Project no. IST-033576

XtreemOS

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

Design and Implementation of High Performance Disk Input/Output Operations in a Federation D2.2.5

Due date of deliverable: November 30th, 2007
Actual submission date: December 21st, 2007

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.5

Responsible institution: INRIA
Editor & and editor's address: Adrien Lebre
IRISA/INRIA
Campus de Beaulieu
35042 RENNES Cedex
France

Version 1.0 / Last edited by Adrien Lebre / December 19, 2007

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

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	16/10/07	Adrien Lebre	INRIA	Initial template + Outline
0.2	20/11/07	Adrien Lebre	INRIA	Taking into account Adolf and Christine remarks
0.3	4/12/07	Adrien Lebre	INRIA	Taking into account Marko remarks

Reviewers:
Adolf Holh, Marko Novak

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.5	Design and implementation of high-performance disk input/output operations in a federation	INRIA*

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

Executive Summary

Most of the available network file systems for clusters are built on the historical model compute nodes *vs* storage nodes. Available hard drives on compute nodes are only used for the system and temporary files, wasting both a lot of space and throughput, predominant criteria in the current High Performance Computing context.

In this document, we present a kernel Distributed File System (kDFS) designed to efficiently exploit storage resources within a cluster. Moreover to provide common distributed file systems functionalities, kDFS aims at providing an integrated cluster file system dedicated to HPC. From our point of view, fine and efficient use of storage resources can only be reached by combining all cluster services.

The first kDFS prototype has been implemented upon kernel Distributed Data Manager (kDDM) mechanisms, a kernel DSM-like manager which allows consistent data sharing clusterwide. Thanks to the kDDM set, kDFS provides a cooperative cache for both data and meta-data. To our knowledge, such an approach is quite innovative since kDFS does not exploit any other mechanisms or network protocol to exchange and synchronize data.

The first prototype implements basic file system functionalities. Advanced mechanisms such as data striping, redundancy and I/O scheduling enhancements are under heavily development.

Even though the current activities are done under the framework of XtreemOS WP2.2 work, The majority of the concepts can be extended to traditional clusters. Finally, kDFS evaluation is under progress, we hope to get preliminary results by the end of 2007.

Contents

Chapter 1

Introduction

The XtreemOS operating system is intended to be executed on all computers in a Grid, making their resources available for use as part of virtual organizations. There will be three XtreemOS flavours, one for each kind of Grid node: individual computers (typically for PCs), clusters and mobile devices.

As described in the ”Description of Work” document [9], the XtreemOS cluster flavour relies on the SSI approach. In a cluster all nodes work closely together so that in many respects they can be viewed as though they were a single computer. A Linux SSI operating system provides the illusion that a cluster is a virtual multiprocessor machine executing Linux. For XtreemOS grid services, a cluster executing LinuxSSI-XOS will be seen as a powerful PC executing Linux-XOS.

Former investigations from WP2.2 work package [10] led to leverage the Kerrighed SSI technology developed by INRIA in cooperation with EDF [14, 17, 21, 23, 32].

In this document, we present the design and the implementation of high performance disk input/output operations within LinuxSSI-XOS. Major functionalities that were implemented by the end of M18 are emphasized.

The most common approach for building a distributed file system in a cluster context consists in exploiting specific nodes to provide the distributed/parallel file system. If the existing systems offer many features, they strongly rely on the hypothesis that they are deployed on dedicated nodes. Thus, the cluster nodes are divided into two groups: the compute nodes and the I/O nodes. The hard drives available on the compute nodes are only used for the system and temporary files, thus wasting both a lot of space and throughput.

In our view of an SSI cluster OS, all nodes could potentially provide both CPU and storage resources. Thus, our proposal should be able to efficiently exploit most of the available storage and, in the meantime, take into account the resource usage of the applications (CPU, memory, network and hard drive). Such an approach in designing a file system for a Linux SSI should lead to several innovative works. Indeed, our proposal should obviously provide the common features of a distributed file system but moreover it should exploit, cooperate with and complete the SSI system itself by improving services and global performance.

Based on this assumption, we have designed and started to implement kDFS, a kernel Distributed File System. The first prototype described in this document aims to:

- Aggregate storage resources available within a cluster,
- Provide a unique clusterwide name-space,
- Provide efficient accesses to both small and large files.

Coordination and integration with other cluster services will be deeply addressed in a future document.

The deliverable is organized as follows. Chapter 2 presents major works done in the distributed/network file system field. Chapter 3 is focused on kDFS design and the implementation. Chapter 4 gives details about current investigations done about integration and coordination of kDFS with other cluster services. Finally, Chapter 5 concludes the document.

Chapter 2

Background

I/O bottlenecks have always been a major issue in Computer Science. As early as 1967, the issue of storage and computation efficiency has been addressed [1]. Almost forty years later, this lack of performances is confirmed [15] and this trend is likely to continue as I/O hardware performance increases slower than CPU and memory. Furthermore, this gap is amplified by the increasing use of clusters as well as the number of scientific and commercial application developments (molecular biology, climatology, nuclear physics, financial, web services . . .) with ever more demanding I/O requirements and different patterns of access.

As a consequence, lots of researchers have attempted to develop new I/O subsystems that take into account both hardware aspects and parallel computing accesses. These new systems address various issues like: security, efficiency, scalability, ...

In this chapter, we first review the major solutions suggested by the community and then we conclude by raising the major disadvantage of the current approaches from our point of view. Most of the available solutions are built on the historical model "compute nodes *vs* storage nodes". Available hard drives on compute nodes are only used for the system and temporary files, thus wasting both a lot of space and throughput, predominant criteria in the current High Performance Computing context.

2.1 Distributed/Network File Systems

Distributed/Network file systems [19] have been suggested early in the eighties. They aimed at providing a way of sharing files among computers.

Created by SUN Microsystems in 1985, the Network File System (NFS) became the first widely used distributed file system. Nowadays, NFS is still the most deployed solution to share data within a LAN or a cluster. The NFS consortium composed of several institutes has finished the version 4.1 of NFS. This new version extend the stateless client/server model to a parallel statefull one.

The AFS family (AFS, OpenAFS, CODA) is also a well known approach. It focuses on providing a secure way of sharing files within a campus for instance. It offers an efficent way to share files securely on a large scale. Yet, for a safe environment such as a protected cluster, this solution is not optimal since it adds many features that are not necessary for a cluster and requires some not-so-obvious tweaking. Such file systems are more suited for Grids.

The serverless network file system, xFS [2] is probably the most finalized solution from conceptual point of view. This file system has been designed under the framework of the Network of Workstation (NoW) project at Berkeley 10 years ago. This is the first fully distributed file system (i e., all mechanisms are present on every node, thus in case of a node failure, the file system would not be impacted). Unfortunately, only one prototype had been implemented and it had not been maintained.

Due to the large scope of constraints (LAN, WAN, security, high-availability, scalability, ...), latter proposals were more focused on particular aspects. SAN and parallel file system approaches have been introduced to mainly improve performances in a cluster context.

2.2 SAN File Systems

SAN file system such as GFS [25], GPFS [26] and SGI XFS [29] are based on the use of specialized technologies (e.g. RAID, fiber optics) as a means to increase performance. This kind of systems relies on the concept of a dedicated network (Storage Area Network) for storage messages (such as SCSI requests). Basically, a SAN defines a com- mon storage ”device” composed of several physical devices. Other projects like Petal/Frangipani [31] are built upon the concept of a distributed virtual disk: a set of daemons running on a number of machines cooperate to form the view of a single storage device. The Shared Logical Disk [13] follows the same idea: the SAN is implemented in software over a network.

