Final Report
Distributed Network Filesystem

Yuanrui Zhang (yz545) Hao Jiang (hj110) Zian Li (zl180) Kai He (kh385)

Introduction

Distributed Network Filesystem (DNFS) is a client/server-based application that allows multiple users to access files and share storage resources via a computer network. Clients would access files on remote hosts in exactly the same way as a user would access any local files. Data accessed by all users can be kept on central hosts, with clients mounting this directory at boot time.

A Distributed Network Filesystem has the following advantages over local filesystem:

- **Availability**: With multiple servers deployed, system will not crash if one of the servers is down.
- **Disaster Recovery**: Servers can be setup in different locations across the world, which minimizes the loss after the disaster occurs and enables data recovery.
- **Network-accessibility**: Allows easy sharing of data between servers and clients via network.
- **Security**: Provides centralized administration where each clients' privilege is limited and file access handled by file permission bits.
- **Scalability**: Can scale on demand up to petabytes without disrupting applications.
- **Less repetition:** All users can share the same environment and no need to install the same software on many different machines.

Distributed Network Filesystem is a common solution to various types of business companies that have the need of storing data. In the era of big data, business enterprises strive on the ability of storing, processing and serving a large amount of data. As business processes are expanding rapidly, there is a surge in demand for data centers providing DNFS services.

Amazon Elastic File System, for example, is a scalable NFS file system for use with AWS Cloud services and on-premises resources. It supports a broad spectrum of use cases from home directories to business-critical applications including big data analytics, web serving and content management, application development and testing, media and entertainment workflows, database backups, and container storage.

2 Design

Our basic distributed network filesystem consists of 2 mirrored servers and one or more clients connecting to them. Our Client-Server model is shown in the figure below:

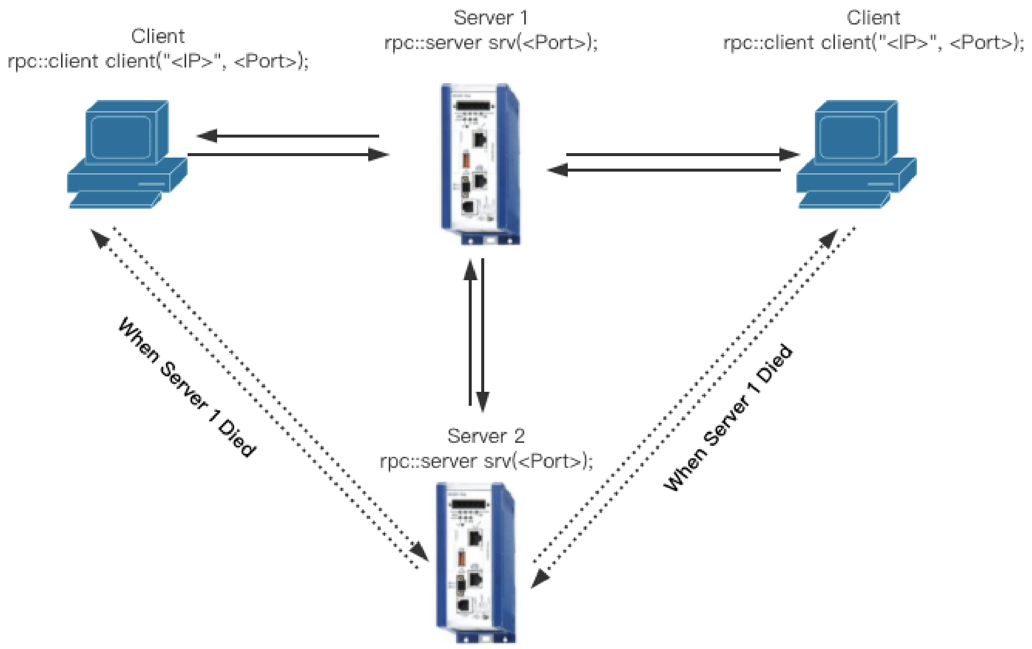


Figure 1: Client-Server model

Our distributed network filesystem is based on FUSE program which catches system calls and RPC protocol which communicates between clients and servers.

