Project no. IST-033576

XtreemOS

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

Design Report for the Advanced XtreemFS and OSS Features D3.4.3

Due date of deliverable: 31-MAY-2008
Actual submission date: 31-MAY-2008

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP3.4

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

Version 1.0 / Last edited by Björn Kolbeck / 22-MAY-2008

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

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	04.04.08	Felix Hupfeld	ZIB	Initial document structure and initial contents from work documents
0.2	10.04.08	Toni Cortes	BSC	RMS description
0.3	24.04.08	Eugenio Cesario	CNR	Advanced Storage Stage features description
0.4	22.05.08	Björn Kolbeck	ZIB	final version
0.5	28.05.08	Felix Hupfeld	ZIB	final version 2

Reviewers:
Adrien Lèbre (INRIA), Roman Dementiev (SAP)

Tasks related to this deliverable:

Task No.	Task description	Partners involved$^{\circ}$
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

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

Contents

Executive Summary

The architecture of XtreemFS as a parallel object-based file system has been designed in a way that it can act as a platform that enables the implementation of advanced features that solve problems of cross-organization cross-site data management. Most of the planned features cannot be found in today's systems and require substantial research effort.

In this report, we have investigate the requirements for these advanced features and provide a framework for the design. This evolution of XtreemFS will be in the areas of replication mechanisms, replica management, caching and storage. Furthermore, we concentrate on the evolution of our work in the Object Sharing Service (OSS) and general quality and testing. In particular, the investigated areas are:

- **Replication Mechanisms.** We provide an architectural framework for implementing replica consistency mechanisms. We present how the evolution steps of replication of read-only files and later writable replicas will be designed. Our design allows us to separate logical replica creation from physical copying and thereby create incomplete replicas, which we will also describe.
- **Replica Management.** Using these mechanisms, the Replica Management Service (RMS) will automatically manage the creation and deletion of replicas and optimize their location in the Grid. It will be able to reactively create replicas on demand and create replicas proactively by following demand forecasts. For these purposes, it will interact with the Application Execution Management (AEM) of the XtreemOS installation.
- **Client-Side Caching.** While the current XtreemFS client already supports caching with NFS semantics, we plan to further improve it towards POSIX compliance. Then it will be able to provide a coherent file abstraction in presence of concurrent cached access from multiple client and on multiple storage servers.
- **Storage.** We will further work on improving our IO performance by investigating possible enhancements that allow us to fully exploit the abilities of the underlying hardware. In particular, that means improving the concurrency of storage access and adapting the optimal layout for the underlying local file system.
- **Object Sharing Service.** We plan to improve the scalability of the OSS by using peer-to-peer technology as the underlying communication substrate. Furthermore, we will investigate in

detail the design of advanced transaction and memory management mechanisms.

1 Introduction

Since the beginning of the project, the members of the 3.4 work package have not only concentrated to design and implement a fully-functional XtreemFS file system and OSS, but also started to discuss the design of advanced features. Most of these features are not commonly found in today's Grid systems and middleware; they will provide XtreemOS users with new opportunities and will ease the use of Grid resources. These developments affect all parts of XtreemFS and the Object Sharing Service (OSS):

1. The XtreemFS client will be enhanced with client-side caching mechanisms that allow for high-performance I/O while ensuring data consistency even in presence of concurrent accesses.
2. The design of the XtreemFS Object Storage Device (OSD) will be refined for better performance.
3. The overall system architecture will be extended to be able to provide file replication. This includes many changes in the OSD, but also introduces a new component, the Replica Management System (RMS) that will use the replication mechanism to automatically improve data access.
4. The OSS will build on an overlay network and support advanced transaction and memory management mechanisms.
5. Also, WP3.4 will further be committed to providing quality software and plans to refine its testing procedures.

2 Design of the XtreemFS Replication Mechanisms

Replication is an important feature in Grid Datamanagement; it increases the failure tolerance of the Grid through higher data availability and it reduces the load on WAN links by using local replicas which also reduces access times.

Many of today's grid data management systems do file replication by copying data to consumers and back to storage resources (e.g. [5]). While this helps access performance and availability for write-once data, it leaves the task of ensuring consistency of all the copies in face of changes to the user.

For XtreemFS we will design a replication mechanism that is transparent to the user. Applications can use XtreemFS just as any local file system; the complexity of replication and data consistency is hidden from users. This implies that replication in XtreemFS must allow files to be modified at any time, only then do we get a real replicated file system. This allows us to implement a replicated file system that can be used with legacy applications and offers enhanced performance for advanced users.

XtreemFS will also support read-only replication as it is an important intermediate step in the evolution of XtreemFS' design and allows higher performance that many users can benefit from. Typical examples are HPC applications that read input data and produce output files. In contrast to grid data management systems, read-only replicas will be under full control of XtreemFS and benefit from XtreemFS data dissemination mechanisms and its support for incomplete replicas.

2.1 Replication Components

Both modes of replication are based on the same architecture. For each file in XtreemFS exists a location list which contains all replicas and for each replica, the list of OSDs the replica is stored on. This list is managed and stored in the Metadata and Replica Catalog (MRC). When a client opens a file it also retrieves the location list from the MRC. The client can then access one of the OSDs from the list and work with the file. The client also passes the list to the OSD which needs it in order to identify the other OSDs holding a replica. The OSD is then able to manage the replication in a transparent way, i.e. the client is not involved in the replication process.

