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

Keywords: clusters, distributed file system, NFS, UDP, spoofing

Abstract

As clusters of workstation get more and more popular (and as a consequence bigger and bigger), disks of these nodes are only used for the system and temporary files. Systems that offer an abstraction of the storage devices in a distributed manner for a cluster are few and far between. In this paper we introduce an extension to the implementation of NFS more suited to the context of clusters because it makes use of the disk space available on the nodes of the cluster instead of the one available on the server only. Our solution relies on the same principles as PVFS: a metaserver and I/O daemons. We present its architecture, the first implementation we did and early performance results.

1 Introduction

The most recent trend in parallelism is the rise of the "poor man" supercomputer, i.e. clusters of PCs connected through a dedicated, sometimes high performance, network. This kind of clusters are now called Beowulf as described by Thomas Sterling in [SSB+95]. 6 years later many works have been done to take full advantage of this architecture in many fields: scheduling, load-balancing, remote login, programming environments and runtime.

Many works have also been done on filesystems, a summary can be found in section 2. However these solutions do not completely take into account the characteristics of a cluster dedicated to heavy computation: high availability, local network, secure environment, large disk space on every node and use of standard, as opposed to highly specialized, software. It is not yet possible to use a subset of nodes to be a distributed file server without using a new protocol, which often requires a kernel modification, or a complete reinstallation, at worst, of all the clients.

*This work utilized resources provided by the ID/HP i-cluster. More information is available at http://icluster.imag.fr

†This work is funded by the French research institute CNRS.

The industrial solutions for storage offers are mainly dedicated NFS servers shipping with several high-performance disks "merged" by means of a hardware RAID technology. This solution works but is quite expensive to buy and doesn't make use of the disk space available on the nodes of the cluster. The system we present in this paper aims at providing a solution that

- enables the use of the disk space of all, or a subset of, the nodes of a cluster,
- gives a unique and unified view of this disk space,
- offers performance good enough to saturate the bandwidth of the network,
- no modifications on the client side.

The last point led us to use the NFS protocol on the client side, the server being basically cut into two entities: one that stores the file system structure (a.k.a metadata) and the other the real data. Using this approach we hope to combine the flexibility and wide availability of NFS together with good performances.

The rest of this paper is organized as follows: in section 2 existing works done on distributed file systems are presented and commented with respect to a cluster environment. Then section 3 presents the hardware and software context in which this work was done and in section 4 we explain our proposal. The following section, 5, presents some technical issues we had to deal with to implement our distributed file system and section 6 give the first results we obtained with our implementation. We conclude and give directions for future works in section 7.

2 Related works

There has been work on distributed/network file servers since the early eigties. They aim at addressing various issues like: security, performances, distributed caching. In this section we review some of them and focus on the issue they want to address. Since we focus on Beowulf clusters, we will only review here solutions available **freely** with an **open source license**.

The AFS family (AFS, OpenAFS, CODA) aims at providing a secure way of sharing files among the nodes, within a campus for instance. There are main servers ("cells") that are often dedicated file servers. The installation cannot be described as trivial, yet once installed it offers an effective way to share files securely on a large scale. Besides the on-disk local cache system it uses is found to be really handy and offers performances boosts. Yet, for a safe environment such as a protected cluster, this solution does not suit since it adds many things not necessary for a cluster and requires some not-so-obvious tweaking.

xFS[1] ([WA93] and [WAD97]), is a part of the Project NoW at Berkeley that finished a few years ago. Thus, it is no longer maintained and has not been ported to Linux.

One of the most advanced project concerning cluster filesystems is hosted by the Parallel Architecture Research Laboratory, at Clemson University: PARL began developing PVFS [CIBT] in the late nineties. From a software view, a central metaserver and I/O daemons are used. This system ships with it own filesystem type (`pvfs`) registered into the system by means of a kernel

[1]This is not SGI's XFS - the journalled filesystem - recently open-sourced and ported to Linux.

module to allow a transparent use by the end-user. The configuration and installation was a bit tricky when it was tested on a few nodes, a few months ago. Currently, PVFS uses its own protocol of communication over TCP streams, which obliges the admin to modify the clients' configuration.

Last but not least, the NFS server is quite simple to install and the client part has been used and tested for several years. It may not be a POSIX-compliant filesystem or sometimes have an awkward semantic but it works for most cases. Yet, it does not scale well as soon as there are several concurrent clients and the storage is centralized.

