Co-funded by the European Commission within the Seventh Framework Programme

Project no. 318521

HARNESS

Specific Targeted Research Project
HARDWARE- AND NETWORK-ENHANCED SOFTWARE SYSTEMS FOR CLOUD COMPUTING

http://www.harness-project.eu/

Performance Modeling Report and Release of Data Storage Component (updated) D5.2

Due date: 30 September 2014
Submission date: 30 September 2014
Resubmission date: N/A

Start date of project: 1 October 2012

Document type: Deliverable
Activity: RTD
Work package: WP5

Editor: Christoph Kleineweber (ZIB)

Contributing partners: ZIB

Reviewers: John McGlone, Alexandros Koliousis

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	Description
0.1	2014/07/01	Christoph Kleineweber	ZIB	Outline
0.2	2014/09/08	Christoph Kleineweber, Matthias Noack, Patrick Schaefer, Thorsten Schuett	ZIB	Draft for external review
0.3	2014/09/17	Christoph Kleineweber, Matthias Noack	ZIB	Applied reviewers comments
0.4	2014/09/18	Christoph Kleineweber	ZIB	Finalised plots
0.5	2014/09/19	Christoph Kleineweber, Thorsten Schuett	ZIB	Finalised deliverable
1.0	2014/09/29	Alexander Wolf	IMP	Final Coordinator review

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T5.1	Define heterogeneous storage resources model	ZIB*, IMP, SAP
T5.2	Design/develop storage management infrastructure	ZIB*, IMP, SAP
T5.3	Design/develop heterogeneous storage virtualisation	ZIB*, IMP, SAP

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

Executive Summary

It is the main goal of the *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) project to make it easier to use heterogeneous resources in cloud platforms. These resources include technologies like *general-purpose graphics processing unit*s (GPGPUs), *field-programmable gate array*s (FPGAs), *solid-state drive*s (SSDs) and *programmable network routers*. While existing cloud platforms allow users to reserve resources, there is very limited support for users to figure out which resources to use. The HARNESS platform takes an application manifest as input and automatically figures out which type of compute, network and storage resources to allocate. For each of the three types of resources there exists a separate resource provider that tries to maximise utilisation by implementing multi-tenancy through virtualisation and by using efficient resource scheduling algorithms. This deliverable presents the storage resource provider.

Note that we have merged Deliverable D5.2 (Performance modeling report) and Deliverable D5.4.2 (Release of data storage component (updated)) into this deliverable. In D5.1, we already presented storage performance models for simple workloads: either random or sequential file access. In this deliverable, we extend these models for mixed workloads. We also show how the software, e.g. local file systems and the XtreemFS client, between the raw device and the user affects performance. For example, the disk layout and the level of fragmentation can have a huge impact on sequential throughput. These factors are influenced by the on disk file system and the way XtreemFS stores file content to it. In addition, we present the second prototype of the HARNESS storage component. Since D5.1, we have extended the scheduler, improved the performance isolation on *object storage device*s (OSDs) and implemented a new XtreemFS client. We developed a feedback system, which provides usage statistics for volumes to the HARNESS platform. This deliverable will be followed by Deliverable D5.3 after month 36.

Contents

Acronyms

AM *Application Manager*. 26, 41

API *application programming interface*. 1, 4, 5, 26, 30, 41

CBFS *Callback File System*. 4, 19, 29

CPU *central processing unit*. 33

CRS *Cross-Resource Scheduler*. 5, 7

DFE *dataflow engine*. 33

DIR *directory service*. 1, 3, 5, 19, 21

FPGA *field-programmable gate array*. i, 5

FUSE *Filesystem in Userspace*. 3, 4, 7, 19, 29–31, 37, 39

GPGPU *general-purpose graphics processing unit*. i, 5

HARNESS *Hardware- and Network-Enhanced Software Systems for Cloud Computing*. i, 1–5, 9, 11, 19, 21, 24, 27, 30, 33, 41, 42

HDD *hard disk drive*. 9–11, 21, 25, 29

IOPS *input/output operations per second*. 2, 9, 10

IRM *Infrastructure Resource Manager*. 5, 7, 41

JAX-RS *Java API for RESTful Web Services*. 5

JSON *JavaScript Object Notation*. 5

MRC *metadata and replica catalog*. 1, 3–5, 19, 21, 24, 25, 41

OSD *object storage device*. i, 1–5, 9–11, 13, 15, 19, 21, 22, 24–27, 29, 33, 34, 36, 37, 41, 42

PaaS *platform-as-a-service*. 41

POSIX *portable operating system interface*. 1, 24, 29, 33, 41

QoS *quality of service*. 4, 9, 22, 24, 37

RAID *redundant array of independent disks*. 21

REST *Representational State Transfer*. 1, 5

RTM *reverse time migration*. 11, 33, 34, 36, 41, 42

SEDA *staged event-driven architecture*. 13, 22, 26

SLO *service-level objective*. 9, 19

SSD *solid-state drive*. i, 9, 10, 13, 15, 21, 25, 29

VM *virtual machine*. 5, 7

XtreemFS-IRM *XtreemFS Infrastructure Resource Manager*. 5, 7, 19

ZIB *Konrad-Zuse-Zentrum für Informationstechnik Berlin*. 3, 11, 31

1 Introduction

The *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) cloud provides a platform layer that is able to orchestrate a set of heterogeneous infrastructure resources for different cloud applications. Resource management is separate for computational, networking, and storage devices. This deliverable provides an update on the performance behaviour of different storage device types for different workloads (Task T5.1), which includes mixed access patterns, *in-memory* storage and reliability considerations. Furthermore, we provide an updated description of the storage component in the HARNESS prototype architecture, (Task T5.2 and Task T5.3), which is an extension of the XtreemFS object-based file system [9, 17].

The present cloud storage market offers a wide range of products with different interfaces and consistency guarantees. A very simple and flexible class of cloud storage are (virtualised) block devices that allow users to run an arbitrary (distributed) file system on top. While this type of cloud storage allows a maximum flexibility, it requires additional effort by the user to maintain a file system on top of the block storage layer. A commercial product of this category is Amazon's Elastic Block Store (EBS)[1].

Another type of cloud storage are object stores such as Amazon's Simple Storage Service (S3)[2]. Objects stores offer the possibility to store an arbitrary content with a given identifier. Such object stores are usually accessible via a *Representational State Transfer* (REST) interface and offer data access at a high abstraction level. As object stores are not *portable operating system interface* (POSIX) compliant, using this type of cloud storage requires usually the adaptation of applications to proprietary *application programming interface*s (APIs).

A third type of cloud storage are databases. The commercial cloud market offers a wide range of cloud databases with different interfaces and consistency models, beginning at relational databases (RDBMSs) with a strong transactional consistency like Amazon's Relational Database Service (RDS)[3] to NoSQL databases with a relaxed consistency model like Amazon's DynamoDB[4].

Most of these cloud storage products use vendor specific interfaces and thus require to adapt cloud applications to a particular environment. Also the capabilities in terms of performance guarantees are limited. The storage component of HARNESS provides a POSIX-compliant file system. POSIX compliance has the advantage that a wide range of applications is supported without modifications. Furthermore the interface abstracts from the physical hardware and, thus, it enables the use of heterogeneous storage devices.

XtreemFS is an object-based file system [11], that is well scalable in terms of capacity and throughput and, thus, it is well suited for cloud environments that host a large number of users with high capacity and high throughput storage requirements. XtreemFS consists of a *directory service* (DIR) that tracks status information of all system components, *object storage device*s (OSDs) that store file contents, and *metadata and replica catalogues* (MRCs) that store metadata about files.

[1] Online at http://aws.amazon.com/ebs

[2] Online at http://aws.amazon.com/s3

[3] Online at http://aws.amazon.com/rds

[4] Online at http://aws.amazon.com/dynamodb

In HARNESS, we aim to extend XtreemFS to provide performance guarantees for an application, building upon the already inherent reliability of the storage system. The HARNESS storage component provides logical file system volumes with either sequential or random throughput and capacity guarantees on a shared infrastructure. Throughput is defined in MB/s for sequential access and *input/output operations per second* (IOPS) with a size of 4 KB per request for random access.

The architecture of the storage component extends XtreemFS in three major aspects. The first extension is a reservation scheduler, that places logical file system volumes on XtreemFS OSD servers. This part was implemented in the first year and presented in Deliverable D5.4.1 [6]. The second major extension is throughput and capacity reservation enforcement for logical volumes on OSDs. This part is described in chapter 5 of this deliverable. The third HARNESS contribution to XtreemFS is a performance profiling tool that measures the performance behaviour of XtreemFS OSDs in a heterogeneous infrastructure and provides this knowledge to the reservation scheduler. The performance profiling tool has been developed during the first year of the project, but its integration with the reservation scheduler is described in this document.

The remainder of this deliverable is structured as follows. Chapter 2 describes the XtreemFS open source project and marks out the contributions made by HARNESS. We discuss the integration of this storage component with the overall HARNESS architecture in Chapter 3. Chapter 4 presents device and performance models that have not been covered previously by Deliverable D5.1 [5]. Chapter 5 presents an updated architecture of the storage component and the current state of the prototype. We present our evaluation in Chapter 6. Chapter 7 discussed future plans for Year 3 and concludes.

2 Background

The storage component of HARNESS is based on the object-based file system XtreemFS [18, 9]. This chapter presents an overview of XtreemFS and the contributions made by HARNESS.

2.1 Current State of XtreemFS

The development of XtreemFS was initiated by the *Konrad-Zuse-Zentrum für Informationstechnik Berlin* (ZIB) and funded by the EU projects XtreemOS[1] and Contrail[2] and the German projects MoSGrid[3], "First We Take Berlin"[4] and FFMK[5].

XtreemFS implements a POSIX compliant, object-based file system architecture [11].[6] The source code is published under the BSD license. XtreemFS evolved from a research prototype to an open source project with a growing user community during the last years. An XtreemFS cluster consists of three different kinds of services: one DIR, several MRCs and several OSDs. In a subproject, we develop BabuDB[7] [19]. BabuDB is a non-relational database used as the datastore for the MRC.

The focus of the existing implementation is on deployments that span across multiple datacenters and on files that can be replicated. This is the reason why XtreemFS is partition-tolerant and consistent in terms of the CAP theorem [7]. It supports multiple replication policies for different use cases, integrated transport level encryption and features to overcome high latencies like caching and asynchronous write operations.

Before HARNESS, XtreemFS already had multi-tenancy capabilities by creating multiple logical volumes in an XtreemFS cluster. Each logical volume has its own namespace while multiple volumes may share a set of OSDs. Volumes with overlapping sets of OSDs potentially influence the performance of each other. All requests are handled equally. Additionally, there is no mechanism to manage the available capacity. Any users can claim the available space.

The set of OSDs used for a volume is determined by an OSD selection policy. These policies are an mechanism to define where files and their replicas are stored. Users can implement their own policies or use existing ones. XtreemFS already provides a few policies, like selection at random, based on the distance to the client, or a user defined OSD list.

XtreemFS offers C++ and Java libraries (*libxtreemfs*) for client access and management tasks. Existing XtreemFS clients and management tools are built on top of these libraries. XtreemFS has *Filesystem in Userspace* (FUSE) based clients running on Linux, Mac OS X and FreeBSD. The Windows client is

[1] Online at http://www.xtreemos.eu/
[2] Online at http://contrail-project.eu/
[3] Online at https://mosgrid.de/
[4] Online athttp://www.zib.de/en/das/projects/details/article/first-we-take-berlin.html
[5] Online at http://www.zib.de/en/das/projects/details/article/labimif-kopie-1.html
[6] Online at https://github.com/xtreemfs/xtreemfs
[7] Online a https://github.com/xtreemfs/babudb

based on *Callback File System* (CBFS). Additionally, an adapter to use XtreemFS in a Hadoop cluster is available. The adapter can be used to replace HDFS with XtreemFS (see also Section 5.7).

2.2 XtreemFS extensions in HARNESS

XtreemFS has been extended in HARNESS to ensure *quality of service* (QoS) guarantees for a logical file system volume. We extend XtreemFS by the following components to achieve this goal.

- The first part is an OSD performance profiler that determines the performance capabilities for all available OSDs in a multi-tenant scenario. This benchmark is based on the Java client. It does not require any server side code modifications.
- The second component is the reservation scheduler. Based on the performance profiles and the available capacity of the OSDs, it places logical volumes on a set of OSDs. Typically requests are received from the HARNESS platform. The schedule is communicated to the MRC and OSDs. MRC receives an OSD selection policy, which limits the volume to the selected OSDs. The scheduler is build as an independent service which uses existing APIs. It could also be used with the original XtreemFS services without modifying them.
- The third contribution by HARNESS is the enforcement of reservations. It is covered in Chapter 5. The benchmark and the reservation scheduler use existing interfaces and do not require any modifications to existing XtreemFS components. The reservation enforcement is extends the XtreemFS MRCs and OSDs.

We continue to develop the XtreemFS Hadoop adapter in HARNESS to be compatible to Hadoop 2.x. The Hadoop integration is required as the AdPredictor demonstrator application has a MapReduce implementation using Hadoop. Supporting Hadoop 2.x makes XtreemFS available for a huge set of additional applications that use the YARN [20] platform. We furthermore develop a new client that can be linked to an application using LD_PRELOAD. LD_PELOAD is a mechanism to link a library dynamically to an existing application. This client can access XtreemFS where FUSE is not available or due to performance issues not an option.

3 Heterogeneous Cloud Platform: Storage

A central component in HARNESS is the *Cross-Resource Scheduler* (CRS) which is in charge of handling resource provisioning requests. In particular, it requests a group of heterogeneous resources, with optional placement constraints between resources. It delegates the actual provisioning of the resources to corresponding *Infrastructure Resource Manager*s (IRMs), which are in charge of managing specific types of heterogeneous resources, including *virtual machine*s (VMs), *general-purpose graphics processing unit*s (GPGPUs), *field-programmable gate array*s (FPGAs), storage and network devices.

One of the key design features of the HARNESS platform is its extensibility to new forms of heterogeneity. This is achieved by incorporating dedicated IRM components into the platform, allowing existing heterogeneous and specialised resources to be seamlessly managed by the HARNESS platform. We focus on the *XtreemFS Infrastructure Resource Manager* (XtreemFS-IRM) which manages storage resources for the CRS.