2.2 Read-only Replication

In XtreemFS users have the option to declare a file finally read-only. If this flag is set, the file system can easily keep multiple copies of the file. There is no need to coordinate the replicas since they are immutable. The main challenges in this replication mode are the efficient transfer of file data and the management of replicas. The latter is handled by the Replica Management Service (RMS), described in Section 3. The efficient dissemination of data has been extensively studied, e.g. [4]. We will evaluate existing strategies in terms of suitability for XtreemFS. The *object dissemination layer* will manage the actual data transfer between OSDs and implement different strategies suitable for read-only replicas as well as full replication.

2.3 Full Replication

Replication of regular files must also allow a file to be modified. This means that we need a mechanism to coordinate the updates and to guarantee the consistency of replicas. Basically, this mechanism must ensure a total order on the modifications which are then disseminated using the object dissemination layer described in the previous section.

2.3.1 Approaches to Replica Consistency Coordination

There are two schemes to create such a total order. Quorum based mechanisms negotiate the total order by using e.g. distributed consensus protocols among a subset of the replicas. These mechanisms are often resilient to host and network failures but at the cost of many rounds of message exchange [8]. In contrast, the primary/secondary scheme relies on a single replica chosen as primary. All modifications must be made at the primary which automatically creates the total order. The primary then disseminates the updates to the secondaries. As the primary is a single point of failure, an additional fail-over mechanism is required.

A completely different approach was taken by OceanStore [11], Pangaea [14] which relies on reconciliation of files when updates create conflicts. This reconciliation, however, cannot be used in a general purpose file system as it requires the file system to understand the contents of files.

2.3.2 Primary Failover with Leases in XtreemFS

In XtreemFS we will use the primary/secondary approach as it has better performance [8] and is easier to implement and test than quorum approaches [2]. This will constitute the *replica consistency layer* on top of the object dissemination layer.

For the primary failover we use a distributed lease negotiation based on Paxos to implement the *primary failover layer*. The details of this mechanism have been published by ZIB and BSC [6]. While the implementation of the leases mechanism is fairly complex, it is separated from the actual data replication.

Leases are a proved method to grant a process exclusive access to a resource for a limited amount of time. In our setting, the resource is to be primary replica for a specific file. Since the lease is only valid for a limited amount of time, we do not need additional failure detectors.

There are different situations which require another replica to take over the primary role. Apart from a failure of the primary, it is possible that another replica is close to the data consumers. Such a situation requires the ability to hand over the primary role between replicas. Our leases mechanism is designed to allow replicas to return a lease before it expires. This can be used to implement such hand-over mechanisms.

To avoid a central lock service, we developed a lease negotiation mechanism that is based on distributed consensus. This allows the replicas (more precisely the OSDs) to negotiate the lease among themselves. By using Paxos which is based on quorums the algorithm can tolerate host and network failures as long as a majority of replicas is available. In addition to the better failure tolerance, we also remove the bottleneck of centralized lock servers. In [6] we demonstrate that the algorithm is able to handle high workloads, i.e. concurrent requests, which makes it suitable for a file system.

Finally, the *replica set management layer* must ensure correct operation of the underlying layers even when replicas are added or removed. Especially, the primary failover layer is based on quorum decisions for which changes in the participating replicas must be carefully coordinated. A solution for this layer will be developed.

The layered architecture presented here makes the implementation and testing considerably easier since each layer can be tested individually.

2.4 Incomplete Replicas

Many grid data management systems can only create complete replicas, i.e. copy an entire file. This means that an application has to wait for the full file to be transferred before it can continue operation. Moreover, the entire file will be transferred even when an application needs only a few bytes. Those two issues can lead to delayed application start-up or execution and a waste of network bandwidth.

Since XtreemFS has full control over the replicas, it is possible to fetch only those objects of a file which an application really needs. When a replica is created it is initially empty but the application can instantly access the new replica. Data is fetched as the application requests it.

These incomplete replicas can, however, pose a problem when replicas are used to enhance availability and data safety. Therefore, XtreemFS will support different replica update policies which allow users to choose between performance (incomplete replicas) and data safety (complete replicas).

3 Design of the XtreemFS Replica Management System (RMS)

3.1 Functionality

In this section we will describe the advanced functionality that will be offered by the Replica Management System (RMS).

3.1.1 Choosing the best replica

One of the most important mechanisms in XtreemFS is the possibility to have several replicas of a file distributed over the Grid. The problem appears when a given client (or an OSD) has to access the file (which replica should it access?). The client should be able to detect which replica will give the better performance.

The idea to solve this problem is to build a virtual 2D or 3D 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. As we will not have more than few tenths of replicas, this computation should not be a significant overhead. In addition, if a given replica becomes unavailable (i.e. the OSD holding it fails), the client can pick the next best replica using the previously computed euclidian distance (it is important to notice that this operation does not need any communication between clients and RMS). If new replicas are added, the client will see them once the capability is renewed and then it can decide to switch replicas by computing again the euclidian distance of the new replicas.

