# nfsp: A Distributed NFS Server for Clusters of Workstations *

Pierre Lombard, Yves Denneulin
Laboratoire Informatique et Distribution - IMAG
ENSIMAG - Antenne de Montbonnot - ZIRST
51 avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France
{Pierre.Lombard,Yves.Denneulin}@imag.fr

## Abstract

*A consequence of the increasing popularity of Beowulf clusters has been their increasing size (in number of nodes). Yet, the hard drives available on these nodes are only used for the system and temporary files, thus wasting a lot of space (several TiB on large clusters !). The systems that might help recycling this otherwise-unused space are few and far between. This paper presents a NFS server that aims at using the unused disk space spread over the cluster nodes and at offering performance and scalability improvements (compared to the plain NFS servers). The architecture of our solution uses a metaserver and I/O daemons. The client only sees plain NFS, thanks to NFS over UDP spoofing techniques. A first implementation and early performances are shown for this approach.*

## 1 Introduction

As part of a project to provide a complete infrastructure for computing on top of corporate intranets, HP provided the *i-cluster*, a cluster built of 225 iVectra (desktop PC's made of P3-733, 256MiB, 100Mb ethernet cards and 15GB IDE hard drive). This kind of cluster is categorized as a Beowulf cluster [18] and has proved its efficiency in dealing with several kinds of tasks (for instance, see [17]). After several months of use, we noticed that a lot of disk space is left unused on the cluster nodes: the base system requires around 4GiB... which leaves 11GiB unused, which sums to more than 2TiB unused for the i-cluster... We have started looking for a distributed file system to prevent this loss. As the context is a cluster used on a daily basis by several users, the following requirements have to be considered: enables the use of the disk space of several nodes of a cluster, offers a common naming space (one network volume), has performance good enough to saturate the bandwidth of the network, needs minimal modifications on the client side (no new protocol) and no extra administration cost.

The results of a survey on several existing systems (see section 2) did not satisfy our needs. Existing systems often have too many features and do not take into account the peculiarities of a typical cluster: highly available nodes, local network, secure environment, disks on most nodes, "very" standard hardware... As we wanted to use a subset of nodes as a distributed file server but without using a new protocol because it often requires kernel alterations and/or an heavy installation process, we decided to develop a simple drop-in replacement for the classic NFS server, but with a distributed storage back-end to balance the I/O's load among nodes. Therefore, the plain NFS server has been split into two distributed entities: one that stores the file system structure, the metadata and the other one, the content of the file, the "real" data. By using this approach, we hope to make an interesting combination of the flexibility and wide availability of NFS and expect to have good performances.

The rest of this paper is organized as follows: in section 2, we present a quick survey on existing distributed file systems. Then, section 3 presents an overview of our system. The following part (section 4) shows some technical issues we had to deal with to implement our prototype and section 5 gives some early results we could obtain with this implementation. Eventually, section 6 concludes and describes several directions for future works.

## 2 Related works

There has been a lot of work done on distributed/network file servers [11] since the early eighties with various issues addressed: security, performances through caches, high availability, disconnected mode, consistency, ...

One of the most known set is by the AFS family (AFS and its direct heir OpenAFS [5], CODA [2], Intermezzo [3]) which address several of this issues in an efficient way, ac-

*This work is done in the context of the joint research project APACHE supported by CNRS, INPG, INRIA, and UJF. Computer resources are provided by the ID/HP i-cluster (further information on `http://icluster.imag.fr`).

cording to the flavor used. The main drawback is the intrusive installation and all the requirements needed (but it has a lot of not necessary features in the case of a Beowulf cluster).

There has been several projects related to storage in Network of Workstations in the past - now over/stopped: the Swarm Scalable Storage System [14, 15], University of Berkeley's xFS [19] , ...

Not all of those are available for Linux, which is quite problematic in the case of a Beowulf cluster.

One recent and advanced project of cluster file system is PVFS [12, 6]: it uses a simple yet interesting approach based a central metaserver and I/O daemons. It requires a kernel module and a local `pvfsd` daemon which really handles the network protocols (over TCP streams). Though all daemons run in user-space, it is a bit too intrusive as it requires a new kernel module and some setup on the clients.

