# Improving performances of a distributed NFSP implementation*

Pierre Lombard, Yves Denneulin, Olivier Valentin, and Adrien Lebre

Laboratoire Informatique et Distribution-IMAG
51 avenue J. Kuntzmann, 38 330 Montbonnot Saint-Martin, France
{plombard,denneuli,ovalenti,lebre}@imag.fr

**Abstract.** Our NFS implementation, NFSP aims at providing some transparent way to aggregate unused disk space by means of dividing a usually centralized NFS server into smaller entities: a meta-server and I/O servers. This paper illustrates what are the issues related to increasing the performances of such an implementation by using two different approaches: distributing the load across several servers or implementing the server in a more efficient and intrusive way. Performances of the different versions are given and compared to the first implementation.

## 1 Introduction

Today's low-cost clusters are often built by using off-the-shelf hardware: each node has its own storage capability, usually only used to store the operating system and the runtime environment. As the hard disk capacity increases, most of the disk space of the nodes remains unused but for temporary files since the users prefer having their files available on every nodes. Partial solutions imply investing in an expensive storage architecture (SAN or RAID servers), yet the disk space is still wasted on the disks of the nodes. Systems providing an aggregation of the unused disk space and the existing ones often implement new protocols or file system types, which may not be considered as a seamless integration for the clients.

Such issues try to be solved by the NFSP project. When the NFSP project was started in mid 2001[1], we chose to use standard and well defined protocols to implement a new kind of NFS server. The first prototype implemented was based on the Linux user-mode server. The first experimental results we got with this implementation highlighted the cost of running the daemon in user-mode. To improve this we tried two methods: balancing the load between several servers and making a more efficient implementation of the server itself. This paper presents these two approaches and compare them from a performance point of view. After this introduction, some related works in the distributed file systems field are shown in 2. Then the NFSP principles are explained in section 3 and the two

* This work is a part of the research project named "APACHE" which is supported by CNRS, INPG, INRIA and UJF. Some resources were provided by the ID/HP *i-cluster* (More information is available at http://icluster.imag.fr/)

methods for improving performances are detailed in sections 4 and 5 which contain performances evaluation. Eventually, some future tracks of research will be tackled in section 6.

## 2 Related works

A large amount of work has been carried out in the network file system since the 1980s. Among the first ones, still used nowadays are Sun NFS and Carneggie Mellon's AFS. NFS is aimed at sharing files among nodes in the same LAN whereas AFS is more suited for WAN architecture. A NFS [2, 3] server is made of a node exporting its local file system to the clients who access it through a remote mounting operation. NFS is a stateless protocol, no state is kept on the server side so every operation is self sufficient. This gives NFS some protection against temporary faults. However since the access point is unique for all clients the implementation is inherently centralized and so the storage space is limited to the one on the server. This is not the case for AFS which is a fully distributed file system: servers across different sites cooperate to share the same space and offer all the data they contain to their clients which use as a mounting point a server node part of the global architecture. Contrary to NFS, AFS is a stateful system and so coherency is different from the one found in NFS: when a node opens a file a memory of this operation is kept on the server so when another node access the same file for a write operation a cache invalidation message is sent to all the nodes who opened it. However, this strong coherency implies in high cost in terms of network latency, and thus requires a fast network.

In both cases, the goal of these systems is to provide shared storage for users, which is usually different from the needs of current cluster workloads. Indeed, the development of scientific applications has incurred in new constraints (huge amount of data, level of coherency, fine-grained sharing) on the previous file systems, which led to the design of new storage systems.

A first group of solutions, in order to meet the above needs, might be seen as hardware-based. File systems such as Sistina's GFS[4] and IBM's GPFS[5] are thought for specialized SAN architectures. Both systems have their data and metadata distributed across the SAN and offer advanced locking and sharing facilities of files. However, the performances of such a system is intimately related to the performances of the storage system underneath. For instance, the GFS handling of coherency relies on an extended SCSI instruction sets. As for GPFS, providing things such as fine-grained coherency by means of software requires a fast and low-latency network like those of the SAN's. Another quite promising new system, LUSTRE, being developed since 2000[6, 7] by ClusterFS Inc. aims at satisfying huge storage and transfers requirements as well as offering a Posix semantics. To achieve these goals, clients, meta-data servers (MDS) and object storage targets(OST)[1] are connected by means of a fast network.

