A Perfectly Normal Namespace for the DESY Open Storage Manager

Patrick Fuhrmann,

DESY Hamburg, Notkestr. 85, 22607 Hamburg

This small write up introduces the implementation of a pretty weird network file system service. It was designed to fill the gap between a raw Mass Storage System and the needs and habits of several hundreds of DESY users. On top it appears to be a *Perfectly Normal File System* but offers a large number of hidden features. Each file-like entry is identified be a systemwide unique identifier which is guaranteed to stay with that entry for its lifetime. Moreover it has a name and eight layers of Data - Metadata pairs. The top layer may simply consist of meta data representing a data portion on a removable medium. In addition directories may contain tags which follow the concept of inheritance. Each directory can be declared to be a mount point with a specific profile. Methods exist to access special information or configuration directories from everywhere in the file system without knowing where they originally reside.

Key words: mass storage; network file system ; name space ; client server architecture ;

1 Background

In the beginning of '97 the DESY Central Computer Centre had to face the management of data in the order of 30 TBytes in several forms of disk and tape media. It is not unlikely that this amount will increase by 15 TBytes/year for the next years. To organise this portion DESY decided to use a piece of software known as the *Open Storage Manager[OSM]*. During the last years quite a lot of modifications had to be done on that system to make it suitable for the specific needs at DESY. One important step towards the use of OSM by non experts was the introduction of PNFS [1] a network file system which offered the well known file system view to the underlying OSM repository. Although

[1] A Perfectly Normal File System

we are running *PNFS* together with *OSM* there exists no direct interconnection. PNFS doesn't rely on any OSM supplied features. The interconnectivity consists of tools, using both, the PNFS and the OSM. The methods used by Mass Storage Systems to store and retrieve data to and from their repositories are fairly similar. The total media space is divided into partitions which we will call *Storage Groups*. On one hand those Storage Groups define a set of media units together with the corresponding robot system. On the other hand they consist of a selection of *Bitfiles* which build the smallest data unit. Bitfiles are identified by a system determined and rather arbitrary ASCII string. Those strings build a flat name space and they are the only handle to the corresponding stored data portion. Being used to navigate through hierarchical file systems, a flat name space containing arbitrary and system determined filenames doesn't appear to be state of the art and is by no means acceptable for a large community of users. Consequently we decided to design a network capable and Unix like Namespace, which hides things like Storage Groups and Bitfiles from the end user.

2 The Objectives

Beside the requirement listed above, a lot more issues came up during the design phase :

- One of the main objectives must be to provide a network wide available hierarchical name space. The mapping between this name scheme and the Mass Storage name scheme should be totally hidden from the user.
- All file meta-data, like file size, owner and permissions should be always available in the well known file system like manner.
- In addition it would be nice to be able to stick some kind of additional user data to the file name entry. This data must also be always accessible without retrieving the whole file.
- To avoid specifying the Storage Group when storing data, the directories must be able to hold some kind of tag which determines the Storage Group they are responsible for.
- Consequently this directory connected data has to be inherited by subdirectories to allow the user to create directories without any special administration tools.
- Concerning the file system tree itself :
  - The whole file system should originate from one single root to simplify administration tasks.
  - On the other hand group dependent mount points must exist with different access privileges to separate user groups from one each other.
  - Although the different user groups should only view their own part of the pie, there must be a method to access system wide valid tools, config-

urations and status informations without duplicating this data for each mount point.
- There shouldn't be to much specific requirements to the client host to participate this distributed file system.
- The file system shouldn't at any point rely on the fact that it is used in conjunction with OSM.

3 Tieing things together

There is no doubt that most of the requirements above point to a solution pretty near to NFS [2] Version 2. Nearly all operating systems include the NFS2 client capability. The protocol itself is well known and in the meantime pretty much shock proofed. Providing an NFS server gives us full control of whatever we want the client to think is behind our file system. The decision to use NFS2 to distribute our name space implies that we have to face the following drawbacks :

- NFS uses a restricted and fixed number of function calls to map the client file system operations to NFS operations. Adding new functionalities is only possible if they can be mapped to file system operations supported by the NFS protocol.
- The statelessness of the NFS protocol, which makes an NFS server easy to implement, can become really painful if one needs to rely on a specific sequence of NFS requests.
- Another discouraging point is the weak I/O performance. Even on fast networks the data rates are unacceptable.