In this process, we will need to study and evaluate what is the best metric to make this 2D or 3D placement. Already existing software use round trip times to compute distance, but we will evaluate if bandwidth or a combination of both gets better results in our environment.

Another problem that needs to be solved is how to decide the distance of a striped replica that is placed in several OSDs (thus different parts of the file will have different distances to the client). We will try simple solution to ease the task of selection a replica.

3.1.2 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.

Reactive replica creation. In this scenario we will implement mechanisms that detect when replicas are currently needed in other parts of the Grid. Using the distance mechanisms we just described in Section 3.1.1, we will detect if clients request a replica from large distances. In this case we will try to decide a better location for a replica and create it.

Proactive replica creation. In this scenario we will try to learn the usage of files to decide what files will be needed in the future (and where). To perform this prediction we will use access tries as was done in previous work by Kroeger [9, 10]. With this information we will try to create a replica in close-by place. This will allow applications to find the files they may need in a near OSD once they start running.

In both cases reactive and proactive, we will also 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).

Finally, we do not want to have an unlimited number of replicas and thus each file will have a limit in the number of possible replicas (see Section 3.1.3). This means that all replication mechanisms will take this maximum into account and never create more replicas than the ones allowed.

3.1.3 Automatic setting of the number of replicas

The problem of having many replicas is that updates imply that a coordination mechanisms has to be started. This coordination will reduce performance and the magnitude clearly depends on the number of replicas available.

For this reason we have decided to set a limit in the number of replicas a file will have.

On the other hand, it is clear that the overhead this coordination will imply also depend on the frequency at which files are modified. For instance, if a file is only modified once a month (and only small modification are done) we could keep many more replicas than for a file that is constantly modified.

The objective of this mechanism is to detect the access pattern of files and find the ratio between reads and writes. With this information the RMS will decide the maximum number of replicas that obtains a good tradeoff between the benefit of multiple replicas in read operations and the penalty of coordination in write operations.

3.1.4 Replica deletion

On the one hand, if we want to be able to replicate files whenever needed but still maintain the maximum number for replica per file, it would be interesting to keep the number of replicas a bit smaller that 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 give 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.

3.1.5 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.

In the first step, the AEM will ask which nodes from a list (the preselected nodes before the scheduling step) are closer to a set of files. Once again, XtreemFS will use the distance between files and clients described in Section 3.1.1 to answer this query to the AEM. At this step, it is important to realize that the AEM will only check file closeness of a reduced set of nodes that have been previously selected by other characteristics, thus it will be frequent that files needed are not close to the preselected 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 (3.1.2).

3.2 Design of the mechanisms needed

In order to implement the functionality the RMS will offer, we need to implement a few general mechanisms. In this section we will describe the main mechanisms, the integration in the XtreemFS architecture, and the way they will be implemented to guarantee a very high level of scalability.

3.2.1 Building the 2D or 3D virtual space

In order to build a 2D or 3D virtual space we plan to use a mechanism similar to Vivaldi [3]. Using this mechanism, each OSD or client just needs to contact a few other members of the space. With the round trip time (or any other metric we might evaluate), we find distance to these nodes and then compute their real location in the 2D or 3D space.

We plan to use both special messages (to find the initial location) and regular communication (to refine the location). It is important to notice that we do not need a very accurate placement but something good enough to allow the system to choose among available replicas. In addition, this constant checking using regular communication will allow moving clients to always have an updated position in the 2D or 3D space.

Finally, we will study what is the best option for our case: a 2D or a 3D space.

3.2.2 Gathering file access pattern

The information from the access pattern will be gathered in the OSDs and with no coordination among other OSDs. Each OSD will keep track of both the relationship among files (maintaining a trie as described in Section 3.1.2) and the important information of the way each file is accessed (the granularity of this information still needs to be decided).

As we want to have a more global view than what each OSD see by itself, we will work on some kind of aggregation.

Regarding the relationship between files, we will propose a decoupled and off-line aggregation of the tries. Once in a while (still to be determined), OSDs will 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.

4 Design of the XtreemFS Client-Side Caching mechanisms

XtreemFS is a distributed file system for Grid environments. As such, its components can be linked over low-bandwidth links with possibly high latency. File systems employ caching in order to improve performance in these situations. Caching becomes even more relevant in a Grid context, even if – like in XtreemFS – replicas are brought near the consumer. In this section we will describe the design of client-side caching in the advanced version of XtreemFS.

4.1 Problem Description

Client side caching is essential for the usability of the file system as a whole. It must make sure that I/O requests of an application using XtreemFS are finished within a reasonable amount of time. The application must not be delayed by I/O too long. It is also crucial that caching does not destroy the consistency of the file system.

As a Grid file system XtreemFS can have a potentially large number of clients. Some of these clients might also access the same file concurrently. So if each client caches data these caches must be synchronized in order to keep consistency of the whole file system. It is impossible for the clients to know of each other because they might be located in different sites that are separated by firewalls. So the clients cannot synchronize their caches among each other and they need the server components of XtreemFS for that purpose.

