Title : Minimal implementation of distributed filesystem similar to pNFS

People involved

Ravi Kumar Manjunath : CWID – A20238832

Sanjeev Bagewadi : CWID - A20237384

Abstract :

As part of the course work for CS550 we have designed and implemented a miminal distributed filesystem which is similar to pNFS using the filelayout method. It provides scalability by striping the data across multiple data servers and a single metadata server.

This report looks the implementation details of a minimal implementation of a distributed filesystem similar to pNFS using FUSE on the client side and communication over RPC. It describes the approach taken for scalability by means of striping the data across multiple servers. We then look at the tests conducted to evaluate the implementation and conclusions from them.

1. Introduction and Motivation

With the increase in the amount and rate of data being generated there is a growing need to develop filesystems which can scale up to handle these needs. With the traditional model of a single server serving this capacity the server tends to become the bottleneck. Splitting this data across multiple filesystems is one of the solutions. However, with this the simgle namespace requirement will be broken. Hence, there is a need for scaleout filesystems.

Several parallel/distributed filesystems are developed or being developed to cater to these needs. Some of the examples are GFS from Google, GPFS, PVFS, Lustre and pNFS. And each of these filesystems use different techniques to achieve scalablity.

This project implements a minimal distributed filesystem based on the pNFS as described in NFS V4.1 specs. pNFS specs allow different implementations of metadata server namely Object based, SAN based and Filelayout based. We chose the filelayout option as it is simpler and is easy to implement in a short span. We have implemented the following features :

- Basic file operations : create, open, read, write, close and delete
- Directories : create, open, getents etc.

The key focus of this project is to implement striping of data across multiple servers and hence increase the available bandwidth. We plan to use the file-based layout similar to the one described in the pNFS 4.1 RFC Section 13.

2. Background Information

Distributed filesystems are becoming more important with the increased stress on parallel computing. There are several distributed filesystems catering to the scale out model which include : pNFS, Lustre, ibrix, GPFS, GFS etc. However, these are not the first ones in this family. There have been older generation distributed filesystems like : AFS and later DCE/DFS derived from AFS.

With the increase in the amount of digital information being created by various apps, right from digital photographs, high defination videos to large amount of data generated by sensors like weather data, single servers become a bottle neck for scalability. Distributed filesystems play a key role in achieving high scalability (both in terms of speed and capacity) by spreading out this data accross multiple servers and still providing a single namespace for it.

We explored the various currently available distributed/parallel filesystems and compared the techniques they adopt to achieve scalablity. This information is described in the Related work section of the paper.

2.1 Problem Statement

As stated in the section above we know that a single server model of a network filesystem where the entire filesystem is served by a single server becomes a bottleneck in HPC world. Also the increasing disk capacities make it harder for a single server to scaleup to the need.

This problem can be addressed by separating the metadata from data and then spreading the data across multiple servers. The next sections describe our approach and implementation details.

3. Proposed Solution

Our implementation is heavily based on the pNFS RFC (NFS Version 4 Minor Version 1). Section 12 of this paper describes the pNFS protocol and Section 13 describes File Layout Type. We have derived our implementation based on these sections and have implemented a the File Layout Type (the other layout options being Block Layout and Object Layout).

The following is the proposed solution. In this project we have implemented a basic filesystem which allows file read/write operations. For the work we have used the following frameworks :

- FUSE : The FUSE infrastructure of Linux which allows us to implement filesystems in userland with very little difficulty.
- RPC : We use the RPC framework for communication between the fuse filesystem client and metadata and data servers.

The following are the main modules of our implementation :

- pnfs client : This is the fuse plugin which acts as the client and contacts the metadata and data server
- Metadata Server : This is an RPC server which maintains the directory layout of the filesystem and the metadata of the files. It also maintains the layout details of the files. i.e. the details of where stripes of the files are layed out.
- Data Server : This an RPC server very similar to a regular NFS server.

3.1 Pnfs Client :

This is the FUSE plugin and which is used to mount he remote filesystem locally. It takes the name of metadata server name, the remote directory name that is being served and the local mountpoint where it is to be mounted.