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 (AFS) or need careful installation and maintenance (PVFS). This is what motivated us to develop an extension to NFS to provide these characteristics.

3 Context of the Work

As part of a project to provide a complete infrastructure for computing on top of corporate intranets, Hewlett-Packard gave to the ID lab a cluster of 225 PCs connected with a low cost network. The i-cluster is built using 225 off-the-shelf HP's iVectra: Intel Pentium III 733MHz, 256MB RAM, 100Mb ethernet cards and 15GB IDE hard drive. It has been ranked 385th in the latest Top500[2] list of the most powerful systems (see [RAM+01]).

After several months of use, it occured to us that the disk space available on the nodes was seldom used, the base system and the swap taking at most 4GB, the other 11GB left are lost. So we started looking for a distributed file system that could avoid this loss. Since we are in the context of a cluster used every day by many users we had two important requirements:

compatibility do not force a major rebuild of the installation,

stability do not break the whole system by using standard (heavily tested) software and avoing messing with the kernel.

As explained in section 2, none of the solutions we tried satisfied these constraints, that is why we decided to develop our own solution: designing a simple drop-in replacement for the network filesystem NFS that would dispatch the load of I/O's among a pool of machines.

4 The nfsp proposal

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

The traditional architecture of a network file server is shown on figure 1. There are communications between the server and the clients, the architecture of the server, and, in particular, the

[2] http://www.top500.org

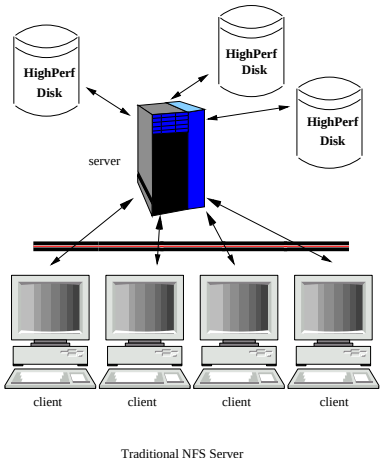

Figure 1: Typical architecture of a network file system

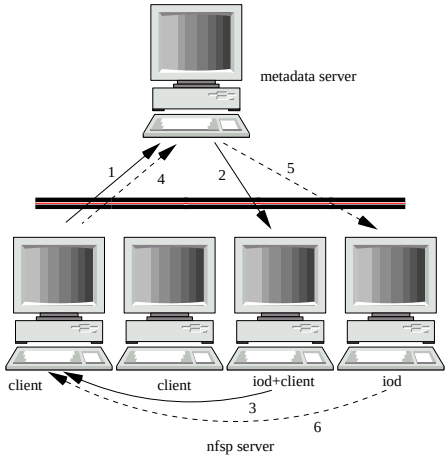

Figure 2: Architecture of our parallel network file system

number of disks it has, is hidden to the clients. The bandwidth used between them does not depend on the number of disks the server contains or on the number of clients it has to serve but on the network between them. Such an architecture is though not very scalable, the server will always remain a bottleneck. Thus it seems interesting to try to distribute the server on a set of nodes to avoid this, such works are presented in section 2. The problem is then to succeed in maintaining a coherent state in the filesystem without centralizing it. This implies an overhead that is not always acceptable and makes more complex, and therefore difficult to design such a system “bug-free”. It also means designing a new protocol and forces the clients to implement or install it as well.

The approach we propose is to keep a central server that fully implements the NFS protocol, so no change is needed on the client side, while distributing the storage of the data on several nodes of the clusters. Hence the transmission of the “real” data will occur between nodes and not between the server and the clients[3].

The architecture of our proposal is shown on figure 2. When a client has a read or write operation to do it will send it to the server, to comply to the NFS protocol. If it is a read operation then the server will simply forward it to the node that has the data requested. This node will reply directly to the client using the server address to do that[4]. If case of a write operation the server will forward it to the node that will store the data.

This is shown on figure 2 by the sequence of messages necessary to handle a first request(1 to 3), in this case a read one: the request is sent to the server that forwards it to the node that has the data requested which then replies directly, not through the server, to the asking node. The messages 4 to 6 illustrate the processing of another request.

[3]Only for the read operations, we will see why later.

[4]This operation is called spoofing the server address in the literature.

