disk if they still exist in the MRC. Requires an admin password in the XtreemFS authentication data.

`remove_zombies`	delete files which have been deleted on the MRC
`remove_unavail_volume`	delete files if the MRC holding the volume is not available (DANGEROUS!)
`lost_and_found`	do not delete files but re-create them in a lost+found directory

`void xtreemfs_cleanup_stop()`

Aborts the OSD cleanup process. Requires an admin password in the XtreemFS authentication data.

`void xtreemfs_cleanup_status( out string status )`

Returns a human readable status string from the cleanup process. Requires an admin password in the XtreemFS authentication data.

`out status`	human readable status text (in English)

`void xtreemfs_cleanup_is_running( out boolean is_running )`

Check if the cleanup process is running. Requires an admin password in the XtreemFS authentication data.

`out is_running`	true, if the process is running

`void xtreemfs_cleanup_get_results( out StringSet results )`

Returns a list of messages from the cleanup process. Requires an admin password in the XtreemFS authentication data.

`out results`	list of messages

`void xtreemfs_cleanup_shutdown()`

Shuts down the OSD. Requires an admin password in the XtreemFS authentication data.

1.6.6 Interactions

This section illustrates the interactions between XtreemFS clients and servers.

delete

Files are deleted as described in the following (see Fig. 1.5):

1. The client receives a delete request from the VFS. It removes the file on the MRC via the unlink operation and receives the file credentials, which contain the globally unique XtreemFS fileID, a deletion capability and the replica locations list.
2. The client initiates the deletion of file content by invoking the `unlink` operation on the head OSD (i.e. the first OSD of a stripe).
3. The head OSD delays the deletion until all clients have closed the file, i.e. all capabilities known to the head OSD have timed out. In turn, the head OSD initiates the deletion of file objects on the remaining OSDs via `unlink`.

read

Files are read as described in the following (see Fig. 1.6):

1. The client receives an `open` request from the VFS. It opens the file on the MRC for reading, and receives the file credentials, which contain the globally unique XtreemFS fileID, a read capability and the replica locations list.

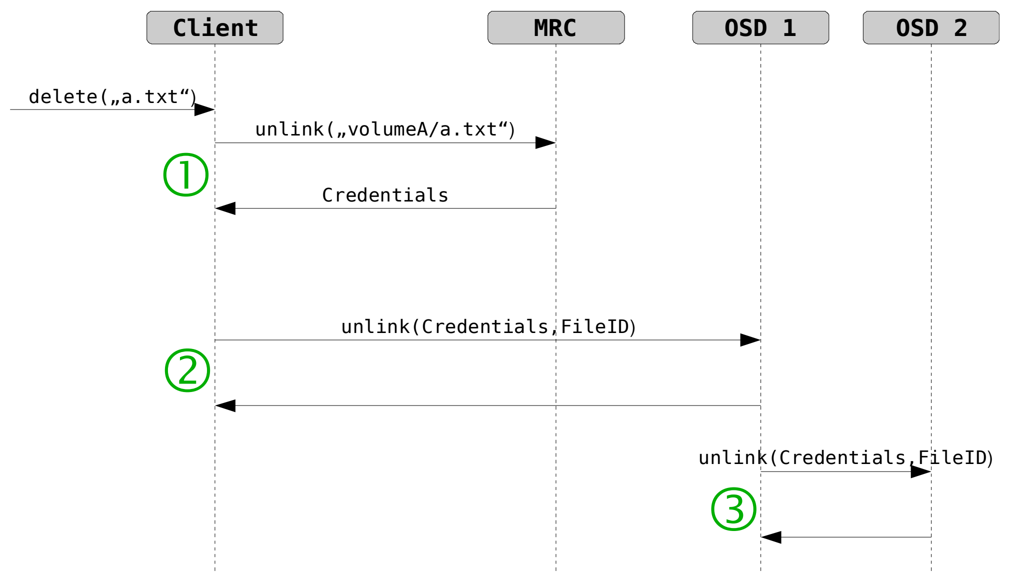


Figure 1.5: Deleting a file

2. The client sends a request for reading Y bytes of data from the offset Z of the file identified by the fileID. The OSD returns a buffer containing object data, as well as additional information like checksum failure notifications or padding flags.

3. If multiple read requests are send, the client has to ensure that the capability is renewed before it times out, in order to keep the file open.

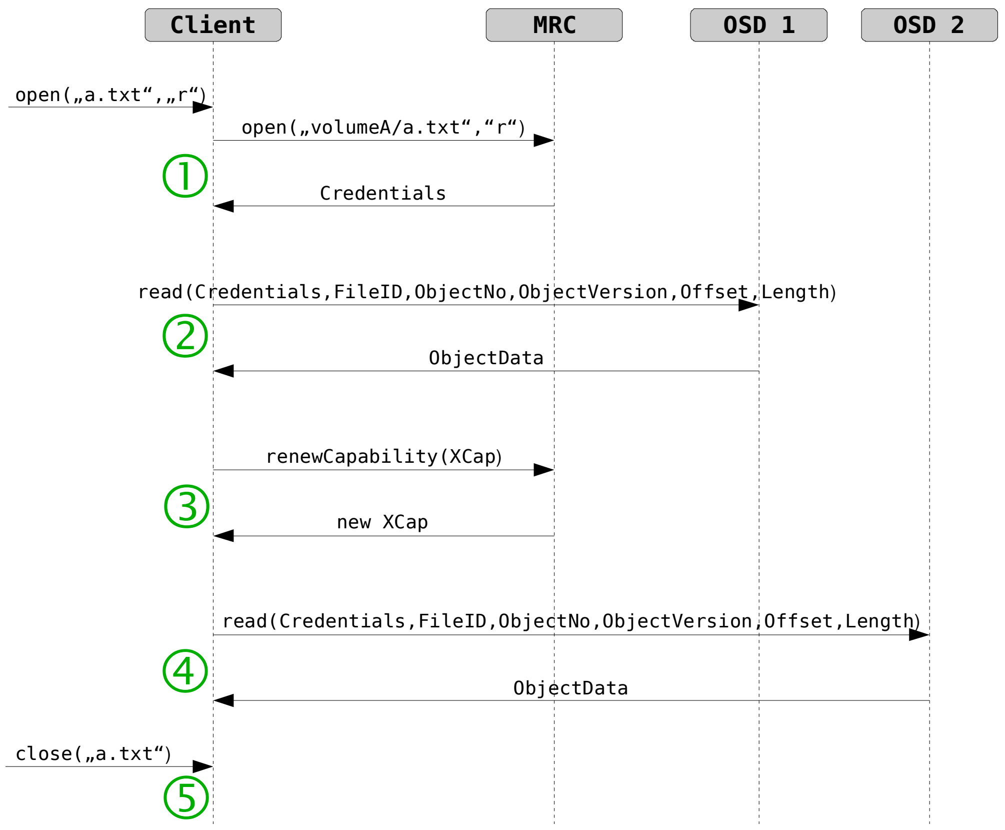
4. The client reads more data from the file, e.g. another object.