Once the client is runing for all the filesystem requests for that mountpoint FUSE kernel module calls into this client. The client inturn contacts the metadata server for all the metadata operations and then contacts the data servers for any data operations.

Later in this document we explain how the control flows for each filesystem operation.

3.2 Metadata Server :

This is an RPC server which handles all the metadata of the filesystem that is being shared. This server handles the following functionalities :

- It maintains a list of data servers serving the given filesystem
- It handles all the metadata of the filesystem. This includes the directory structure and directory entries, file metadata (size, permissions etc) and the layout of individual files which describes where (i.e. Dataserver name) each extent of the file is located.
- The pnfs_client contacts the metadata server for all the operations. We will describe the operations which are slightly different a little later,

NOTE : Today the metadata server is capable of sharing only one instance of filesystem. However, it is easily possible to extend it to share more filresystems. We could implement a **share** command which would add a

new filesystem to be shareed.

3.3 Data Server :

This is very similar to a plain NFS server. It maintains the extents of individual files (as layed out by the metadata server). The data server is only responsible for maintaining the data of the files and not the metadata. This is very similar to the chunk servers of CFS in a way.

3.4 Striping of files and READ/WRITE operations :

As mentioned in the earlier paragraphs our implementation stripes the data of the files across multiple data servers. And also when small files are created they are assigned to different data servers. This allows achieving good distribution of load amongst the data server. Today we use a simple round-robin method for this purpose. However, better load spreading mechanism can be easily added.

In the following paragraphs we describe a typical **create, read** and **write operations.** However, before doing that we describe the typical layout of a file on the metadata server. If we have file by name *file1* the contents of it would look as below :

```
# cat mds_share/file1
      24144
0     16384 dshost1     file1.ext0
16384 16384 dshost2     file1.ext1
```

Here the first line "24144" indicates the current size of the file ***file1.*** The subsequest lines describe the details of extents of that file :

- the first extent starts from offset ***0*** of size 16K is on dataserver named ***dshost1*** and is named ***file1.ext0***
- the second extent is from offset ***16K*** and is of ***16K*** in size and is on dataserver named ***dshost2***

NOTE : For the current implementation we have selected a extent/stripe size of 16Kbytes. However, for realworld scenarios this can be easily increased to 1M.

3.4.1 Create operation :

During the create operation, the pnfs-client contacts the metadata server. The metadata server just creates the file and writes a size of 0 in the first line.

3.4.2 Write operation :

For this operation the pnfs-client makes the getlayout() to the metadata server. The metadata server searches for the matching extent record for the given offset and length. If no matching extent is found a new extent is allocated. It then sends the extent record back to the pnfs-client. The pnfs-client then contacts the dataserver to write that extent.

3.4.3 Special handling on dataserver for writes :

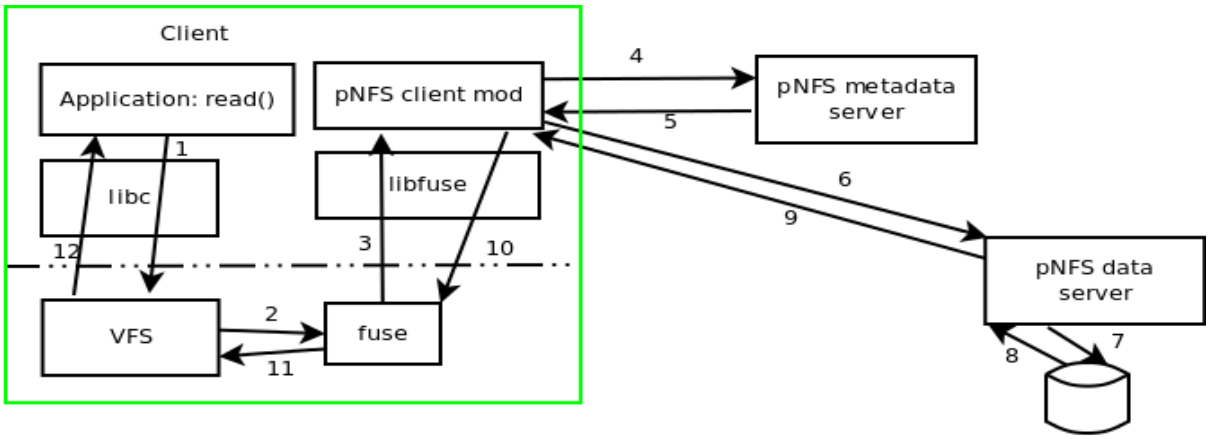
The directory creation (mkdir operation) from the pnfs-client is carried out only on the metadata servers. This info is pushed to the data-servers selectively only when extents of a file in these directories are written to the dataserver. For this the dataserver does the following additional work :

