Distributed Hybrid-Storage Partially Mountable File System

Abstract: We have extended the classical approach of a disk partition to a distributed environment. In this way, a *NetFS* partition is composed out of several storage *fs_nodes* that are stored on different type of media and/or computer systems. The file system can be partially mounted, meaning that mounting can be done using only a subset of the *fs_nodes*. Due to the design of this kind of system, a partition can be dynamically extended or shrunk while it is mounted without any data loss. Our paper presents the implementation of this file system using FUSE (which in term offers some advantages – usage of library functions, stability, portability), the problems encountered and some future work yet to be done.

Key–Words: distributed, file system, hybrid-storage, copy-on-write, extensible, memory, network, partial mounting

1 Introduction

File systems are one of the core components of any modern computer system. This allows users to store, organize and retrieve their data. Nowadays, computers aren't anymore the only ones using file systems. Mobile phones, PDA's, portable media players starting with the very simple $10 player to the very famous iPod have storage organized using computer file systems and partitions.

Moreover, today computers *do not work alone anymore*. Almost any *smart* electronic device, computer, PDA, mobile phone etc., is connected to a network, usually to the Internet. The amount of data that it needs to store and process has increased significantly over the past years. If some years ago watching a movie on a computer was great, today most of the mobile phones are capable of this. A lot of research has been done in creating and improving the way in which large amounts of data can be stored distributed over several storage devices [6].

The file system that we propose, *NetFS*, is a distributed, hybrid-storage and partially mountable user space file system. There are several file systems that use concepts similar with *NetFS*. During the following paragraphs we shall focus only on few examples.

UnionFS [4] is a service that implements a union mount for different types of file systems, providing a unified view for all of their files. Those file systems are known as branches. *NetFS* is not mounted on top of other files systems, but uses the same concept of files stored on different storage devices known as *fs_nodes*. Like UnionFS, *NetFS* supports dynamic insertion and removal of *fs_nodes* (or branches).

Even if *NetFS* is implemented in user space, which makes it slower than kernel based UnionFS, it has several advantages over UnionFS like flexibility and partial mounting option.

4.4BSD Union Mounts [7] is a file system-namespace management tool. It merges several file systems into one virtual system. Union Mounts has some similarities to UnionFS and as UnionFS it allows to dynamically add and remove the top branch. If we compare the features of Union Mounts with those offered by UnionFS or *NetFS*, we will notice that there are some features which it does not implement yet. It does not permit for any nodes to be dynamically added and removed from the system while it is mounted. Also it cannot have multiple writable branches, only the file system at the top of the union stack can be modified.

MRAMFS [2] is an in-memory file system. It uses for meta data and data storage the non-volatile RAM (NVRAM). This file system offers performance improvement over a disk based file system. Unlike *NetFS*, MRAMFS only uses memory for storage, which means that it is very limited in terms of available space. To compensate this limit, MRAMFS uses data compression. The data compression mechanism is applied to meta data structures and to file blocks. Because *NetFS* is a hybrid system that can use storage adapted to the size of the data to be stored, the implementation of data compression cannot be justified under current circumstances.

During the paper, we will present our implementation solution, the way we have solved several distribution problems (conflicts, error-handling, etc.) and some tests that we have performed on the system.

2 Architecture

2.1 Overview

The goals that we wish to achieve with this file system are listed below.

1. A *NetFS* partition is distributed over several independent parts, called *fs_node*;
2. The *fs_nodes* belonging to a partition can be of several types (stored on different types of media – RAM, file, network, etc.);
3. The *fs_nodes* of a *NetFS* partition can be mounted independently or in subsets;
4. All the meta data about the *NetFS* partition and its files will be stored only on its *fs_nodes*, no additional control files or storage systems will be used;
5. The meta data about files will not be replicated among the *fs_nodes*, meaning the *inodes* structure will not be duplicated in any way;
6. The files will be partially accessible (meaning only the blocks that are on the currently mounted *fs_nodes* are available) if they are present in a folder which is on one of the currently mounted *fs_nodes*;
7. The system needs to be extensible, meaning that new file storage types may be added without changing the file systems source code.

Distribution The *NetFS* partition is split into several independent *fs_nodes*. Each node contains enough meta data about the partition so that it can be mounted separately and also allow the extension (adding nodes) or shrinking (removing nodes). This is very useful when used in a network environment where several *fs_nodes* of the partition can be split over the network computers. Moreover, it also allows the combination of storage space on several removable media (USB drives, CD-ROMs, CD-RAMs, etc.).