If such solutions increase performance in both cases (soft or hard SAN), efficiency and scalability heavily depend on the underlying communication technology.

2.3 Parallel File Systems

Parallel file systems were suggested to solve the cluster constraints. They exploit hardware capabilities as efficiently as possible while taking into account distributed file system constraints (coherency, fault tolerance, remote accesses, ...) in a cluster context.

Several solutions have been suggested [7], [20], [27], ... Roughly in order to achieve high performance on I/O operations, they distribute file system mechanisms across a set of nodes in a cluster. Two types of nodes are used: the I/O server and the manager, which is a metadata server. Data is striped across several

I/O nodes. Thus, clients can access files in parallel. The way the files are striped is handled by the metadata manager, which is also responsible for managing file properties and a global name space to applications. The server, however, does not particpate I/O operations. The I/O servers are accessed directly by the clients to deal with data transfers. Finally, users mount the file system using a POSIX interface, and access to files as any other file systems.

Lustre [27] is currently one of the most powerful solution. It is an object-based file system designed to provide performance, availability and scalability in distributed systems. It relies on the meta-data/I/O server model previously described. Lustre runs on many of the largest Linux clusters in the world (as large as 25,000 nodes) [**?**].

2.4 Compute *vs* Storage Nodes Paradigm

Even if available solutions such as parallel file systems provide some ways to exploit distributed storage resources, they do not directly rely on resources available within the cluster. Instead, a set of nodes (storage nodes) is dedicated to provide file system mechanisms. Such a model is mainly historical: file management in distributed environments has been mostly addressed by external and independent solutions such as centralized or distributed servers. This approach makes a clear separation between applications and data management and thus facilitates design and implementation of file systems (cf. Figure 2.1). However, it wastes a lot of both storage space and throughput.

The current architecture of the Rennes Grid5000 clusters relies on such a model.

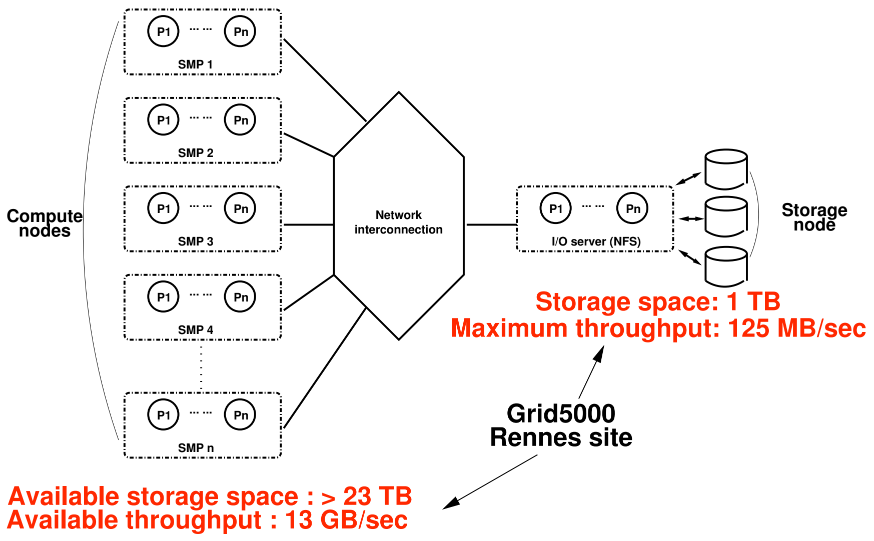

Figure 2.1: Compute nodes *vs* Storage nodes
Inefficient use of disk capabilities (space and througput)

In a the same way as most of small and medium size clusters, the ”storage nodes” group is composed by only one NFS server. This server is used to share 1TBytes with a maximum throughput of 125MBytes/sec[1], whereas on the compute nodes side, more than 23TBytes of space are available with an aggregated throughput closed to 13GBytes/sec.

In a context where fine and efficient building blocks are required for building higher level mechanisms, the design and the implementation of an highly efficient file system exploiting all available hard drives in a cluster is needed. From our point of view, such a file system should obviously provide common functionalities but also exploit, cooperate with and complete the SSI system itself by improving services and global performance.

The next chapter presents the design and the implementation of the first kDFS prototype.

[1]In this typical NFS architecture, the bottleneck appears at network level on server side. In the Rennes case, a GigaEthernet card.

Chapter 3

Kernel Distributed File System

Based on Linux and implemented at kernel level, LinuxSSI offers a Posix compliant interface. Hence legacy applications can be directly executed without any modification or recompilation on top of the SSI. Kerrighed implements global and dynamic resource management by a set of distributed services.

In this chapter, we first describe one of the key components of Kerrighed: the Kernel Distributed Data Manager. This service is the building block of our proposal, kDFS, presented in the second part. We have voluntary chosen to design and implement a fully distributed file system based only on the kDDM service. By reducing Kerrighed dependencies, we hope to facilitate the insertion of the LinuxSSI mechanisms into the Linux mainstream development more easily.

3.1 Kernel Distributed Data Manager

The Kernel Distributed Data Manager, kDDM [21, 22], allows consistent data sharing cluster wide. All operating system services access the physical memory through the kDDM service. In a cluster, each node executes its own operating system, which can be roughly divided into two parts: (1) system services and (2) device managers. The kDDM is a generic service inserted between the system services and the device managers layers in order to give the illusion to system services that the cluster physical memory is shared as in an SMP machine. This concept was formerly called *container* and has been renamed to kDDM to avoid confusion with current kernel container mechanisms.

kDDM *sets* are integrated in the core kernel thanks to *linkers*, which are pieces of software inserted between existing device managers and system services.

3.1.1 kDDM Sets

A kDDM set is a specific structure to store and share similar objects clusterwide. Each set can store up-to 2^{32} objects of 4KB. Each set is linked to high level services (virtual memory, file system, ...) *via interface linkers* and to devices storing data (memory, disk) *via I/O linkers*.

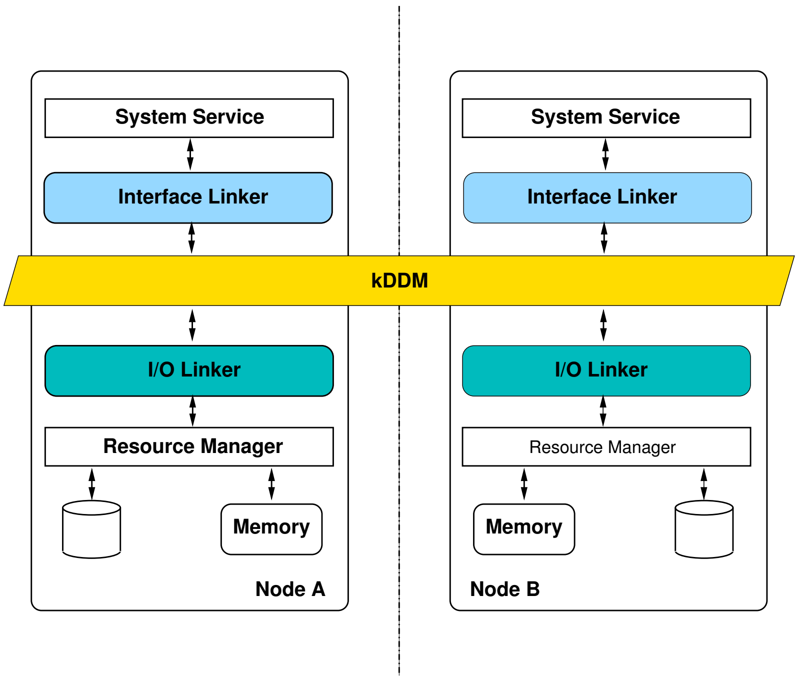

Figure 3.1: kDDM overview

Thus, all kDDM sets are completely transparent to user level software. Data is stored in a kDDM set on host operating system demand and can be shared and accessed by the kernel of other cluster nodes. Pages handled by a kDDM set are stored in page frames and can be used by the host kernel as any other page frame.