5. The client receives a `close` call. There is no need to explicitly close the file on the servers; this is implicitly done when the capabilities in the OSD cache time out.

write

Files are written as described in the following (see Fig. 1.7):

1. The client receives an `open` request from the VFS. It opens the file on the MRC for writing, and receives the file credentials, which contain the globally unique XtreemFS fileID, a write capability and the replica locations list.



Client
MRC
OSD 1
OSD 2
open(„a.txt“,„r“)
open(„volumeA/a.txt“,“r“)
1
Credentials
read(Credentials,FileID,ObjectNo,ObjectVersion,Offset,Length)
2
ObjectData
renewCapability(XCap)
3
new XCap
read(Credentials,FileID,ObjectNo,ObjectVersion,Offset,Length)
4
ObjectData
close(„a.txt“)
5

Figure 1.6: Reading a file

2. The client sends a buffer containing the data to the OSD on which the object is stored. In return, the OSD sends an `OSDWriteResponse`, which must be cached and sent to the MRC when the file is closed or fsync is called. The client can also decide to send pending filesize updates from time to time between capability renewals, as the lifetime of a capability can be in the range of tens of minutes.

3. If multiple write requests are send, the client has to ensure that the capability is renewed before it times out, in order to keep the file open.

4. The client appends data to the file by sending more `write` requests, each being answered with an `OSDWriteResponse`.

5. The client receives a `close` call. It sends any pending `OSDWriteResponses` to the MRC, in order to update the file size. There is no need to explicitly close the file on the servers; this is implicitly done when the capabilities in the OSD cache time out.

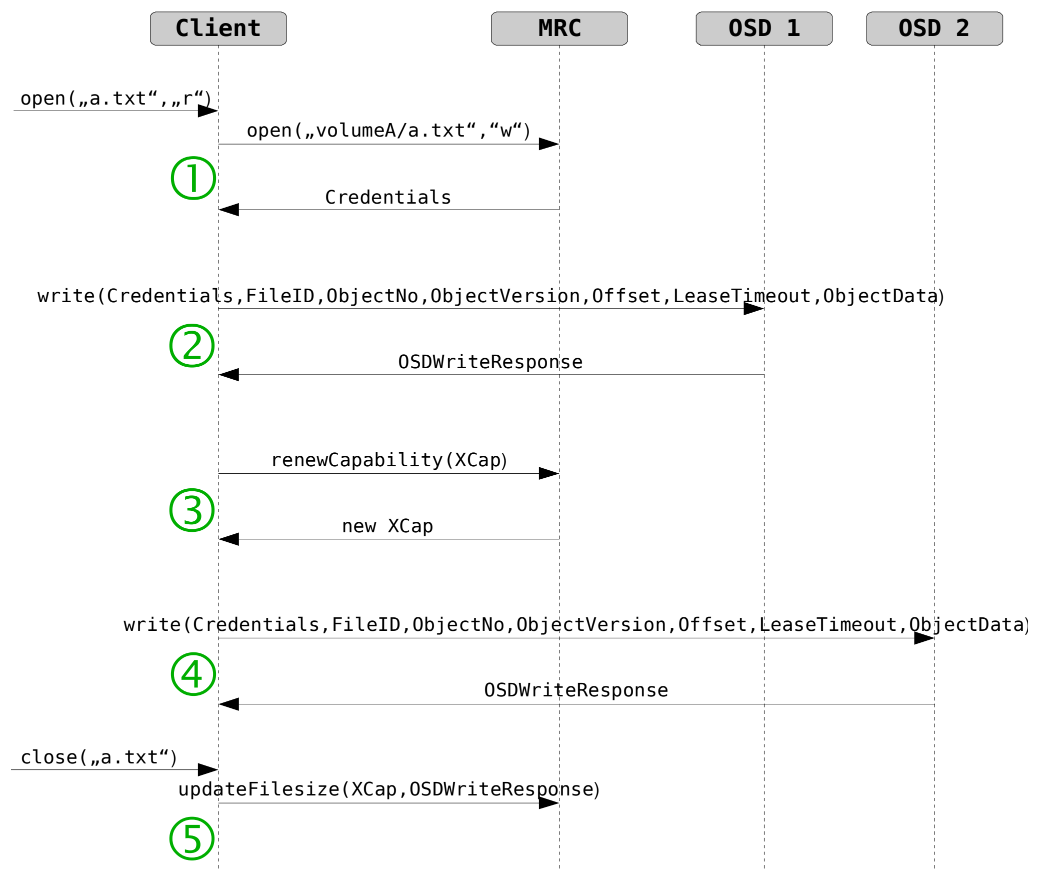

Client
MRC
OSD 1
OSD 2
open(„a.txt“,„r“)
open(„volumeA/a.txt“,“w“)
①
Credentials
write(Credentials,FileID,ObjectNo,ObjectVersion,Offset,LeaseTimeout,ObjectData)
②
OSDWriteResponse
renewCapability(XCap)
③
new XCap
write(Credentials,FileID,ObjectNo,ObjectVersion,Offset,LeaseTimeout,ObjectData)
④
OSDWriteResponse
close(„a.txt“)
updateFilesize(XCap,OSDWriteResponse)
⑤

Figure 1.7: Writing a file

fsync

Files are fsync'ed as described in the following (see Fig. 1.8):

1. The file was opened and modified.
2. The client receives a `fsync` request from the VFS. If the file is opened all pending data will be written to the OSD.
3. The client sends any pending `OSDWriteResponse`s to the MRC, in order to update the file size.
4. The file is further modified or closed.