4.2 Architecture

In XtreemFS, the client is the mediator between an application that uses the file system and the XtreemFS servers, like MRC or OSDs. See fig. 1 for an overview of the infrastructure in which the client operates.

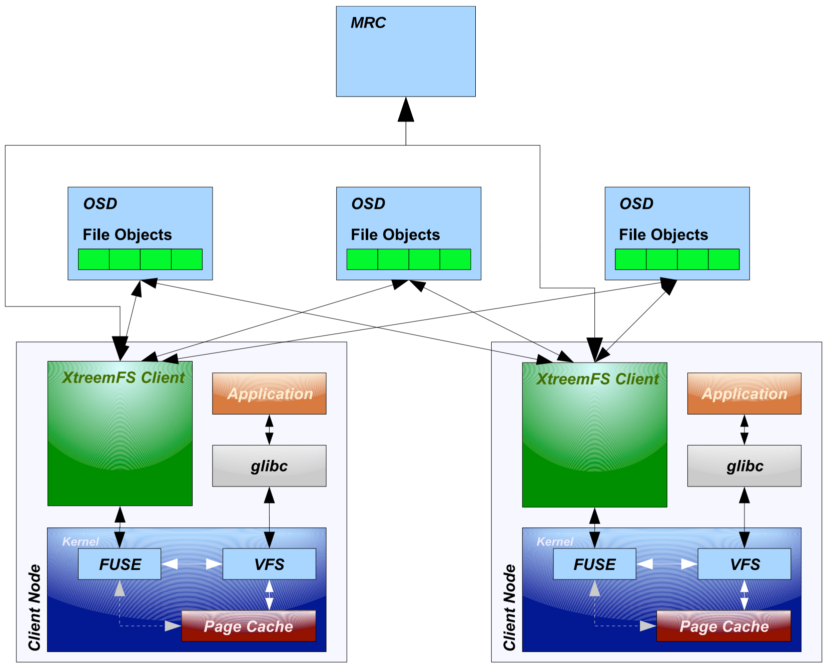

Figure 1: Overview of the relevant parts of the XtreemFS architecture for client side caching. I/O requests from an application are forwarded through glibc, VFS and FUSE to the client and via the client to the XtreemFS servers.

The application uses the libc library to communicate I/O requests to the kernels virtual file system (VFS). The VFS distributes these requests to the underlying file system. In the XtreemFS case this is the FUSE kernel part. FUSE communicates with the FUSE user space library on which the client is based. The client then handles the requests and employs the MRC or OSD accordingly. The problem is then to keep the caches of several clients consistent.

4.3 Design Proposal

The design of client side caching consists of several parts. One part is the inter-operation with other parts of XtreemFS and another part is the design of the cache within the client itself.

4.3.1 Inter-operation with Other Parts of XtreemFS

Due to the widely distributed architecture of XtreemFS it is not possible to synchronize the clients caches each time an I/O operation occurs. This would decrease I/O performance dramatically and contradict the design goals.

We base our design on the concept of leases. These leases allow a client to keep its cache for a certain amount of time. If the client wants to keep its cache longer than it has to renew the lease. There will be two kinds of leases: leases for writing and leases for reading. If there are only clients that want to read from a file, then there will be only read leases. Read requests can be executed concurrently without any harm. Data consistency is only threatened when data are changed by one of the clients.

Clients who want to cache data that they are likely to change must acquire a write lease. This lease contains the right to read as well. Because XtreemFS is an object-based file system this is necessary. The smallest possible unit of data movement is a file object which contains data from an application that is to be stored in the file system. The client caches will operate on this file object granularity, too. Writing to parts of an object requires fetching the whole object from the OSD. So the right to read is required for write leases. Write leases will be exclusive for parts of a file that a client wants to write to. This ensures that no two clients can have the same data cached and modify them in a different way. File system consistency is not violated. The locking mechanisms for file object locking must also make sure that consistency among replicas is not violated.

As clients do not know of each other, the OSDs have to synchronize the leases among themselves. They must make sure that only one write-lease at a time is issued to one client only. Requests for a read-lease will be blocked or denied if a write-lease is issued. The client will transport information about possible replica of a file that it has received from the MRC on file opening to the OSDs. The OSDs can then use that information to synchronize access to the replica as described in section 2.

4.3.2 Client Internal Cache Design

In order to serve a large number of applications concurrently the client is heavily multi-threaded. FUSE supports this concept by providing a multi-threaded interface to the client.

As stated above the client's main role is a mediator between an application and the XtreemFS infrastructure. As such it translates I/O requests from an application into something suitable for XtreemFS. This translation is done in several stages.

- **File R/W stage** This stage translates incoming application I/O requests into requests for file objects. Application I/O has the form of `offset` and `length`.
- **File object stage** This takes file object requests and translates them into stripe object requests. XtreemFS is intended to support different RAID levels and stripe objects are part of a stripe set according to a given RAID level. File objects contain application data only. RAID mechanisms create additional redundant information that allow a file system to recover the application data if parts of a file are lost. This stage creates the additional redundancy information.
- **Stripe object stage** This stages takes the stripe object requests and initiates the actual data transfer to and from the OSDs.