Hybrid *NetFS* partition *fs_nodes* can be stored on different type of media. The system will function the same, no matter what kind of storage media is being used for the *fs_nodes*. Storage examples are *files*, *RAM memory*, *physical disks*, *removable media*, *network computers*, etc. By the time of writing of this paper, we used the first two and the last one.

Partially Mountable The novelty of this file system, is the ability to mount partially. This means that not all the *fs_nodes* have to be available at all time. One can read the files which have the meta data (*dirent*) and part of the data (*inodes*) on the currently mounted *fs_nodes* and may write files in the limit of the available space. Moreover, new *fs_nodes* may be added *on the fly* with certain conditions.

User Space The goals of this type of file system can be best achieved by implementing it in user space. This means that even if it might not be as fast as a kernel space file system, it is able to implement more functions. We are talking here about the ability to use network connections, virtual memory and provide more stability to the system. In case of failure, the system kernel will not be affected.

Extensible As stated before, *NetFS* is a hybrid storage file system, *fs_nodes* being able to be stored on different media. Due to its construction, support for new storage types may be added without changing the code of the entire file system. As an example, we have added an improved RAM memory storage system, which assures persistence. The files from memory can be backed-up on disk while the system is mounted if the CPU has an average load less than 2%. When this *fs_node* is remounted the data are restored from the disk.

Due to the complexity of the file system, we have chosen a layered architecture inspired by the OSI model. This makes the development and maintenance very easy. Layers can be changed without affecting the other file system components.

The internal organization of *NetFS* is inspired by MinixFS [8]. Consequently the file system meta data are stored in *inodes*, directories are stored using classic *dirent* structures and each *fs_node* contains a *superblock*. The main difference is that data blocks addresses contain the *fs_node* id besides the block id.

Building the file system driver in user space means actually building a normal program. This was possible by installing a the FUSE [3] kernel module which redirects system calls to the file system into this program, and forwards the responses to the system's virtual file system. This approach makes the operating system more stable as crashes of the file system will not affect the kernel.

Surprisingly, as our tests will show later on, there is no performance loss due to the system calls forwarding. Moreover, using the cache features of the kernel module, the file system might even be faster.

2.2 NetFS Layers

NetFS is designed using three layers: *File Manager*, *Partition Manager* and *Read/Write Modules*. Each layer provides functions for the layer above and uses the functions provided by the layer below. They basically communicate using a standard interface, so changing one layer does not require changes into the others. The schematics of the architecture is illustrated in figure 1.

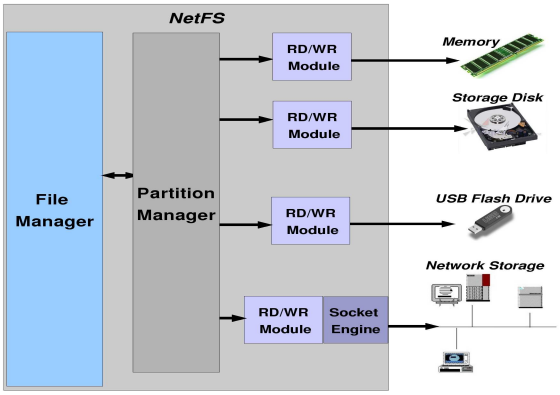

Fig. 1: NetFS Structure

We will discuss now each layer in detail, pointing out the main features of each one.

2.3 File Manager

The File Manager is the top most component of *NetFS*. Its main purpose is to provide the interface with the user programs, keep track of the partition's meta data (*inodes*) and manage files (create, modify and delete). It receives all the system calls redirected by the kernel module to the file system.

The File Manager relies on the Partition Manager for interaction with the distributed file system. This means that it will send requests for meta data allocation, modification and deletion, and reading and writing requests for data. From the point of view of the File Manager, the file system is not distributed, it looks like any normal partition.

2.4 Partition Manager

2.4.1 Overview

The middle layer of *NetFS* is the Partition Manager. It implements most of the key features of the file system: *distribution*, *partial mounting* and *file system management*. It is the core of the file system.

2.4.2 Distribution

One of the most important feature of *NetFS* is the distribution. The file system is able to transparently distribute a single partition on several *fs_nodes* of different storage types (memory, file, USB stick, network, etc.).