Last but not least, NFS is perhaps the most standard and available file system so far despite its known drawbacks (loose coherency, scalability) but it is simple to install and administer.

To summarize, none of the systems presented has the two most important characteristics for our cluster: easiness of use and administration (minimal modifications of the system). They provide too many unneeded things in the context of a cluster, like an elaborate security protocol or require some reconfiguration of the cluster nodes (new kernel modules).

This has been our motivation for developing a NFS parallelized server (simply acronomized into `nfsp` ) behaving like PVFS.

## 3 System overview

The design of this system is to split the standard NFS server in a subset of dedicated smaller servers: a metadata server that holds the attributes (timestamps, permissions,...) whereas the content of the files are physically stored on other machines, on which runs an I/O daemon.

A traditional NFS system is built on a client-server architecture: clients send requests to the server which sends them back replies. The former does not know what is the server (disks, filesystem used for storage, ...), they just see a storage area. Yet, centralized architectures are known to not scale very well should no expensive and dedicated hardware be used: this applies to NFS servers, too.

Thus, one way to design a "cheap-yet-efficient" NFS server (cf. with Beowulf and its "poor guy's supercomputer") might be to use a distributed back-end (i.e. not seen by the user) to store the data and to keep as a front-end a plain NFS server.

The approach chosen has been to keep a central server using the plain NFS protocol with the clients but that only serves metadata. The requests requiring heavier I/O are forwarded to I/O daemons (`iods` ) running on other nodes and whose role is to handle the storage of file stripes. These `iods` should logically send the reply to the client but they have to impersonate the server (as the clients are not aware of their existence), which is achieved by UDP spoofing [13].

Hence, the transmission of the "real" data will occur between `iods` and clients, not between the server and its clients, not wasting the bandwidth of the server. This is especially interesting for read operations as explained in section 5.4.

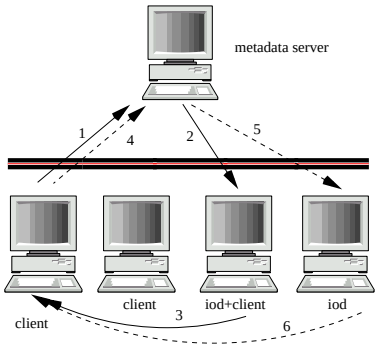


**Figure 1. Architecture of a NFSP system**

The architecture of our proposal is illustrated on figure 1 (numbers 1 to 3 represent a first request, 4 to 6 a second one). When a client does a read/write operation it sends a NFS request to the server (number 1 or 4). Then the server forwards it it to the node that has the data or that will store it (number 2 or 5). The node will reply directly to the client by *spoofing* the server reply (number 3 or 6) by spoofing the UDP packet.

## 4 Technical Issues

This section presents the technical aspects of our server and shows what has to be modified in an existing NFS implementation (Linux user-mode server) to handle distributed storage.

### 4.1 The NFSv2 protocol

The NFS protocol (version 2) is defined in a IETF's RFC [9]. It relies on SunRPC [8] which use XDR [7]. Though these RPC's support TCP or UDP sockets, the use of UDP appears as a natural choice in the case of NFS as this protocol was designed to be stateless. Besides, the RPC mechanism adds a fiability layer to the NFS protocol by use of acknowledgements and request resends, so TCP may not offer huge improvements but for flow control which does not occur that frequently in modern switched local networks.

Once the client acquired a handle on the NFS volume (by the `mount` protocol), the server has to handle the requests of the client and process the I/O's. These requests are NFS over UDP packets exchanged between $\text{IP}_{client}$:$\text{port}_{client}$ (usually a system port below 1024) and $\text{IP}_{server}$:$\text{port}_{server}$ (usually 2049). This kind of "pseudo connection" is what identifies the NFS mount.