By integrating this generic sharing mechanism within the host system, it is possible to give the illusion to the kernel that it relies on top of a physically shared memory. On top of this virtual physically shared memory, it is possible to extend to the cluster, traditional services offered by a standard operating system. Such an approach allows first, to keep the OS interface which is known by users, and second, to take advantage of the existing low level local resource management. The figure 3.1 presents the general architecture.

Last but no least, the memory model offered by kDDM service is sequential consistency implemented with a write invalidation protocol. This model is the one offered by a physically shared memory.

3.1.2 Linkers

Many mechanisms in a kernel rely on the handling of physical pages. Linkers divert these mechanisms to ensure data sharing through kDDM sets. Each kDDM set is associated with one or several high level linkers called *interface linkers* and a low level linker called *input/output linker*. The role of interface linkers is to divert device accesses of system services to kDDM sets while an I/O linker allows a kDDM set to access a device manager.

System services are connected to kDDM sets thanks via interface linkers. An interface linker changes the interface of a kDDM set to make it compatible with the high level system services interface. This interface must give the illusion to these services that they communicate with traditional device managers. Thus, it is possible to "trick" the kernel and to divert device accesses to kDDM sets. It is possible to connect several system services to the same kDDM set. For instance, a kDDM set can be mapped in the address space of a process P1 on a node A and a process P2 on a node B can access to it thanks to a read/write interface.

During the creation of a new kDDM set, an input/output linker is associated to it. The kDDM set then stops being a generic object to become an object sharing data coming from the device it is linked to. The kDDM set is said to have been instantiated. For each semantically different data to share, a new kDDM set is created. For instance, a new kDDM is used for each memory segment to share or to be visible cluster wide.

Just after creation, a kDDM set is completely empty, i.e. it does not contain any pages and no page frame contains data from this kDDM set. Page frames are allocated on demand during the first access to a page. Similarly, data can be removed from a kDDM set when it is destroyed or in order to release page frames when the physical memory of the cluster is saturated.

3.2 kDFS Overview

In our vision of a fully distributed file system for cluster, all nodes can potentially provide both CPU and storage resources. Thus, our proposal should be able to efficiently exploit most of the available storage and, in the meantime, take into account the resource usage of the applications (CPU, memory, network and hard drive). In other words, in addition providing the common fonctionalities of a distributed file system, our proposal should exploit, cooperate with and complete the cluster services itself by improving usage and global performance.

Keeping that in mind, we have designed and implemented a first prototype of a distributed file system named kDFS (kernel Distributed File System). This section gives an overview of the current implementation of the distributed file system. Preliminary investigations on coordination with other cluster services are addressed in Chapter 4.

The design of kDFS relies on two main concepts:

- Using the native file system available on each node (avoiding block device management),
- Using kDDM sets to provide a unique and consistent clusterwide name space.

The use of kDDM sets makes the implementation of a global distributed cache with Posix semantics easier since kDDM mechanisms directly insure meta-data and data consistency.

Figure 3.2 illustrates the overall architecture of kDFS.

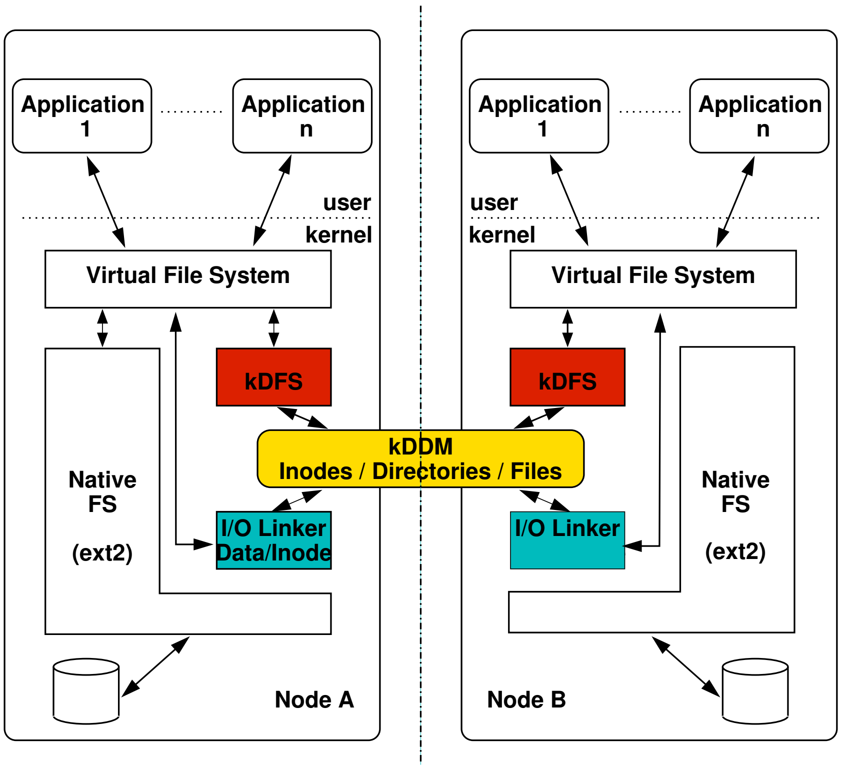

Figure 3.2: Overview of the kDFS system

3.2.1 Disk Layout

To avoid block device dependencies and make storage management easier, we have chosen to build kDFS upon native file systems provided by the cluster nodes. As a consequence, each node has to specificy if it takes part in the kDFS storage space or not. Storage space provided by a particular cluster node is considered as a kDFS partition only if it has been kDFS formatted. As kDFS does not directly manipulate block devices, a kDFS partition actually refers to a local directory with a particular hierarchy. This section introduces different sub-directories and different files contained in a kDFS partition. For instance, the '...' one which stores kDFS superblock informations.

To get a storage space which can be used afterward in the kDFS physical structure, administrators has to use the `mkfs.kdfs` command[1] [12]. This command takes two arguments: `DIR_PATHNAME` and `ROOT_NODEID`. The former one corresponds to the absolute path to store kDFS meta-data an data, the latter one is the node identifier for the kDFS root entry (the node that stores the kDFS root inode).

`mkfs.kdfs`, creates the kDFS "superblock" file (...) for the node. This file is stored on the local native file system in the given directory. If the current node identifier (the IP generally) equals to the given *id*, `mkfs.kdfs` creates the root entry.

[1]Associated to kDFS, a dedicated command, `mkfs.kdfs` has been implemented. This command formats a directory which can be used afterward in the kDFS physical structure.