In HARNESS, a typical XtreemFS installation consists of the XtreemFS-IRM, a Reservation Scheduler, a DIR, and multiple OSDs and MRCs (Figure 3.1). XtreemFS uses a proprietary protocol based on Google Protocol Buffers [1] for the communication between its services. The XtreemFS-IRM provides a REST based interface which allows to interact with the scheduler of an XtreemFS installation. The XtreemFS-IRM itself is stateless. Its only task is to translate REST calls into calls to *lixtreemfs*. The communication between the XtreemFS-IRM and the XtreemFS Reservation Scheduler can be secured with X.509 service certificates. X.509 certificates offer authentication and transport layer security.

The CRS component is the only user of the XtreemFS-IRM. It uses the XtreemFS-IRM to retrieve the available storage capacity and to reserve and release storage reservations.

The XtreemFS-IRM is implemented as a *Java API for RESTful Web Services* (JAX-RS) service which consumes *JavaScript Object Notation* (JSON) streams. Jackson[2] is used to marshal Java objects to JSON and unmarshal JSON streams to Java objects. It is deployed using a webserver like Apache Tomcat. The XtreemFS-IRM is implemented as a stateless service. Tomcat provides multithreading and isolates concurrent TCP requests from each other, like concurrent resource reservation requests by the CRS.

The specification of the RESTful API of the XtreemFS-IRM consists of calls to reserve, verify, list, calculate and release XtreemFS storage reservation:

getResourceTypes(): Returns the types of supported resources and their attributes.

calculateResourceCapacity(): Calculates released/reserved capacity of a resource for the CRS scheduling phase (prepareReservation).

calculateResourceAgg(): Calculates aggregated resource capacity for the CRS scheduling phase (during prepareReservation).

[1] Online athttps://code.google.com/p/protobuf/

[2] Online at http://jackson.codehaus.org/

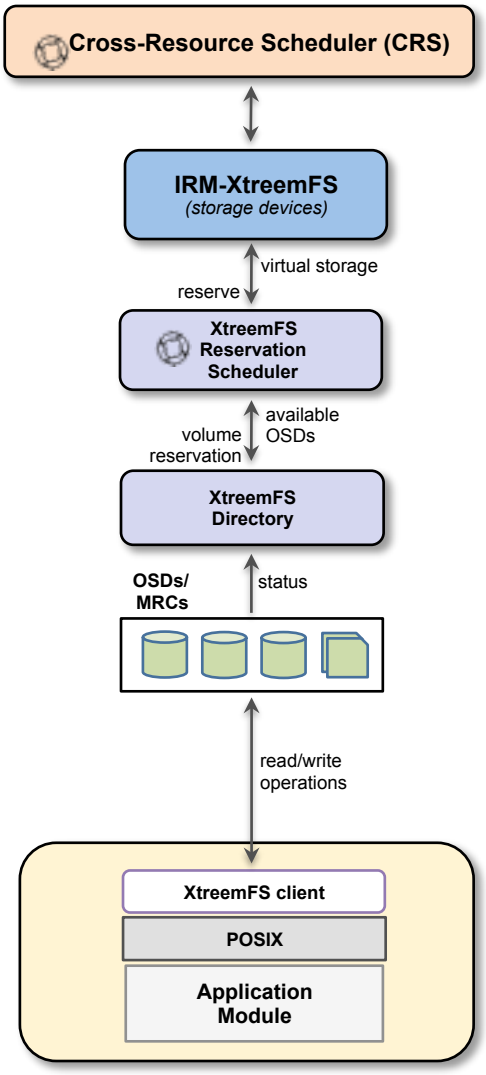


Figure 3.1: XtreemFS Harness Architecture

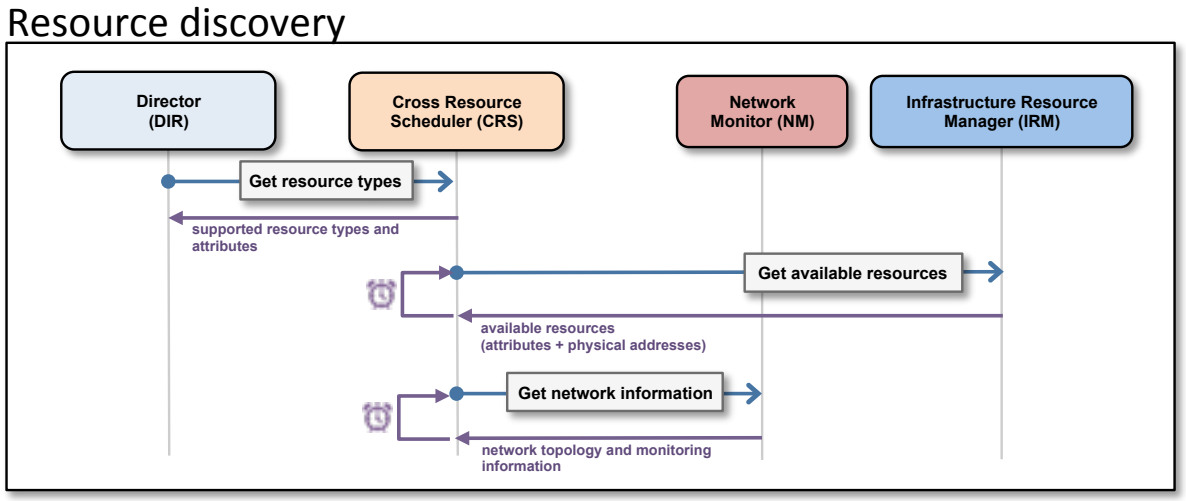


Figure 3.2: Resource discovery.

getAvailableResources(): Returns a list of resources that are available for reservation.

reserveResources(): Reserves a group of resources of specific type (as defined by the IRM).

verifyResources(): Checks if an infrastructure resource reservation is ready for launching the application.

releaseResources(): Releases resource reservation after the application terminates.

The XtreemFS-IRM provides two calls for resource discovery (Figure 3.2): (a) *getResourceTypes* and (b) *getAvailableResources*. In a typical workflow the CRS calls the XtreemFS-IRM using the *getAvailableResources* to obtain the list of resources for the following scheduling phase.

The XtreemFS-IRM provides multiple calls for application deployment (Figure 3.3): *calculateResourceAgg(), calculateResourceCapacity(), reserveResources(), verifyResources(), releaseReservation()* and *releaseResources()*. During the scheduling phase, the CRS can make multiple calls to *calculateResourceAgg()* and *calculateResourceCapacity()* to verify the remaining resources after a reservation.

After the XtreemFS-IRM has finished the scheduling phase, it reserves (*reserveResources()*) the scheduled XtreemFS resources, verifies that these have been reserved *(verifyResources())* and finally releases the resources using *releaseReservation()* after execution.

An application does not interact with the XtreemFS-IRM. Instead the platform directly mounts the reserved XtreemFS volume using an XtreemFS client into a VM. The FUSE client provides a POSIX interface. Files are read and written to using the mounted XtreemFS volume. In addition, applications can access the file system directly using *libxtreemfs* or the Hadoop adapter.

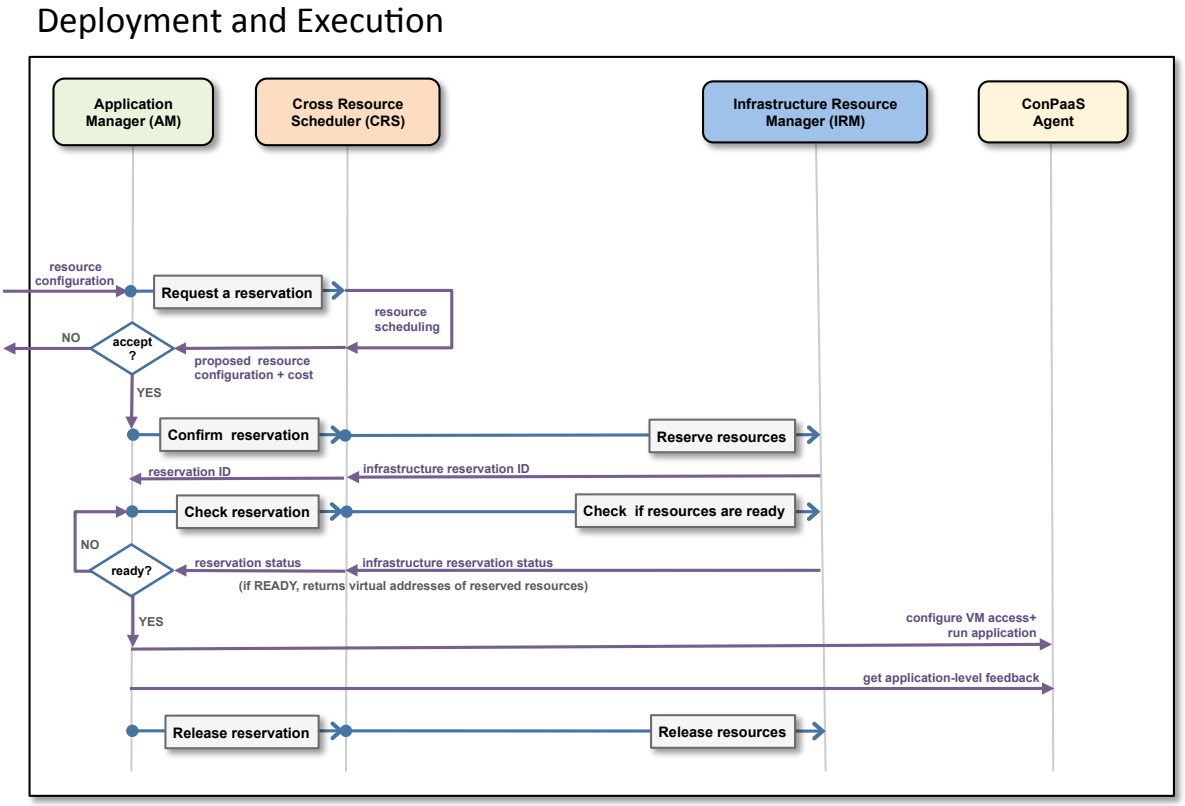


Figure 3.3: Deployment and execution.

4 Device and Performance Models

Understanding the performance behaviour of storage infrastructure is essential to build a cloud platform that offers the possibility to formulate QoS objectives. We use empirical models to describe the behaviour of XtreemFS OSDs in the HARNESS cloud, due to the complexity of modern storage devices, which may be composed of multiple physical devices, different cache layers and a software stack. Our performance models have to cover all kinds of devices and different access patterns in a multi-tenant scenario.

We presented device and performance models for sequential and random access on *hard disk drive*s (HDDs) and *solid-state drive*s (SSDs) in Deliverable D5.1 [5]. Our empirical models show that the performance of sequential access decreases with an increasing number of concurrent readers on HDDs while write access is not slowed down by multi-tenancy. This observation was not confirmed for sequential access on SSDs where the performance was independent from the number of tenants. Furthermore, our evaluation has shown that the random throughput of an XtreemFS OSD is independent of the number of concurrent readers or writers for both, HDDs and SSDs.

In this year, we develop a scheduling algorithm for storage reservations that has been implemented in the first year of the HARNESS project and described in Deliverable D5.4.1 [6]. We develop advanced device models that cover the effects of mixing workloads on rotating HDDs and consider the impact of the block placement on a disk. Additionally, we introduce a new class of storage device, namely *in-memory* OSDs, and reliability as a new *service-level objective* (SLO).

4.1 Mixed Workloads

In the reservation scheduling algorithm developed during the first year of the HARNESS project, we followed the strategy to place random access and sequential access workloads on separate OSDs. Mixing workloads on individual OSDs would result in a wider range of possibilities for volume placement. This could lead to an improved resource usage and reduce costs for cloud providers. Mixing sequential and random access inside a volume might be beneficial for users. Many applications do not have a purely sequential or purely random access pattern. Using this kind of application might break the contract with the cloud provider or might require the reservation of unnecessarily high throughput when sequential access is formulated as an IOPS reservation.

We analysed the performance of mixed workloads on a single HDD by running one volume with random read access and a varying number of volumes with sequential read access. Figure 4.1 shows the summed throughput of the sequential access volumes, depending on the number of active readers. We compare the throughput of mixing sequential and random access (*MIXED_READ*) to the pure sequential throughput (*SEQ_READ*, we already presented in Deliverable D5.1 [5].The throughput of the random access volume is less than 1 MB/s, which is negligible small and thus not considered in the plot. The plots show the average of three runs, the error bars the minimum and maximum throughput.

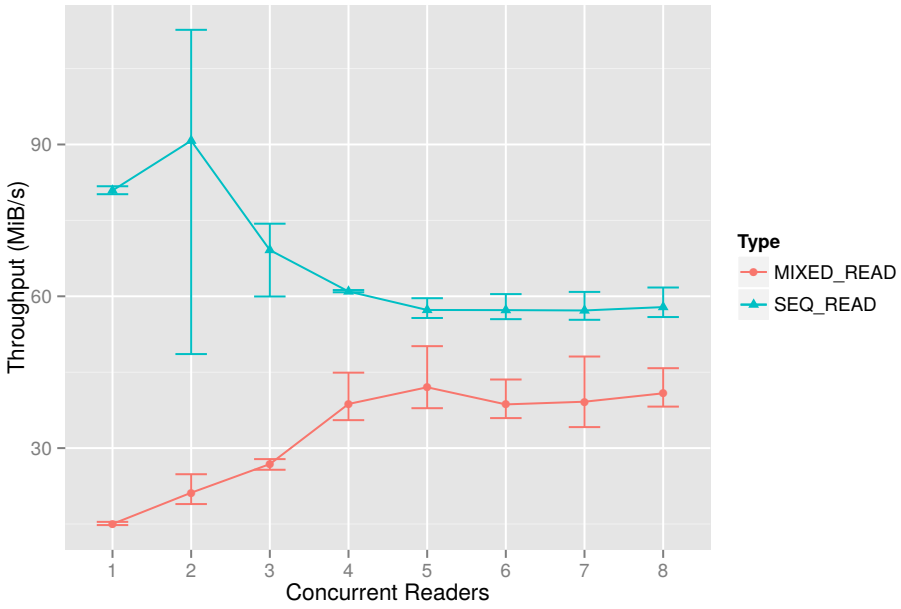


Figure 4.1: Mixed read access on HDDs

Compared to the benchmarks we presented in Deliverable D5.1 [5], the sequential throughput of an OSD serving mixed access patterns is significantly below the performance of an OSD serving purely sequential access.