Our Design has the following features:

- Transparency: Clients connect to the storage server and access the mounted directory in exactly the same way as a user would access any local files.
- Security: By default, clients can only view other clients' files but cannot modify or delete them. File access control are properly handled with file permission bits. It is also possible to set file permission to allow access, modification and execution by other users.
- Redundancy: master server forward client requests to secondary server to maintain identical states between the two servers. The secondary server will server as a replication to the master server.
- Failure handling: When master server went offline, clients will automatically switch to secondary server and the system enters degraded mode. Client will continue to visit the NFS as usual, while secondary server takes over and handle everything and wait for master server to go back online.

Clients will always try to connect to master server at first, if successful it enters normal mode; if not it tries to connect to the secondary server, if successful it enters degraded mode; if that still doesn't work it will exit. The flow diagram is shown below:

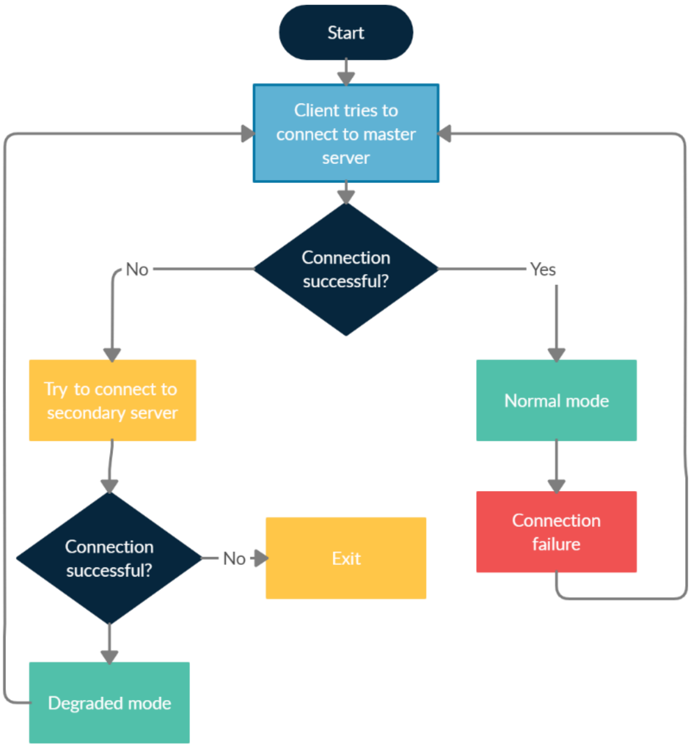


Figure 2: Client connection logic

Server will receive some information concerning whether it is master server and whether the degraded mode flag is enabled, based on that it decides how to react. The flow diagram is shown below:

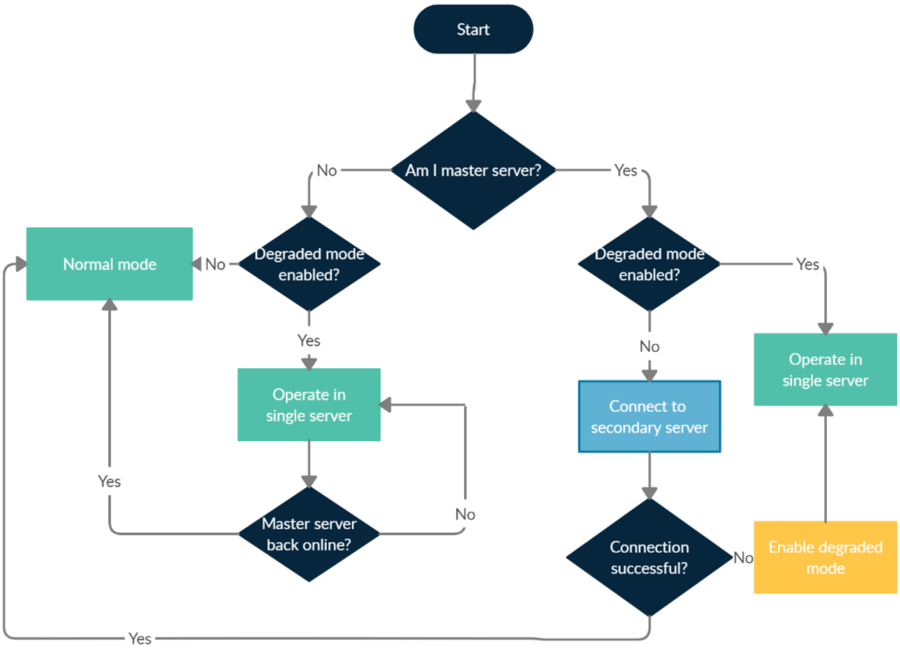


Figure 3: Server connection logic

3 Implementation

Our programing language is C/C++.
The system is generally divided into client side and server side.
Client side is based on FUSE program which catches the system calls made by clients. However, it doesn't handle the requests directly as FUSE used to do, instead the client side packs all requests in RPC and send it to the server. The operations will be carried only after the server responds back with the needed information.
client-to-server request are packed in structs named as function_arg, server-to-client response structs are named as function_ret. Server-to-client response includes a data member ret, it returns 0 if RPC succeeds, otherwise it'd be assigned -errno. For read/write requests, a len member is included, which indicates the actual bytes to read or write. Every *_arg has a data member called ip, which is used to differentiate separate path for distinct clients. To create path at server side, it's represented as an unsigned integer.

Server side is responsible of receiving clients' requests, perform the actual operation to files, and give back a response that contains data needed by clients for visualization. Server side behaves differently based on whether it is running as master server or secondary server, and whether it is in degraded mode.
The master server only forward requests that would update the state of the file system to the secondary server. For example, write, mkdir, rmdir, etc. It first transmits these requests to secondary server, then wait for secondary server to respond success upon completing the operation, and finally perform the operation on master server file system. This chain of waiting is needed to ensure consistency between both servers. Other read-only requests will only be handled on the master server.
Master server goes through the "handle flow" of opendir - readdir - releasedir, or open - read - release, so it could use the fd acquired from previous operations, while the secondary server has to rely on full path.

All requests are handled in the following order:
(1) transmit request to secondary server if needed
(2) check rwx validity, if needed
(3) lock the file or directory, if needed
(4) execute the operation
(5) unlock the locked resource

Locking mechanism:

- add shared lock for file read and directory read
- add exclusive lock for file write
- add exclusive lock for directory mutation, eg: rename, unlink, etc

Despite multiple clients sharing the same space, clients' privacy is protected by file permission bits. The entire file system is stored on server at /DFS. All files are considered visible by all user mounted, but other operations, like updating, removing, reading needs further authentication. Note that root directory of our NFS is designed to be readable/writable/executable by all users, but users cannot delete files or directories directly under root directory because it is owned by root user but not clients.

Currently our NFS does not support user groups with multiple users. We simply treat each user as a user group. By default, client create a file with rw-r--r—permissions, which indicates that the owner can read or write the file but cannot execute it, while other users can only read it. Clients can then change the permission of his own files with chmod command.

Our NFS follow the following rules for access control:

- file open is checked by read and write bit
- directory listing is checked by read bit, open is checked by execute bit
- file creation, rename and deletion need the rwx bits of parent directory, related operations include operation: unlink, mkdir, rmdir, mknod, rename, symlink, link
- chown needs root privilege, so we simply disabled the function for all clients
- only the owner of the file can chmod and utime

Our data recovery is half automatic. Assume that after the server went offline and the problem was fixed by technicians, we have provided a python script for cold restoration. Eg, you could run

```
sudo python rsync.py localadmin@esa08.egr.duke.edu:/9962309 /
```

to recover the data so that both servers are again in identical states.