Table 3.1 describes the creation of a kDFS structure distributed between two nodes:

On node A: (nodeid = 1)	on Node B: (nodeid = 2)
`mkfs.kdfs /PATH1 1`	`mkfs.kdfs /PATH2 1`
Create kDFS local '...' Create kDFS root entry	Create kDFS local '...'

Table 3.1: kDFS structure creation (two nodes)

For each entry (a directory or a file) of the kDFS hierarchy, a "native" directory is created on one kDFS partition. This directory contains two files:

- The `.meta` file stores meta-data associated with the entry (size, timestamp, rights, striping informations, ...)
- The `.content` stores real data.

The name of the "native" directory is defined by the kDFS local inode identifier (each kDFS superblock contains an identifier bitmap to define next free inode *id*).

To avoid large directory issue, we have arbitrarily chosen to sort each kDFS partition on a one hundred range basis. For instance, when `mkfs.kdfs` creates the kDFS root entry, (the first entry of the kDFS hierarchy) the command first creates a sub-directory '`DIR_PATHNAME/0-99/`'. Then, it creates the corresponding "native" directory which is the `DIR_PATHNAME/0-99/1/` directory. Finally, the file '`.meta`' and the file '`.content`' are created inside this latest directory.
Every hundred entries, kDFS adds a new sub-directory corresponding to the appropriate range ('`DIR_PATHNAME/100-199/`', '`DIR_PATHNAME/200-299/`', '`DIR_PATHNAME/300-399/`', ...).

To access kDFS file system, users has to mount it by using the following command:

```
mount -t kdfs ALLOCATED_DIR|NONE MOUNT_POINT
```
`ALLOCATED_DIR`: native file system directory formatted with `mkfs.kdfs`
`MOUNT_POINT`: traditional mount point.

The table 3.2 describes kDFS mounting procedure from two nodes:

On node A: (nodeid = 1)	on Node B: (nodeid = 2)
`mount /PATH1 /mnt/kfds`	`mount /PATH2 /mnt/kdfs`

Table 3.2: Mount kDFS partitions
`/mnt/kdfs` is now a global kDFS namespace for both nodes

Since files and directories are stored in a distributed way on the whole cluster, we need mechanisms to find the kDFS entry point and thus be able to join the file system. kDFS provides two ways of retrieving the `root` inode. The first one is based on the superblock file stored in the formatted kDFS partition. As mentioned, the kDFS superblock file provides several informations including the kDFS root inode id. Thus, when a mount operation is done, the '`...`' file is read from '`ALLOCATED_DIR`'and the root inode id is used to retrieve the kDFS root entry. The second mechanism concerns diskless nodes or nodes which do not want to take part of the kDFS physical structure. In such a case, users do not give any device but have to provide the kDFS root inode id as an additive mount parameter.

We stil have not extended the mount operations with specific kDFS parameters. Thus, for the moment, There are few limitations:

- One kDFS partition per node
- Only one mount point per node
- Diskless mechanisms not available.

Moreover, we plan to take advantage of the kDFS superblock file to add some QoS parameters such as allowed storage space, rights, ...for each kDFS "partition".

3.2.2 File System Architecture

The first version of kDFS has been implemented by using three kinds of kDDM:

- Inode kDDM set, one clusterwide. It provides a cache of inodes recently accessed by processes.
- Dir kDDM set, one per directory. each Dir kDDM set contains directory entries (roughly names of subdirectories and files).
- File kDDM set, one per file. It stores data related to the file contents.

Figure 3.3 depictes the kDDM entities. To make read and understanding easier, Table 3.3. gives a potential representation of the regular files for each kDFS entries mentioned in the figure.

Next sections introduces each of these three kinds of kDDM sets. A fourth kDDM set is depicted in Figure 3.3: the dentry kDDM set. This unique kDDM set provides a distributed cache to manage all dentry objects. It is currently not implemented and requires deeper investigations.

3.3 kDFS Inode Management

kDFS inode management relies on one global kDDM set. This set exploits a dedicated I/O linker for inserting/removing and maintaining each kDFS entry on the corresponding local file system (sub-direcory creation/deletion and updating of

On node A: (nodeid = 1) `DIR_PATHNAME`: /PATH1	on Node B: (nodeid = 2) `DIR_PATHNAME`: /PATH2
`mount /PATH1 /mnt/kdfs`	`mount /PATH2 /mnt/kdfs`

`/mnt/kdfs/`
`/PATH1/0-99/1/.meta`
`/PATH1/0-99/1/.content`
	`/mnt/kdfs/data/`
	`/PATH2/0-99/1/.meta`
	`/PATH2/0-99/1/.content`

`/mnt/kdfs/lib`
`/PATH1/0-99/2/.meta`
`/PATH1/0-99/2/.content`
	`/mnt/kdfs/data/xtreemos/`
	`/PATH2/100-199/104/.meta`
	`/PATH2/100-199/104/.content`

`/mnt/kdfs/lib/xtreemos.so`
`/PATH1/200-299/280/.meta`
`/PATH1/200-299/280/.content`
	`/mnt/kdfs/data/xtreemos/config`
	`/PATH2/100-199/105/.meta`
	`/PATH2/100-199/105/.content`

...
...
...
	`/mnt/kdfs/lib/linuxssi.so`
	`/PATH2/100-199/180/.meta`
	`/PATH2/100-199/180/.content`

Table 3.3: Translation between kDFS entries and regular files stored on harddrives
The global kDFS namespace is distributed on two nodes