- When a write of an extent file is requested, it check if the corresponding directory of the extent file and extent file exist. If not, the directory path leading to the extent file are created on the fly and then the extent file itself is created. It then writes the actual contents of the extent file.

With this approach the mkdir() operation is more scalable as the metadata server need not propagate the directory creation to all the dataservers. Instead, this is propagated lazily only to those dataservers that need it.

3.4.4 Read operation :

This is very similar to the write operation described above. The only difference is that if metadata server does not find a matching extent it does not allocate a new one. The pnfs-client on receiving the extent record issues a read request to the corresponding data server. The below diagram depicts a typical read operation.



Control flow for a read operation in the proposed pNFS setup

3.4.5 Parallelization of Read/Write operations :

In order to increase the efficiency of the pnfs-client we split the given read/write operations in to chunks of 16K and issue them in parallel using separate threads. This is implemented in the routine pnfs_p_rdwr() (pnfs_client.c : pnfs_p_rdwr() and pnfs_rw_ext()).

This should give us a good performance boost in case of large read/write operations.

4. Evaluation :

As described earlier our main intention was to make sure that we are able to stripe the files across multiple data servers and make sure that we are able read/write files and make sure that the data is intact.

We evaluated the work by testing the following basic functionalities :

- Create small files (less than stripe size – which is choosen as 16k currently) and make sure that the files are spread across multiple data servers available.
- Deletion of files (make sure that the stripes of the files spread across the data-servers are cleaned up as well).
- Create large files (> stripe-size) and make sure that data is striped across multiple data servers.
- Read the large file and ensure that the contents are intact (using cksum or md5sum).
- Renaming of a file (and make sure that the stripes are renamed as well) and to verify that the contents of the new file are intact by reading them.

Details of this of the test carried out are available in Verification.pdf.

We also tested the **scalablity aspect** of the system. Read and write operations were carried out on a large file with varying number of data servers. Below are the results.

File Size : 1.4 Mb		
No of Data Servers	**Read**	**Write**
1	0.417	4.457 s
2	0.409	3.803
3	0.397	3.119

We have not been able to implement the following :

- Renaming of directories.
- Links (both soft and hard links). This is partially implemented but behavior is not correct or complete.
- Extended attributes
- setattr (to modify the time). Hence commands like touch fail.

Also, the page caching of the files is not implemented. We could have tried to use the page-caching support of FUSE using "-o kernel_cache" or "-o auto_cache". However, due to lack of time we did not.

5. Related Work :

There are several distributed filesystems which address the problems described in the previous section.

- **AFS** : Andrew FileSystem developed by Carnegie Mellon University as part of the Andrew Project.
- **DFS** : This filesystem was derived from AFS and was developed as part of DCE in response to the ONC NFS from Sun Microsystems.
- **GFS** : The Google FileSystem was developed by Google to provide efficient, reliable access to data using large clusters of commodity hardware. The GFS master server and chunk servers are very similar to metadata server and data server of our implementation.
- **GPFS** : The General Parallel File System (GPFS) is a high-performance shared-disk clustered file system developed by IBM. In GPFS the servers handle both metadata and data and this is distributed. However, with our implementation there is one metadata server and a multiple data servers.
- **ibrix** : The IBRIX Fusion Software Suite comprises a scalable parallel file system, integrated logical volume manager, availability features, and a management interface. In case of ibrix the client once it gets the inode would not need to contact the central server for anything else for that inode. Because, the inode number has the segment-id embedded into it and the client contacts the segment server directly. Unlike this in our case the client would need to contact the metadata server for the details of the layout for every read/write.
- **lustre** : Lustre is a massively parallel distributed file system, generally used for large scale cluster computing.
- **NFS** : This was one of the first popuplar distributed filesystems which was widely adopted. However it was not a scalable filesystem until pNFS (NFS V4.1).
- **pNFS (NFS v4.1) :** This is the scalable version of the NFS. And our implementation is heavily based on this and some ideas from GFS.