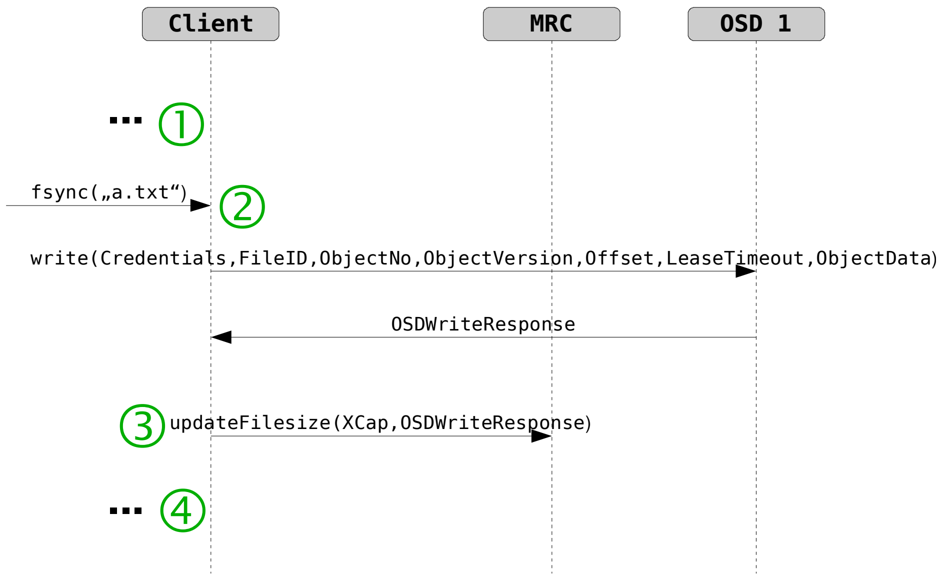

Client
MRC
OSD 1
... ①
fsync(„a.txt“)
②
write(Credentials,FileID,ObjectNo,ObjectVersion,Offset,LeaseTimeout,ObjectData)
OSDWriteResponse
③ updateFilesize(XCap,OSDWriteResponse)
... ④

Figure 1.8: Synchronizing data with the underlying device

mkvol

New volumes are created as described in the following (see Fig. 1.9):

1. The client receives an `mkvol` request. In response, it creates the volume on the MRC.
2. The MRC registers the volume at the DIR.
3. If no problems occur, the MRC responds to the client with an acknowledgment.

rmvol

Volumes are deleted as described in the following (see Fig. 1.10):

1. The client receives a `rmvol` request from the console. In response, it removes the volume on the MRC.
2. The MRC deregisters the volume at the DIR.
3. If no problems occur, the MRC responds to the client with an acknowledgment.

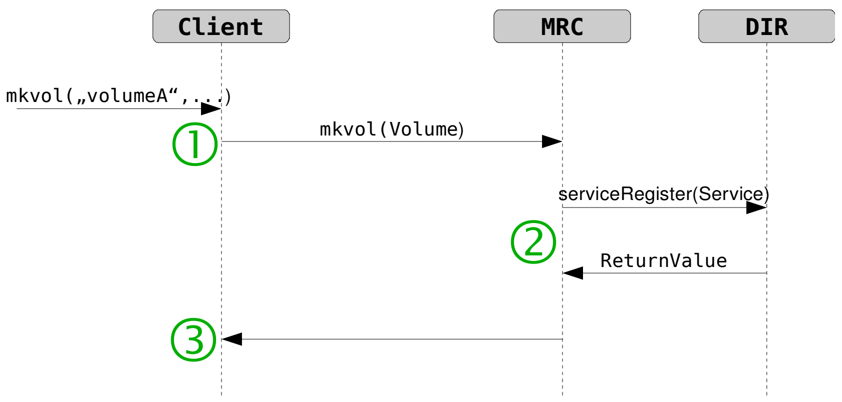


Figure 1.9: Creating a new volume

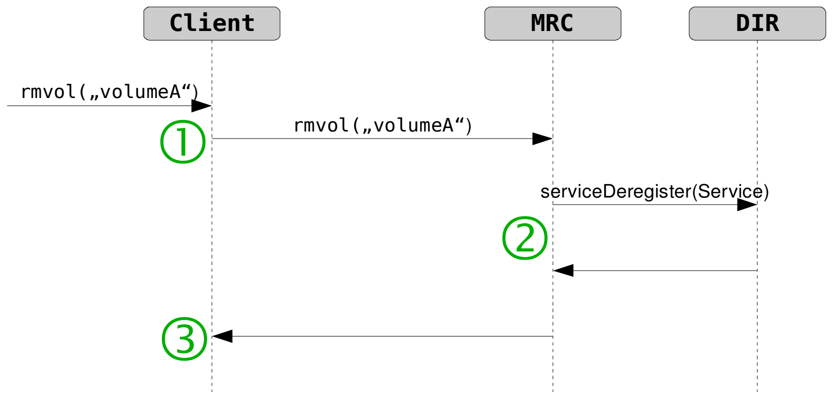


Figure 1.10: Deleting an existing volume

removeReplica

Single replicas of a file are deleted as described in the following (see Fig. 1.11):