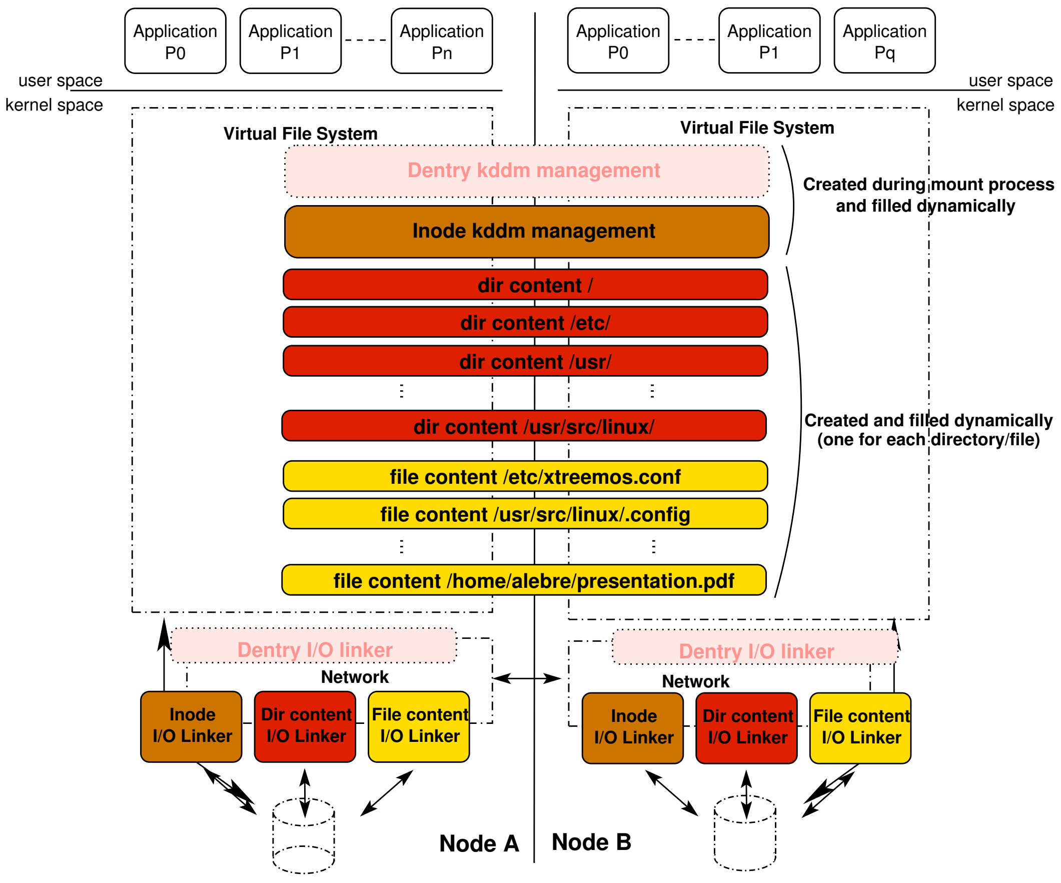

Figure 3.3: KDFS internal layers

the '`.meta` file'). The inode set is created during the first kDFS *mount* operation within the cluster. At the beginning, it only contains the kDFS `root` inode. Then, when a process wants to access one file/directory, its corresponding kDFS inode is added into the set (providing by this way a fully distributed inode cache). All file/directory creations are performed locally whenever possible. That means, when an process wants to create a new file or a directory, kDFS looks for a kDFS partition. If there is one directly available on the node, a new inode identifier is get from the superblock file otherwise, kDFS stores the new entry on the same node as the parent directory.

When a mount operation is done, the root inode identifier is used to retrieve the root inode structure directly inside the inode kDDM set. If the kDDM set already exists, the structure is already cached and the inode is simply returned to the node. In the opposite case, the I/O linker associated to the inode set exploits the inode id to retrieve required informations from the relevant hard drive within the cluster. KDFS inodes are currently based on 32 bits, the 8 MSB bits provide the node id within the cluster and the 24 LSB ones correspond to the local id.

The inode management proposal has some scalability limitations due to our current implementation of a 32 bits inode. kDFS can federate at most 256 nodes and manage only one kDFS partition with a maximum of 2^{24} files for each node. We plan to fix such issue by extending the inode size to 64 bits as it is already done by several file systems (XFS, NFS, . . .).

3.4 kDFS Content Management

Since kDFS file hierarchy is based on native file systems, both directories and files are simply stored as regular files. Thus, as contrary as traditional file systems, the management of a large kDFS directory containing lot of directory entries is similar to the management of a kDFS file content. Whatever the kDFS entry, its content is stored in its respective '`.content`' file.

After briefly describing directory and file content manipulation, the section focuses on optimization mechanisms such as read-ahead, write-behind orr I/O scheduling and how they could be exploited to improve the overall performance of kDFS. The last paragraph introduces data-striping and redundancy mechanisms.

3.4.1 kDFS Directory Management

When an application tries to list the contents of a directory stored in the kDFS storage space, kDFS creates a new directory kDDM set. This new set is linked to the inode object stored in the inode kDDM set and caches all directory entries on a page basis. In other words, all file and subdirectory names stored in the directory are loaded in objects of this new kDDM set (one object corresponding to one page). After that, all filename manipulations such as create, rename and remove apply modifications of these pages. The associated dir I/O linker is in charge of propagating changes to the proper hard drive (into the '`.content`' file)

3.4.2 kDFS File Management

In this first prototype, kDFS file management is similar to directory management: when an application tries to access a file f, if f is not already "cached", kDFS creates a new file kDDM set. As for directory management, this new set is linked to the corresponding inode object of the file f. The associated file I/O linker is in charge to read/write data from/to the `.content` file of f and remove/put pages in the set.

3.4.3 Read-ahead and Write Behind

Caching is a widespread technique used to reduce the number of accesses to hard drives and thus to improve performance of the I/O system [6]. In a local context, such mechanisms mainly appear at high level in the OS I/O stack (in the VFS for Linux, cf. Figure 3.4). Due to kDFS placement within the I/O stack and the kDDM design, cache mechanisms could be applied twice in kDFS: at low level (when I/O linkers access the native file system, cf. 1 on Figure 3.4) and at high level (when linkers access kDFS, cf. 2 on Figure 3.4). At the first sight, it seems logical to exploit such techniques to improve performance whatever the level they can be applied. However, after a deeper study, the naive use of read-ahead and write-behind at high level lead to a negative impact on both efficiency and consistency.

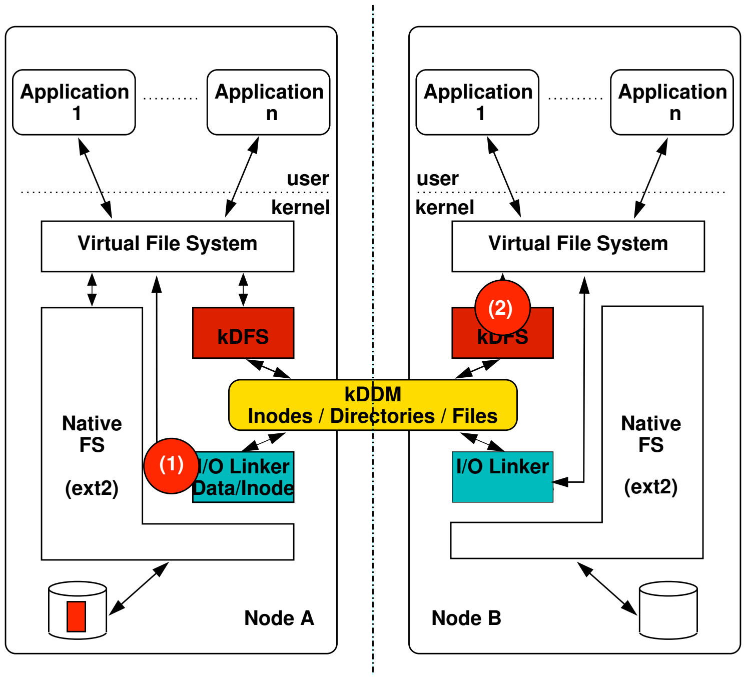

Figure 3.4: Cache in kDFS

At low level, read-ahead and write-behind strategies only depend on the local file system and can be compared to an internal buffer cache available in hard drives. In other words, if a process reads or writes in one kDFS file stored on a remote node, cache mechanisms applied on the remote node at the I/O linker does not change the consistency within the associated kDDM set. At high level, it is a bit more sophisticated. Interest and impact of each strategy require to be tackled one by one.

- The read-ahead strategy has been suggested to mask disk latencies. This technique is mainly based on the sequential behaviour of readers and can even decrease performance in case of random accesses (wrong pages are prefetched). In a distributed context such as the high layers of kDFS, the use of read-ahead can either improve performance in some cases or impact network traffic in some other ones. Typically, for a distributed application, some useless chunks of one file may be prefetched from several nodes. These pages go through the network several times, increasing the network congestion probability. Thus, a performance tradeoff between prefetch mechanisms and network congestion should be defined. The disk latency is about $9ms$ in current harddrives whereas network latency tends to be negligible at cluster level (< 1 μs). So, we have chosen to leave aside read-ahead at high level for the moment.