5 Technical Issues and implementation

This section presents the technical aspect of our server and shows where we had to modify a standard implementations of NFS.

5.1 Standard NFSv2 protocols: NFS, RPC, UDP

The NFS protocol (version 2) is normalized in the RFC1094[IET89]. It is built over the SunRPC[IET88] mechanisms using XDR[IET87] for data formats. SunRPC's may be used over TCP or UDP sockets, but concerning NFS, most server only offer UDP support[5] Besides, as the RPC system brings a fiability layer to the NFS protocol (acknowledgements/resends of requests, ...), TCP may not offer great improvements, except to avoid congestion, which is relatively uncommon on modern switched local networks.

The NFS server set is split into two processes, as shown in figure 3:

- the **mountd** process that implements the mount protocol (some kind of initialization protocol),
- the **nfsd** process that does the I/O and processes the requests.

Once the client has mounted the NFS volume, the communications happen between the client and the nfsd by means of UDP packets. The server waits on port 2049 and client on a system port (below 1024). Thus, the client sends a request contained in a UDP packet: IP_{client}:port_{client} to IP_{server}:2049, then it expects to receive a reply in a UDP packet which source is IP_{server}:2049 and destination, IP_{client}:port_{client}.

5.2 Why a user-mode server?

Most NFS servers are multithreaded and runs at the kernel level to increase the throughput. As we don't want to be intrusive regarding configuration, we decided to use a user-space implementation of the NFS server instead of modifying the kernel one.

The Linux User-Space NFS server was implemented a few years ago by Mark Shand, and further enhanced by Donald Becker, Rick Sladkey, Orest Zborowski, Fred van Kempen, and Olaf Kirch. It did not evolve much since 1998, when the version 2.2 was released.

It is a bit slower than the kernel mode server, mainly because of the overhead added by data copies: disks to kernel buffers to user space to kernel space (network buffers) for a read request, the write being in the reverse order.

Yet, it is simpler to modify and a crash will not hang the whole machine (compared to a kernel level server). Besides, installation is also quite simple and requires minimal administration and configuration provided there is already NFS support available, which is the case of most systems.

5.3 A few definitions

In the following parts, the following terms will be used as defined below:

[5] maybe because most clients only have only support for the UDP mode...

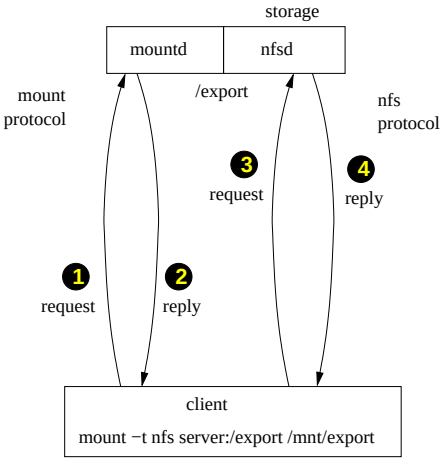

Figure 3: Traditional NFS server

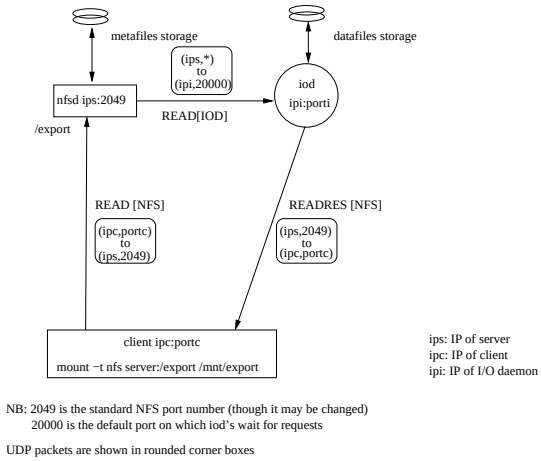

Figure 4: NFSP server

real file: the file as seen by the user

metafile: the file on the nfsp server

metadata: data concerning a file - in nfsp, metadata are handled at two levels:

- the underlying file system for timestamps, owner, etc...
- another part stored in the metafile (size of the real file, seed)

datafile: a file containing data of the real file. We will see later that these data are stored on the iods.

To illustrate this: a real file is made of a metafile holding metadata on the metaserver and of datafiles striped on I/O daemons (iods) running on I/O nodes.