Unlike GFS and GPFS being based on very specific hardware, Berkeley's xFS[8], as well as LUSTRE, only requires a fast network in order to implement

[1] Some kind of specialized smart storage.

its cooperative multiprocessor cache. This serverless design results from LFS[9] and Zebra[10] file systems. It is built as a totally distributed system where data and meta-data are spread (and may migrate) among the available trusted machines. A different approach is Frangipani[11]/Petal[12] which aims at providing a distributed shared file system (similarly to GFS). The lower-level layer, Petal, implements a logical disk distributed over physical disks. The Frangipani part builds a file system on top of it.

All those systems each offer interesting performances heavily depending on the underlying hardware which doesn't make them well-suited for Beowulf clusters built with common hardware. So another way was developed using purely software solutions and thus, more suited to Beowulf clusters. For example, Intermezzo[13] is a distributed file system relying upon concepts developed in CODA[14][2], which intends to solve high-availability and scalability issues. A kernel module on the client side handles local and remote operations by means of user-mode helpers, it makes this solution somewhat intrusive since it supposes modifications of the configuration on the client nodes.

On the other hand, the omnipresence of NFS centralized servers has led to develop new designs to improve the throughput without tackling the other specificities, such as temporal coherency, security and fault tolerance. The most common solution has been to aggregate several nodes, either by putting some smartness into the client (Bigfoot-NFS[15], Expand Parallel File System[16]) or by putting some kind of load balancer between the client and the servers (NFSˆ2[17]).

An alternative is to modify the NFS server by using a meta-data server and storage daemons similarly to the PVFS [18] architecture. Standing from this point, this led us to develop NFSP as a way to offer non-intrusive use and administration.

## 3 `NFSP` overview

`NFSP` [1] is a NFS server implementation using techniques developed in PVFS.

The architecture falls into three parts: the clients, the meta-data server (referred to as meta-server or `NFSPd` for simplicity) and the storage servers (referred to as `iod(s)`, which stands for I/O daemon(s)).

The figure 2 illustrates the sequence of events occurring when a client wants to access a file. The numbers *1-2-3* and *4-5-6* correspond to clients accessing files. This figure also illustrates the fact that a same physical machine may host a client and a storage entity. For both sequences, the meta-server acts as a fixed point (the client only knows it) as it knows to which storage server it has to forward the request to have it processed.

In the NFS protocol (see figure 1), the first step to manipulate files is always to get a NFS handle on the file. This operation is achieved by sending a `LOOKUP` request to the server which will reply by computing a unique file handle based on

[2] A child of AFS.

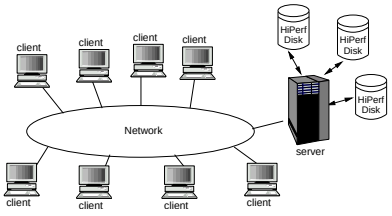


**Fig. 1.** Architecture of a NFS system

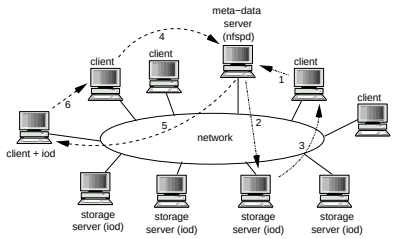


**Fig. 2.** Architecture of a `NFSP` system

some file layout properties (inode, device, etc ...). The way the handle is found does not matter to the client as this field is opaque, which means the client has only to use this field as a file identifier. Once the client has managed to get a file handle, the following sequence of events occurs to read the file: 1) it sends a request to the server containing the file handle, the offset, the size and 2) it receives the results sent by the server.

As we have chosen to split the server into smaller entities (a meta-data server and storage servers), this scheme is slightly modified (from an overall view):

1. send a request to the server containing the file handle, the offset, the size, ... (See fig. 2, arrow #1 or #4)
2. the server receives the client's request and checks the meta-data it holds,
3. based on some information stored in the meta-data it looks for the storage node that holds the requested data,
4. the request is then modified (a copy of the meta-data required to generate the NFS reply is added) and forwarded to the storage node (see fig. 2, arrow #2 or #5),
5. the storage node processes the request (I/O) it has just received from the server on behalf of the client,
6. the storage node sends the reply to the client (see fig. 2, arrow #3 or #6).