## 4.2 Architecture overview

First, we would like to define a few words to avoid ambiguities. A "real file" (or simply "file") is the file as the client sees it or as it would be in a local file system. A "metafile" is a file in the local file system of the metaserver; this metafile holds metadata (name, owner, group, permissions, timestamps, striping parameters, etc...) for a real file. A "datafile" is a file on the local file system of an `iod` ; it holds a part of the data of a real file called a stripe.

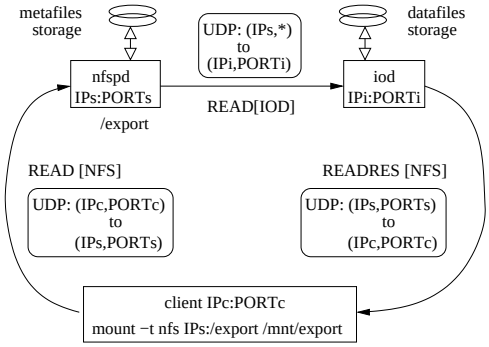


**Figure 2. NFSP server**

The nfsp system uses two kinds of server:one deals with metadata (filenames, permissions, ...) and several I/O daemons handle the data. To simplify the notations the former will be called metaserver (or `nfspd` ) and the latter `iods` .

The metaserver currently keeps metafiles in a standard Unix directory (on disk). This directory will be exported by means of the standard NFS mechanisms (`mount`). The `nfspd` stores metadata at two levels in the metafiles:

- in the attributes of the metafiles: this is handled by the underlying file system on which lives the metafile. This concerns attributes not altered by striping: creation time, owner, group, permissions, etc...
- in the metafile itself: attributes such as the real file size, the "seed" of the file, striping parameters...

## 4.3 Differences with a standard NFS server

This part presents how we implemented the most common operations in the prototype we developped above the the Linux User-Space NFS server. This server implements NFSv2 and is running in user space, which does not require any modifications of the kernel and provides a quick installation. It is a bit slower than the kernel mode server since some kernel-to-user overheads are added.

### 4.3.1 Creation

When the clients creates a regular file, the metaserver creates a metafile with the same name. Then it writes into this metafile a few information such as: a magic number (to ensure it was created properly), a seed (or cookie to give the file a pseudo-version), the size of the real file. Informations to write may be extended to handle extended attributes (striping, storage nodes, ...). If a cookie mechanism was not used, the `iods` might serve stale data belonging to the previous file with the same inode number. Indeed, `iods` only reference the data they should have stored with the metafile inode number and this pseudo version.

As a consequence of this implementation choice, when the user hardlinks two real file, two metafiles are hardlinked on the metaserver. Another one happens when a client creates a special file, the metaserver also creates a special file. Last example: when you compare the files list on both a client and the metaserver, they are identical (same names, file hierarchy) but the sizes differ: the files stored on the metaserver all have the same size (about twenty bytes) whereas the clients see them with their real size.

### 4.3.2 Deletion

On a classical NFS server, a deletion request is sent by the client to the server: it removes the file and acknowledges the NFS request. This deletion process, though looking simple is trickier to handle in a distributed environment. When a client requests a file deletion to the metaserver, the metafile is `stat()`'ed and its hardlink count checked to see if the disk space should be freed. If this metafile is the last hard link left, then the `nfspd` informs an ancillary process (`unlinkd`) that the file has to be erased. Then `nfspd` removes the metafile and acknowledges the NFS request.

Meanwhile, `unlinkd` merges deletion requests till a maximum number is reached and/or a timeout has expired. Then, it contacts each `iod` through a TCP stream and requests the datafiles of the removed file to be erased.

As `nfspd` communicates with `unlinkd` by means of a standard pipe and blocking `read`/`write`, the system regulates itself, should lot of deletions be requested.

This mechanism is a bit *heavy* but otherwise there would be disk space leaks if all the datafiles were not properly removed.

#### 4.3.3 Read

On a NFS server, a READ request is sent by the client to the server which sends the data requested together with NFS attributes. In nfsp, the client sends the READ request to the metaserver. It reads metadata, finds the iods that will hold data and then forwards the request to it. The iod processes the request it received and spoofs the reply the metaserver should have sent to the client. The spoofing with ports and hosts is illustrated in figure 2.