5.4 Architecture overview

The file system is handled by two kinds of server: one that deals with metadata (filenames, permissions, ...) and several I/O daemons that 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`).

There exists a mapping:

- one meta file $\Leftrightarrow$ one real file on the metaserver
- one "special file" (directory, pipes, blocks or character devices) $\Leftrightarrow$ one "special file" seen by the client

Thus if you do a `find /export` on the metaserver or on a client you will see the same files. Yet, if their size is looked at, it will appear that on the metaserver every file has the same size (a

few bytes) and on the client their sizes are different (you will find an example in the following section).

File attributes on the metaserver are stored in two places:

- in the metadata of the underlying filesystem (ext2fs for example), for "standard" attributes, that is the one not altered by striping: creation time, owner, group, permissions, etc...
- in the metafile itself: size of the real file, "seed" of the file...

5.5 What are the differences with a standard NFS server?

This part presents how we implemented the most common operations in the prototype we developped.

5.5.1 Creation

When a regular file[6] is created by the client, the metaserver creates a metafile whose name is the same as the one specified by the client. This metafile is then opened and the following things are written into it:

a magic number: used to be a bit more sure it is a correct metafile,

a seed: this figure is generated using the libc `rand()` function and is currently used to give a "version" (or "cookie") to the inode a metafile lives in; it will also be used to choose the first node of striping,

the size of the real file: this value is modified when data is appended to an existing file or when this file is cut.

A seed is used since iods only know the data they store by the inode and seed of the metafile. The problem that could happen if such mechanism were not used would be that stale data could be read if the metaserver is short on inode numbers and reallocates an inode before the data is flushed on the iods.

With this seed, even though the inode is reallocated and the iods did not flush all of the data they had about the previous "version" of the file, stale data will not be read (some kind of lazy deletion). This mechanism is similar to the one found in the inodes of the `ext2` filesystem (the file version may be found by means of `lstattr -v`).

Another consequence of the mapping used is that when the user hardlinks two real file, in fact, two metafiles that hardlinked.

When a special file is created by the client, the metaserver creates the corresponding special file for storage.

As a result, if you compare the directory listing on both a client and the metaserver, they will be the same. However, if you consider the sizes of the files, you will notice that regular files stored on the metaserver all have the same size (a few bytes) whereas the clients see them with different sizes.

[6] that is: not a directory or special file(pipes, block or character devices)

Example:

On the metaserver: `$ls -l /META`

```
-rw----- 1 user group 12 Sep 10 14:27 test.0
```

On a client: `$ ls -l /MNT/`

```
-rw----- 1 user group 295456 Sep 10 14:27 test.0
```

5.5.2 Deletion

On a classical NFS server, a deletion request is sent by the client to the server which call `unlink()` on the file and then acknowledges the RPC.

This process, though simple is a bit trickier in nfsp. When a client wants to erase a file, it sends a request to the metaserver. The metafile is `stat()`'ed and its hardlink count is checked: if it is above 1, then the metafile is removed and the underlying filesystem will decrease by 1 the count of the other hardlinks. Yet, if it is 1, then the metaserver sends a message containing the inode and seed of the file to an ancillary process (`unlinkd`) that was spawned at the launch time of `unfspd`. The communication pipe is a standard Unix pipe. Then, it sends an acknowledgment to the client.

Meanwhile, the `unlinkd` process gathers remove requests till a maximal number of messages is reached or some specified inactivity delay has been observed. Once one of these events happen, it multicasts to every iod through TCP streams the files that need to be erased.

For your information, in our first implementation, we used to send remove notifications via UDP and did not care if they did not succeeded sometimes, thus accepting some disk leaks that would be fixed by another maintenance tool. Yet, when a burst of remove notifications had to be sent, they flooded the UDP socket and were silently lost, which caused massive disk leaks on the iod, so we used another approach.

5.5.3 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. The latter reads metadata, finds the iods that will hold data and then forward it the request. 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 4.

5.5.4 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 the same way as for a READ request as illustrated in 4.

5.5.5 Other requests

Every other NFS request are handled directly by the metaserver, without interacting with the iods.

Some care has to be taken though: sometimes, metafiles have to be read so as to send a correct reply to the client.

For instance, when the NFS request `STAT` - used to get metadata of a file - is required[7], the NFS server simply calls the `stat()` function.