This scheme is inherently synchronous and adds obviously some latency since a network hop is added. Yet, one has to keep in mind that there will most likely be several clients that want to access some files on the server, which permits to have overall performance gains by handling several requests at the same time using multi-threading techniques.

Another source of performances increase in the `NFSP` model comes from the fact that the time spent to forward the request on the meta-server is much smaller than the time required to process the I/O's. If the meta-data happen to be cached on the meta-server – which is most likely as they are only a few bytes – then the meta-server does not even have to do slow disk I/O. Another performance boost dwells in the fact that by having several storage servers (`iods`), we have indeed much more cache available than on a single server.

The first implementation of `NFSP` [1] was done extending an existing user-mode NFS server. Unfortunately we found the performances disappointing due

to a saturation of the metaserver processor and I/O. More precisely, for 18 iods on our cluster, described in section 4, the optimal bandwidth is roughly 180Mbytes and the bandwidth we obtained was only 55Mbytes with the CPU of the metaserver used at 100%. In the following we will study two approaches to improve the performances of our prototype.

## 4 Multiplying `NFSPd` (or the number of access points)

Our preliminary work has shown that the main limitation of the current implementation lies in the fact that all the clients use the same meta-server as a mounting point, which causes contention. To bypass this bottleneck, we chose initially to multiply the number of entry points, that is NFS servers.

The natural idea, in this approach, is to introduce several `NFSPd` that would share the same pool of `iods`. However, the main underlying problem, though eased by the NFS coherency, consists in keeping a synchronization between several meta-data servers. We also tried to keep the level of modifications on the meta-servers as low as possible in order to maintain the good level of performances of the current implementation.

This preliminary work has been carried out to implement such a policy by mixing NFS and `NFSP` exports and re-exports: the basic idea is that a set of `iods` is handled by a `NFSPd` server only and that other `NFSPd`'s can mount it as a regular NFS server. By using this technique, it is possible to share several sets of `iods` with different `NFSPd` servers while keeping it completely transparent for the user that always use a single mount point. Of course, if performances are mandatory, it is important that a client mounts the `NFSPd` that will contain most of the data it will access to minimize communication overhead.

Our tests have been launched on the i-cluster[3] (Intel Pentium III 733MHz CPU's - 256MB RAM - 100Mb/s switched network). The bench we use is quite simple: a 1GB file is stored on a `NFSP` volume and is then read again concurrently by a varying number of clients. The aggregated bandwidth is found by dividing the total amount of data served by the time of completion of the last client. The graph in figure 3 contains three curves illustrating the aggregated bandwidth of a `NFSP` system composed of 16 `iods` and successively 1,2 and 4 meta server. As expected, the use of several meta-servers is much more efficient using only one. The simple nfspd curve tends to stagnate then decrease slowly as the number of clients increases. The 2meta-mode curve has almost the same behavior yet the figures are often around at least twice higher. The curve for the optimal bandwidth indicates the maximal peak performance expected (we considered 11.5MB per Ethernet 100 card) and grows till 16 (there need to be at least 16 client to saturate all the `iods`). The 4meta-mode curve decreases as the number of clients increases. The growing communication cost implied by the message passing between meta servers could explain this. An attempt with 20 iods, 64 clients and 12 meta servers (4 clients per server) gave 80% of the optimal

[3] `http://icluster.imag.fr`

throughput. Nevertheless, in this particular case, each server was saturated and that's a real issue from scalability point of view. Hence, even if a good balance between the number of clients and meta-server nodes could considerably improve the performance, the meta-server is still the bottleneck. We try to address this issue in the next section which presents a different implementation of the meta-server itself.