Mixing sequential and random workloads on OSDs requires an OSD model that respectively considers the active reservations of the other workloads. For instance, in case of an OSD already serving a random access reservation, the available sequential throughput would be reduced by the random access reservation. We cope with this issue by converting a sequential throughput reservation that has to be placed on an OSD, which already serves random access reservations into 4 KB IOPS. Our evaluation in Deliverable D5.1 [5] shows that this conversion is only applicable to SSDs due to the bad random access performance of HDDs.

The random access performance of modern SSDs is very close to its sequential throughput. For instance, the Samsung 840 Pro devices we used for our evaluation allow a sequential read throughput of 540 MB/s and a random throughput of 100,000 IOPS. Converting IOPS into throughput would result in a 400 MB/s. In Deliverable D5.1 [5], we have shown that the random throughput on this type of SSD used with XtreemFS is significantly below the raw device performance. The reason for this is additional latency of the network and the additional software stack of the XtreemFS OSD. The evaluation of our reservation scheduling algorithm [10] has shown that mixing workloads under the given constraints is generally possible, but does does not save any resources as long as an XtreemFS cluster has free resources. However, this extension of the reservation scheduler prevents rejecting reservations when the OSDs runs out of resources.

4.2 Performance Impact of Disk Fragmentation

The performance models we developed during the first HARNESS year covered multi-tenancy and different access patterns. The experiences that have been made while operating the HLRN supercomputer[1] at the ZIB have shown that the performance of the Lustre [1] parallel file system, which is part of the HLRN, fluctuates significantly.

The fluctuation in the file system performance is caused by two characteristics of rotating HDDs [13]. First, the performance on the outer tracks of a disk is higher than on the inner tracks. This is caused by the larger number of sectors, which pass the read/write heads during a single rotation of the disk, on the outer tracks. File system implementations usually try to allocate the outer tracks first. Hence, the performance of a HDD managed by a file system like *ext4* drops over time when a disk runs out of the fast outer tracks.

The second reason for performance fluctuations are fragmented block allocations. Local file systems like *ext4* allocated disk space in fixed sized blocks, usually having a size of 4 KB. Sequential access performance to a file is optimal only if its file system blocks are mapped to a contiguous set of device blocks. During the lifetime of a file system, files are created, deleted, and mapped to different positions of the used block device. Deleting files causes a fragmentation of the device blocks and results in a state where large files cannot be mapped to a contiguous sequence of device blocks. The probability of this happening increases in time, occupied capacity, and file size. A sequential read or write access to a file that is mapped to a non-contiguous sequence of device blocks results in additional disk head movements an thus a performance drop.

To demonstrate the impact of the allocated disk capacity and the performance difference of inner and outer tracks, we ran sequential write benchmarks at a varying level of disk space usage on two different types of disks. Figure 4.2 shows the sequential write throughput on an *ext4* file system using a Seagate ST9250610NS. The allocated capacity is varied between 10 and 160 GB. We repeated the same experiment on a SATA disk with a capacity of 2 TB. The result is shown in Figure 4.3. The plots show the average of three runs, the error bars the minimum and maximum throughput.

The experiments demonstrate that the performance difference between a nearly empty disk and a highly utilized disk is significant.

4.3 In-Memory Storage

We introduce a new type of storage device, named *in-memory* OSD. All files stored on a volume of *in-memory* OSDs are stored in the main memory of the used machines without any persistence. Modern servers have memory capacities in the range of multiple terabytes. This is adequate to fulfil the storage requirements of many applications. As the performance of I/O bound applications can be improved significantly by fast storage, using in-memory file systems is adequate as long as reliability requirements are fulfilled. Reduced reliability is in particular acceptable for batch applications like the *reverse time migration* (RTM) or AdPredictor demonstrators of the HARNESS cloud. The reliability of *in-memory* storage can be improved by using replication. We analyse this relationship in Section 4.4. Providing main memory via a file system interface has the advantage that existing applications do not have to be modified and memory can be shared between multiple compute nodes. We describe the prototype implementation of our *in-memory* OSD in Section 5.6.

[1] Online at https://www.hlrn.de

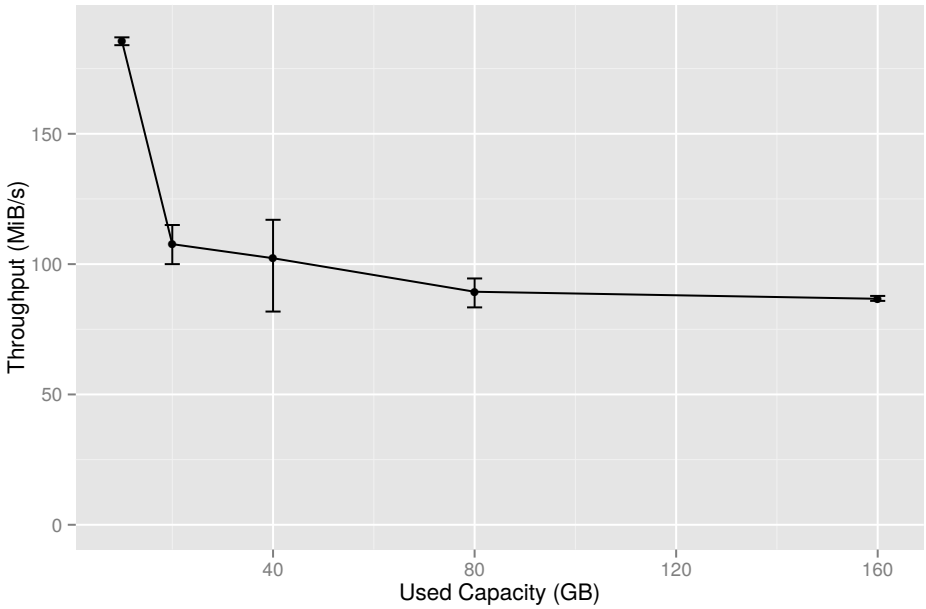


Figure 4.2: Sequential throughput depending on allocated capacity on a 250 GB disk

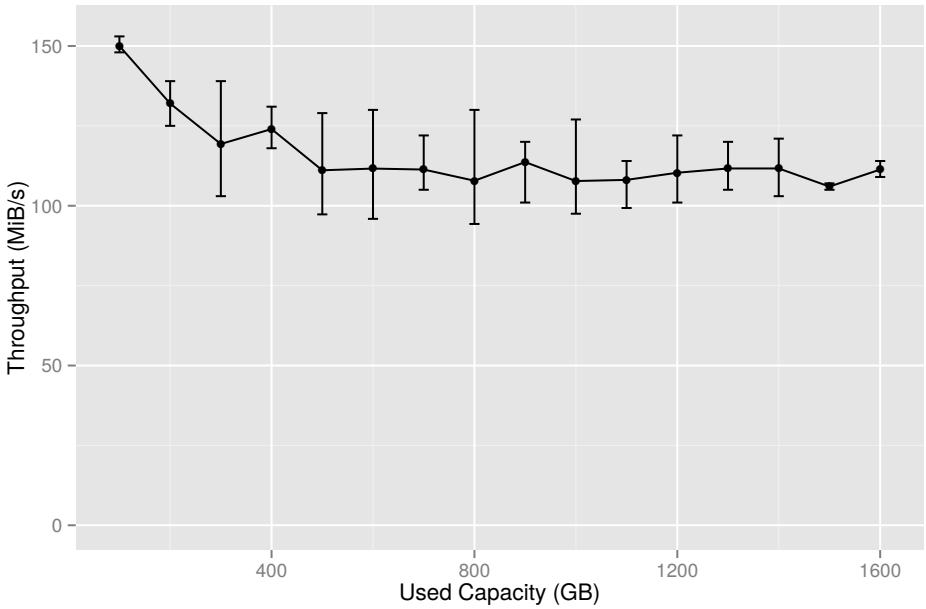


Figure 4.3: Sequential throughput depending on allocated capacity on a 2 TB disk

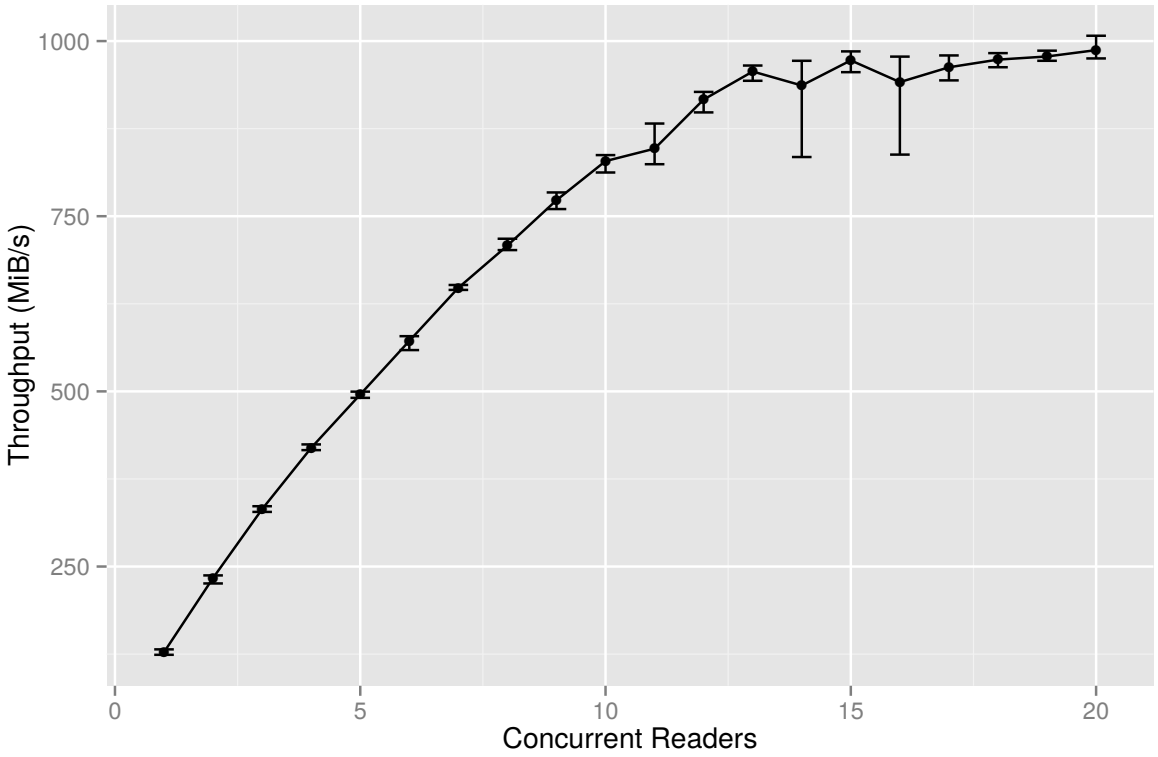


Figure 4.4: In-memory OSD access over 10GbE

The *staged event-driven architecture* (SEDA) architecture of the XtreemFS OSD consists of multiple stages, which each request has to pass. This causes a considerable latency for processing a request. To get a good performance out of an *in-memory* OSD, a certain level of parallelism generated by the clients is necessary. The most time consuming part of the request processing on the OSD side is the *StorageStage*, which accesses the underlying local storage system or memory management system for *in-memory* OSDs. We make use of multiple storage threads, which have been introduced with XtreemFS 1.5 for efficient SSD usage. Handling file access is distributed by means of a hash function used to map files to storage threads. Each file is served by one particular thread. This approach avoids locking, but requires a certain level of parallelism to get an optimal throughput.

Figure 4.4 shows the sequential write throughput of an XtreemFS OSD using *in-memory* storage. The x-axis shows the number of concurrently written files, while each file is served by its own storage thread. Client and OSD have been connected via a 10 GBit/s Ethernet connection during this experiment. The plot shows eventually, the 10 GBit/s link becomes the bottleneck for the OSD throughput.

In order to eliminate the network as limiting factor, we repeated the same experiment running the XtreemFS client and OSD on the same machine. Figure 4.5 shows that the throughput of an OSD can be improved beyond 10 GBit/s with a faster network as long as a sufficient level of parallelism is generated by the clients.

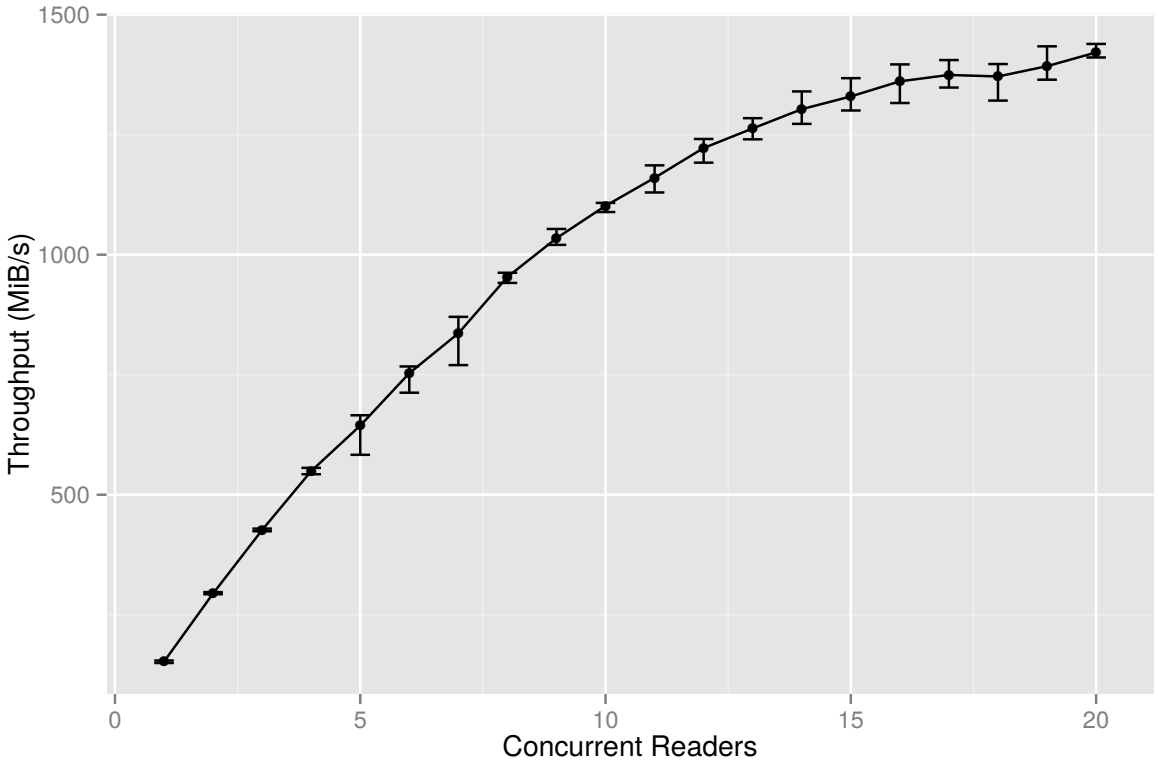


Figure 4.5: In-memory OSD access with local client

We can conclude that a certain level of request parallelism is necessary to achieve a good throughput on high performance storage devices. This requires a workload spread over multiple files on the client side and a parallel request processing on the OSD side. We will make use of these insights to also improve the performance of OSDs using an SSD.