The partition's files have to be distributed across the *fs_nodes*. This implies distributing file blocks and directory entries. Depending on the file system's usage, *NetFS* supports several algorithms:

- **Closest space** - if it is possible, new file blocks will be distributed on the same *fs_node* as the *inode* of the file, if not another *fs_node* will be chosen randomly, first from the locally stored *fs_node*, and than from the *fs_nodes* stored on the network and, as a last resort, from RAM stored *fs_nodes*.

  In this case, we have a classic file system, which uses the distribution just in case it runs out of space. The local stored *fs_nodes* are chosen first due to speed considerations and high availability (they will persistent and most probably available all the time, regardless of the computers connectivity). The second choice are the network stored *fs_nodes*, as they are persistent, but not always available. The last solution are the RAM *fs_nodes*. They are always available, but not persistent, so data will be deleted on shut down.

- **Priority** - to each *fs_node* is given a priority, so new blocks will be distributed starting with the highest priority *fs_node* which has free available space.

  This could be useful for systems where a partition is distributed over several persistent storage devices with different speeds. The fastest device will have the *fs_node* with the highest priority, while the slowest device will have the *fs_node* with the lowest priority. For instance, small laptops with solid state hard drives (usually a small and fast one and a large and slow one) could use this system.

  Another usage of this system would be a combination between some read-only media (with no free available disk space) and writable media. It is similar to the approach of Union-FS used on Live CD's.

- **Equal distribution** - new blocks will be distributed in such a way that they fill up the currently mounted *fs_nodes* equally. The *fs_node* with the largest free available space will be chosen first.

This leads to a uniform distribution of data. This system is similar to implementing a partition over several hard drives using RAID without any redundancy.

2.4.3 Partial Mounting

Our file system is able to be mounted using only a subset of its *fs_nodes*. Moreover, no centralized meta data about the entire file system is stored, meaning that the only meta data available regarding the partition is the one stored on the mounted *fs_nodes*. This however creates several consistency and redundancy problems.

The first problem that arises due to distribution and partial mounting is creating and deleting *fs_nodes*. Each *fs_node* is identified by a numeric id. As *fs_nodes* can be mounted separately and at different times and no centralized meta data about the partition is kept, an *fs_node* cannot know for sure how many *fs_nodes* the file system has. This is due to the fact that an *fs_node* might have been added while the current *fs_nodes* have not been mounted. Consequently there is no way of knowing which one is the following *fs_node* id.

The solution that we propose is based on the *divide and conquer* method. Each *fs_node* has a unique id stored on 32 bits. We allocate to each *fs_node* an *empty fs_node space*, which composed out of the list of id's that an *fs_node* might give to a new *fs_node*. This is stored on each *fs_node* as a bitmap. When a new *fs_node* is mounted, the Partition Manager will merge its bitmap with the rest of the bitmaps obtaining the allocation space of the current mounted set.

Each time a new *fs_node* is created, it receives a proportional amount of the free empty id's of the set of *fs_nodes* which where mounted at the time of its creation.

Using this algorithm, each mounted set of *fs_nodes* will know how many free *fs_nodes* it can create. This is also a method of knowing approximately how much of the file system is mounted.

Another problem that we have encountered is duplication of folders and files. Since each *fs_node* may be mounted separately or in subsets, it may be possible that a folder or file having the same name and path to be created on more than one *fs_node*. In case of folders, the solution is very simple; the content of the folders will be merged. The problem still remains for the files. We have applied here the solution of symbolic links.

If a file with the same name exists in the same folder on different *fs_nodes*, two virtual folders will be created, having a special name, representing each *fs_node* on which the file is located. The original files will be located in the virtual folders. The file displayed in place of the conflicting file is a symbolic link towards one of the two files (chosen by one of the data distribution priority algorithm, described in the section above) in the virtual folder. The symbolic link has been chosen instead of a hard link, so that software designed for *NetFS* would be able to recognize the conflict.

2.4.4 File System Management

Standard file systems provide special tools for management (creating, formatting, deleting and querying for information). This is usually done by accessing directly the device files associated with the physical media. This is rather hard to do with *NetFS*, as it's *fs_nodes* stored over multiple media types.

Our approach for this problem is using special file stored on the *NetFS* partition. The reading and writing into these special files using certain parameters performs actions over the partition. The main advantage of this solution is that *fs_nodes* can be managed dynamically, without the necessity of being unmounted (on the contrary, they have to be mounted). In this way we can add, modify or delete *fs_nodes* while the system is working. This is a very important feature if the file system is used in production environments, where downtime is not acceptable.