- The write-behind strategy at high level is a real issue. From the performance point of view, such a technique is crucial to avoid page migration from one node (high level) to a remote one (low level). Let's consider a process that is writing to a remote file byte per byte. A write-through strategy generates a page migration to acknowledge each write operation whereas a write-behind strategy significantly reduces network traffic by aggregating several writes before flushing data to low level. However, in the event of a failure, such an approach may lead to a consistency issue. Indeed if a crash occurs on a node writing into the file, data contained in the write-behind buffer is simply lost, although these writes have been acknowledged. To avoid such critical situations, we have decided to only implement a write through strategy in the first version of kDFS: each write is directly propagated to the corresponding I/O linker. We, currently work on the design of derived mechanisms to improve kDFS performance in case of write requests.

3.4.4 I/O scheduling

Several scheduling algorithms have been proposed to minimize completion time of a batch of I/O operations. These algorithms usually fall into two categories: disk scheduling [28] and parallel I/O scheduling [8, 16]. The first one tends to limit disk head movements while the second one distributes parallel I/O operations to different I/O servers to minimize the overall response time.

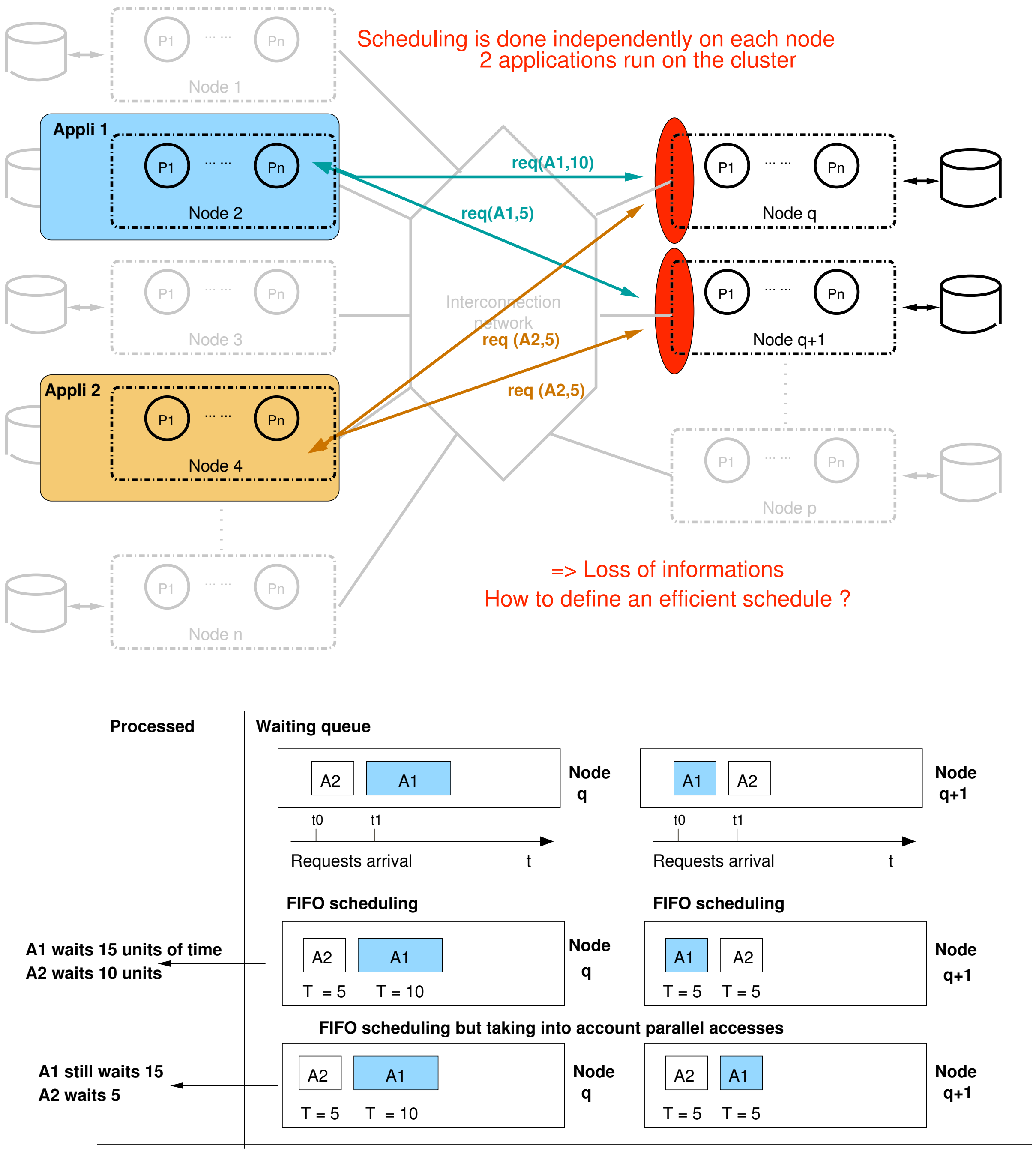

Figure 3.5: Parallel scheduling constraint

In order to provide a good tradeoff between performance and fairness in a distributed context, we have to study the integration of both approaches: exploit low level schedulers and complement them with new strategies more suited to distributed constraints. Indeed, recent work showed the importance of such strategies in a multi-programmed cluster where several applications are executed concurrently, competing for accessing storage subsystems [18]. The I/O subsystem layer has to perform optimizations that take advantage of accesses regularity from each application while balancing storage access between them. Most of the available file systems do not use I/O scheduling strategies as they are just built on schedulers located in the block device layer. At this low level, due to kernel and file system implementation applications information is not available and I/O access patterns cannot be exploited for throughput optimization.

Based on the aIOLi work [18], we currently investigate a first approach to add similar I/O scheduling strategies at I/O linker levels on each node. As Figure 3.5 shows, to be optimal, I/O scheduling strategies need to have a global point of view. However such a global view only appears inside the kDDM manager itself. Implementing such I/O scheduling mechanisms at this level will require strong development efforts since it will directly impact the kDDM design itself. For the moment, we have chosen to focus on the ”node basis” and expect that it will contribute to improve throughput by maintaining a QoS for each application.

3.4.5 Striping and Redundancy

The objectives of striping and redundancy policies are twofold: first, to balance I/O requests over several nodes in order to decrease the amount of requests per node (and thus increase the scalability) and second to benefit from aggregating throughputs provided by several nodes. Indeed in a conventional cluster, some dedicated nodes attached to RAID devices are exploited to deliver the expected bandwidth for each application. In an SSI cluster, each node can provide its storage support which corresponds in most of cases to one traditional hard drive with a throughput peak around 60MBytes/sec. In such a case, striping and replication policies[2] are mandatory.

As mentioned in the D2.2.1 deliverable [10], kDFS is designed to provide two striping modes: the first one is automatic and transparent whereas the second one is based on user parameters. Currently, we have almost finalized the design of the automatic approach and the implementation is under heavily development. As a reminder, in the transparent mode, all data is stored locally. That is, all write requests are propagated on local hardrives. Two cases should be considered, sequential and parallel access:

- Sequential access: only large files (out-of-core) could suffer from such an approach. A way to solve this issue is to stripe a file on multiple hard drives

[2]We distinguish striping policies to improve performance from the ones that target fault tolerance aspects. Fault tolerant mechanisms are not addressed in this deliverable.

when its size is bigger than a defined threshold.