#### 4.3.4 Write

In a standard NFS server, a write request has to acknowledged by a short packet containing the RPC acknowledgment and the attributes of the file (metadata).

In nfsp, requests holding data to write are sent to the metaserver. It reads metadata and forwards it, together with the data to write, to the iod chosen by the hash function. Then, the iod processes the request and spoofs the acknowledgement the metaserver should have sent to the client. The spoofing works in the same way than for a READ request as illustrated in figure 2.

#### 4.3.5 Other requests

The `nfspd` handles every other NFS request without interacting with the `iods` . Some care has been taken though to handle the metadata and send a correct reply to the client: for instance, some wrappers for the `stat()` family of functions are used.

### 4.4 I/O daemons (or `iods` )

I/O daemons are currently multithreaded servers. The reception part is designed as follow:

- a thread gets requests from UDP packets from the metaserver (READ/WRITE/PING) in one of the iods buffers then wakes up an I/O thread,
- a thread gets requests from the TCP stream (opened by the metaserver at lauch - indeed only PURGE requests, see why in section 4.3.2) in one of the buffers then wakes up an I/O thread,
- I/O thread(s) are sleeping until they are notified some work has to be done, whatever it may be.

The sending part uses a RAW socket: such a low-level socket is required to control the way packets are sent. A simplistic UDP stack on which we have complete control is then used to send messages.

The simple protocol used between the metaserver and the `iods` is simply called **iod protocol** and consists of a few messages:

READ is sent by the metaserver to the `iod` , contains inode number, offset, seed, the 4-uple ($IP_{client}$, $port_{client}$, $IP_{metaserver}$, $port_{metaserver}$), the Sun RPC `xid` (request identifier) and the file attributes. The `iod` will spoof the NFS reply to the client.

WRITE contains the same data than a READ request but also has to include the data to write. The `iod` will spoof the NFS acknowledgement to the client.

PURGE: is used to flush every datafile still existing on the `iods` and has (currently) to be sent to every `iod` .

Two other messages types are available for administrative and troubleshooting purposes:

- PING may be sent by any host to any `iod` ;
- PONG is the the acknowledgement sent back to the PING requester and is sent by means of our spoofing UDP stack (to test if an `iod` works correctly).

### 4.5 Installation

The installation on the master node (the one that holds the metaserver) is meant to be quite simple: edition of the `exports` file (usual format), edition of the list of *host:port* of the `iods` , create a storage directory for the datafiles, start the `iods` , and eventually launch `mountd` and `nfspd` on the metaserver node.

And *voilà*: the server is now up and ready to serve ; the client has now access to the exported NFS volume with an adequate "`mount -t ...`" line.

## 5 Preliminary results

This section presents the results of the early tests we have run on our prototype. (B stands for byte and b for bit.)

### 5.1 Description of the testbed

All the nodes (metaserver, `iods` ,clients) have the same hardware (see section 1) and run a plain Linux Mandrake release 7.1 with a compiled standard Linux kernel 2.4.4. The current dedicated NFS server of the i-cluster is a P3-1GHz 512MiB RAM and 1GiB swap powered by a 2.4.5-xfs-1.01 Linux kernel.

To lower the influence of the cache system offered by the kernel, the memory available may be set at bootime at a low value (thanks to a boot parameter) but such an approch is not easily applicable in our environment with the batch and reservation system used. That's why we chose the other non intrusive (and classic) approach to defeat the cache and VM issues by disabling the swap devices and using a file twice as big as the RAM size.

### 5.2 `bonnie++` tests

This parts present how `nfsp` behaves when running a `Bonnie++` 1.0.2 test[1]. This tool is a benchmark suite performing several simple tests on file system performance: database type access to a single file, creation, reading, deleting of many small files (like in a web cache for instance).