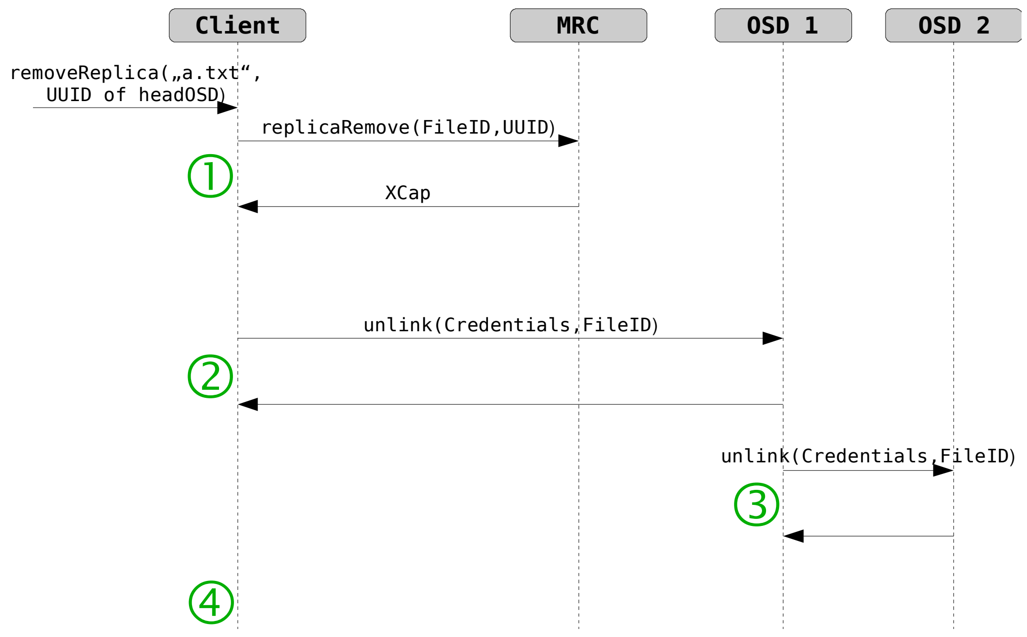
1. The client receives a `removeReplica` request from the console. It sends a request to the MRC to remove the replica with a matching head OSD. In response, it receives the file credentials, which contain the globally unique XtreemFS fileID, a deletion capability and the list of replica locations.

2. The client initiates the deletion of the replica's file content by invoking the `unlink` operation on the head OSD (i.e. the first OSD of a stripe).

3. The head OSD delays the deletion until all clients have closed the file, i.e. all capabilities known to the head OSD have timed out. In turn,

the head OSD initiates the deletion of file objects on the remaining OSDs via `unlink`.

4. If a client finds out that its replica locations list for the file is outdated, it has to retrieve the new replica locations list from the MRC.



Client
MRC
OSD 1
OSD 2
removeReplica(„a.txt“, UUID of headOSD)
replicaRemove(FileID,UUID)
1
XCap
unlink(Credentials,FileID)
2
unlink(Credentials,FileID)
3
4

Figure 1.11: Deleting a replica of a file

Chapter 2

The OSS Developer Guide

2.1 Introduction

The Object Sharing Service (OSS) implements shared objects for grid applications. OSS is built as a shared library. Linking OSS to user applications allows sharing of objects residing in volatile memory across multiple nodes in the grid. An object in this context is a replicated volatile memory region, dynamically allocated by an application or mapped into memory from a file.

Objects may contain scalars, references, and code. Therefore, OSS handles concurrent read and write access to objects and maintains the consistency of replicated objects. Persistence for objects stored in files are provided by XtreemFS, fault tolerance in contrast is provided by the grid checkpointing mechanisms developed in WP3.3. Currently, OSS supports IA32 and AMD64 compatible processors.

This report is structured as follows. Section 2.2 describes how OSS implements the XOSAGA API. Section 2.3 explains OSS' modular architecture and its network protocol. Section 2.4 documents the internal interfaces of the modules. Finally, section 2.5 describes step by step how to extend OSS with custom consistency models.

2.2 API

OSS's services are available via the XOSAGA API [1]. Besides, we have implemented a POSIX support library, which emulates POSIX's malloc and free calls for unmodified legacy applications. Both the XOSAGA API and the interface of the POSIX support library are based on the internal OSS interface which has been described in the OSS interface and user guide [2].

2.3 Architecture

Developed as a modularized system, OSS contains three main modules *cache management*, *consistency models* and *network communication* (see figure 2.1).

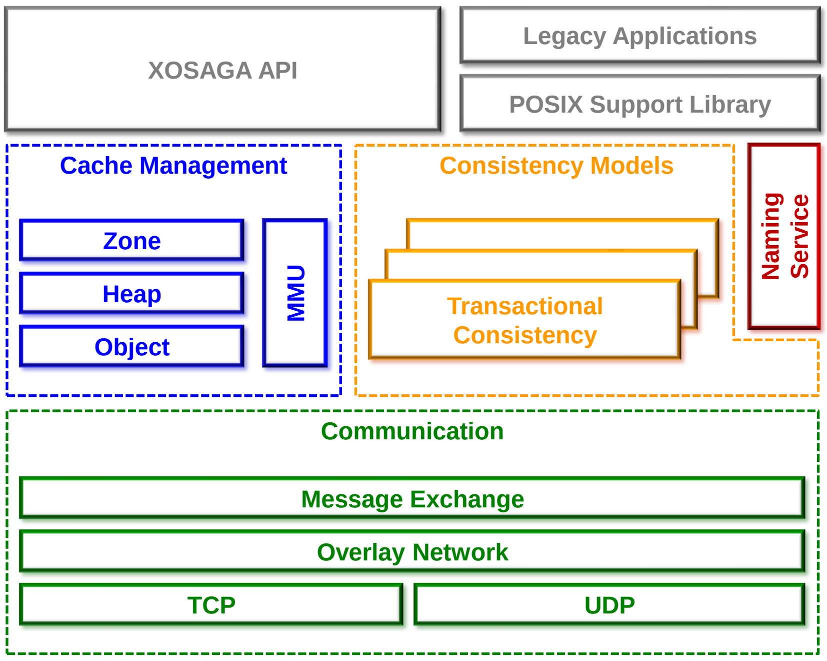


Figure 2.1: OSS architecture

2.3.1 Cache Management

OSS's cache management allocates objects from a global distributed object space. At a high level of abstraction, objects are represented as chunks of memory. The interface to the cache management is modeled after dynamic memory allocation on the heap, a technique well-known to most programmers. OSS does not interprete the content of objects, such that it allows to share objects in virtually all programming languages using XOSAGA language bindings. The cache management comprises object allocation, replication and basic synchronization services used by the consistency models.