XtreemFS is intended to deal with different RAID levels. Therefore it is reasonable to cache stripe objects as part of a RAID stripe rather than file objects that contain pure data only. The stripe object cache will be placed between the (yet to be created) RAID stage and the actual stripe object stage that fetches and stores stripe objects. The RAID stage will allow for different RAID levels and eventually other data redundancy mechanism. It will generalize creation of redundancy information and distribution of data among OSDs that are incorporated into the file object stage right now.

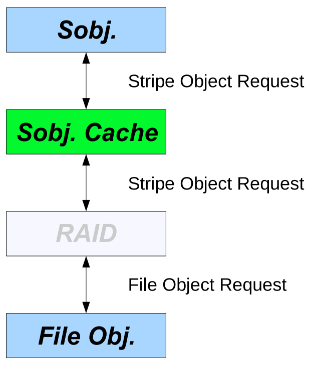

Figure 2: The new stripe object cache will be inserted between the file object stage and the stripe object stage.

The stripe object cache takes care of maintaining local copies of stripe data and takes also care of handling the leases. Only if it needs to involve OSDs is the stripe object stage invoked. So the file object stage (and later the RAID stage) only talks to the cache.

5 Design of the XtreemFS Advanced Storage Stage

5.1 Introduction

In this section we will describe some advanced mechanisms of the OSD Storage Stage, that we propose to study and implement, in order to improve its functionality and performance. To do that, first we will give a brief introduction by recalling the OSD Architecture and its components (including the Storage Stage), then we will discuss some improvements and advanced features that could be added to the current implemented solution.

5.2 The OSD Architecture

In order to give a self-contained document describing the file access functionality of XtreemFS, we want to give a brief preliminary introduction to the whole OSD.

The OSD is a fundamental component of XtreemFS, and it is responsible for storing file content. It relies on an event-driven architecture, composed of stages, according to the SEDA approach [16]. The internal architecture is depicted in Figure 3. The *Parser Stage* is responsible for parsing information included in the request headers. The *Authentication Stage* checks signed capabilities. The *Storage Stage* is responsible for storing and managing objects; for the sake of performance, the Storage Stage relies on caching that is used for fast access to objects which are still in memory.

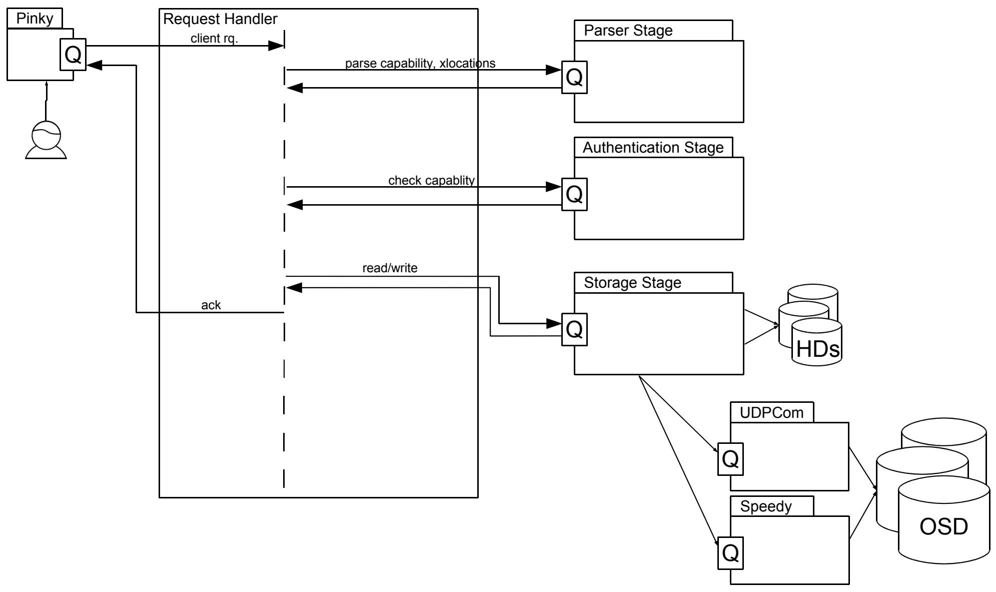

Figure 3: OSD internal design.

5.3 Storage Stage Improvements

The Storage Stage is a fundamental component of the whole OSD, and its performance really affects the whole OSD performance.

In order to make the Storage Stage dispatching the requests as efficiently as possible, in the following we describe some ideas that will be evaluated in order to improve the Storage Stage performances and functionalities. We divide them in two subsections. In the first, we analyze the way to make the Storage Stage able to process more requests concurrently by describing how a multi-threaded infrastructure could improve its performances in general. In the second, we describe some approaches for extending the functionalities of the StorageLayout, that defines how objects are stored on the physical storage device.