4.4 Reliability of Storage Volumes (replication and erasure codes)

4.4.1 Definitions

In this work, we focus on reliability in the sense of durability of data storage, in contrast to availability, the probability that data is accessible.

Definition 1. The reliability $R(t)$ of a system is the probability that the system survives until some time t. Thus, $R(t) = P(L > t)$ with L as the lifetime of the system, can be understood as the probability that the system is operational for at least a time t.

One assumption is that the system is working at start time $t = 0$, $i.e.$, $R(0) = 1$. $R(t)$ is also called the survival function. In the scope of storage systems, it is the probability, that no data loss events occur during time t. Contrary, the unreliability $F(t) = 1 - R(t)$ is the probability that data loss events occur during the mission time. In the following we use $SDF(t)$ (Single Disk Failure) as the unreliability of a single disk.

Different methods can be used to determine data storage reliability:

1. One possibility is a *real world analysis*. For this purpose, data is recorded in a large system over a long period of time, in order to derive failure probabilities and predict system behaviour. Real world data analysis is the most realistic way of a reliability analysis. But the efforts are high. For example, the very long duration of the analysis is necessary, especially for events with a low probability.

2. *Simulation* is another method for reliability analysis. On one hand, it is flexible in the type of system that can be simulated and faster than real world data recording. On the other hand, a very accurate simulation of complex systems can also last very long. Thus, most simulations need further optimisations to reduce the execution time.

3. *Analytical approaches* are the most flexible method for assessing the reliability of a system, for instance Markov chains. A variety of system types can be expressed and the calculation is faster compared to simulations.

To model reliability in XtreemFS, we focus on real world data analysis. The analysis of a production system provides realistic information for a certain system type. But the availability of data from real systems, especially for large installations is a problem. Large-scale reliability studies of real systems have been presented in [8, 12, 15, 14]. The study in [12] is based on data from large production systems at Google for high-performance computing and internet service sites. Their analysis shows that the real disk *mean time to failure (MTTF)* is worse compared to data sheet MTTF and real disk replacement rates are higher than expected. The Table 4.1 summarizes Google's real world results from 100,000 hard drives operated in their data centers. Some sources report that SSDs are more reliable in terms of the MTTF

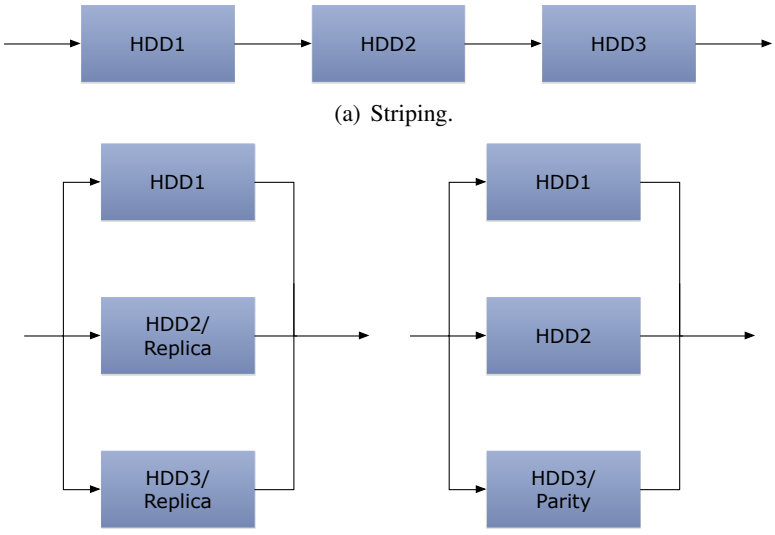
Years of Operation	Approx. Failure Percentage
1	~2%
2	~8%
3	~9%
4	~6%
5	~7%

Table 4.1: Google Hard Drive Failure Statistics [12].

with 1,000,000 hours compared to HDDs with a MTTF of 600,000 hours. However, Google pointed out that the actual failure rates are higher than those published by the vendors.

4.4.2 Reliability in XtreemFS

XtreemFS has three operational modes that affect reliability:



(a) Striping.

(b) Replication.

(c) Erasure Coding.

Figure 4.6: XtreemFS operational modes.

1. *Striping*, Figure 4.6(a): Striping stores the data of each file evenly across n OSD services. This mode does not provide any fault tolerance or redundancy. When a single service crashes, the data is lost and not accessible anymore. The probability of data loss in striping scenario with n disks is given by:

$$F(t) = 1 - (1 - SDF(t))^n$$

where SDF(t) is the percentage of a single drive failure at time t. The SDF can be estimated using the numbers given in Table 4.1 or the MTTF given by the hard disk vendors.

2. *Replication*, Figure 4.6(b): Replication stores n identical replicas of every data block on n OSDs. Replication can tolerate up to $n - 1$ disks to fail. If all n disks fail, all data is lost. The probability of data loss in a replicated scenario is given by:

$$F(t) = SDF(t)^n$$

3. *Erasure Coding*, Figure 4.6(c): Erasure coding distributes the data across n OSD services of which k are redundancy blocks. Thus, erasure codes tolerate up to k simultaneous failures, i.e. the data is lost if $k + 1$ services fail. The probability of data loss for erasure coding is given by:

$$F(t) = \sum_{j=k+1}^{n} \left(\begin{array}{c} n \\ j \end{array} \right) \cdot (1 - SDF(t))^{n-j} \cdot SDF(t)^j$$

4.4.3 SMART Status

SMART (Self-Monitoring, Analysis, and Reporting Technology) captures drive error data to predict failures far enough in advance, so that measurements can be taken in advance. However, SMART focuses on mechanical and surface errors, while a high percentage of disk drive errors is of electronic nature. Thus, SMART misses sudden drive failure modes due to power component failures. A Google team discovered [12] that 36% of all failed HDDs didn't show a single SMART-monitored failure. Google concluded that SMART-monitoring is almost useless for predicting the failure of a single drive. That's why we use the SMART status only to detect and exchange defective HDDs. Other than that, we do not use the SMART status for reliability.

4.4.4 Use Cases

Given a usage pattern that requires a specific reliability, the required setup can be computed based on the formulas given above.

- Use case 1: Suppose we have HDDs with a maximum age of 1.5 years. Assuming we need a reliability of 80% and the job runs for 6 months. We can to use *striping* to speed up disk accesses. The SDF of these disks is given by approximately 8% in the second year of operation:

$$\begin{aligned} R(2y) &= 1 - F(2y) \\ &= (1 - SDF(2y))^n \\ &= (1 - 0.08)^n \\ &= (0.92)^n \\ \Rightarrow \quad n &\leqslant 2 \end{aligned}$$

  i.e. we can use at most 2 HDDs with striping enabled.

- Use case 2: Suppose we have HDDs with a maximum age of 0.5 years. Assuming we need a reliability of 99% and the job runs for 1 month. We need to use *replication* to obtain this reliability. The SDF of these disks is given by approximately 2% in the first year of operation:

$$\begin{aligned} R(1y) &= 1 - F(1y) \\ &= 1 - SDF(1y)^n \\ &= 1 - 0.02^n \\ \Rightarrow \quad n &\geq 2 \end{aligned}$$

  i.e. we need a replication factor of at least 2 (2 HDDs).

5 Prototype of the Storage Component

This chapter presents the current prototype of the storage component in the HARNESS cloud.[1] The storage component is implemented by a distributed file system with a POSIX compliant interface based on an extension of XtreemFS. XtreemFS is integrated with the HARNESS platform via the XtreemFS-IRM (Chapter 3).

This Chapter starts with an introduction to the XtreemFS architecture and a summary of the prototype developed in the first year of the HARNESS project. We continue with the contribution of the second year.

5.1 Architecture

5.1.1 XtreemFS Architecture

XtreemFS has a typical object-based file system architecture [11], consisting of MRCs, OSDs, a central DIR, and clients. Figure 5.1 presents an overview of the roles of the XtreemFS components. The DIR is the entry point for all other services and clients. Each service registers at the DIR and sends periodic heartbeat messages to indicate their own status. The MRC servers are responsible for metadata management, i.e. maintaining the file system hierarchy, file attributes and the data location of files. The MRC selects a subset of the OSDs for new files. The MRCs are responsible for client and user authentication and issue an *XCap* object to the client that is used for the authorization at the OSDs. The OSDs store the file content in the form of objects and handle file replication. Files can be striped over multiple OSDs or stored on a single OSD in a single object. After the initial authorization, clients interact directly with the OSDs and can access files in parallel on multiple OSDs.

XtreemFS has different client implementations that are based on a common client library named *libxtreemfs*. The client library is available in *Java* and *C++* variant. The *C++* library is used to implement the Linux client using FUSE and the Windows client using the CBFS interface. Furthermore we implemented a new client that can be linked directly to an application. FUSE requires a system call for every operation which can reduce performance. Our new client can avoid this overhead. This client is presented in Section 5.7. The Java *libxtreemfs* is used to implement internal maintenance tools and the *Hadoop* integration.

5.1.2 HARNESS Extensions

The storage prototype of the HARNESS cloud enhances XtreemFS in two aspects, reservation scheduling and reservation enforcement. Figure 5.2 illustrates the resulting interactions of the enhanced system.

XtreemFS can already provide virtualised file systems implemented by logical volumes. With HARNESS, we can now offer volumes with an SLO. An SLO contains a capacity requirement, a

[1] Available for download at https://github.com/xtreemfs/xtreemfs/archive/D5_2.tar.gz

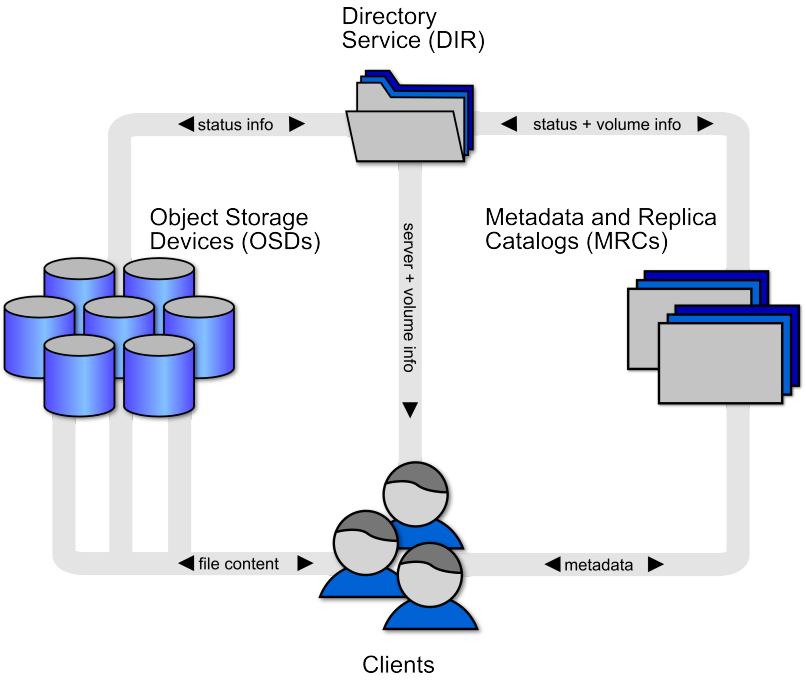


Figure 5.1: XtreemFS Architecture

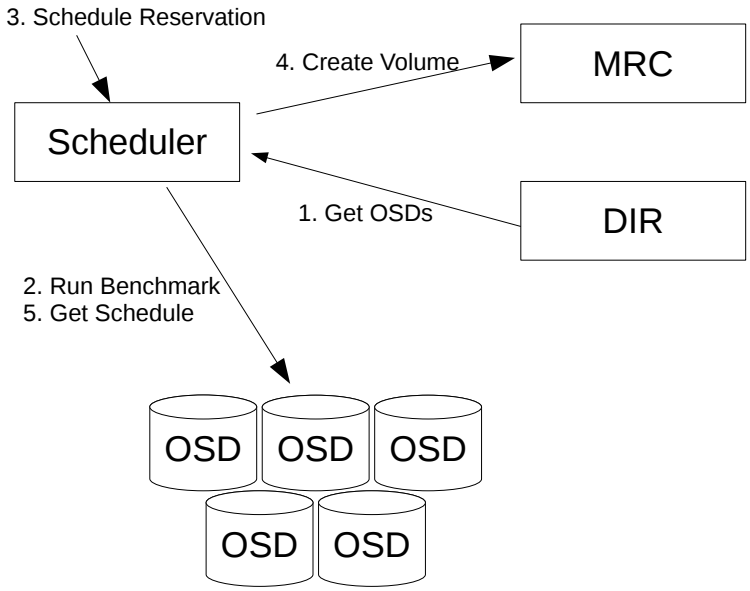


Figure 5.2: HARNESS Extensions

performance requirement and the access pattern. Currently, we support sequential and random access patterns. Sequential access requires a performance description in MB/s. Random access patterns require a performance description in IOPS, i.e. requests with a size of 4 KB at a random position of the file system. We currently do not distinguish between read and write performance.

As presented in Figure 5.2, the new scheduler service requests periodically all available OSDs from the DIR. The response from the DIR also contains information about the available capacity. On each new OSD, the scheduler service runs a profiler to determine the maximum sequential and random bandwidth the OSD can serve. Users can create reservations after the scheduler collected performance profiles for a set of OSDs that serves the reservation. Details of the reservation scheduling are summarized in Section 5.2. The reservation scheduler creates a logical volume for each scheduled reservation on an MRC and communicates the schedule to the MRC by setting an OSD selection policy. After the volume has been created, the reservation can be used by clients by mounting the volume. Different reservations may occupy different amounts of resources on one OSD. To allow an OSD to provide exactly the reserved reservations in terms of capacity and performance to each volume, an OSD requests all reservations that are scheduled to that particular OSD from the scheduler, when the OSD gets a request to an unknown volume. This approach avoids a persistent storage of the schedule on OSDs. Deleted reservations are automatically dropped from the OSDs when a new reservation is scheduled to the OSD and a client sends a first request.