To bypass the performance issue, we can divide the NFS operations into name space and I/O commands. All name space operations which result from commands like *cd* , *mv* , *rm* , *pwd* ... have to be answered in the assumed way, so that the client sees a perfectly normal file system. This must be different for the I/O operations *read and write*. We need to provide a tool which is tuned to achieve the maximum performance for the real I/O operations. See figure 4 how this is done. To remind the user of this behaviour the file system replies with an I/O error if an NFS I/O command is requested.

Adding additional functionality can be achieved by extracting a small portion of the total name space and using this part for non regular file system operations. As long as the filename syntax of this part is exotic enough, it won't do any harm to the regular user.

[2] Network File System

4 What it finally does

Our actual *PNFS Server* implementation supports all NFS2 commands. For a given set of directories the *read and write* functions can be disabled to prohibit the storage or retrieval of large data files via the NFS path.

There is a specific I/O program similar to the *OS cp* command which allows to perform the actual I/O operations.

A well defined part of the name space has been reserved for specific non standard operations. File system entries holding those names never appear in the *readdir or ls* listing.

We introduced data objects called *tags* which always belong to a directory object. Once created, they can't be moved or removed. They can be seen as an integral part of the directory itself. All subsequently created subdirectories inherit a pointer to those tags. Changing the content of a master tag will change the content of all inherited tags in the directory tree below the master.

The file object itself consists of the *Filename* and *8 layers* with different behaviours (see Fig. 3 on the right). The top layer, which is the visible part of the structure simply contains meta data, like file size, file mode, owner and group. This meta data actually belongs to the stored tape file. This top layer doesn't contain any regular data. I/O operations performed on that layer will always end up in an I/O error. The other 7 layers consist of the meta data and optional data up to 4K. All 8 levels of meta data can be set independently. The *Open Storage Manager* and the planed *Disk Layer[SUSE]* make use of two of those data forks. A third level may be used as *user data* repository. Although each of the layers may have its own uid, gid and permission mask, they all are connected to one filename and can't be separated. The create, move and remove operation will work on all 8 layer together. The construct owns a unique identifier which stays with it for its lifetime. This 128 bit identifier builds a bidirectional link between the actual data in the Mass Storage repository and the file object in PNFS.

The move and remove operations are filtered by an event handler before the requested action is performed. For the move operation, this filter checks whether the file or directory is moved inside the context of a *Storage Group*. Otherwise the filter denies the move request. The remove filter copies the contents of all available layers of the file into a predefined place. This enables the Mass Storage System to remove the tape file asynchronously. Currently those event handler routines are still written in C-language and they have to be linked together with the dbServer code. We plan to implement those routines in java. This enables a dynamic loading of action filters depending on the directory the action is performed on.

Each directory can be configured to be a mount point using an individual mount profile.

Each directory contains an invisible entry which points to a configuration directory. It's designed to contain system wide informations but it need not be part of the mounted file system. Figure 1 is a possible PNFS file system layout. The surrounded directories *h1, hermes and zeus* are mount points. The shaded area can be seen under the mount point *h1*. The directory tree below *osm* is visible from each point inside PNFS using the link .(config)(osm). This special directory is meant to contain setup and configuration informations. File object residing beyond that directory tree behave like regular files and don't produce the I/O error code when being read or written.