2.3.2 Consistency Models

OSS is designed to support multiple consistency models for shared data synchronization. Application developers are able to allocate multiple objects, each coupled with a consistency model suitable for application semantics. Currently, OSS supports *strong, transactional* and *explicit consistency.*

Strong Consistency

Using the *strong consistency* memory modifications are immediately visible on subsequent reads. It is implemented using a modified version of the MESI cache coherence protocol, where the state *Exclusive* has been omitted. Strong consistent objects (in current OSS version allocated at page granularity) can obtain the following states (see figure 2.2):

`MODIFIED EXCLUSIVE` Object is modified and exclusive accessible by one node

`SHARED` Object is shared among multiple nodes

`INVALID` Object is invalid

In general, if a node gains write access to a specific object, it changes into modified state. Simultaneously, on all other nodes the object changes into invalid state. Gaining read access instead, the object changes into shared state. Any other node having exclusive access also changes into shared state.

Transactional Consistency

Transactional consistency also provides strong consistency but multiple operations are bundled into atomic transactions. Possibly occuring conflicts among transactions will be resolved transparently to applications in the background. The developer has only to define transaction boundaries by placing the following two function calls into the program code, defining begin and end of transactions:

```
oss_transaction_id_t oss_bot(...)

int oss_eot(oss_transaction_id_t)
```

Transactional consistent objects can obtain the following states (see figure 2.3):

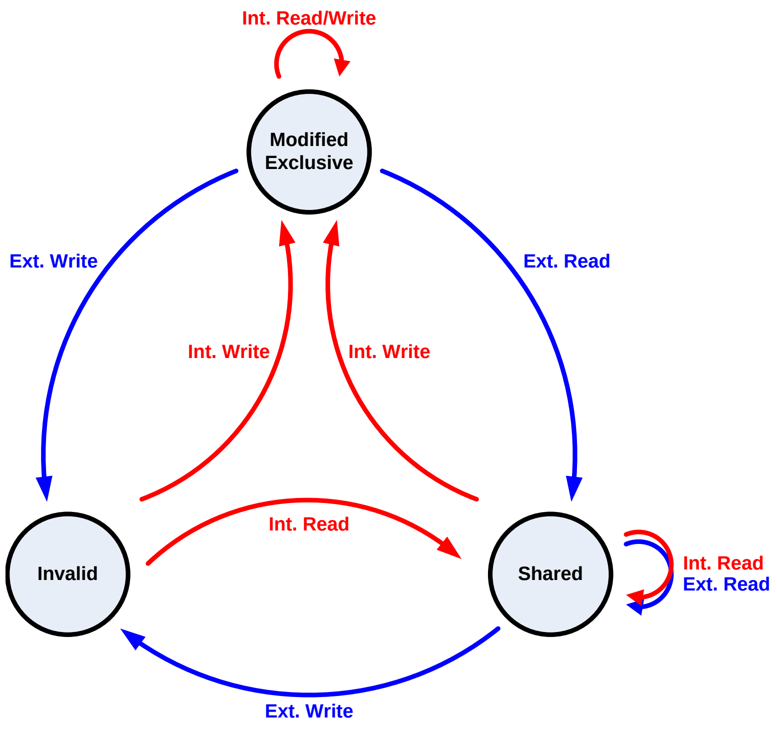


Figure 2.2: Finite state machine of strong consistency

`UNBOUND` Memory page has not been accessed

`BOUND READ` Memory page has been accessed for reading

`BOUND WRITE` Memory page has been accessed for writing

Explicit Consistency

Explicit consistency is a pseudo consistency model. Memory allocated under this consistency constraints will never be synchronized and behaves like local allocated memory (e.g. memory allocation via *malloc*). But the memory allocation scheme still follows the semantics of OSS. As in other consistency models, objects using this consistency model reside at the same memory address on all peers.

2.3.3 Name Service

OSS contains a simple internal name service, which applications can use to store and retrieve object IDs. The name service has a tree structure, with

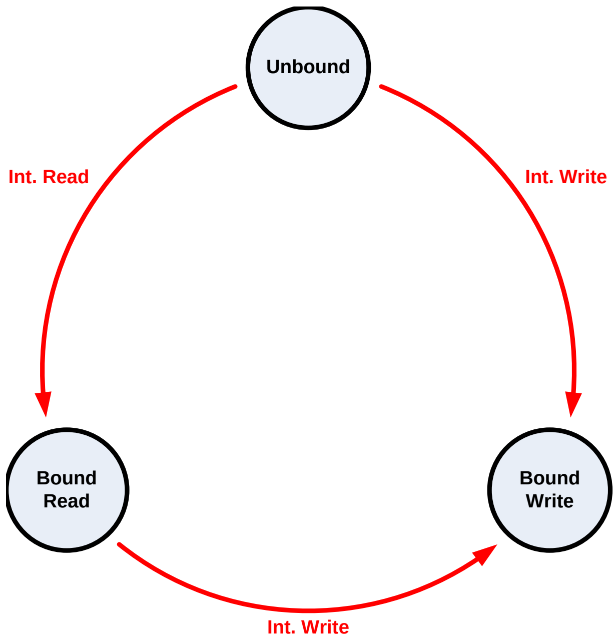


Figure 2.3: Finite state machine of transactional consistency

slashes (/) separating directory levels. An application or OSS module can set a value for a name by calling `oss_nameservice_set` and retrieve a value by calling `oss_nameservice_get`. A value that has not yet been set is treated as object ID `NULL`.