In the tables below, the size of the test file is 512MiB (twice the RAM size).

| | Write | | | Read | | Random |
|---|---|---|---|---|---|---|
| | /char MB/s | /block MB/s | rewrite MB/s | /char MB/s | /block MB/s | seeks /s |
| NFS | 9.07 | 10.70 | 4.99 | 8.21 | 9.68 | 211 |
| NFSP(4) | 7.51 | 7.25 | 4.38 | 7.89 | 8.55 | 623 |
| NFSP(8) | 7.26 | 8.03 | 4.54 | 8.96 | 8.97 | 679 |
| NFSP(16) | 8.47 | 7.91 | 5.30 | 9.04 | 10.07 | 627 |
| NFSP(32) | 8.49 | 8.28 | 5.09 | 9.03 | 9.63 | 662 |

NFSP(x) indicates x `iods` and there is 1 client doing the test.

Performances with a single client are a bit disappointing as they are worse than for a plain NFS server. An explanation comes from the latency increase since an additionnal message (`nfspd` to `iod` ) has to be sent, which implies a decrease of the speed with NFSv2 (since it only does synchronous calls). Another thing to keep in mind is the NFS server results may be tainted by the fact it is a server (more RAM, SCSI disks, etc...)

| | Write | | | Read | | Random |
|---|---|---|---|---|---|---|
| | /char MB/s | /block MB/s | rewrite MB/s | /char MB/s | /block MB/s | seeks /s |
| NFS 1c | 5.40 | 5.73 | 2.53 | 4.70 | 5.17 | 273 |
| NFS 2c | 6.27 | 5.75 | 3.36 | 4.67 | 4.56 | 204 |
| NFSP(32) 1c | 5.32 | 5.56 | 4.68 | 8.14 | 8.13 | 684 |
| NFSP(32) 2c | 5.74 | 5.81 | 4.08 | 8.30 | 10.46 | 698 |

NFSP(x) indicates x `iods` ; 1c indicates the 1st client, 2c the 2nd client.

Writing is bound by the ethernet 100 (around 11MB/s) and this upper limit appears when summing the writing speeds of both clients. Yet, unlike the NFS server, more bandwidth is available for reading since several network cards are used simultaneously which shows an increase of the total bandwidth usage of the system.

### 5.3 Concurrent sequential tests

A 1GiB (4 times the RAM size) file was created and stored on 16 `iods` with a simple `dd` command. This operation took around 110s, which gives an average speed of 9.3 MiB/s, which is not bad considered it is NFSv2 through a 100Mb/s pipe.

This tests consists in having a common 1GiB file read sequentially by a variable number of clients launched (almost) simultaneously. Several `dd` commands were launched on several hosts by means of `ka-run`, a parallel job launcher[2].

[1] `http://www.coker.com.au/bonnie++/`

[2] `http://ka-tools.sf.net/`

This tests may illustrate the behavior of clients being started and wishing to fetch a common big data file. The figure 3 illustrate the results obtained.

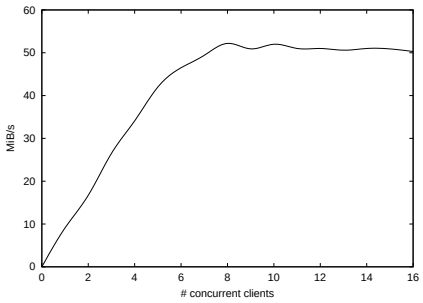


**Figure 3. Total bandwidth used (MB/s)**

Starting at 6 clients, the cumulated bandith served by the `nfspd` does not increase and becomes stagnant, which is a bit disappointing, and reveals the presence of a bottleneck.

We first thought that some buffer in the switches (HP Procurve 4000) used were being saturated but a quick look at their specifications showed a maximal backplane speed of 3.8 Gbps, which lets some margin... Then, we noticed 50MB/s corresponded roughly to 12,000 4KiB requests sent from the `iods` to the clients. Therefore 12,000 requests have been sent by the clients to the `nfspd` , analyzed and forwarded to one of the `iods` , which grossly corresponds to 2000KB/s input and 2000KB/s output and therefore does not saturate the `nfspd` bandwidth. Then, we suspected that the metafile handling was the problem as it required file I/O (at least a `stat()`) for each request received. Yet, the file system layer of Linux is pretty efficient and this I/O activity could only account for around 10% CPU. This part still needs profiling as we found later on that 100% of the CPU was eaten by the `nfspd` .