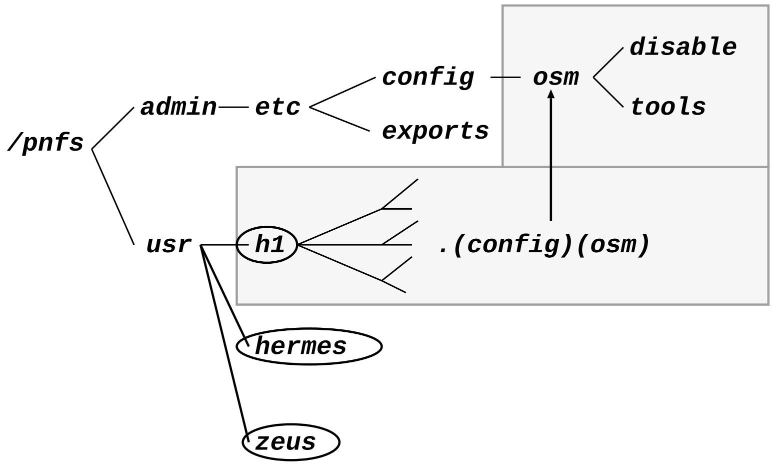

Fig. 1. A possible PNFS file system layout. The surrounded directories are mount points and the shades area is visible under the mount point *h1*.

5 How it is done

Figure 2 gives an impression how the file system is built. As usual, there is the mount daemon and a set of NFS daemons listening for NFS requests. Those daemons determine the database server responsible for that particular request and forward the command. The process interconnection is done by *shared memory portions and semaphores*. The communication is designed to allow multiple PNFS servers to talk to different dbServers simultaneously without interference. The whole file system is simulated by the dbServer with the help of a database representing the file system structure. Or in other words, there is no host where our network file system is local. Even on the server machine the file system has to be mounted to be accessible. The file system simulation can be divided into three layers (see Fig. 3 on the left). The bottom layer is just a key-value database with fast key search capability. The next layer constructs the different file system objects like files, directories, links a.s.o..

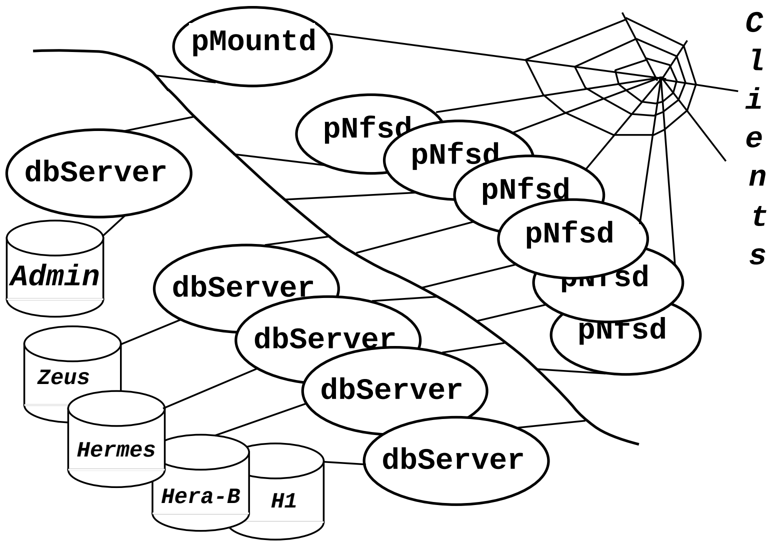

Fig. 2. Involved PNFS Processes. The pMountd and the pNfsd's are connected to the dbServer via shared memory. The client hosts are connected over the network using NFS Version 2.

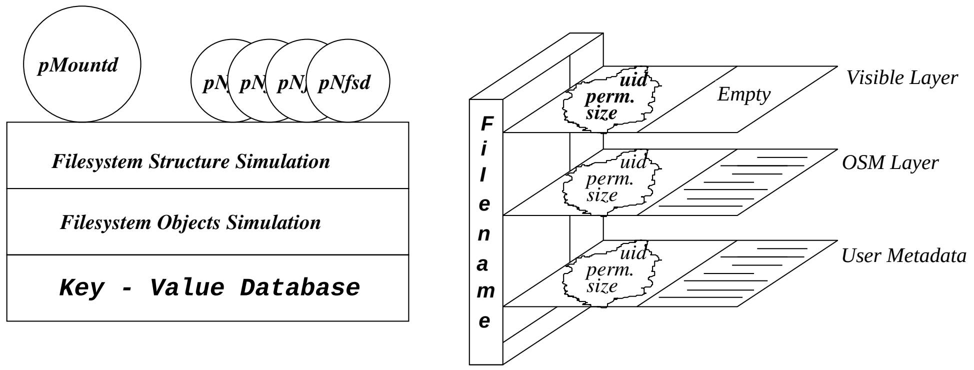

Fig. 3. Left : Software Layout of the PNFS file system simulation. Right : View of a file object in PNFS being used by OSM. Only 3 of the possible 8 layers are shown.