2.4.5 Copy-on-write Function

A very important function of the partition manager is *copy-on-write*. This concept is borrowed from the virtual memory system. This allows very fast file copying inside the partition. This means that copying a file will actually result in creating a copy of the original file's meta data and marking all the file data blocks as *copy-on-write*. Each time one of the two files needs to be modified (the original or one of the copies), the system will check if the data block is marked. If so, it will create a copy of it and them modify the copy.

The blocks bitmap of each *fs_node* stores two bits for each block, instead of one. The number stored is the number of files that point to that respective *inode*. If this number is greater than 1, the block is considered marked as *copy-on-write*. Each time a write request is issued for this kind of block, the system creates a copy of it and decrements the number in the bitmap. The *copy-on-write* functions has though a limit of maximum three files copies. When this limit is reached, the next copy request will perform an actual copy of the block.

This function also applies to entire *fs_nodes*. If an *fs_node* is read only and marked as *copy-on-write*, any write access to the file will be done by creating a *copy-on-write copy* of the file on another *fs_node* (if possible). Once the new file is created, due to the fact

that *copy-on-write fs_nodes* have smaller priority, further accesses will be considered accesses to a duplicate file (described in the previous section) and, based upon access algorithms, the newly created file will be used instead of the read only one.

2.5 Read/Write Modules

The last layer of the file system is represented by the Read/Write Modules. This layer is actually composed out of several modules which perform reading and writing of bytes from and to different media. This includes from simple RAM reading and writing to complicated socket engines which retrieve and store data located on other computers.

Each mounted *fs_node* has an associated read/write module.

As stated before, on of the goals of our file system is to be easily extensible. Read/Write modules are implemented as dynamically shared linked libraries. They behave like *plug-ins*. This allows us the extend the set of usable storage devices *on-the-fly*, without even recompiling the file system's source code. In this way, anyone may extend the file system's storage capabilities without having any knowledge at all about the file system's structure.

The modules have no knowledge of what kind of data they read or write. They communicate with the Partition Manager through a very simple interface composed out of the functions:

- *read* - reads data from the storage media;
- *copy* - copies data from a location to another;
- *write* - writes data to the storage media;
- *flush* - assures that data are completely written to the storage media;
- *mount* - mounts a new storage media;
- *unmount* - unmounts a storage media.

The **Socket Engine** is used for communication between network *fs_nodes*. It uses TCP connections that are established on-demand when the need to talk to a remote node appears. The TCP connections are only terminated upon shutdown. The data received from the File Manager or from a remote Socket Engine are stored in circular buffers.

Data for writing is tagged with numbers. Upon issuing a write command with a certain tag, all other commands for the same tag will be delayed until a response is received from the same tag.

This module also implements a rudimentary cache: a circular buffer is kept with write commands and their data. If a read command arrives that can be satisfied by only looking in this write cache, no other request will be made, and the reply will be copied (or assembled from the data in multiple writes) and the read will be finalized.

Upon detecting a lost connection to a remote node, the Socket Engine will keep retrying the requests that are already in its buffers, for a certain time. During the retries, further requests received with a tag on the node that seems to be down will be instantly answered with an error. If the connection is re-established before the time is up, pending operations will be satisfied. If the connection is not established, pending operations will be discarded, and they will return errors to the issuing module.

3 Evaluation

NetFS is still under development, so we have performed only a few performance tests until now. We have used the file system on a Debian distribution and on OpenSolaris 2008.11. The tests consisted on copying files to *NetFS* having the storage in the memory and to *RAMFS*. Solaris tests results are inconclusive as the Solaris port of FUSE really misbehaves on high load. As Sun Microsystems states, it is still under development. Only Linux tests will be described further on.

The first test that we have performed was copying an audio file (3 MB) and a video file (40 MB) into the both file systems. As expected, *RAMFS* performed better.

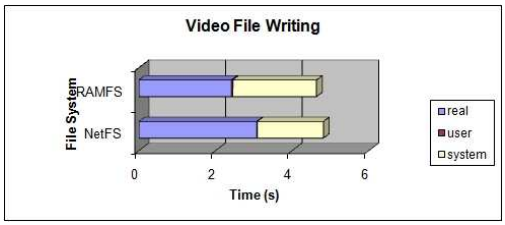

Fig. 2: **Video File Writing**

Further on, we have performed tests by increasing the number of files that where copied. As depicted in figure 3, the higher the file number, the better the performance of *NetFS* over *RAMS*.