In NFSP, those calls are wrapped in functions handling the metadata stored in the metafile, which allow the nfspd to generate a valid answer in the view the client has.

5.6 I/O daemons (or iods)

I/O daemons were first designed as small server loops that waited and processed UDP packets sent by the nfspd. Yet, this approach has been found soon to be not easily extensible to correctly handle the deletion of files and we needed to add a TCP stream (see the explanations in 5.5.2).

That is why we moved to a multithreaded approach since synchronous I/O's with multithreading tend to offer a good compromise for performances and simplicity of code (compared to asynchronous I/O's). Iods have been rewritten to use POSIX threads and now we have a more simple and maintainable architecture.

There is an array of buffers used to spool request received and wait for I/O.

- one thread gets requests via UDP packets from the metaserver (READ/WRITE/PING) in one of the buffers then wakes up an I/O thread,
- one thread gets requests via the TCP stream the metaserver opens when it is launched (indeed only PURGE requests) 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 socket dedicated to output is a RAW socket. It is necessary to use such a low-level socket to control the way packets are sent. A simplistic UDP stack is then used to send messages (fragments them according to the MTU of the used network interface).

The simple protocol used between the metaserver and the iods is (imaginatively :) called **iod protocol** and consists of a few messages:

READ sent by the metaserver to the iod, contains inode number, offset, seed, the 4-uple (IP_{client}, $port_{client}$, $IP_{metaserver}$, $port_{metaserver}$), the RPC xid[8] and the file attributes. The iod spoofs the reply to the client.

WRITE this request contains the same data as the read but it also includes the data to write. The iod is to spoof the acknowledgement to the client (NFS/RPC reply).

PURGE this request is used to flush every data file still existing on the iods. Currently, it is sent to every iod in a row but is scheduled to be replaced by a multicast or broadcast UDP packet.

[7] if the user requests on the client a `ls -l`

[8] identifier number of the Sun RPC request

Two other messages types exist, mainly for administrative (or troubleshooting !) purpose and more specifically to test whether the iods are running fine.

They may be sent from any host to the iods and are:

- PING: the PONG reply will be sent by means of a UDP socket from "our" UDP stack, thus it may spoof another IP address to test whether spoofing works,
- PONG: the acknowledgement message is sent back to the requester.

5.7 Considerations of security

The protocol in itself is not really secured and rather aimed at a safe environment, such as the one found in a cluster.

An additional level of security might be added by forcing iods to listen on ports below 1024 which would mean that they were launched by a privileged user. The `nfspd` might also be told to use such ports to offer a relatively small improvement in security.

Iods must be launched with superuser privileges to open the RAW socket but these privileges are dropped as soon as the RAW socket has been opened.

5.8 Installation

The installation on the master node, the one that hosts the metaserver, is meant to be quite simple:

- edit an `exports` file (same format as for standard NFS server)
- edit an `iods.conf` file (list of the pool of iods)
- launch the iods on the hosts/ports you declared in `iods.conf` on all the I/O nodes
- launch the `mountd` daemon
- launch the `nfsp` deamon

And *voilà*, the server is now up and ready to serve.

The client has just to mount the NFS volume it described in the `exports` file:

```
mount -t nfs metaserver:/export /mnt/foo -o rsize=4096,wsize=4096
```

An `iod_ping` utility is available to check the state of the iods, and may help to check if spoofed UDP packets are allowed to be sent to the network.

6 Preliminary results

This section presents the results of the tests we ran. They should be considered preliminary since the code has places where our nfspd daemon really needs optimizing.

Please note that in this document, B stands for byte and b for bit.

6.1 Description of the testbed

All the nodes metaserver, iods or clients have the same hardware (described in section 3) and run an off-the-shelf Linux Mandrake release 7.1, powered by a custom compiled standard Linux kernel 2.4.4.

To lower the influence of the cache system offered by the kernel, the memory available for the system should be set to a lower value: for instance at boot time, one may use the `mem=16M` option may be used to tell the Linux kernel to consider only 16MB out of the total system RAM in the system. However, this approach is not yet usable since it would require restarting the system, which our reservation system does not handle quite well.

The other approach we chose since it was less intrusive was to defeat the cache by disabling the swap devices and using a file being twice the RAM size.

The NFS server of our cluster (running a P3-1GHz 512MB RAM and 1GB swap powered by a 2.4.5-xfs-1.01 Linux kernel).