The upper layer links those objects together to the actual file system with the corresponding methods like create, move, remove a.s.o.. On top, the mount and the NFS daemons make use of those methods to manipulate the simulated file system. We are thinking of replacing the two base layers by an object oriented persistent database like *Objectivity[DBASES]* to gain performance and being independent of the actual size of the database. Figure 4 gives a brief view of how PNFS works together with the Mass Storage System and the I/O client. There is no direct connection between PNFS and the Mass Storage System. The client takes all information it needs to store or retrieve the data to/from a removable medium out of PNFS and uses this information in control requests to the Mass Storage Server Processes. After all resources could be allocated the client is connected by the *Mover Host* and the data is transferred in either direction using TCP/IP and the fastest available network hardware between the Mover and the Client host.

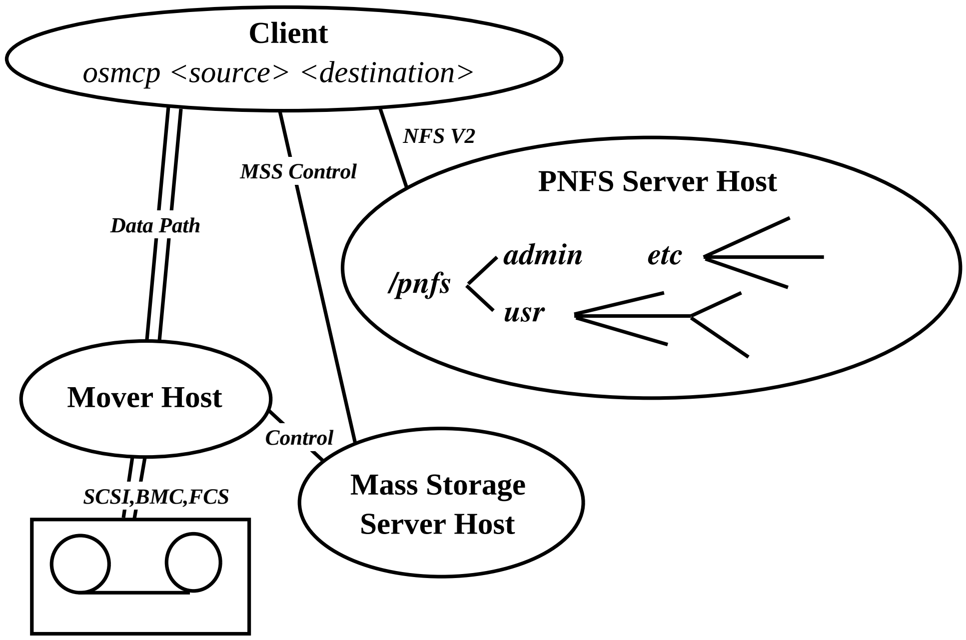

Fig. 4. The process interconnections between PNFS, the mass storage system and the I/O client.

An alpha release of the described system is in production at DESY Zeuthen since April 1996 and the full server at DESY Hamburg since November 1996. Mid of February 1997 we counted about a quatre of a million entries in our name space databases and the server was mounted by more than 50 clients simultaneously. Last but not least I want to thank the people from DESY Zeuthen for their patience concerning all those tiny little bugs they found in the pre-releases.

References

[OSM] Lachman Technology, Inc. : The Open Storage Manager.

[SUSE] Susanne Brand : A Distributed Disk Layer for Mass Storage at DESY. Chep97

[IEEE] IEEE P1244. Reference Model for Open Storage Systems Interconnection - Mass Storage System Reference Model Version 5, September 1994.

[DBASES] Objectivity : Objectivity/DB Version 3.

[NFS] Santifaller : TCP/IP und NFS in Theorie und Praxis. Addison-Wesley