5.3.1 Multi-threaded

Besides the basic functionalities of the Storage Stage, concerning to the dispatching of requests and the way they are served out by the storage device, a fundamental aspect that needs more investigation is how to improve its performance. A multi-threaded infrastructure, that concurrently serves out more requests, might give advantages in terms of performances and functionalities. There are two main reasons that should make multi-threading improving performances of the Storage Stage, thus of the whole OSD component. The first reason is that, in a scenario contemplating more threads dispatching requests concurrently, some requests could be served out by the cache (with respect to the cache policy) and other requests could be served out by the disk. It is clear that the first operation is much quicker than the second. Therefore, meanwhile a request is dispatched by the disk (handling an I/O operation), other requests can be dispatched by the cache. The current prototype implements the single-threaded case, with only one running thread taking care of dispatching requests. In this case, thus, the thread (the request q has been assigned to) tries to serve out q by the cache and, if it can not do that, sends it to the storage device and waits for its response. Since there is just a single thread running for request dispatching, during this I/O burst no other operations on the Storage Stage are performed.

In order to make the StorageStage able to concurrently process more requests, we have to deal with two main issues. First, the creation of a thread pool infrastructure, whose solution can be simply implemented by an array of threads, i.e. *numThreads*, each one running and polling for the next request to be processed (the assignment of the requests to the threads depends on a dispatcher policy that needs to be designed). Second, and more complex, the implementation of a suitable strategy for managing mutual exclusion in accessing objects. This second aspect, that is a real case of the well-known readers-writers scenario at the granularity of the single object, could be solved in different ways, and in the following we will describe some of the feasible strategies to do that. Our proposal is to investigate some (or all) of them, and evaluate which one can be the most suitable for our purpose.

A first solution that could be adopted consists in using some software-based locking mechanism provided by the development language. For example, since the OSD (as the most of the XtreemFS components, excluding the client) are actually developed in Java, a Lock instance can be used for locking each data object (XtreemFS object) in reading or writing. In this case, the idea consists in having a shared structure that contains an array of locks, each one associated to each object currently accessed for reading or writing operations. In this way, if any thread (executing a new request) has to perform an operation on the object obj, its access is synchronized by the Lock associated to obj. For example, if a new read request has to be dispatched, and other read requests have been executing on the object, the Lock will contemplate that the thread can execute its request concurrently with other requests. If it can not (for example, because the object has been accessing in writing), the thread will go in a wait state (handled by its Lock) until its request can be dispatched (a fairness policy can be set for handling the waking-up of the threads in a FIFO order). A possible drawback of this solution consists in an additional CPU overhead due to the software structure for lock allocation (in memory) and handling (CPU overhead). Indeed, we should have as many Locks as the number of objects concurrently accessed, and this could be a large value. An interesting aspect to be evaluated is how much high this computational cost is.

However, it does support an optional fairness policy. When constructed as fair, threads contend for entry using an approximately arrival-order policy. When the write lock is released either the longest-waiting single writer will be assigned the write lock, or if there is a reader waiting longer than any writer, the set of readers will be assigned the read lock.

A second solution that could be adopted consists in distributing the requests to threads with respect to their $objNo$. According to such a scenario, we have a pool of threads running in background and waiting for the next request sent to them. This solution contemplates that, as soon as a new request r (read or write) has been received, the index of the thread that has to dispatch the request is computed by an hash function applied to the objID. In this way all the requests related to the same object are executed in sequence, by naturally solving the synchronization problem. Advantages of this solution are twofold: it solves the synchronization problem in a simple way and it does not add extra computational overhead (for locking mechanisms, as seen in the first case). Nevertheless, the drawback of this solution is that all the requests concerned to the same object are not concurrently executed (because they are assigned to the same thread, that executes them according a FIFO policy). This is not a problem for the write requests (this is exactly what the readers/writers scenario contemplates), but it could be a limitation for the read requests. We are confident that this limitation should not slow down the stage, also considering the fact that frequent requests are served out by the cache. Anyway, this needs more detailed investigation.

A third solution [12] that could be investigated consists in having only two threads running, the first aimed at serving out just requests from the kernel cache, and the second from the disk. This solution could speed up the requests, but we still need to clarify some aspects more in detail. As an advanced variant of such a proposal, we could also use a pool of cache threads and a pool of disk threads, in order to take benefit from the current multicore/multidisk infrastructures. Also this solution could be investigated.

A further issue to be investigated is the optimal setting of *numThreads*. Obviously, it concerns the first two solutions, but not the third. The value could affect the Storage Stage performance, because having both too many or too little threads could degrade performances. It is very hard to fix the best value a-priori, because the most suitable level of multi-threading depends on the hardware properties of the device where the OSD is currently running on (disk speed, number of disks, etc.). Our idea is to study some suitable metrics that can suggest the optimal value at run-time, and dynamically change it.

5.3.2 StorageLayout

The StorageLayout component is a sub-component that works as interface between the Storage Stage and the storage device. In other words, it takes care of mapping logical objects to physical objects stored on the disk. The way objects are arranged on the local file system is an issue to be studied. Actually, each file f is represented by a physical directory, and each object (part of f) is represented by a physical file (storing its data content). In order to avoid that the number of physical files in a directory becomes too large, each directory in turn is arranged in a directory hierarchy, by means of hash prefixes of their logical fileID. Indeed, we think that this is an acceptable solution (in terms of efficiency and functionality), but we will look for other solutions for optimizing the storage on disk. An idea could be to modify the StorageLayout, by driving the allocation of objects contiguously on disk in presence of an high intra-object locality of references (in order to reduce disk seek overhead) [15].