When the master server goes back online, clients will automatically connect to master server and return to normal mode. However, if the master server is ok, clients will never try to connect to the secondary server and therefore will not be aware if the secondary server were online or not.

4 Evaluation

Our system is fully functional in terms of being a file system, failure handling and user privilege separation. Our criteria of success are the functionality of the features of our NFS. However, if we look into some file system benchmarks and compare it with original FUSE program, we can reveal the weaknesses in our system.

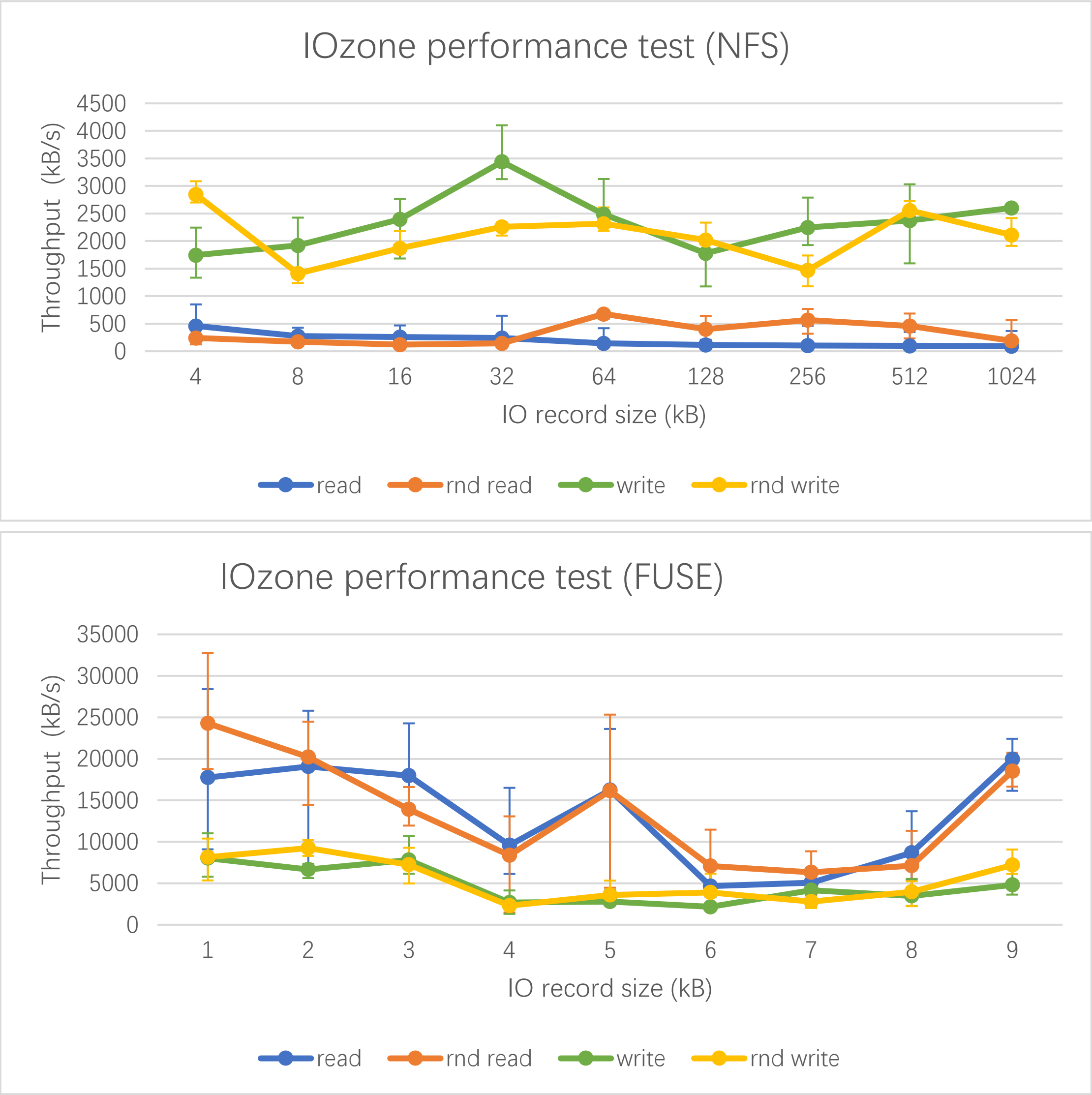
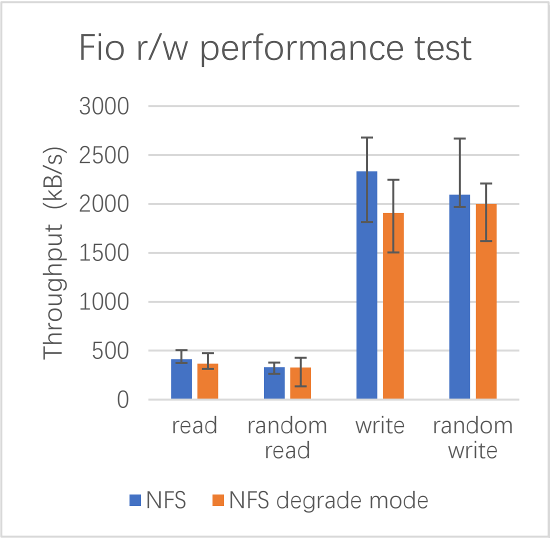