The measurements have been performed several times using the *time* command. The copying was performed using the *cp* command. The tests results are an average of several successive tests. It must be said that tests where performed on initial versions of *NetFS* and performance parameters might change due to changes in the implementation.

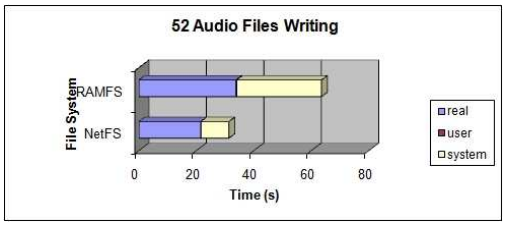

Fig. 3: **52 Audio Files Writing**

4 Future Work

The research for *NetFS* is only at the beginning. We are already taking into account the improvement of the file system by creating much **more efficient Read/Write network storage modules**, which would be able to perform optimal caching functions.

Another important feature that we are looking into is **redundant data storage** using the Read/Write modules. In order to protect data integrity, we are planning to create persistent storage Read/Write modules which would be able to store data redundantly, thus simulating full RAID behavior.

For the moment, an *fs_node* can be mounted only into one file system at a time. As further work, we considered a peer-to-peer approach [1]. In the next version of *NetFS* a partition could be **mounted partially on several computers at the same time**. The new system will have to offer location transparency, as all participating peers must see the same directory structure. Because we will use a peer-to-peer storage infrastructure, some policies must be defined which will specified how the storage space will be managed between peers.

Another very important improvement that we are looking into is a **better handling of concurrent system calls**. For now, calls that modify the *fs_node*'s *inodes* or data blocks bitmaps are serialized, which leads to performance loss on heavy load.

Background data migration is another feature that we would like to implement. If the *Closest space* distribution algorithm is used, data are written to the *closest* space available. In time, space may become available closer. In this case, if the system is idle, the file system should move data blocks *closer*. This function could be thought of as a defragmentation extension.

5 Conclusion

We have presented in this paper a distributed, hybrid-storage partially mountable file system. We believe that this approach is very important for the development of data storage. Computers need to take advantage as much as possible of all storage devices, regardless of their type making it transparent to the user.

Partial mounting is another advantage of our file system, as parts of the file system can be dynamically mounted, unmounted, created or deleted. There is no downtime involved, making it perfect for usage in a production environment.

Due to its modular implementation, upgrades and extension to the file system can be create very easily. As Read/Write modules are implemented as dynamic shared linked libraries, the storage capabilities of the file system can be extended without having to stop the file system.

References:

[1] A.R. Butt, T.A. Johnson, Y. Zheng and Y.C. Hu, Kosha: A Peer-to-Peer Enhancement for the Network File System, http://people.cs.vt.edu/ butta/docs/sc04.pdf.

[2] N.K. Edel, D. Tuteja, E.I. Miller and S.A. Brandt, MRAMFS: A compressing file system for non-volatile RAM, http://ssrc.cse.ucsc.edu/Papers/edel-mascots04.pdf.

[3] Filesystem in Userspace, http://fuse.sourceforge.net/.

[4] P. Gupta, H. Krishnan, C.P. Wright, M.N. Zubair, J. Dave, and E. Zadok, Versatility and Unix Semantics in a Fan-Out Unification File System, http://citeseer.ist.psu.edu/old/702953.html.

[5] F. Isaila, G. Malpohl, V. Olaru, G. Szeder and W. Tichy, Integrating Collective I/O and Cooperative Caching into the ClusterFile Parallel File System, *Proceedings of the 18th annual international conference on Supercomputing*, Malo, France, 2004, pp. 58-67.

[6] The OceanStore Project, UC Berkeley, http://oceanstore.cs.berkeley.edu/.

[7] J.-S. Pendry and M.K. McKusick, Union mounts in 4.4BSD-lite, *Proceedings of the USENIX 1995 Technical Conference* New Orleans, Louisiana, 1995, p. 3.

[8] A.S. Tanenbaum and A.S. Woodhull, *Operating Systems Design and Implementation*, 3/E, Prentice Hall, 2006