- Parallel access: the performance should be excellent. For instance, in MPI applications, each MPI instance writes its data locally according to the striping policy. This policy is selected by the application (CYCLIC, BLOCK/ BLOCK, . . .). From the file system point of view, there is no fixed striping size. This mode should improve the performance since it avoids all striping issues which may appear when the file striping does not correspond to the application one.

We have chosen to focus our effort on implementing the transparent mode since, in contrast to the second mode, it does not require to extend the POSIX API[3]

All placement information for each file is stored within the associated kDFS meta-data file (cf. Section 3.3). The striping geometry is based on an "object" granularity (from 1 to n block). For instance, the meta-data lists all objects: a first object which is p blocks long is stored on the first drive of node x and a second one, q blocks long, is available on the drive of node $y, \ldots (p \neq q)$.

The use of replication is also a well-known approach to improve efficiency. Instead of accessing only one file stored on one hard drive, clients are balanced on replicas available on distinct nodes. In the particular case of kDFS such an approach is not required. Indeed, when a client accesses a file, this file is immediately stored in the associated kDDM set. Thus, all subsequent accesses are directly satisfied by the clusterwide cache and do not reach hard drives. Based on current trend of RAM memory size increasing, we can imagine that each node will have soon a memory size close to 10 GB. Deploying the kDFS solution on such a cluster will provide a large and efficient clusterwide buffer cache.

We expect to finalize the first mode of striping mechanisms by the end of 2007.

[3]In the "user" mode, each user will be able to define a specific striping policy on a per file or per directory basis.

Chapter 4

kDFS: Towards an Integrated Cluster File System

As we mentioned in Chapter 2, there has been a lot of work done on distributed file servers [5, 19] since the early eighties with various issues tackled: security, performance through caches, high availability, disconnected mode, consistency,...

By designing kDFS, we should look one step beyond providing just another new distributed file system. We should take advantage of the XtreemOS framework to suggest innovative contributions to the HPC file system field. From our point of view, fine and efficient use of storage resources within a cluster can only be reached by combining all cluster services. We expect from such an approach to improve obviously storage performance but also global usage of the cluster. This chapter addresses preliminary investigations regarding the integration and the coordination of kDFS with other cluster services in the framework of LinuxSSI.

4.1 Linux SSI Scheduler and I/O probes

To increase global cluster performance, schedulers have to take into account resource usage by each process. In most of works done, schedulers take into account processor load and in few case memory usage. The MOSIX operating system [3] has probably suggested the most complete scheduler. However, all these studies consider CPU and memory monitoring mechanisms without taking I/O apsects into account I/O aspects. From our point of view, one of the major contributions of kDFS will consist in coordinating file system information and scheduler decisions. Two work directions have been identified:

- First, we have to improve data locality. Processes should be launched where required files have been stored. Moreover to increase I/O efficiency, such an approach also impact network by significantly decreasing traffic.
- Second, we have to provide I/O probes which can measure average hard drive load and hard drive usage per application (concurrency within a node). In that sense, we plan to analyze how processes impact each other within

a node when hard drives are stressed (network intensive process versus I/O intensive process, CPU intensive versus I/O intensive, memory intensive versus I/O intensive, number of processes, ...). Moreover taking into account the load generated by the "user" processes, we should consider the resources exploited by the SSI file system itself [4] (mainly `swap` impact). Such a model will make design and implementation of the I/O probes easier. Finally, I/O probes will be also directly used by kDFS to define which nodes are the best candidates for storing data.

In the context of LinuxSSI-XOS, kDFS will communicate with the new scheduling framework implemented in task T2.2.4 [11]. This framework has been designed for facilitating integration of new probes and customized scheduling policies. By using such mechanisms, we expect to improve both performances and global usage of the cluster.

As we mentioned, our main objective consists in going one step beyond traditional cluster file systems. We want to go toward an integrated cluster file systems. To continue to improve efficiency and global usage of the cluster, we will combine kDFS, SSI scheduler and migration mechanisms.

As an illustration, we present an example based on the execution of the well-known IOR benchmark. The I/O Stress Benchmark Codes[1] emulates the behaviour of a parallel I/O intensive application. It consists of a parallel code developed by the Scalable I/O project at Lawrence Livermore National Laboratory. This parallel program performs parallel writes and reads to/from a file using the `POSIX` or `MPI I/O` API and reports the throughput rates. Roughly, it repeats the two following steps:

1.
   ```
   For each MPI instance:
   Reads particular data from one common file (according to its MPI rank),
   Processes them,
   Writes results in a second common file
   ```

2.
   ```
   Permutation between processes is made
   Restart step 1./ from last result file.
   ```

In a traditional distributed file system, such an experiment leads to both an I/O and a network intensive application. Each MPI instance has to read and then write data from/to the file system: data is read from I/O server(s) goes through the network, is processed and results get back to the file system.

The use of local hard drives as it is implemented in kDFS already contributes to the reduction of network traffic. For the first phase, the SSI scheduler launches each MPI instance on the proper node where data is read and written. The use of, in the first hand, I/O probes and in other hand, migration mechanisms contribute to reduce network traffic. Instead of accessing remote data during the second phase, I/O probes notify kDFS about remote accesses and kDFS informs the scheduler to migrate processes on the appropriate nodes storing the data.

[1]`http://www.llnl.gov/asci/purple/benchmarks/limited/ior/`

Distributed FS		kDFS (no migration)		kDFS (with migration)	
HDD	Net	HDD	Net	HDD	Net
8GB	8GB	8GB	2GB	8GB	negligible

Table 4.1: IOR benchmark on a 2GBytes file size
Magnitude order of data exchange for one IOR iteration
(HDD: Hard Disk Drive)

Table 4.1 summarizes expecting I/O requests and network traffic for one iteration of the IOR benchmark with a file-size of 2 GBytes (each MPI instance is running on a dedicated node). As we can see, in addition avoiding congestion due to the multi-application phenomena on the distributed file system, the integration of migration processes mechanisms in kDFS should significantly improve performance by directly exploiting hot cache present on each node.

4.2 Checkpoint/Restart Mechanisms

Checkpointing mechanisms are exploited to recover applications in the event of failures. Unfortunately, most of available solutions consist in saving only the process state without taking into account open files. Such a feature is needed to restart processes in a consistent state from the files point of view. In this section, we present preliminary investigations on the coordination between checkpointing mechanisms and kDFS.

The major objective is to provide an efficient file checkpointing functionality: file checkpointing mechanisms provide a way to reverse changes performed in a file between two process checkpoints when an application is restarted from a checkpoint. This feature is missing in the current checkpointing service.

An audit of different available solutions is currently in progress [13] [24] [30]. Our first investigations led us to propose an extension to Linux I/O API (more precisely to the Virtual File System) by adding some calls which enable to give specific states to files: ”normal” or ”checkpointable”. When the file is tagged as a checkpointable file, a copy-on-write policy is associated with it. Thus, all subsequent writes accesses on it trigger copying of the old data in another shadow file. We choose to extend the POSIX API instead of just use traditional open flags. So, it is possible to change the state of a file at any time. This could be useful to checkpoint a whole application which did not anticipate to exploit checkpointing services (for instance, when a system administrator makes a manual checkpoint before stopping a node). When an application restarts from its last checkpoint, another VFS call enables to reverse changes in the file and to restore the correct image. We could imagine several versions of such a checkpointable file: the most simple is based on one copy which is overwritten at each checkpoint and a more sophisticated one might be an incremental checkpoint based on several shadow files.