5.2 Resource Reservation

The reservation scheduler service has to find a subset of the OSDs that can serve the necessary resources that a reservation requires. We presented a scheduling algorithm that breaks down the reservation scheduling to a multi- dimensional bin packing problem in the first year of the HARNESS project [10, 5, 6].

An XtreemFS cluster can consist of a heterogeneous set of OSDs, each having different performance and capacity capabilities due to different storage hardware. Possible device types are HDDs, SSDs, a *redundant array of independent disks* (RAID), or hybrid machines, e.g., having a HDD and an SSD cache (see Figure 5.3). The reservation scheduler gets a sequence of incoming reservations, containing resource requirements in multiple dimensions. The possible dimensions are sequential throughput, random throughput, and capacity. The scheduler has to find immediately a valid schedule, i.e. a set of OSDs that can provide the required resources. Reservations are not time dependant: the requested resources have to be provided until a reservation is deleted by a user or the cloud platform.

Multiple reservations can share one OSD and reservations can be striped over a set of OSDs if a single OSD cannot fulfil the requirements. The overall objective of the reservation scheduler is to fulfil all incoming requests with minimal resource usage.

5.3 Reservation Enforcement

Scheduling storage reservations on a shared infrastructure might result in a situation where users with different requirements share resources. In such a multi tenant scenario we have to ensure that each user gets at least the reserved resources by a reservation enforcement mechanism. Our reservation enforcement covers two parts, performance isolation and capacity limitation. While the performance isolation has to

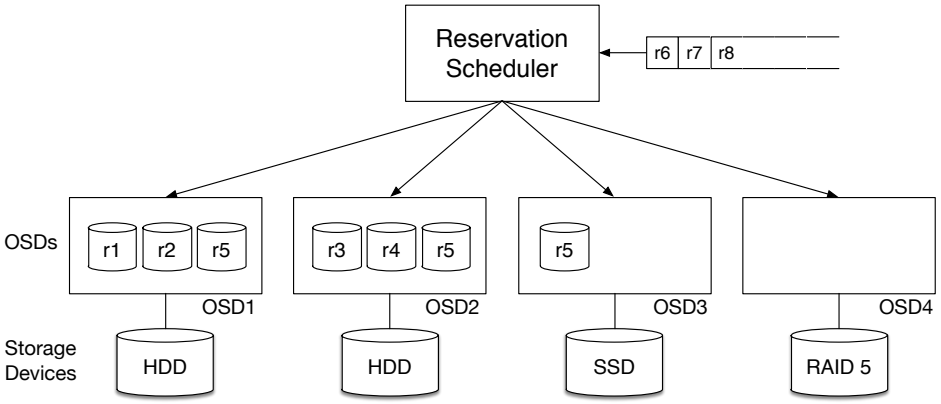


Figure 5.3: Reservation Scheduling

guarantee a lower bound of the available throughput for a user, the available capacity have to be a hard limitation. Sequential or random throughput that is reserved but unused by a tenant, might be used by a different tenant without slowing down the owner. The situation is different with capacity. Reserved capacity that is not used, must not be given to another tenant, since the owner might need it at some unforeseeable point in the future.

We enforce the throughput reservations by a new request scheduling strategy on the XtreemFS OSDs and the capacity reservations by introducing file system quotas on volume level.

5.3.1 Weighted Fair-Share Queuing

We introduce a queueing strategy that ensures proportional fairness for the request scheduling on OSD side. The proportional fairness property is ensured using a weighted fair queueing [3] strategy.

We integrate the request scheduling to the XtreemFS OSD by adding a new stage to the SEDA [21] which is responsible for reordering incoming requests while respecting the proportional fairness property. This architecture modifications are minimal.

The entry point for new requests in the network of stages in the XtreemFS OSD is a *OSDRequestDispatcher* which accepts all incoming network connections. Each request is passed to a preprocessing stage that parses requests and rejects invalid ones. We plugged a new *QueueingStage* after preprocessing to ensure proportional fairness. Each request that leaves the *QueueingStage* is passed to different stages to satisfy it. The exact path in the stage network depends on the request type. Any read or write request ends in a *StorageStage* that perform the local file system access; write requests are optionally replicated by the *ReplicationStage*.

The introduced *Queueing* contains of a single thread that forwards incoming request to a set of queues, while all requests that belong to one volume, i.e. belong to the same QoS class, are place in a common queue. The *WightedFairQueueStage* monitors the costs of all served requests per volume. Requests are always taken from the queue that has the largest difference between its weight and the actually served requests. The counters of served requests are reset periodically to avoid inactive volumes to save their quota and slow down other volumes at a later point of time. We consider the reserved bandwidth as the weight for each volume.

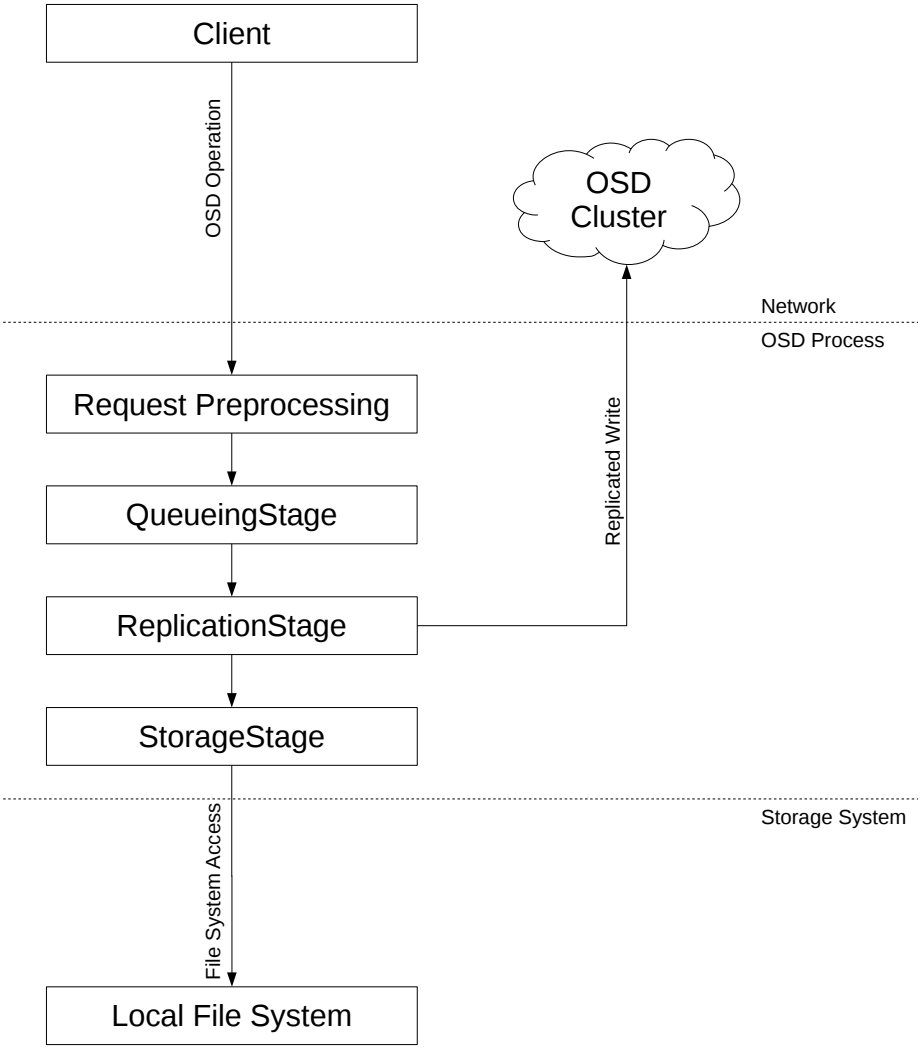


Figure 5.4: OSD Stage Network

The costs of requests can be determined by different metrics. Our implementation distinguishes between sequential and random access volumes in the cost metric. For sequential access volumes we consider the number of read or written bytes to be the costs of a request. This metric adopts the assumption that requests cover usually a large byte range and the overhead for disk head movements, etc. are negligible. We assume a constant costs for random access volumes. This metric underlies the assumption that requests are quite small and the overhead for disk seeks, network transmission, etc. is dominating.

A sufficient request cost metric can also be used to penalize access patterns that differ from the QoS reservation. Random access on a OSD that is used for sequential access volumes for instance would also slow down other volumes on the same OSD (see Section 4.1). Performing large sequential read or write requests on an OSD that is used for random access might increase the request latency for other tenants. Requests that differ from the reserved access pattern can be penalized by adding a constant overhead for all sequential access volumes. This causes disproportionate large costs for small (random) access. Large requests on random access volumes can be penalized by adding extra cost for requests with a size that expires a given threshold.

5.3.2 Capacity Limitation

Beside the performance isolation of different file system volumes, limiting a volume to the reserved capacity is an important task. We name this part of the reservation enforcement *volume quotas*. Implementing quotas in object-based file systems is challenging in general, but can be simplified under certain conditions. Here we discuss different ways to implement volume quotas quotas in object-based file systems and present our first prototype.

We start with an introduction to the authorization mechanism of XtreemFS to get a better understanding of the necessary steps to limit the amount of data a user may write to a set of files. Figure 5.5 presents the necessary operations that have to be performed by a client to access a file. In the first step, the client sends an *open* operation to the MRC. The MRC answers with the file ID and the location of the data chunks. This answer contains a *XCap* object with cryptographic signature and the allowed access mode. The client presents this *XCap* to the OSDs with each read or write request. A client reports file size updates after write requests to the MRC. XtreemFS has a scrubber process, which can correct missing file size updates, e.g. caused by a crashed or malicious client. The scrubber can for instance run periodically, initiated by an operator. We extend the *XCap* to communicate quota information from the MRC to the OSDs.

In the case that a volume runs out of space, this information has to be communicated to the clients. The POSIX standard provides the error code *ENOSPC* to state that a device is out of space. We pick up this error code as the behaviour of applications should be identical in the case of an exceeded quota in a distributed file system.

The effort to implement a quota protocol depends on the assumptions, made about the storage system in terms of the trustworthiness of the clients and the used data distribution on the OSDs. In the general case, we assume untrusted clients and an unknown distribution of files over the OSDs.

The HARNESS cloud uses a single OSD or striping per volume, thus the amount of data that is stored on one OSD is known. As the volume placement that is computed by the reservation scheduler is propagated to the OSDs to enforce reservations, the OSDs can enforce volume quotas locally. Each OSD has to track the written data per volume and returns the *ENOSPC* error code if a volume is full.

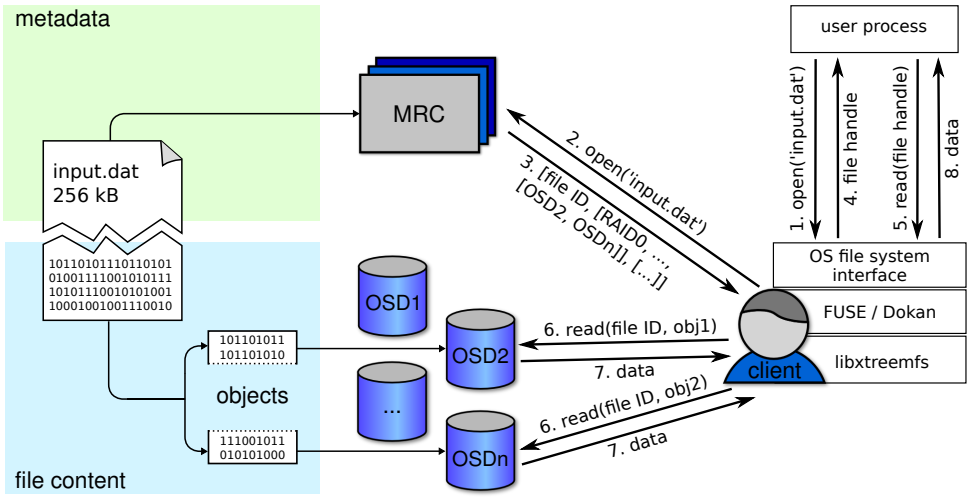


Figure 5.5: File access with XtreemFS

Volume quotas can be implemented in a simplified way by only modifying the MRC. The MRC monitors the occupied capacity per volume and returns the *ENOSPC* error code if the volume is out of capacity and a client tries to open a file for write access. This approach relies on correct file size update reports by the client to the MRC. In the case of missing or incorrect file size updates, a client can pass the quota up to the next scrubber run. Furthermore, a client can override the volume quota with files that have been open before the quota exceeded up to the expiry of the *XCap*.

We implemented volume quotas in the MRC of our current prototype, which implies the mentioned limitations. We will extend the quota implementation to the OSDs during the third year. An XtreemFS user has to set a predefined extended file attribute on the root node of a volume to set a quota. The reservation scheduler sets this attribute after the volume creation to the reserved capacity. The MRC compared the used volume space to the quota whenever a file is opened for write access or created. In the case that the volume is out of capacity, the MRC returns an *ENOSPC* error code instead of a *XCap* object.

5.4 OSD Performance Exploration

For the process of scheduling storage reservations to XtreemFS OSDs, a detailed knowledge of the performance characteristics of the used devices is necessary. The behaviour of OSDs differs for access patterns, the level of multi tenancy and the used storage devices.

There is an large body of research regarding performance modelling for HDDs [13] and SSDs [22, 4]. These works analyse disk performance on a block device level. The behaviour of an XtreemFS OSD may differ significantly, due to the XtreemFS OSD architecture, consisting of an OSD software stack, a local file system, and various cache layers. As the software stack may differ between XtreemFS OSDs and cause many uncertainties, we developed an OSD profiler to measure the capabilities of each machine before scheduling reservations.

We presented the performance behaviour of selected HDD and SSD models in Deliverable D5.1 [5]. The evaluation has shown that the sequential throughput drops while increasing the number of concurrent

streams while random access is not affected by multi tenancy. We also noticed that the read performance for both workloads is beyond the write performance.

We integrated the profiling tool, originally developed for performance modelling, to the reservation scheduler. The profiler can be used in two different ways. The first use case, which was already available in the previous prototype, uses static OSD performance profiles. These are sufficient for an XtreemFS cluster consisting of a large set of identical OSDs. The performance measurement has to be done once and can be reused for all OSDs. The second method is an automatic OSD performance exploration where the reservation scheduler performs a OSD profiler run for each unknown OSD.

We measure the sequential read performance for a varying number of concurrent streams and the random read performance for a single tenant. These values are the minimal information that are necessary to schedule reservations while the profiling runtime is kept fast.