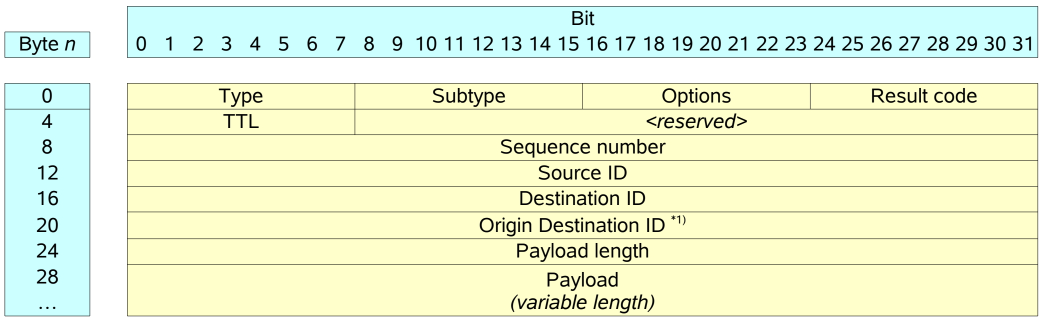
2.3.4 Network Communication

The network module has two layers. The lower layer implements the binding of transport protocols like TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) to the *overlay network* and assembles incoming data fragments to PDUs (Protocol Data Units). Furthermore, it implements fault tolerance mechanisms if not supported by the transport protocol itself. Currently, OSS uses TCP, only.

The upper layer implements functionality for establishing and managing the overlay network. Peers will be grouped together, coordinated by one *super peer* node which manages inter group communication and group internal tasks (e.g. transaction validation). The super peer will be elected on the basis of its properties (e.g. performance, network latency and bandwidth, average cpu load, ...). Moreover the overlay network routes messages among nodes and is dynamically reconfigurable by using statistical data collections.

The communication module implements an interface to abstract messaging from the underlying network structure.

For efficiency reasons, OSS implements its own binary request/reply network protocol. Every PDU begins with a header followed by an optional payload part. All fields of the PDU are described in detail (see figure 2.4)



Byte *n*	Bit 0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	Type	Subtype	Options	Result code
4	TTL	*<reserved>*		
8	Sequence number			
12	Source ID			
16	Destination ID			
20	Origin Destination ID *1)			
24	Payload length			
28 …	Payload *(variable length)*			

Figure 2.4: Structure of Protocol Data Units in OSS

Type Primary type of a network message. This field coincide with the modules which have registered a specific message type. All messages of a type are passed to the registered handler of a module. The communication module distinguishes request and response messages by inspecting the MSB (Most Significant Bit). An unset bit defines a request message, a set bit a response message. A pair of request and response messages differ only in the MSB referred to the type field.

Subtype Subtype of a network message. This field is used by modules to distinguish various messages of the same type.

Options Internally used bit field.

Result code In response messages this field carries the result code of the previously processed network request.

TTL Time-to-Live field, which prevents an endless routing/forwarding of messages in the overlay network. This field is decremented on every message routing/forwarding.

Sequence number Consecutive number for assigning reponse to request messages and to preserve message ordering.

Source ID Node ID of the sender.

Destination ID Node ID of the recipient (modified in case of message forwarding).

Origin dst. ID Node ID of the origin recipient (still unmodified in case of message forwarding).

Payload length Length of payload.

Payload Payload.

2.4 Internal Interfaces

As mentioned in section 2.2, OSS' services are available through the XOSAGA API. The internal interfaces of OSS' modules are used only inside the shared library and therefore will not be exported to user applications. The following sections describe the module interfaces of *cache management* and *network communication*.

2.4.1 Cache Management

Each object has a unique identifier within the shared object space. To enable efficient parallel allocation of objects, nodes pre-reserve heaps of objects using the distributed allocator. The cache management comprises different allocators to partition the per-node heaps. When creating an object, an application can specify which allocator to use, depending on the object's intended usage. Access control and storage management provide basic mechanisms to keep objects consistent.

High-level object management (object)

The high-level object management module dispatches object creation and object deletion to the respective heap. The module's interface consists of the following four functions:

- The function `memory_alloc` reserves memory for the object on a heap, binds it to the specified consistency model and attributes, and returns a reference to the newly created object.

- The function `memory_free` frees the memory pointed to by the reference.

- The function memory_mmap creates an object as a copy-on-write mapping of the specified file. If the file is unspecified, the function creates an anonymous mapping.
- The function memory_munmap destroys an object which is a file mapping or an anonymous mapping.

Object allocation (heaps)

During object creation, the high-level object management allows an application to choose the heap on which the object will be allocated. OSS currently implements three heap allocators:

- The *page allocator* always reserves at least one hardware page (4 KB). This allocator perfectly avoids false sharing situations, but it incurs a high memory overhead for small objects.
- The *mspaces allocator* integrates the standard allocator from GNU/Linux's standard C runtime library. This allocator is very efficient in terms of memory usage, but depending on the application's object access pattern, false sharing situations can degrade performance.
- The *millipage allocator* implements the Multiview/Millipage approach avoiding false sharing [6]. This allocator tries to avoid false sharing despite low memory overhead. It uses special features from the access control module to efficiently place objects.

Access control and storage management (mmu)

The access control and storage management module abstracts from hardware and operating system features. By means of this module, consistency models access object data and keep track of object state. The following functions are related to information about objects:

- The function mmu_get_consistency_model retrieves the consistency model that a given address is bound to.
- The function mmu_is_valid_address determines whether an address is known to the local node.
- The functions mmu_set_state and mmu_get_state access an object's state as defined by the respective consistency model.

- Consistency models can store more information about an object's state using the functions `mmu_set_data` and `mmu_get_data`.
- The functions `mmu_lock`, `mmu_unlock` and `mmu_trylock` synchronize access to object metadata.