A second aspect concerns placement of checkpointed daa. First, we have to identify which nodes are susceptible to store the data. To solve this issue, kDFS should consider the ”Mean Time Before Failure” value of each node but also current I/O load. As for swap or other cluster services, checkpointing mechanisms will impact I/O performances. Thus, the coordination between checkpoint services and kDFS is also important to minimize overhead. Another issue will be to deal with live migration implied by the scheduler: a process should not migrate where its checkpointing data has been stored. More generally, placement of checkpoint data requires deeper investigations. We expect to have first results by M24.

Chapter 5

Conclusion

There have been a lot of works done in network/distributed file systems with various issues tackled: security, efficiency, availability, consistency, ... Most of available solutions are built on the historical model compute vs storage nodes wasting a lot of space as well as throughput, predominant criteria in a HPC context.

In this document, we have presented kDFS, a new kernel Distributed File System. Based on several concepts suggested in KerFS (Kerrighed 1.02), the kernel Distributed File System has been developed from scratch. The first challenges consisted in developing a distributed file system pluggable under the VFS and only based on the kernel Distributed Data Manager component of Kerrighed. kDDM is used to build a cooperative cache for both data and meta-data.

The first prototype has been implemented with 3 kinds of kDDM sets: inode, dir and files. It provides a simple clusterwide file system. Current efforts are focused on efficiency aspects, mainly on cache page replacement and striping policies.

Associated to kDFS, a dedicated command, mkfs.kdfs has been implemented. This command formats a directory which can be used afterward in the kDFS physical structure. This command is released with the current kDFS prototype. Documentation of kDFS and mkfs.kdfs command appear in the D2.2.7 deliverable [12].

Our proposal has to go one step further than just being another new distributed file system. Moreover to federate storage resources within a cluster, kDFS should exploit, cooperate with and complete the SSI system itself by improving services and global performances. In that sense, we have introduced our preliminary investigations on the integration and the coordination of kDFS with the other cluster services. First results about kDFS, an integrated cluster file system for High Performance Computing, are expected by M24. A dedicated section[1] on the kerrighed website has been created. It contains presentations, documentations and will give latest news about kDFS.

Finally, this deliverable focuses on the design and the implementation of kDFS

[1] `http://www.kerrighed.org/wiki/index.php/KernelDevelKdFS`

and thus we do not direclty address the integration with XtreemFS, the grid file system. [2] The main issue for XtreemFS consists in exploiting the parallel bandwith provided by kDFS. After several exchanges with WP3.4 members, it appears that this issue has to be taken into account not only for kDFS but for all parallel file systems (such as Lustre or PVFS for instance).

[2]As a remainder, the integration of kDFS within XtreemOS components only concerns XtreemFS since other XtreemOS services do not interact with kDFS [10].

Bibliography

[1] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *In AFIPS Conference Proceedings vol 30, AFIPS Press, Reston, Va,*, pages 483–485, 1967.

[2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *Computer Science Division, University of California at Berkeley, CA 94720*, 1995.

[3] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[4] K. Yoshii Beckman, P. Iskra and S. K. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedinfs of IEEE International Conference on Cluster Computing*, pages 1–12, sept 2006.

[5] Peter J. Braam. File systems for clusters from a protocol perspective. In *Extreme Linux Workshop #2, USENIX Technical Conference*. USENIX, June 1999.

[6] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[7] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[8] F. Chen and S. Majumdar. Performance of parallel i/o scheduling strategies on a network of workstations. *Eighth International Conference on Parallel and Distributed Systems*, 2001.

[9] XtreemOS consortium. Annex 1 - description of work. Integrated Project, April 2006.

[10] XtreemOS consortium. Specification of federation resource management mechanisms, November 2006.

[11] XtreemOS consortium. Design and implementation of customizable scheduler, November 2007.

[12] XtreemOS consortium. Prototype of the basic version of LinuxSSI, November 2007.

[13] B. CORNELL, P. DINDA, and F. BUSTAMANTE. Wayback: A user-level versioning file system for linux, 2004.

[14] Pascal Gallard. *Conception d'un service de communication pour systèmes d'exploitation distribué pour grappes de calculateurs: mise en oeuvre dans le système à image unique Kerrighed*. Thèse de doctorat, IRISA, Université de Rennes 1, IRISA, Rennes, France, December 2004.

[15] J. L. Hennessy and D. A. Patterson. Computer architecture: A quantitative approach, 1996.

[16] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[17] Kerrighed website. http://www.kerrighed.org. http://www.kerrighed.org.

[18] Adrien Lebre, Guillaume Huard, Przemyslaw Sowa, and Yves Denneulin. I/O Scheduling service for Multi-Application Clusters. In *Proceeding of the IEEE International Conference on Cluster Computing, Barcelona, SP, to appear*, Sept 2006.

[19] Eliezer Levy and Abraham Silberschatz. Distributed file systems : Concepts and examples. *Departement of Computer Sciences, University of Texas at Austin, Texas 78712-1188*, 1990.

[20] Pierre Lombard, Yves Denneulin, Olivier Valentin, and Adrien Lebre. Improving the performances of a distributed nfs implementation. In *Proceeding of the 5th International Conference on Parallel Processing and Applied Mathematics, Czestochowa, Poland*, September 2003.

[21] Renaud Lottiaux. *Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en œuvre dans le système Gobelins*. Thèse de doctorat, IRISA, Université de Rennes 1, December 2001.

[22] Renaud Lottiaux and Christine Morin. Containers: A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid (CCGrid '01)*, pages 66–73, Brisbane, Australia, May 2001.

[23] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou, and Isaac Scherson. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *Proc. of Cluster 2004*. IEEE, September 2004.

[24] Dan Pei, Dongshen Wang, Meiming Shen, and Weimin Zheng. Design and implementation of a low-overhead file checkpointing approach.

[25] Kenneth W. Preslan, Andrew Barry, Jonathan Brassow, Russel Catalan, Adam Manthei, Erling Nygaard, Set Van Oort, David Teigland, Mike Tilstra, and Matthew T. O'Keefe. Implementing journaling in a linux shared disk file system. *in the 8th NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the 7th IEEE Symposium on Mass Storage Systems*, 2000.

[26] Roger L.Haskin Frank B. Schmuck. Gpfs: A shared-disk file system for large computing clusters. *In Proceedings of the 5th Conference on File and Storage Technologies*, January 2002.

[27] Phil Schwan. Lustre : Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium, Ottawa*, July 2003.

[28] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of USENIX*, pages 313–323, 1990.

[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, 22–26 1996.

[30] Vítor N. Távora, Luís Moura Silva, and ao Gabriel Silva Jo˙ Distributed checkpointing mechanism for a parallel file system. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 137–144, London, UK, 2000. Springer-Verlag.

[31] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[32] Geoffroy Vallée. *Conception d'un ordonnanceur de processus adaptable pour la gestion globale des ressources dans les grappes de calculateurs : mise en oeuvre dans le système d'exploitation Kerrighed*. Thèse de doctorat, IFSIC, Université de Rennes 1, France, March 2004.