5.5 Providing Feedback to the HARNESS Platform

An upcoming version of the IRM API will contain a method to report feedback about running applications to the platform to improve the process of building performance models by the *Application Manager* (AM). The feedback that is reported by the storage component to the platform might contain time dependant resource usage information. We designed and implemented a scalable I/O tracing infrastructure to monitor file system access during the runtime of an application and analyse this trace in terms of the resource usage after an application terminated.

The objective of the tracing infrastructure is a scalable way to analyse file system accesses with a small overhead and thus a negligible impact on the running applications. Application have to be analysed without any code modifications. The system has to work with a large number of OSDs while multiple clients are performing I/O operations in parallel on them.

5.5.1 Scalable I/O Tracing

We implemented a tracing infrastructure that makes use of the Hadoop integration of XtreemFS to analyse file system traces in a scalable way. We extended the SEDA of the XtreemFS OSD by a tracing stage that extracts information from each incoming request using a user-defined policy. The extracted trace information is written to an XtreemFS volume and thus can be accessed by Hadoop using the XtreemFS adapter. In the most general case, the user defined trace policy extracts all metadata from a request and a user may perform arbitrary analytics on this trace using a MapReduce program. Relevant request metadata is, for instance information about the request type, the affected file, the current time-stamp, and the affected byte range of the file.

To achieve an optimal performance, even while the file system activity is traced, we introduce various optimizations. The trace handling is performed asynchronously. The request is handled in the tracing stage concurrently to the regular handling to ensure a minimal impact on the file system throughput. The trace is aggregated in memory and written to the persistent file system in batches to minimize the interference with regular I/O operations. We make use of the weighted fair-share queue of the OSD and insert tracing file write requests with a minimal priority. We furthermore track the current queue length of the OSD to chose a sufficient point in time to write the batched trace information to disk. We use an

OSD selection policy that tries to write the trace locally or at a random OSD in the case that a local file placement is not possible, e.g if the OSD is out of free capacity.

Inspired by the *combiner* function of the MapReduce [2] paradigm, we introduce a preprocessing of the collected trace data. Such a *combiner* is part of the user defined trace policy and is applied to the collected trace data before writing a batch of entries to the persistent file system. The intention of this function is to reduce the data that has to be written to the file system and filter needless information as early as possible.

As shown by Figure 5.6, each request is, in addition to the regular processing, also passed to a *TracingStage* that applies the *map* function to extract relevant information. The intermediate result is written to a *TraceBuffer*. The buffer content is written at a suitable time (e.g. when the system is idle) to persistent storage. The *combiner* is applied before that. The request is written to a regular file using *libxtreemfs* to perform preprocessing (e.g. a *reducer*) using the regular XtreemFS interfaces.

5.5.2 Analysing File System Traces

To provide feedback to the HARNESS platform layer, we have to extract the resource usage on a XtreemFS volume from the traced data. Resource usage means the used capacity and the maximum sequential or random throughput caused by the running application.

We use a trace policy that extracts from any *read* or *write* access that arrives at an OSD the request type, the current time stamp, the id of the accessed file and the accessed byte range given as an offset and a request length.

Analysing the file system trace using MapReduce has the advantage that all resources that have been available for a compute job can be used to analyse the file system trace on the available compute nodes in parallel.

Computing the Used Capacity

The maximum used capacity during the runtime of a job in the HARNESS cloud can be determined by summing up the maximum written offset for each file that is seen in the access trace.

This strategy can be implemented as a MapReduce algorithm with the following *map* and *reduce* functions. The *map* function emits a key-value pair for each file system write access. The key is the file identifier and the value is the sum of the offset and length of the write access. The *reduce* function picks the maximum offset for any file and sums up the offsets of all files before terminating. If the trace is analysed by multiple reducer tasks, the results off all reducers have to be summed up in a post-processing task. The algorithm can be optimised by a combiner that emits only the maximum offset per file.

Computing Sequential and Random Throughput

The goal of the throughput analysis is to compute a list of sequential or random throughput, grouped by time slices. The computed throughput is averaged for each time slice. A computed time series contains either sequential or random throughput.

Our MapReduce implementation consists of a *map* function that emits a key-value pair with the time slice, the request is assigned to, as a key and the original request description as the value. The size of the

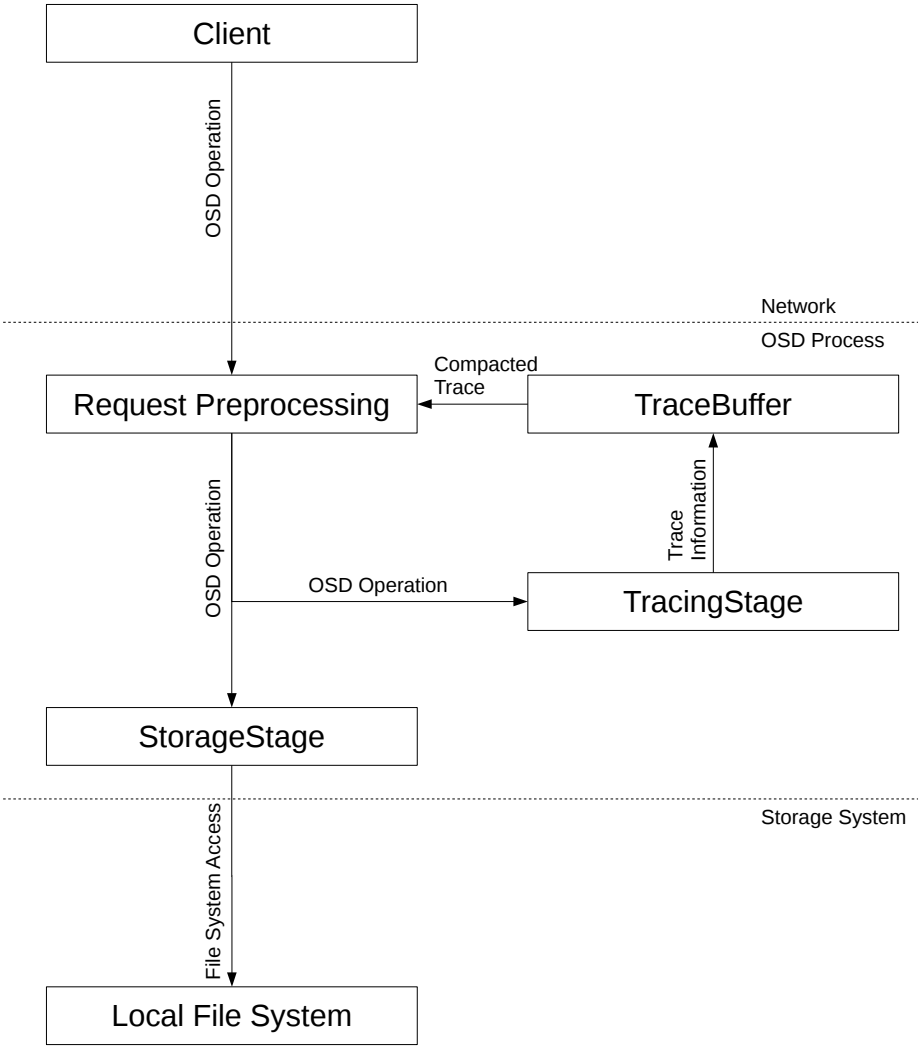
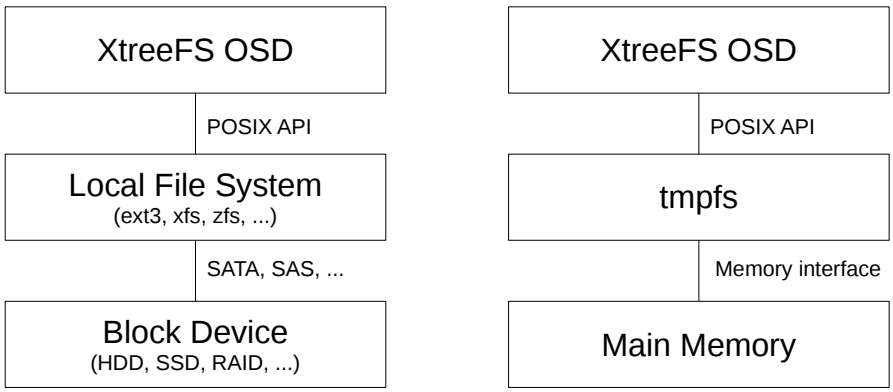


Figure 5.6: OSD Request Tracing



(a) Persistent storage. (b) *In-Memory* storage.

Figure 5.7: OSD software stack

time slices is computed based on a predefined granularity. The *reduce* function computes the average throughput for all requests in one time slice.

5.6 In-Memory OSDs

We analysed the performance of in-memory OSDs in Section 4.3. In this section, we describe their implementation. As the OSD software stack makes use of an existing local file system like *ext4*, we can replace the local file system by using tmpfs [16]. As presented in Figure 5.7(a), the OSD process sends requests to a local file system using the POSIX API. A file system, which is usually implemented in the operating system kernel, e.g *ext4* in the Linux kernel, stores its data on a block device like a HDD or SSD. The block devices are accessed in the kernel via a block device interface like SATA or SAS.

One possibility to transform this stack to in-memory is replacing the local file system by tmpfs (see Figure 5.7(b)). Tmpfs stores all data directly in the main memory of the OSD server without any block device access.

Our evaluation has shown that the throughput of tmpfs is – depending on the used machine – limited to 2.5 to 3 GB/s. To improve the performance of an in-memory OSD, we will implement a new object store in the XtreemFS OSD that handles the memory allocation internally and does not rely on the POSIX interface.

5.7 LD_PRELOAD based XtreemFS client

XtreemFS provides multiple client solutions. For Linux and Unix systems, there is a client based FUSE, which allows the user to mount and use an XtreemFS volume like a local file system with POSIX semantics. For Windows, a similar solution based on CBFS from EldoS is available. On the other side, there is the *libxtreemfs* library for C++ and Java, which allows application developers to directly integrate

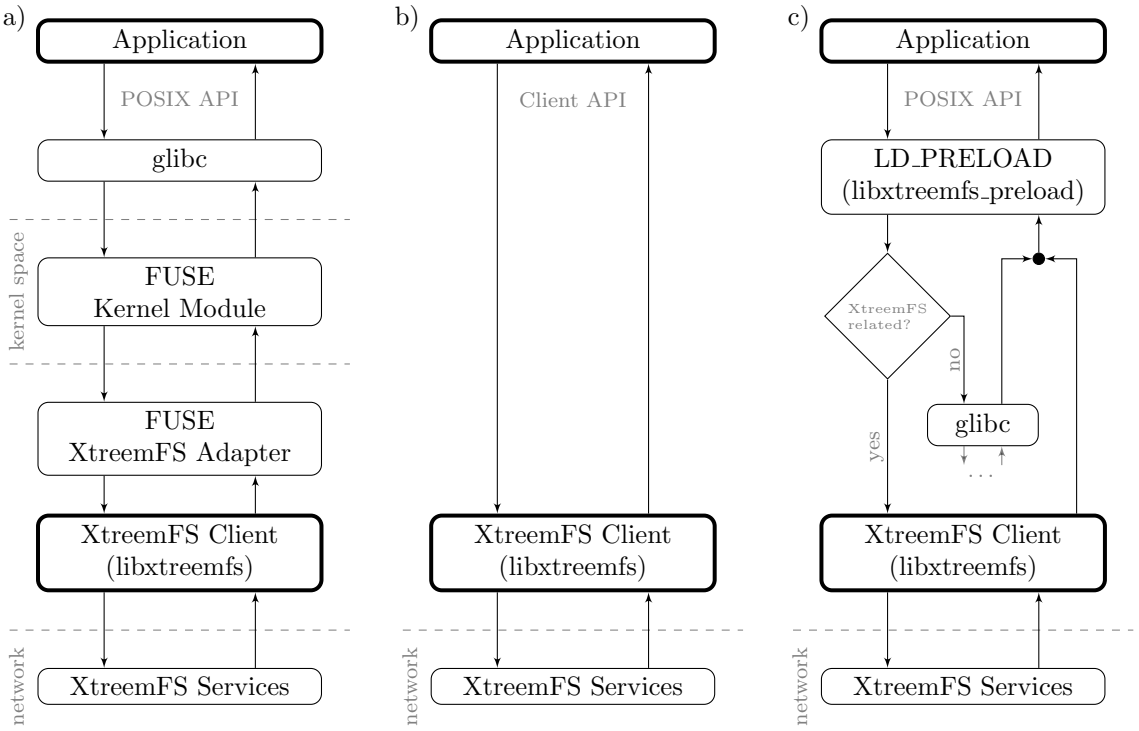


Figure 5.8: The different client solutions for XtreemFS: a) shows how the application transparently interacts with XtreemFS via FUSE, b) shows the direct use of the client library which avoids overhead but is intrusive, and c) shows the LD_PRELOAD based interception of file system calls which is non-intrusive and avoids FUSE as a bottleneck.

XtreemFS support into applications. Both, the FUSE client and *libxtreemfs* have characteristic advantages and disadvantages. While the FUSE approach is the most transparent one and is compatible with every application, it requires the FUSE subsystem to be available on the client system. Also does channelling data through FUSE limit the achievable performance. Those limitations are not present if an application directly uses the API provided by *libxtreemfs*. So currently, there is a trade-off between transparency and performance. If neither FUSE is available nor is modifying the application an option, XtreemFS cannot be used at all.

In HARNESS, we need both transparency and performance. Transparency is needed to support a wide range of applications, while the client performance determines the guarantees the system can make, and how much they cost. For instance, if the achievable bandwidth or latency is limited by the client, this poses an upper bound to the guaranteeable values. On the other hand, if the fraction of the random access latency caused by the client can be reduced, the same latency can be guaranteed with less hardware cost. So, to overcome the current client situation and meet the requirements of the HARNESS project, we developed a new solution based on the *LD_PRELOAD* mechanism.

LD_PRELOAD allows loading libraries that are used to resolve dynamically linked symbols before other libraries are considered. We implemented a *libxtreemfs_preload* that uses this mechanism to intercept and substitute file system calls of an application. If an intercepted call relates to an XtreemFS

volume or file, it is translated into its corresponding *libxtreemfs* call, which is similar to what the FUSE adapter does. If it is not, calls are passed through to the original *glibc* function which would have handled it without the pre-load mechanism in place. Whether or not XtreemFS should be used is determined via a configurable path prefix that can be thought of as a virtual mount point. Figure 5.8 shows all three client solutions in comparison.