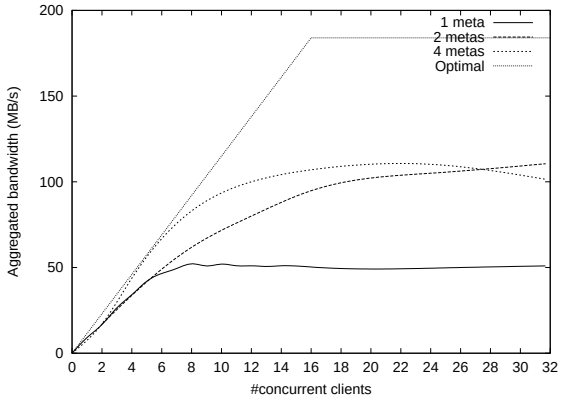


**Fig. 3.** Aggregated bandwidth with a varying number of meta-servers and clients - 16 iods

## 5 Kernel Implementation

Another way to improve performances is to make a more efficient implementation, especially to avoid time consuming context switches between user and kernel mode by porting the `NFSPd` daemon to kernel. This section describes some specificities and provides some indications on how the issues related to this kernel port have been solved.

The kernel mode port has been devised to alleviate the performance limitations observed with the user-mode port. Indeed, this has been found to be necessary as extended tests have shown that at maximal speeds the user server has its CPU completely saturated. The main reasons for this are the high user-mode overheads (memory copies, I/O system calls and context switches). Moreover, for architectural designs and history, `UNFSPd` is a mono-threaded application and performant servers are nowadays based on a multithreaded paradigm. As the NFS implementation of the Linux kernel was already multi-threaded (for obvious reasons), it has been much more easier to start directly with a multi-threaded architecture for `KNFSP`.

To manage several types of exports, the existing set of `nfs-tools` have been extended by setting an unused bit when for a `NFSP` type export. This way, the

meta-server is able to handle both NFS and NFSP file systems exports at the same time.

To illustrate this, the exports file only requires adding a nfsp option:

```
/nfs_export     1.2.3.4/255.255.255.0(rw,root_squash)
/nfsp_export    1.2.3.4/255.255.255.0(rw,root_squash,nfsp)
```

This example file assumes that /nfsp_export contains a meta-data tree and /nfs_export a regular files tree.

We only present results for read operations, as write is mainly limited by the meta-server bandwidth (currently 100Mbps). The bench we use is quite simple: a 1GB file is stored on a NFSP volume and is then read again concurrently by a varying number of clients. The aggregated bandwidth is found by dividing the total amount of data served by the time of completion of the last client.

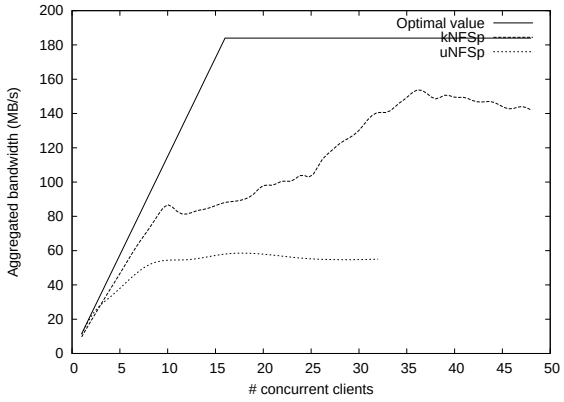


**Fig. 4.** Comparison user-mode server - kernel-mode NFSP server - 16 iods

The graph in figure 4 contains three curves illustrating the aggregated bandwidth of a NFSP system composed of 16 iods. As expected the kernel version is much more efficient than the user-mode one. The user-mode curve tends to stagnate then decrease slowly as the number of clients increases. The kernel-mode curve has almost the same behavior yet the figures are often around at least twice higher. The curve for the optimal bandwidth indicates the maximal peak performance expected (we considered 11.5MB per Ethernet 100 card) and grows till 16 (there need to be at least 16 client to saturate all the iods). The irregularity of the KNFSP curve is due to the timeout policy of the NFS protocol. The slowdown from 10 to 25 clients may match with the first timeouts as the meta-server is being increasingly stressed, then as these wait costs are recovered when there are more clients, it grows again.