To start the clients almost simultaneously, and thus heavily stress the server processes (nfspd+iods), we used the parallel launcher `ka-run` [9] developped by Cyrille Martin (see [MR01]).

6.2 `bonnie++` tests

In this test, nfsp has been tested by `Bonnie++` 1.0.2 by Russell Coker [10].

We have had some slight troubles with running the massive file creation and deletion, which most likely comes from the recent rewriting of most of this code.

In the two following tables, the size of the test file is 512MB (twice the RAM size).

	Write						Read				Random	
	/char		/block		Rewrite		/char		/block		seeks	
	KB/s	%CPU	KB/s	%CPU	KB/s	%CPU	KB/s	%CPU	KB/s	%CPU	/s	%CPU
NFS: 1c	9065	96	10701	8	4986	4	8211	81	9683	4	210.7	1
NFSP(4): 1c	7514	72	7248	6	4376	4	7889	80	8550	4	623	5
NFSP(8): 1c	7255	69	8032	6	4541	4	8955	93	8973	5	679	5
NFSP(16): 1c	8474	82	7907	7	5301	5	9044	94	10067	6	627	5
NFSP(32): 1c	8486	80	8284	7	5090	5	9034	94	9627	5	662	4

With a single client, the performances are not wonderful, and even a bit worse than the one we evaluated on the NFS server. This may be explained by the overhead implied by the approach, in latency for instance since messages have to be transfered via the nfspd.

In the table below, two clients are launched almost simultaneously with `ka-run`.

	Write						Read				Random	
	/char		/block		Rewrite		/char		/block		seeks	
	KB/s	%CPU	KB/s	%CPU	KB/s	%CPU	KB/s	%CPU	KB/s	%CPU	/s	%CPU
NFS: 1st c	5401	49	5730	4	2531	2	4698	42	5174	2	273.4	1
NFS: 2nd c	6274	57	5753	4	3361	3	4666	43	4575	1	204.4	1
NFSP(32): 1st c	5315	48	5555	4	4677	4	8135	82	8130	4	683.5	4
NFSP(32): 2nd c	5737	53	5810	4	4080	3	8300	83	10462	6	698.4	6

[9] Further information at `http://ka-tools.sf.net/`.

[10] Check `http://www.coker.com.au/bonnie++/` for further information.

As one may see it, the write operations are limited by the 100Mbps link (around 11MB/s), which is observed if client 1 and client 2 write speeds are added.

Yet, for the read, unlike the NFS server, more bandwidth is available since several network cards may be used, which appears when the read speed of both clients are added.

6.3 Concurrent sequential tests

We tested a massive sequential read with an increasing number of clients.

To illustrate this, we created a 1GB (4 times the RAM size), and used 16 iods. This file using the following command: `dd if=/dev/zero of=file bs=1M count=1024`, which took around 150s.

To have this 1GB file read concurrently by several clients we issued the following command on every client: `dd if=file of=/dev/null bs=4096`.

The performances we obtained are illustrated below.

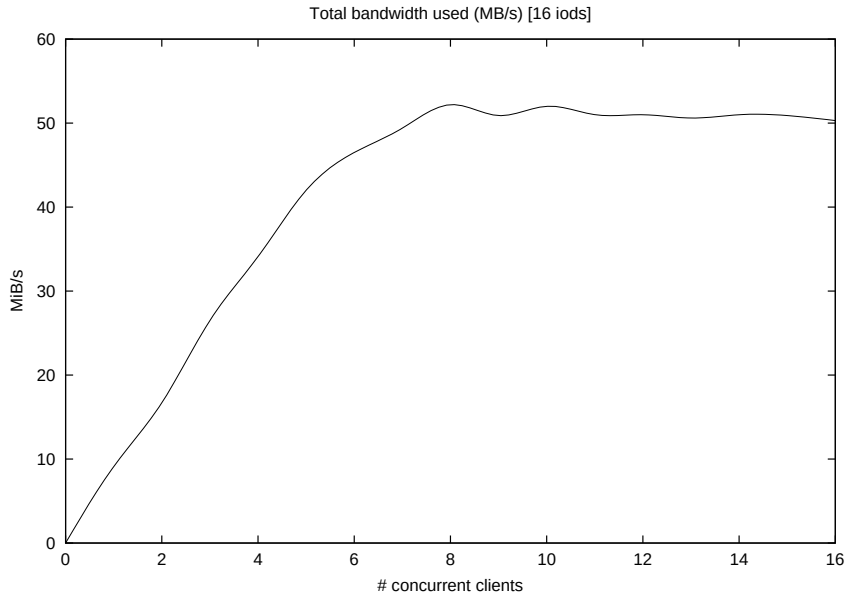

Figure 5: Total bandwidth used (MB/s)

Starting at 6 clients, the cumulated bandith served by the nfspd does not increase and finishes by becoming almost constant, which is a bit disappointing, and reveals the presence of a bottleneck.