For example, copying a file to an XtreemFS volume via FUSE using `cp` as application would performed as follows:

```
$> mount.xtreemfs remote.dir.machine/myVolume /xtreemfs
$> cp myFile /xtreemfs
$> umount.xtreemfs /xtreemfs
```

The same operation without FUSE but with the *LD_PRELOAD* and *libxtreemfs_preload* could be achieved with the following command:

```
$> XTREEMFS_PRELOAD_OPTIONS="remote.dir.machine/myVolume /xtreemfs" \\
   LD_PRELOAD="libxtreemfs_preload.so" \\
   cp myFile /xtreemfs
```

This example can be easily generalised into a shell script that wraps `cp` or other commands, such that the environment setup is hidden.

In the development of *libxtreemfs_preload*, we collaborated with Quobyte, an SME and ZIB spinoff working on commercial software-only enterprise storage-solutions based on XtreemFS. The current implementation still is in an experimental state and does not yet implement the full set of POSIX file system calls. Nonetheless, we already used it successfully to set up transparent distributed checkpointing for an unmodified scientific HPC application (BQCD) on a Cray XC30 supercomputer. Section 6.2 provides performance results on the different XtreemFS client solutions.

6 Evaluation of the Storage Component

This chapter describes the evaluation of the storage component prototype. We focus on the evaluation of the performance isolation on OSD side, which is necessary to enforce reservations, and the evaluation of client side improvements.

The first part of this chapter demonstrates the behaviour of the HARNESS demonstrator applications RTM and AdPredictor in a multi-tenant scenario on a shared storage infrastructure. In the second part, we compare the performance of the different XtreemFS client implementations. This is done by a synthetic benchmark, as the early stage of the *LD_PRELOAD* based client does not support all POSIX file system calls and thus, the range of supported applications is limited.

6.1 Performance Isolation

We evaluated the performance isolation that is guaranteed by the request scheduling we presented in Section 5.3 through running two instances of a demonstrator application. Both of the instances access their own XtreemFS volumes, which are placed on one shared OSD. Each instance uses dedicated compute nodes.

The used OSD is equipped with a Seagate ST9250610NS that provides a sequential throughput of approximately 90 MB/s with two concurrent sequential access streams. This is the end-to-end performance, considering the latency of the software stack and the network. We create two logical volumes, where the fraction of the possible throughput of each volume is varied.

6.1.1 Reverse Time Migration

RTM is a multi-threaded application that runs on a single node. There are multiple implementations of the RTM demonstrator, a compute intensive and a storage intensive version, both having a *central processing unit* (CPU) and a *dataflow engine* (DFE) implementation. We used the storage intensive CPU implementation for our evaluation.

Figure 6.1 illustrates the setup of the RTM evaluation. Each RTM instance runs on its own compute node with 12 CPU cores with hyper threading and 32 GB of memory. Both program instances access their own logical volume located on a shared XtreemFS OSD.

The runtime of RTM depends on the number of migrated shots and the frequency of the shots. As multiple shots are migrated sequentially, we migrate a single shot. We ran RTM with shots having a frequency of 9 Hz, 18 Hz and 36 Hz. We used two volumes where the faster one has two times the weight of the slower one, i.e. one with a sequential throughput of approximately 60 MB/s and one with 30 MB/s. Figure 6.2 shows the performance difference between the faster and the slower XtreemFS volume, depending on the frequency of the migrated shot. The runtime of the application was in a range of 29 and 36 seconds at a frequency of 9 Hz, between 519 and 640 seconds at a frequency of 18 Hz and between 8442 and 9343 seconds at 36 Hz. Figure 6.3 shows the relative, which drops by increasing the frequency. We use shots with a frequency of 36 Hz for the next experiment, because the higher runtime

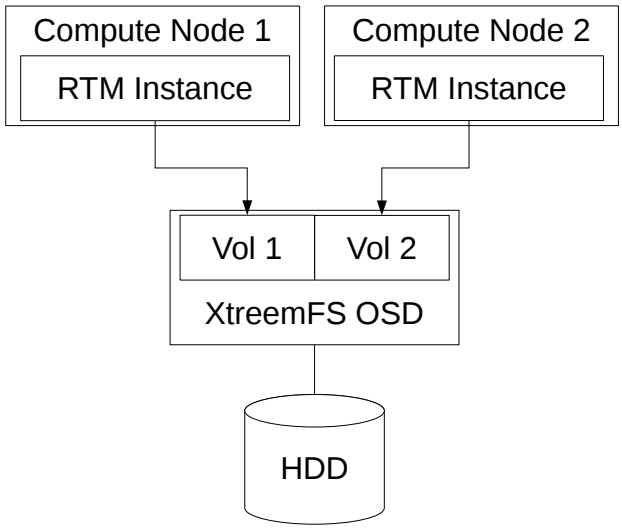


Figure 6.1: RTM setup

decreases the expected measurement error. The plots are based on the average of three runs. The error bars denote the minimum and maximum.

In our next experiment, we vary the weights of the used volumes. The weights are varied from a ratio of 0.75 to 0.05. We evaluate the impact of the ratio of the volume weights on the runtime of the RTM demonstrator. We again executed two instances of RTM on different compute nodes, which access one storage node concurrently. Figure 6.4 shows the performance difference of both RTM instances, depending on the weight ratio of the XtreemFS volumes. Again, the plotted values are the average of three runs.

Compared to the total runtime of each RTM run, which ranges between 8373 seconds and 9605 seconds, the difference between the two volumes is relatively small. The small effect of the volume weight can be explained by the following reasons. First, the OSD request queuing implementation requires multiple pending requests in the queue to prioritise one of the volumes. Many applications perform I/O calls synchronously in a single thread, thus there is not more than one request of each application instance in the OSD queue at the same time. We bypass this problem partially by using the asynchronous write feature of the XtreemFS client, but the situation is unchanged for read operations. The second reason for the tiny runtime difference is that the RTM instance accessing the lower prioritised volume gets for full I/O bandwidth after the first application terminates and a significant speedup for this time period.

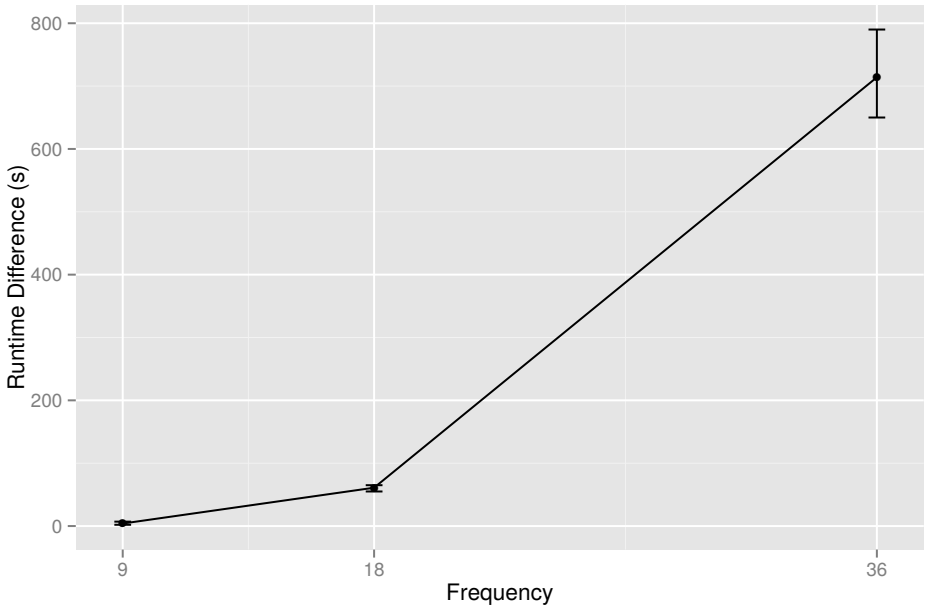


Figure 6.2: RTM absolute runtime difference while doubling the volume weight

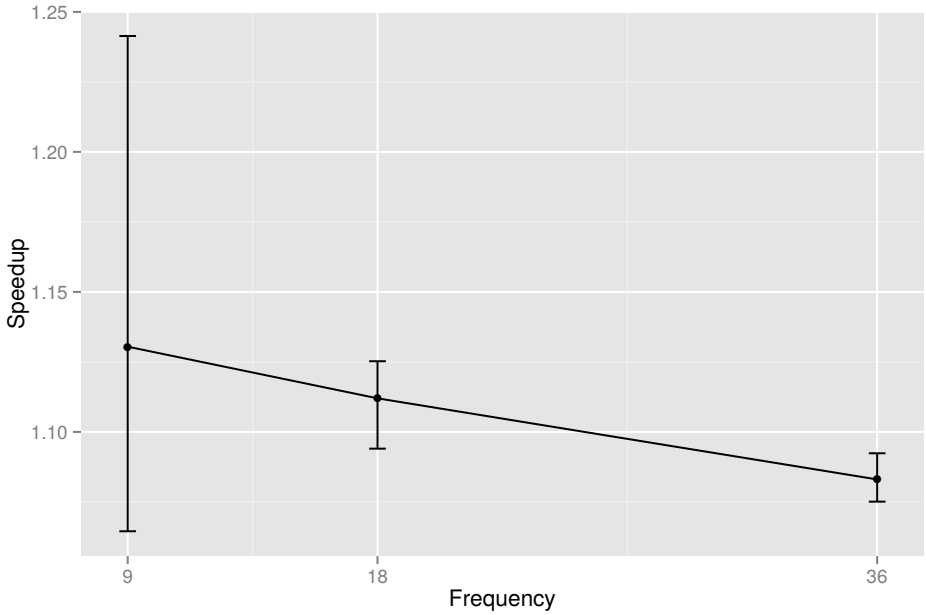


Figure 6.3: RTM relative speedup while doubling the volume weight

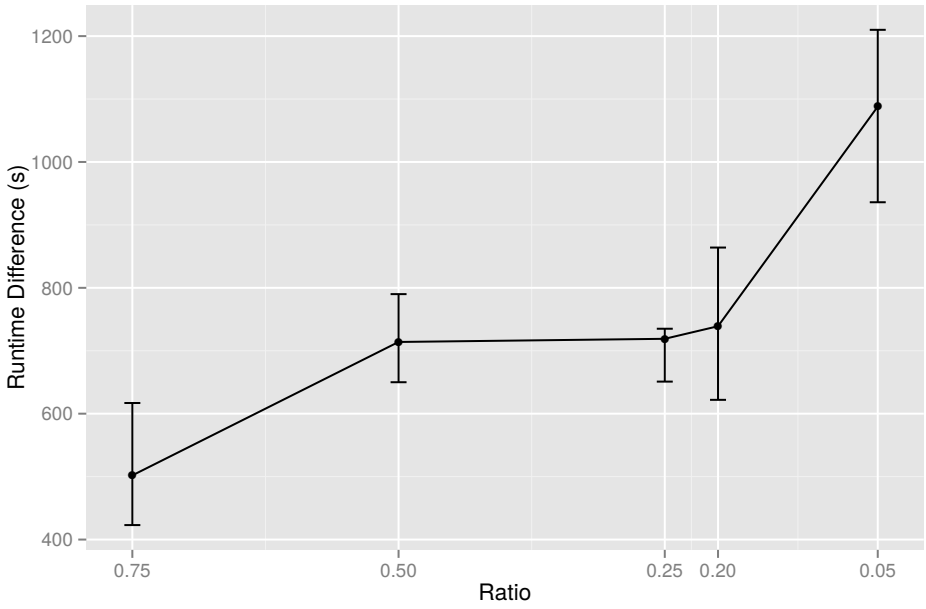


Figure 6.4: RTM runtime difference depending on the weight ratio of the volumes

6.1.2 AdPredictor

We repeated the same experiment with the AdPredictor demonstrator. We used the C++ implementation of AdPredictor with the KDDCup 2012 data set as input.[1] During each run, 10,000,000 training steps and 4,000 test steps were performed. The setup used is identical to the RTM setup: two instances of the application are running on dedicated compute nodes while both instances access a shared XtreemFS OSD. Each AdPredictor instance accesses its own logical volume.

Again, we varied the weight ratio of the volumes from 0.75 to 0.05. The absolute runtime of each iteration was in the range of 308.4 seconds and 330.2 seconds. Figure 6.5 depicts the runtime difference of both instances depending on the weight ratio. The plot shows the average of three runs while the error bars mark the minimal and maximal difference.

We can observe a slight increase of the performance difference when the ratio, i.e. the priority of the slower volume, decreases. The impact of the volume weight on the application runtime is low compared to the absolute runtime. The reason for that is that I/O operations are performed synchronously by the applications. This means there is at most one pending request per application, while the prioritisation depends on choosing between multiple simultaneously queued requests of both application instances. Since the I/O access pattern of the application is dominated by read requests, the asynchronous write feature of XtreemFS does not solve this problem, as it did for the RTM evaluation.

From this evaluation, we can conclude that a substantial throughput difference between different quality classes can only be achieved by combining our performance isolation mechanism with asynchronous

[1] Online at http://www.kddcup2012.org/c/kddcup2012-track2/data

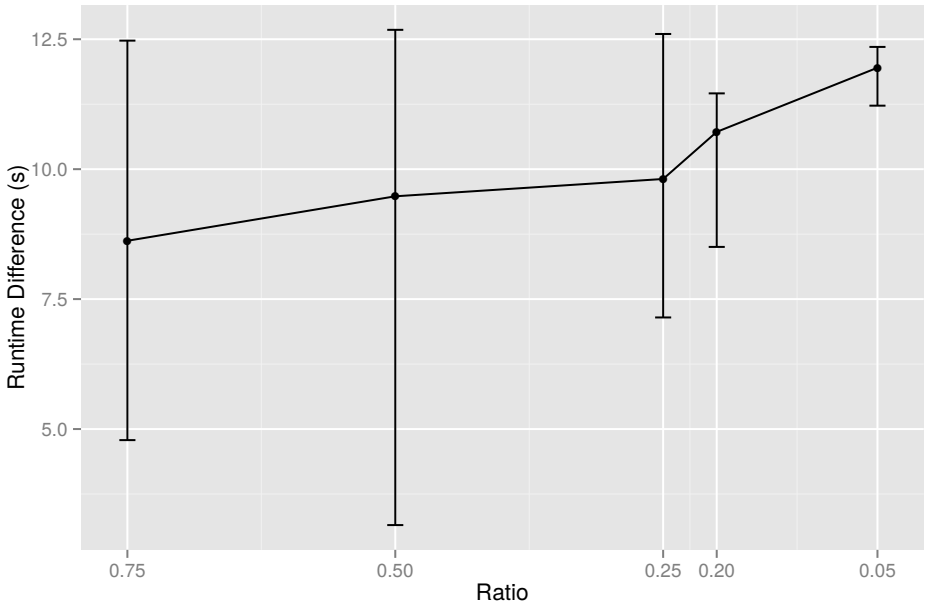


Figure 6.5: AdPredictor runtime difference depending on the weight ratio of the volumes

writes. The applicability of this depends on the I/O characteristics of the used applications. Nevertheless, performance isolation is an important enhancement to provide QoS aware file system volumes.