The figure 5 illustrates the performances reached as the number of iods varies. For 8 iods the performances soon become quite good, yet for a higher

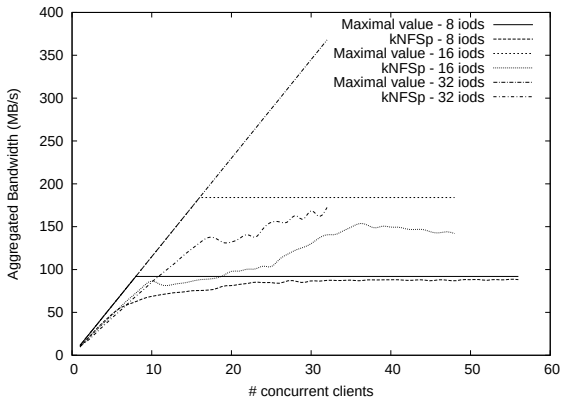


**Fig. 5.** Comparison user-mode server - kernel-mode `NFSP` server - with a varying number of iods

number the optimal level is much higher. We think this is due to the saturation of the Ethernet switch. Nevertheless the performance increases compared to the user-level version are significant.

## 6 Conclusion and future works

This paper has shown the evolution through which our previous prototype has gone. It also illustrates the costs of a pure user-level implementation of the meta-server against a kernel one. There are currently several improvements underway: NFSv3 port, implementation of the kernel meta-server replication and developing a RAID mode to improve fault tolerance regarding `iods`. Some work is also currently being carried out to add a GRID support to WAN transfers between 2 `NFSP` clusters. Assuming each node of a cluster may be linked to a switch and that they may be IP-connected to another cluster, we expect to obtain efficient cluster-to-cluster data transfers by connecting directly remote and local `iods` thus filling more easily the multi-gigabit pipes within a WAN.

## References

1. Lombard, P., Denneulin, Y.: nfsp: A Distributed NFS Server for Clusters of Workstations. In: 16'th IPDPS, IEEE (2002)
2. IETF: NFS: Network file system specification. RFC1094 (1989)
3. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D.: NFS version 3, design and implementation. In: Proceedings of the USENIX Summer 1994 Conference. (1994) 65–79
4. Preslan, K.W., Barry, A., Brassow, J., Catalan, R., Manthei, A., Nygaard, E., Oort, S.V., Teigland, D., Tilstra, M., O'Keefe, M.T.: 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)
5. Schmuck, R.L.F.B.: Gpfs: A shared-disk file system for large computing clusters. In Proceedings of the 5th Conference on File and Storage Technologies (January 2002)
6. Braam, P.J., Zahir, R.: Lustre technical project summary (attachment a to rfp b514193 response). Technical report (2001)
7. Cluster File System Inc.: LUSTRE: A Scalable, High-Performance File System (2002)
8. Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S., Wang, R.Y.: Serverless network file systems. Computer Science Division, University of California at Berkeley, CA 94720 (1995)
9. Ousterhout, J., Douglis, F.: Beating the i/o bottleneck : A case for log-structured file systems. Computer Sciende Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, CA 94720 (January 1992)
10. Hartman, J.H., Ousterhout, J.K.: Zebra striped network file system. Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, CA 94720 (1993)
11. Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: A scalable distributed file system. In: Proceedings of the 16th ACM Symposium on Operating Systems. (1997)
12. Lee, E.K., Thekkath, C.A.: Petal: Distributed virtual disks. In ACM, ed.: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. (1996) ASPLO-7.
13. Braam, P.J., Nelson, P.A.: Removing bottlenecks in distributed filesystems : Coda and intermezzo as examples. Carnegie Mellon University and Western Washington University (1999)
14. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on computers, Vol 39, N4 (April 1990)
15. Kim, Minnich, McVoy: Bigfoot-NFS: A Parallel File-Striping NFS Server (1994)
16. Caldern, A., Garca, F., Carretero, J., Prez, J.M., Fernndez, J.: An Implementation of MPI-IO on Expand: A Parallel File System Based on NFS Servers. In: 9th PVM/MPI European User's Group. (2002)
17. Muntz, D.: Building a Single Distributed File System from Many NFS Servers. Technical Report HPL-2001-176 (2001)
18. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: A parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USENIX Association (2000) 317–327