We first thought that we saturated some buffer in the switches we used (HP Procurve 4000) but a quick look at their specifications showed a maximal backplane speed of 3.8 Gbps, which let us some margin!

50MB/s corresponds roughly to 12,000 4KB requests thus 12,000 messages are transfered from the nfspd to the iods, which is around 2,000KB/s, which does not saturate the nfspd bandwith, either.

However, we suspect that the metafile handling has to be optimized (at least by some kind of caching), which is not currently the case, indeed the metafile is read again each time a request has

to be replied to, which costs syscalls and cache file access.

This phenomenon is currently under further investigations and we are confident that this bottleneck will be exactly identified and fixed for the final paper.

6.4 Summary

The write will be limited at the speed of the metaserver ethernet card since using the NFS protocol implies the client talks to its server: data must be received then, forwarded to the iod that will store them.

The read looks more interesting and promising since the requests between the nfsp and the iods are small and do not saturate the metaserver network interface. Then, the iods process read operations and send back the data directly to the client, not wasting the metaserver bandwidth (on a switched network of course).

7 Conclusion and future works

In this paper we introduced a extension to the NFS protocol to adapt it to the context of clusters. Our proposal is to split the NFS server in two parts, one in charge of the metadata and the other of the data themselves. We presented the architecture of our system and an implementation that gave interesting preliminary results.

We currently think of several extensions to enhance the system:

- enhance the metaserver by using caching of metadata and not using the facilities offered by underlying filesystem for storage of metadata. . . ,
- use threads on the metaserver to increase the throughput: the original architecture of the NFS user-mode server has been kept to avoid a major rewrite: it is a monothreaded (possibly forked) applications but is not cleanly and easily extensible; yet, before doing this, the code needs auditing since several parts of the code are not reentrant,
- run as a kernel extension to avoid excessive memory copies inherent to user space applications when our implementation will have proven its robustness,
- implement a distributed NFSv3 server: asynchronous write tend to offer an interesting gain in efficiency. Besides this protocol support large files (over 2GB) unlike NFSv2. Some research has still to be carried out to see how the NFSv3 commit request may be handled in a coherent way and how not to overload the metaserver.

NFSP aims at being simple to install and to run and at using several machines to load balance the I/O bottleneck onto several hard disks. Besides, a side effect of this appears since there are more machines, thus there is more memory and consequently more cache and hence more performances, which is not a negligible point.

Adding redundancy without sacrifying performance has also to be investigated, as well as the use of multicast UDP datagrams (recent switches handle them natively).

Just as you need more power in a cluster, we would like to extend the file system as easily: plugging a new disk or a new machine into a cluster and the available disk space would increase smoothly.

The prototype has been running on our cluster for about one month, and has been used by several users. We have not met any critical problems so far. A detailed description of the installation process, together with a CVS snapshot is currently available at the following URL: `http://www-id.imag.fr/Laboratoire/Membres/Lombard_Pierre/nfsp/`.

References

[CIBT] Philip H. Carns, Walter B. Ligon III, Robert B. Bross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters.

[IET87] IETF. XDR : External data representation standard. RFC1014, June 1987.

[IET88] IETF. RPC: Remote procedure call protocol specification version 2. RFC1057, June 1988.

[IET89] IETF. NFS: Network file system specification. RFC1094, March 1989.

[MR01] C. Martin and O. Richard. Parallel launcher for cluster of PCs. In *ParCo 2001*, September 2001.

[RAM+01] 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.

[SSB+95] 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.

[WA93] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the 4th Workshop on Workstation Operating System*, 1993.

[WAD97] Randolph Y. Wang, Thomas E. Anderson, and Michael D. Dahlin. Experience with a distributed file system implementation. Technical Report CSD-98-986, January 1997.