The following functions manage the physical backing store for several objects:

- The function `mmu_alloc` is called by the heap allocators to set up the physical backing store for several objects.
- The function `mmu_setup_region` sets up a page-aligned memory region at a specified address. This function applies to locally created regions as well as to remote regions.
- The function `mmu_discover_region` discovers a region using the grid memory allocator and sets it up locally.
- The function `mmu_free` frees a region from physical backing store.
- The function `mmu_foreach_page` runs a function for each page in a memory region.

The storage management functions read or write the content of an object:

- The function `mmu_copy_to_shadow` creates a backup copy for an object, whereas the function `mmu_restore_from_shadow` restores an object from a backup copy. The function `mmu_forget_shadow` discards any backup copy for an object.
- Using the functions `mmu_copyin`, `mmu_copyout` and `mmu_copyout_prefer_shadow`, a consistency model can atomically read or write an object's content.

Using the access control functions, a consistency model can request notification of read or write operations on objects:

- The function `mmu_trap_read_write` configures access control such that the consistency model is notified of reads and writes.
- The function `mmu_trap_write` configures access control such that the consistency model is notified of writes.
- The function `mmu_trap_none` configures access control such that the consistency model does not receive notifications.

Zone allocation (zone)

A zone provides backing store for object heaps. The zone allocator coordinates coarse-grained reservations among the nodes.

- The functions `allocator_set_root_memory` and `allocator_get_root-_memory` access the root memory, which is the basis of the distributed name service within OSS.
- The function `allocator_alloc` reserves a memory region in the global allocator for a node.
- The function `allocator_free` marks a memory region as unused.
- The functions `allocator_discover`, `allocator_get_size`, `allocator_get_consistency_model`, `allocator_get_allocator` and `allocator_get_owner` retrieve information about memory regions from the global allocator.
- The function `allocator_is_valid_address` checks whether an address is within the shared object space.

2.4.2 Network Communication

OSS exchanges network messages among nodes to establish and reorganize the overlay network, synchronize cached objects in memory with respect to the applied consistency model and configure nodes. The interface supports sending, forwarding and droping messages. The send functions are categorized into request and response functions, and in blocking and non blocking functions. When a send function is called, OSS builds a PDU (see figure 2.4) which is passed to the overlay network.

Request/Response messaging

In general a conversation among nodes follows a request/reply scheme in which a node sends a request to another node and awaits a response message. In case a request handler itself needs to request further data from other nodes to respond the actual request, the network module supports linking of multiple requests. In particular, if a node receives a request, it is allowed to start a further request from its request handler. The full processing of the request may be deferred until the node has processed the response of the second request. The following functions allow a request/reply comunication among nodes:

- `comm_send_sync_req_to` sends a request to nodes and blocks until the node has processed all response messages.
- `comm_send_async_req_to` is the same as `comm_send_sync_req_to` but never blocks. Response messages are processed in the background.
- `comm_send_async_linked_req_to` sends a new unblocking request from a network request handler. The actual request is linked to the new request and will be reprocessed after the node has processed the response message of the new request. A request handler which calls this function must return with `-E_PDU_LINKED`.
- `comm_send_sync_resp` sends a message in response to a request message. This function will never block.
- `comm_send_async_resp` In the face of response functions will never block, this is only an alias for `comm_send_sync_resp` to keep synchronious/asynchronious messaging semantics.

Informational Messaging

If a message exchange does not await a response, OSS supports sending one way or informational messages.

- `comm_send_async_msg_to` sends a message to nodes. This function will never block and does not await any response messages.

Messaging forwarding

OSS allows forwarding of request messages directly from the request handler of the message itself. It is allowed to forward the same message multiple times. Response messages will be sent directly to the requester without the indirection of the forwarding nodes.

- `comm_forward_req_to` forwards a request to another node.

2.5 How to implement Consistency Models

This section describes in a few steps how to implement an own consistency model in OSS. Consistency models reside in the folder `src/consistency`, new implementations should also be stored there. First of all, the developer should create a new code and header file for his implementation.

For his own implementation the developer can use the implementation of the *null consistency*[1] as a template. The header file starts with an inclusion guard macro definition which should follow the naming semantics of the already implemented consistency models. At least one declaration (the pointer table of the consistency model itself) must be placed in the header file by adding the following line, where `<NAME>` is a placeholder for the name identifying the consistency model.

2.5.1 Function Pointer Table

Every consistency model must define a function pointer table for its callback functions:

```
consistency_model_t <NAME>_consistency =
{
.name = "<NAME> consistency",  // Name
.id = oss_<NAME>_consistency,  // Internal identifier
.init = *ptr,                  // Ptr to init function or NULL
.fini = *ptr,                  // Ptr to fini function or NULL
.read_handler = *ptr,          // Ptr to read fault handler
.write_handler = *ptr,         // Ptr to write fault handler
.event_handler = NULL,         // Unused (shall be NULL)
.alloc_handler = *ptr,         // Ptr to memory alloc handler
.free_handler = *ptr,          // Ptr to memroy free handler
.mspace = NULL                 // Internally used by mspaces allocator
};
```

2.5.2 Consistency Model Registration in OSS

To make the consistency model available to user applications, it has to be registered into OSS. This is done by performing the following steps

[1]File: *consistency/nc.c* (implementation) and *consistency/nc.h* (declarations)

1. Register the consistency model in the consistency control unit[2]. This is done by appending the pointer of the consistency model's local function pointer table to the global function pointer table in the consistency control unit. Beware of reordering the table entries. Additional consistency models may only be appended to this table.

2. Add an appropriate named entry in the consistency model enumeration in OSS' global header file[3]. The entries in the enumeration must be in the same order as in the global function pointer table. The entry `oss_max_consistency` must always be the last entry in the enumeration.