Figure 4: IOzone benchmark performance test

Due to the fact that our NFS does not use any caching techniques, we deliberately tested IOzone benchmark for both our NFS and original FUSE bbfs program without caching. The FUSE program itself suffered from great variances, especially in read operations, so we didn't bother much about the variances of our NFS since there could be issues inside FUSE. Our NFS performed about 30 times slower in reading operations and about 5 times slower in writing operations. This is likely the result of our NFS connecting to the server via RPC upon every system calls and having to open and close a file for each read and write operations.



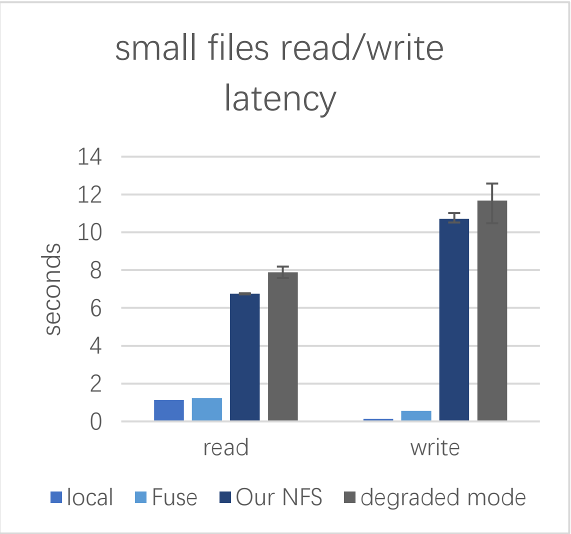


Figure 5: Fio benchmark test

Figure 6: small file r/w latency test

We also used Fio to test the IOPS of our system. To avoid repetition with IOzone results, we only showed the averaged results between NFS and NFS in degraded mode. Our NFS is behaving slightly worse in degraded mode in terms of sequential r/w and random r/w. It is suspected that this result is the tradeoff between less overhead and more retries. On the one hand, some of the overhead was removed when the server enters degraded mode, since it no longer sends requests to another server and wait for its response. On the other hand, the client will try to connect to the master server upon every system call and fail before it connects to the secondary server, which slows down the operation.