### 5.4 Discussion

The write process is bound by the network card capacity on the metaserver: the NFS protocol implies the client only knows one server and here it is the metaserver. Therefore, data packets are sent to the `nfspd` which will forward them to the `iods` in charge of their storage.

The read operation is more interesting and promising since the requests between the `nfspd` and the `iods` are small and do not saturate the metaserver bandwidth. Then, the `iods` handle the read operations and send back the result directly to the client, not wasting the metaserver bandwidth (on a switched network, of course).

## 6 Conclusion and future works

In this paper, we introduced a modification of a classical NFS server to adapt it to the context of clusters. The proposal is to split the NFS server in two smaller servers: a metadata server and data servers. The architecture and the implementation we carried out were presented and preliminary results were shown.

Several extensions may be thought of to improve the system:

- enhance the metaserver by using caching of metadata and not using the facilities offered by underlying filesystem for storage of metadata,
- test other high performances schemes as in [1],
- move into kernel space as other NFS servers to avoid unnecessary memory copies (but the simplicity of installation decreases),
- implement the NFSv3 protocol [10] (NFSv4 has a very different architecture) since it fixes some limitations of the NFSv2 protocol [16].

We aimed at ease of installation and ease of utilization use of the unused disk space. This prototype was first a proof-of-concept, but we were delightly surprised that early results were interesting and promising (though more tests have still to be carried out). Another consequence of having several `iods` on several cluster nodes is that there is globally more memory available for file caching and hence more performances.

Several points have been left apart but would require further investigation:

- storage redundancy: by modifying the metadata handling and what's stored in the metafiles (e.g. mirroring nodes),
- striping: the algorithm is naive and use constant-sized blocks, which might be improved with a smarter handling of metadata,
- dynamic extension: new `iods` joining or leaving dynamically is also desirable: plugging a new node increase the storage space transparently... (this behavior is studied in the peer-to-peer community)
- multicast: recent switches natively support it

The prototype has been tested on our cluster for a few months and no critical problem has been met so far (see [4] for further information).

## References


[1] The c10k problem (website). http://www.kegel.com/c10k.html.

[2] Coda file system (website). http://www.coda.cs.cmu.edu/.

[3] Intermezzo (website). http://www.inter-mezzo.org/.

[4] nfsp: sources + documentation (website). http://www-id.imag.fr/Laboratoire/Membres/Lombard_Pierre/nfsp/.

[5] Openafs (website). http://www.openafs.org/.

[6] The parallel virtual file system (website). http://parlweb.parl.clemson.edu/pvfs/.

[7] XDR : External data representation standard. RFC1014, June 1987.

[8] RPC: Remote procedure call protocol specification version 2. RFC1057, June 1988.

[9] NFS: Network file system specification. RFC1094, March 1989.

[10] NFS version 3 protocol specification. RFC1813, June 1995.

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

[12] P. H. Carns, W. B. L. III, R. B. Bross, and R. Thakur. PVFS: A parallel file system for linux clusters.

[13] CERT. CA-1996-21: TCP SYN flooding and IP spoofing attacks. http://www.cert.org/advisories/CA-1996-21.html, september 1996.

[14] J. H. Hartman, I. Murdock, and T. Spalink. The swarm scalable storage system. In IEEE, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*. IEEE, 1999.

[15] I. Murdock and J. H. Hartman. Swarm : A log-structured storage system for linux. In *Proceedings of FREENIX Track : 2000 USENIX Annual Technical Conference*, June 2000.

[16] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3, design and implementation. In *Proceedings of the USENIX Summer 1994 Conference*, pages 65–79, June 1994.

[17] B. Richard, P. Augerat, N. Maillard, S. Derr, S. Martin, and C. Robert. I-cluster: Reaching top500 performance using mainstream hardware. Technical Report HPL-2001-206, August 2001.

[18] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[19] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the 4th Workshop on Workstation Operating System*, 1993.