2.5.3 Initializer and Finalizer

Code for preinitialization and finalization of the consistency model is placed in initializer and finalizer functions, called before the application starts and after the application terminates. These functions will be called automatically if their function pointers are added to the function pointer table.

```
static void <NAME>_init() {...}

static void <NAME>_fini() {...}
```

2.5.4 Register PDU handlers

To participate in network communication, the consistency model must register callback functions to handle network request and response messages of a specific message type. It is recommended to register two different callback handlers. The developer has to ensure to register only handlers for message types which have not been registered before, because reregistering a message type will overwrite any previous handler registration. The following steps describe the handler registration for a new message type:

1. Add a new request and an appropriate response message type to the PDU header file[4].

   ```
   #define TYPE_<NAME>_REQUEST    ...
   ```

[2]File: *consistency/consctl.c*
[3]File: *oss.h*
[4]File: *net/pdu.h*

```
#define TYPE_<NAME>_RESPONSE   (TYPE_<NAME>_REQUEST ...
```

2. Include the network communication header file into the implementation file of the new consistency model

   ```
   #include "net/comm.h"
   ```

3. Register the PDU handlers by calling the following function. It is recommended to perform the registration in the consistency model's initialization function.

   ```
   pdu_register_handler(*req_handler, TYPE_<NAME>_REQUEST);
   pdu_register_handler(*resp_handler, TYPE_<NAME>_RESPONSE);
   ```

   Registered handlers have the following signature like the example below:

   ```
   static int req_handler(in_pdu_t *pdu)
   ```

2.5.5 Writing PDU handlers

The registered request and response handlers must interprete the subtype code of the messages and delegate them to the correct subhandler. For a clean code structure it is recommmended to implement only the interpretation of subtypes in this function and delegate the messages to subhandlers. A sample implementation could look like the following code snippet:

```
switch (pdu->header.subtype) {
  case SUBTYPE_<NAME>:
    return func1(pdu);
    break;
  case ...

  default:   //ignore messages with unknown subtypes
    dbg_printf(0, "invalid pdu subtype\n");
    return PDU_SUCCESS;
}
```

Subhandlers have the same signature like registered request and response handlers.

2.5.6 PDU subsystem

It is allowed to send new network messages from network handlers. But messaging in this context is covered by the following restrictions

- no use of blocking message functions
- no request messaging from response handlers

Furthermore, network handlers must return immediately after message processing, therefore it is not allowed to block within handlers. The return code of a handler controls the postprocessing of messages. OSS supports the following return codes:

`PDU_SUCCESS` Handler processed succesfully. Request PDUs will be removed from the queue

`-E_PDU_LINKED` Request has been linked with a new request and will be deferred for later reprocessing (request PDUs only)

`-E_PDU_DEFERRED` Request has been deferred for later reprocessing (request PDUs only)[5]

2.5.7 Page Fault Handler

The page fault handlers implement the consistency model as a finite state machine on a per page basis. OSS signals access violations disjoined regarding read and write faults. The finite state machine consists of multiple page states previously defined by the developer. In conjunction with access right controlling[6] and read/write page faults, raised on access violation, the machine performs its state transitions. The read and write handlers must be added to the local function pointer table and have the following structure:

```
static void <NAME>_read_handler(void *addr,
    struct ucontext *context)
{
  int state = mmu_get_state(addr);
```

[5]This return code may be removed soon and therefore shall not be used for new handlers.

[6]Access rights are configurable for all disjunct virtual memory pages and can allow/disallow read and write access to it.

```
  switch (state) {
    case STATE_1:
      ...
      break;
    case ...

    default:
      dbg_printf(0, "address %p: unknown state detected (%u)\n",
          addr, state);
      exit(EXIT_FAILURE);
  }
}
```

The developer can perform state transitions by controlling the state of the faulted pages with the following two functions:

```
int mmu_get_state(void *addr);

void mmu_set_state(void *addr, int state);
```

Additionally, the access rights of memory pages can be modified via

```
void mmu_trap_none(void *addr);

void mmu_trap_write(void *addr);

void mmu_trap_read_write(void *addr);
```

2.5.8 Exporting functions to the API

Functions are exported to the Application Programming Interface by writing wrapper functions prefixed with `oss_`. The function declaration must be included into the global header file of OSS, included by the applications. The wrapper code must be included in the corresponding source code file[7].

[7]File: *oss.h* (declaration) and *oss.c* (implementation)

Index

Bibliography

[1] XtreemOS consortium. Deliverable D3.1.5: Third Draft Specification of Programming Interfaces. Technical report, XtreemOS consortium, 2009.

[2] XtreemOS consortium. Deliverable D3.4.3: XtreemFS and Object Sharing Service: Second Prototype. Technical report, XtreemOS consortium, 2009.

[3] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. *ACM SIGCOMM Computer Communication Review*, 34(4):15–26, 2004.

[4] 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.

[5] Felix Hupfeld, Toni Cortes, Bjoern Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. XtreemFS: a case for object-based storage in Grid data management. In *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB 2007*, 2007.

[6] Ayal Itzkovitz and Assaf Schuster. Multiview and millipage – fine-grain sharing in page-based dsms. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 215–228, Berkeley, CA, USA, 1999. USENIX Association.

[7] T.M. Kroeger. *Predicting File System Actions From Reference Patterns.* PhD thesis, University of California, 1996.

[8] T.M. Kroeger and D.D.E. Long. Design and Implementation of a Predictive File Prefetching Algorithm. *Proceedings of the General Track: 2002 USENIX Annual Technical Conference table of contents*, pages 105–118, 2002.

[9] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *Proc. of NSDI*, 2007.

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

[11] R. Srinivasan. Rpc: Remote procedure call protocol specification version 2, 1995.

[12] Jan Stender, Björn Kolbeck, Felix Hupfeld, Eugenio Cesario, Erich Focht, Matthias Hess, Jesús Malo, and Jonathan Martí. Striping without sacrifices: maintaining posix semantics in a parallel file system. In *LASCO'08: First USENIX Workshop on Large-Scale Computing*, pages 1–8, Berkeley, CA, USA, 2008. USENIX Association.

[13] Osamu Tatebe, Noriyuki Soda, Youhei Morita, Satoshi Matsuoka, and Satoshi Sekiguchi. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 2004 Computing in High Energy and Nuclear Physics (CHEP04)*, 2004.

[14] 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.