Our implementation is heavily based on the pNFS RFC (NFS Version 4 Minor Version 1). Section 12 of this paper describes the pNFS protocol and Section 13 describes File Layout Type. We have derived our implementation based on these sections and have implemented the File Layout Type (*the other layout options being Block Layout and Object Layout*).

Deliverables :

The following are the deliverables :

- FUSE client module
- metadata server module
- dataserver module

The file Program-list.txt contains the entire list of files and executables.

6. Conclusion

As detailed in the above sections we have managed to get a working filesystem which stripes data across multiple servers and still provide a unified namespace (provided the metadata server). In the due course of this project we learnt the following :

1. **Basics of a filesystem :** Since we used the FUSE framework we used the interfaces defined by it. These interfaces closely map to the VFS/VNODE operations of Linux. This gave us a good working knowledge of how various filesystem related system calls translate to inode/vnode operations.
2. **Protocol Definition of Network filesystems :** The process defining IDL definitions in mds_ds.x we learnt the process if defining a protocol for network filesystem
3. **How to implement them over the network ? :** The usage of RPC framework and developing of multithreaded servers to handle these requests is a learning in this direction.
4. **How to achieve scalibility (both for speed and capacity) ? :** The RPC calls are blocking in nature. Hence to achieve parallelization of read operations to multiple servers was done using pthreads. This provided scalibility in terms of speed. Also, the striping of file extents across multiple data servers allows scaling of capacity.

With the current implementation available we are able to create and delete files. Read and write contents to the file and ensure that the data is striped across multiple data servers and reads to multiple dataservers are done in parallel to achieve scalability. The Verification.pdf describes the tests that were conducted and their results. With these results we feel that we have achieveed what we had proposed to implement :

- Create an instance of pNFS like filesystem
- Create files and direcotories on it
- Write data to the files created.
- Read the files created.
- Make sure that we achieve load-balancing/scalability by striping large files and distributing smaller files across multiple dataservers.

6.1 Future Work :

Although the current implementation does the basic stuff it still lacks the following :

- Some of the basic systems calls like renaming of directories
- Soft links and hard links
- Caching of the pages (we could use the FUSE features for this)
- Extended attributes
- Dynamic addtion of new data servers
- Today the stripe/extent size is fixed at 16K. For real world it would make more sense to allow dynamic tuning of stripe size based on the user need.
- Today the layout file on the metadata server is a simple human readable file. This could be inefficient in searhing and updating it dynamially. A more efficient mechanism would be needed to support large sized files.

6.2 Work Break Up :

6.2.1 Work done by RaviKumar Manjunath :

- **Implemented the client side of the code (pnfs_client.c and related code).**
- **Carried out all the testing and related documentation**

6.2.2 Work done by Sanjeev Bagewadi :

- **Implemented the metadata server (mds_server.c)**
- **Implemented parts of data server**
- **Implemented the parallel read/write operations**

6.2.3 Work done jointly :

- **Definition of the mds_ds.x.**
- **Working out the test plan**
- **Initial base NFS implementation**
  - **We initially got a plain NFS implementation working. And then enhanced it to make it parallel/distributed.**

7. References

- The pNFS project documents at : http://www.pnfs.com
- The pNFS RFC 4.1 draft at : http://tools.ietf.org/html/draft-ietf-nfsv4-minorversion1-29
- Google FileSystem paper at : http://labs.google.com/papers/gfs-sosp2003.pdf
- PVFS paper at : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.4297&rep=rep1&type=pdf
- A paper on Lustre Filesystem : http://www.kernel.org/doc/ols/2003/ols2003-pages-380-386.pdf
- Wikipedia for information of various related filesystems : http://www.wikipedia.org
- pNFS implementation on OpenSolaris http://hub.opensolaris.org/bin/view/Project+nfsv41/WebHome
- Some reference documents from FUSE at : http://fuse.sourceforge.net/