6 Design of the XtreemFS Testing

Testing might not be considered as a feature of XtreemFS but is nevertheless important. Indeed, extensive and regular testing is vital for any file system to convince users of the code quality and the stability. Therefore, we created a new task for WP3.4 which is focused on all aspects of testing. The goal of this task is to evaluate, adapt and execute existing file system test suits.

We have already built an automatic nightly test system that runs a small set of tests and file system benchmarks on a single machine. In the future, we plan to create automatic test environments which can execute the tests on several machines in clusters and over the Internet. Unfortunately, there are no ready-made tools available for such testing scenarios which means that we have to develop our own tools or extend existing distributed testing tools.

In addition to the testing environment, we have to evaluate the wide range of existing file system tests. We plan to use tests from the POSIX test suite of the Linux Kernel, stress tests from the Linux Testing Project (`http://www.ltp.org`), the tests developed for the NTFS-3G (`http://www.ntfs-3g.org`) Linux driver and standard file system benchmarking tools such as IOZone (`http://www.iozone.org`), Bonnie (`http://code.google.com/p/bonnie-64/`)or DBench (`http://samba.org/ftp/tridge/dbench/`). Most of the tests are designed for local file system and need to be adapted for XtreemFS.

Another set of tests will be implemented to validate the integration with the other components of XtreemOS that XtreemFS interacts with. This includes the security PAM module, SSL with XOS-Certificates and the Application Execution Management (AEM).

7 Design of Advanced OSS Features

In this section, we will describe the advanced features that we plan to implement for the object sharing service (OSS), especially to support distributed interactive multi-users applications such as Wissenheim (WP4.2).

7.1 Superpeer Overlay Network

In order to achieve scalability in the number of participating nodes, the nodes participating in an OSS instance will organize themselves in a hierarchical structure. So-called superpeers take additional responsibilities, such as high-level memory allocation, transaction validation, and replica location. Superpeers should have good network connections to the other nodes in the OSS instance. Thus, we will use a latency measurement and latency estimation subsystem for OSS similar to Vivaldi [3]. Shared object search among superpeers may be optimized by implementing a DHT over the superpeers [13].

We will consider interaction with WP 3.2 (Infrastructure for Service Scalability and High Availability) and with the XtreemFS Replica Management System. OSS instances will publish their presence either using XtreemFS or the PubSub Service, so that joining nodes can find at least some peers.

7.2 Advanced Transactions

7.2.1 Transaction Monitor

A transaction monitoring facility will be implemented within superpeers. It will be the base for adaptive optimizations and for programmers support. The first includes replica tracking allowing commits with limited scope. For example, if a peer commits it posts its write set to its assigned superpeer. If the superpeer knows that all objects within the write set are not replicated on peers subordinated to other superpeers, it can just perform a validation among its assigned peers. Obviously, this optimization helps avoid unnecessary global commits if possible. Another aspect is to support the programmer by providing statistic information, e.g. access patterns, conflicting transactions, etc. This will help the programmer to simplify the identification of conflicting transactions and to evaluate redefined transaction boundaries. This iterative optimization process also benefits by the underlying validation that always guarantees correctness (even if some transaction may scale poor).

7.2.2 Pipelined Transactions

In order to hide network latency, we will implement pipelined transactions for OSS. As speculative transactions are validated at commit time, write sets need to be propagated to participating peers. Depending on the network latency, number of peers, and size of the write set, this validation phase can get too long for interactive applications. Instead of waiting for the validation result we prefer to start the next transaction. Although this can lead to cascading aborts it is better than waiting. Of course it is not reasonable to implement a lot of pipeline stages but this is anyway not necessary. Typically, network latencies in wide area networks are around 500 ms. The execution time of short transactions is 0.5 - 1s. Thus allowing one outstanding commit per node will in most cases be sufficient to hide the typical wide area network latency.

7.2.3 Hierarchical Validation Among Superpeers

In the grid environment, peers involved into data sharing may be arranged over long distances. The number of communication messages and the network latency have an important impact on validation performance. We plan to implement a validation among superpeers. A peer will not validate its transactions itself but will delegate this task to its assigned superpeer.

7.2.4 Bounded Commit Propagation (Optional)

As an extension of the hierarchical validation protocol we plan optionally to validate transactions only among a subset of peers. In order to accomplish this, page requests will be monitored in the hierarchical overlay network by the superpeers. The latter will allow to detect dependencies between hosted object replicas on the peers and posted write sets of validating transactions. Subsequently, the superpeers can decide if received write sets must be forwarded to other superpeers and peers or not.

7.2.5 Transaction History Buffer

Validating and committing transactions automatically induce posting the appropriate write sets to all other nodes (or to a subset). In some situations (e.g. selective commits, network faults etc.) nodes can miss commits. This could lead to data consistency violations, if these nodes commit their own transactions afterwards. One solution would be to post write sets using a reliable overlay multicast. Numerous efforts are described in literature but to the present there is no ultimate solution. As reliable multicasts are expensive and complex we do not want to implement them for speculative transactions. We plan to use a logical time allowing a node that is about to commit a transaction to detect if it has missed any previous transaction from other peers. If yes, the peer has first to request all previous missed write sets before being allowed to commit its own transactions. To handle requests of missed write sets, all peers implement a transaction history buffer where committed write sets will be stored. This buffer will be of limited space thus allowing peers to recover within a certain time interval. If an on-the-fly recovery is impossible, we will fall back to a checkpoint using the grid checkpointing and restart service from WP3.3.