6.2 Comparing bandwidth and latency of different XtreemFS clients

In this section, we evaluate the three XtreemFS client solutions described in Section 5.7. In order to compare the cost of the different data paths depicted in Figure 5.8, we performed micro-benchmarks of the read and write operations to an XtreemFS volume with each solution. To exclude hardware-specific effects of the benchmark system, like local disk performance, we used a *tmpfs* filesystem on a RAM disk for OSD storage. The different XtreemFS services ran on a single node, so there is no actual network traffic that might pose a bottleneck. Caching mechanisms of the kernel and FUSE were disabled by using the *direct_io* option of FUSE. This ensures that all requests reach the XtreemFS client and the OSD and allows us to quantify the actual client overheads and compare the values with each other. Otherwise we would measure a mixture of client overheads and caching benefits. All result values are averages over multiple runs, the error bars visualise the standard error.

Figures 6.6 and 6.7 show the results for sequential reading and writing, respectively. In both cases, the results match our expectations, i.e. *libxtreemfs* is faster than LD_PRELOAD which is faster than FUSE. For reading, LD_PRELOAD is between 26 % and 72 % faster than FUSE with an average of 51 %. Compared with *libxtreemfs*, it is around 18 % slower on average (between 8 % and 34 %). Writing performance is similar. LD_PRELOAD compared with FUSE is around 52 % faster (between 14 % and 70 %), and 20 % slower when compared to *libxtreemfs*. Narrowing the gap between *libxtreemfs* and LD_PRELOAD will be subject of further investigation.

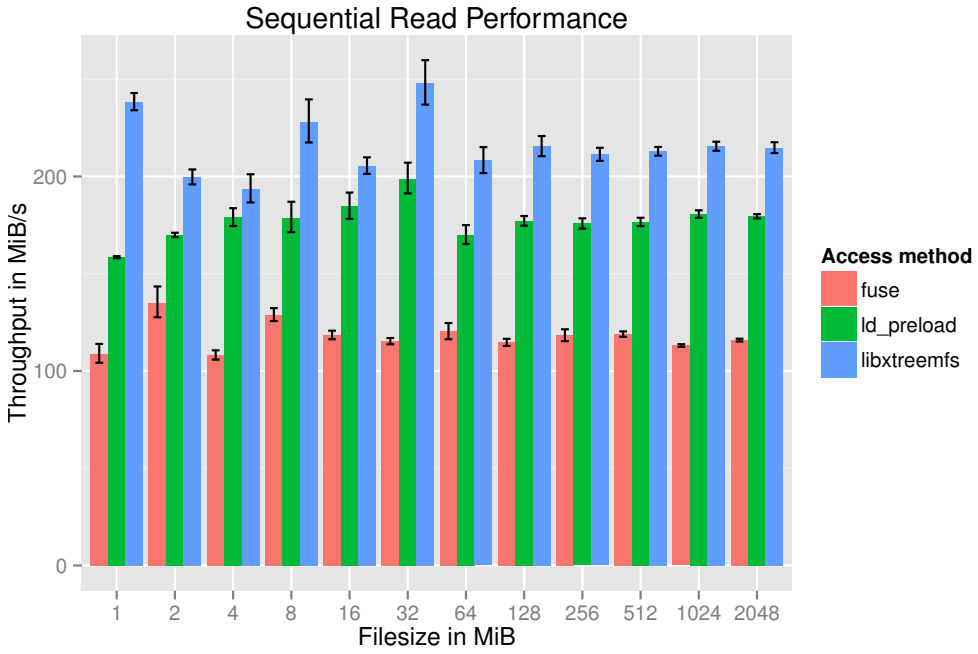


Figure 6.6: Sequential read performance using the three different client approaches.

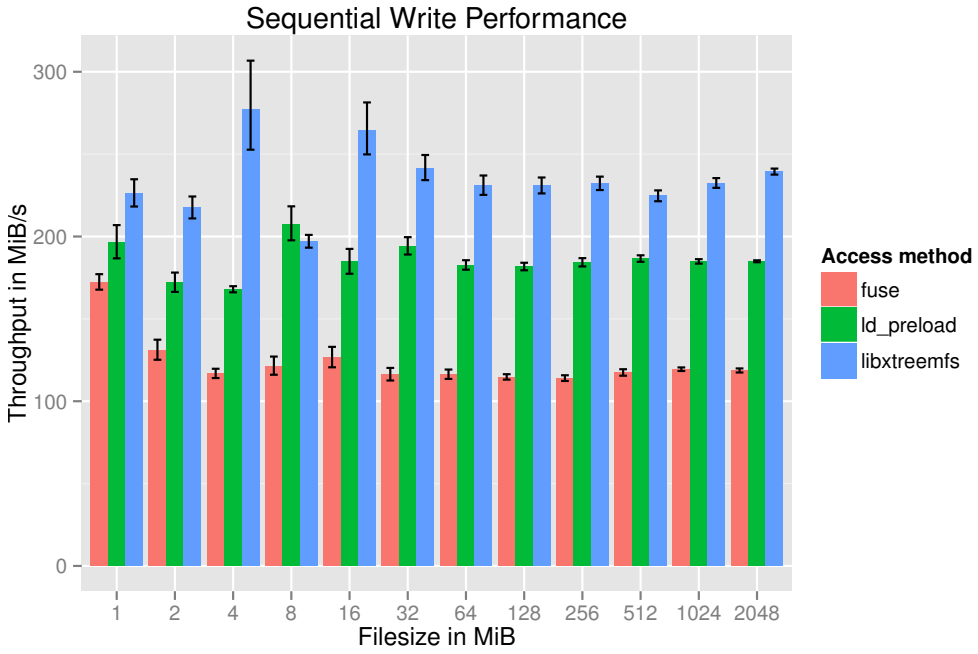


Figure 6.7: Sequential write performance using the three different client approaches.

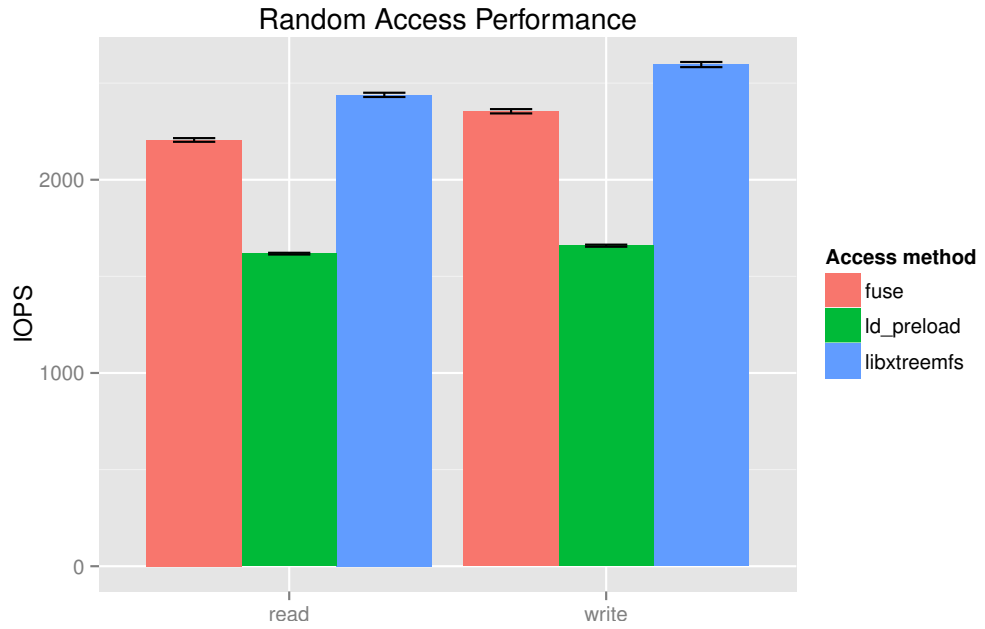


Figure 6.8: Random access read performance using the three different client approaches.

Figure 6.8 depicts the random access performance for reading and writing. We measured accessing 10,000 pseudo-randomly chosen 4-KB blocks from a 1-GB file. The LD_PRELOAD client performs below our expectations. Compared with FUSE, reading is 34 % and writing 36 % slower. Compared with *libxtreemfs*, it is 27 % and 30 % respectively. The gap between FUSE and *libxtreemfs* is around 10 %, so the potential gain is not as high as for sequential access. For each access operation, there seems to be a rather high constant overhead when using the LD_PRELOAD client. Our hypothesis is, that this overhead is caused by the implementation of the interception and pass-through initialisation mechanism, which currently involves a `pthread_once()` call for every operation. This is needed to perform lazy, thread-safe initialisation. Finding a less costly approach will be part of our future work.

All in all, the presented results indicate that the newly developed LD_PRELOAD client approach yields a better sequential bandwidth than FUSE while still being transparent to the application. Its random access performance stays behind expectations but is not prohibitively low. We are positive that this issue can be solved. Regardless of performance, LD_PRELOAD is the only solution for applications that run in an environment where FUSE is not available and where modifying the application code is not an option.

7 Conclusion and Future Work

7.1 Conclusion

The HARNESS storage component, implemented as an extension of the XtreemFS distributed file system, provides reliable file system performance for applications in a *platform-as-a-service* (PaaS) cloud. We enhanced our results from the first project year by extending our performance model to mixed access patterns that occur when running real applications like the HARNESS demonstrators (Requirement R23). We introduced a new device class named in-memory OSDs and analysed its behaviour. Device models have been enhanced to cover reliability requirements.

We presented an updated prototype of the HARNESS storage system which provides performance isolation between different users. Performance isolation is guaranteed by introducing a request scheduling mechanism at the XtreemFS OSDs that helps to establish a lower bound for the available bandwidth to each logical volume (Requirements R20 and R22). Performance isolation is necessary in a multi-tenant scenario and can utilise resources more efficiently. A first prototype to enforce capacity limits on volumes was presented.

A new client variant results in performance improvements for sequential access and extends the number of supported platforms and applications. The new client implementation transparently bypasses the operating system overhead by intercepting POSIX file-system calls and redirecting them to *libxtreemfs*.

The storage component is integrated with the HARNESS platform using the IRM API. The XtreemFS-IRM allows to request and release reservations. We presented preparations to provide feedback to the platform as a next step (Requirement R24).

Our evaluation has shown that the performance isolation effectively enforces reserved resources, which is reflected by the measured runtime differences for two concurrent RTM or AdPredictor instances accessing a shared storage system (Requirements R28 and R37).

7.2 Future Work

During the third year of the HARNESS project, we will continue to work on more detailed device and performance models. We will investigate the impact of OSD internals, like the on disk storage layout or the memory management, on the file system performance. The recent work on the HARNESS storage component was focused on the I/O path, but left out the metadata path of the file system. The impact of metadata performance on the total throughput of the file system will be analysed in Year 3. The goal is to apply the existing solutions for reservation scheduling and reservation enforcement to the XtreemFS MRC.

We will continue the integration with the HARNESS platform and work on extending the API for reliability and monitoring purposes. Relevant metrics for the HARNESS platform will be identified and the tracing framework, that has been presented in Section 5.5.1, will be extended by policies to match the requirements of the AM.

We will extend the reservation scheduler regarding the aspects we presented in Chapter 4, e.g. reliability and disk fragmentation. The capacity reservation enforcement will be extended to the XtreemFS OSD and thus be able to enforce capacity limits even in the case of malicious clients. The prototype will be evaluated using the validation cases RTM and AdPredictor on different platforms. Results from the HARNESS project will be migrated to the stable XtreemFS code base and thus stay available for succeeding projects and the open source community.

Bibliography

[1] Cluster File Systems, Inc. Lustre: A scalable, high performance file system, 2002.

[2] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1), 2010.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM.

[4] P. Desnoyers. Analytic models of ssd write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[5] FP7 HARNESS Consortium. Characterisation report. Project Deliverable D5.1, 2013.

[6] FP7 HARNESS Consortium. Data storage component (initial). Project Deliverable D5.4.1, 2013.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *SIGACT News*, 33(2):51–59, June 2002.

[8] J. Gray and C. Van Ingen. Empirical measurements of disk failure rates and error rates. *arXiv preprint cs/0701166*, 2007.

[9] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreemFS architecture—A case for object-based file systems in Grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[10] C. Kleineweber, A. Reinefeld, and T. Schütt. Qos-aware storage virtualization for cloud file systems. In *Proceedings of the 1st ACM International Workshop on Programmable File Systems*, PFSW '14, pages 19–26, New York, NY, USA, 2014. ACM.

[11] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *Communications Magazine, IEEE*, 41(8):84–90, Aug 2003.

[12] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, volume 7, pages 17–23, 2007.

[13] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[14] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. *ACM Transactions on storage (TOS)*, 6(3):9, 2010.

[15] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *FAST*, volume 7, pages 1–16, 2007.

[16] P. Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference*, 1990.

[17] J. Stender, M. Berlin, and A. Reinefeld. XtreemFS – a file system for the cloud. In D. Kyriazis, A. Voulodimos, S. V. Gogouvitis, and T. Varvarigou, editors, *Data Intensive Storage Services for Cloud Environments*. IGI Global, 2013.

[18] J. Stender, M. Berlin, and A. Reinefeld. XtreemFS – a file system for the cloud. In D. Kyriazis, A. Voulodimos, S. V. Gogouvitis, and T. Varvarigou, editors, *Data Intensive Storage Services for Cloud Environments*. IGI Global, 2013.

[19] J. Stender, B. Kolbeck, M. Hogqvist, and F. Hupfeld. BabuDB: Fast and efficient file system metadata storage. In *International Workshop on Storage Network Architecture and Parallel I/Os*, pages 51–58, 2010.

[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[21] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture ofr well-conditioned, scalable Internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[22] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Trans. Comput. Syst.*, 32(2):6:1–6:48, June 2014.