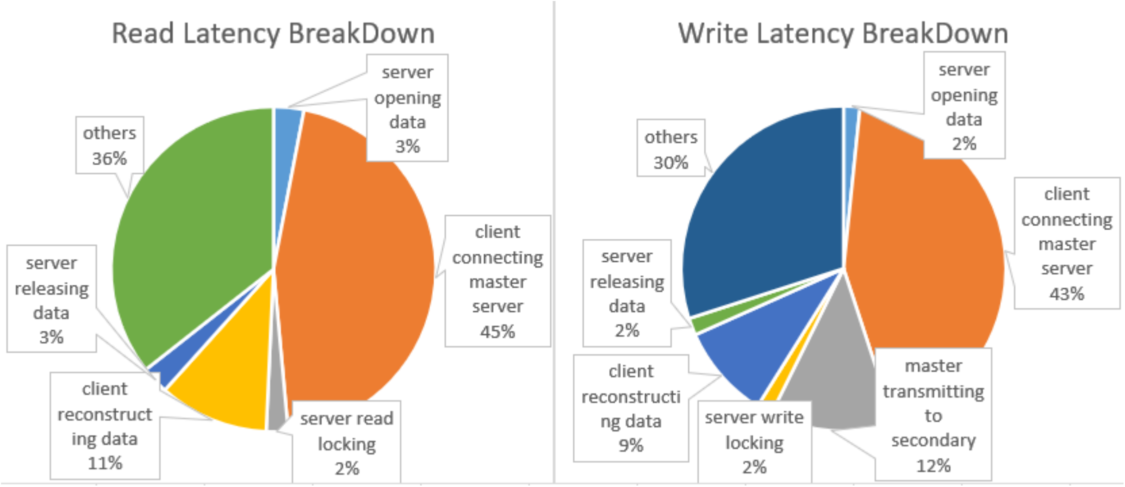


Figure 7: r/w latency breakdown

From Figure 6, the test is timed with Linux's own time command and for looping echo or cat commands. When entering degraded mode, both read and write performance degraded by about 10%. The latency measured by particular chunks using clock() within client and server codes. It is observed that the major cause of latency is client connecting to server via RPC upon every system call. Some commands require 5 or more system calls in series and that would result in client connecting to the server for 5 times, which drastically slows down the operation. If the system were in degraded mode, there would be more time spent in this process because client first tries to connect to master server, and then secondary server upon connection refused or time out.

5 Conclusion

Our NFS successfully achieved the goals of our original proposal. Clients visit the network filesystem as if visiting local filesystems; System does not crash but goes to degrade mode when one of the servers fails; and additionally, we implemented file permission control feature.

Compared to local file system, our NFS has the benefit of availability, recovery, security, and scalability. However, it should be pointed out that our NFS is still very naïve, many fancy features were not implemented, and our NFS has a big overhead which slows down the speed of all operations.

If we were to continue on the project, the following points can be improved:

- File descriptors on the server will not be automatically released after timeout. This could result in resource leakage if a client opens a file and suddenly exit the program. As time grows the file descriptors not freed by clients will accumulate and eventually no more file descriptors are available for new requests.
- Our client code is implemented in a way that it establishes an RPC connection with the server upon every system call. This could be the primary reason of the overhead of our NFS. Instead it should be implemented so that the connection is established at the beginning and maintained throughout all subsequent requests. However, this introduces some difficulty determining when to switch connection to another server. For example, clients need to switch to secondary serve when master server fails; clients also need to switch back to master server when it goes back online.
- There are some small glitches around file permissions for multiple users because we may not have thought about all corner cases. We would like to fix those bugs with a thorough integrated test.
- On the final demo, some problems with concurrent writing was discovered. We should dig deep into this issue and fix the bug.
- If the server not only exited the process, but also quitted the RPC daemon, client and other server will not get active connection refused but have to wait 20 seconds for RPC time out.
- There is a small gap between our cold restoration script running and master server back online. If clients issue any write requests to secondary server during this gap, master serve and secondary server would be inconsistent. We could introduce a locking mechanism where the data recovery script signals the secondary server to lock all files so that any incoming requests would be hang up until master server is back online.
- Given the time, we could implement more features such as caching to improve performance, data compression and decompression to save bandwidth.