7.3 Advanced Memory Management

7.3.1 Hierarchical Allocator

A concurrent object allocator must be synchronization-free in the common case [1]. Therefore, we will implement a hierarchical allocator where allocations are satisfied at the lowest level possible. Nodes reserve chunks of free storage for future allocations from their superpeer. A superpeer can donate some of its free storage to another superpeer.

7.3.2 Support for Fine-grained Memory Allocations

The current allocator in OSS maps objects injectively onto memory pages. The direct mapping is well suited for special applications that handle fine-grained memory allocation themselves. However, an object allocator for fine-grained allocations will simplify using the OSS and can additionally improve object placement within memory. For example, objects should be placed adjacently if their access pattern indicates temporal locality (true sharing), and they should be placed on different logical memory pages if their access pattern indicates accidental collisions (false sharing). If concurrent object relocation is not supported by a programming environment the default allocation scheme is one logical memory page per object [7]. Storing several objects on one page needs to be declared by the programmer during memory allocation.

7.3.3 False Sharing Control

If a lot of small objects are allocated on separate logical memory pages the caching effect is lost, also in case of locality (true sharing). As the sharing pattern changes over time, e.g. false sharing can turn into true sharing and vice versa, it is necessary to implement an adaptive consistency unit management. Therefore, subsequent object allocations that share a physical memory page will belong to a single cache consistency unit (similar to the situation if these objects would have been allocated on the same logical memory page). If conflicts are detected by the planned object access monitoring facility, the false sharing control component can remove objects from this consistency unit. A background task will examine scattered objects trying to merge them again into a consistency unit to avoid loosing true sharing over time for all object groups. We expect that the latter will be a very challenging task.

7.3.4 Transactional Object Allocator

As an advanced feature, we plan to make object allocations more convenient for transactional applications by allowing allocations to occur during speculative transactions. Allocations performed within a speculative transaction must be undone in case the transaction aborts. It seems to be natural to store object metadata within transactional memory.

8 Conclusion

In this report, we have described the design of features that will advance the possibilities of XtreemFS and OSS. These new features will improve all parts of the system, and help it to provide innovative features that are important for its adoption.

In particular, XtreemFS will be the first file system that provides a real replication mechanism, while not sacrificing any POSIX compliance. Furthermore, its replication features can be used in a fully autonomous manner and will be integrated with the rest of the XtreemOS infrastructure.

With its advanced client-side caching mechanisms and protocols, XtreemFS will be allow high-performance access to file data event in presence of multiple clients and storage server, again without sacrificing POSIX compliance. The performance will complemented by a fast access to local storage on storage servers that shall be able to exploit the capabilities of the underlying hardware.

The Object Sharing Service will continue to advance its primitives for data sharing between remote processes, and improve its scalability and memory management mechanisms.

References

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGARCH Comput. Archit. News*, 28(5):117–128, 2000.

[2] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[3] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM'04 Conference, Portland*, August 2004.

[4] Mathijs den Burger and Thilo Kielmann. Mob: zero-configuration high-throughput multicasting for grid applications. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 159–168, New York, NY, USA, 2007. ACM.

[5] C. Grimme, T. Langhammer, A. Papaspyrou, and F. Schintke. Negotiation-based choreography of data-intensive applications in the C3Grid project. In *German e-Science Conference*, 2007.

[6] Felix Hupfeld, Björn Kolbeck, Jan Stender, Mikael Högqvist, Toni Cortes, Jesus Malo, and Jonathan Marti. FaTLease: Scalable fault-tolerant lease negotiation with paxos. In *HPDC 2008*, to be published.

[7] 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.

[8] Ricardo Jiménez-Peris, M. Patino-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[9] Thomas M. Kroeger and Darrell D. E. Long. Predicting file-system actions from prior events. In *Proceedings of the USENIX Annual Technical Conference*, pages 319–328, Berkeley, January 22–26 1996. Usenix Association.

[10] Thomas M. Kroeger and Darrell D. E. Long. Design and implementation of a predictive file prefetching algorithm. In USENIX, editor, *Proceedings of the 2001 USENIX Annual Technical Conference: June 25–30, 2001, Marriott Copley Place Hotel, Boston, Massachusetts, USA*, pub-USENIX:adr, 2001. USENIX.

[11] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[12] Adrien Lebre, Guillaume Huard, Yves Denneulin, and Przemyslaw Sowa. I/o scheduling service for multi-application clusters. In *CLUSTER 2006: Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006.

[13] Yin Li, Xinli Huang, Fanyuan Ma, and Futai Zou. Building efficient super-peer overlay network for dht systems. In *GCC*, pages 787–798, 2005.

[14] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[15] S. A. Weil. Leveraging intra-object locality with ebofs. Ucsc cmps-290s project report, May 2004.

[16] 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.