A Fault Tolerant Multi-Server FUSE Filesystem

Yang Liu, Ruimin Sun, Risheng Wang
Graduate Student
University of Florida

***Abstract*— Previous Fuse File System only supports one client and one server communication. From a client's point of view, when processing speed of server is limited, the time spent on copying a large file to the server will be very long. Especially, when you just want to modify or append very little things on a big file, retrieving the whole file back will be very expensive. In addition, if the server crashes, requests from the client will not be processed. All of these will result into extremely bad customer experience. In this paper, we propose a chunk based fault-tolerant Fuse file system with multiple servers. Each large file is divided into a number of fixed size chunks, with each chunk keeping a key and metadata maintaining both the information of the original file and the specific metadata of the chunk itself. A small portion of write will only affect the related chunks using its unique key. When a server receives a put request, an automatic replicate will be propagated to the next server. In such method, we overcome the bottleneck of transferring large files in long time and ensure fault tolerance by chunk duplication. Tests have shown that this method receives excellent result.**

***Index Terms*—Fuse file system; bottleneck; chunk; fault tolerance; multi-server.**

I. Introduction

Filesystem in Userspace (FUSE) is an operating system mechanism for Unix-like computer operating systems [1]. It enables non-privileged users to create their own file systems without editing kernel code, which makes it more and more popular these days. The FUSE module provides a "bridge" to the actual kernel interfaces, so that users could customarily design the file system code in user space. The following figure Fig.1 shows the path of a filesystem call in a hello world example.

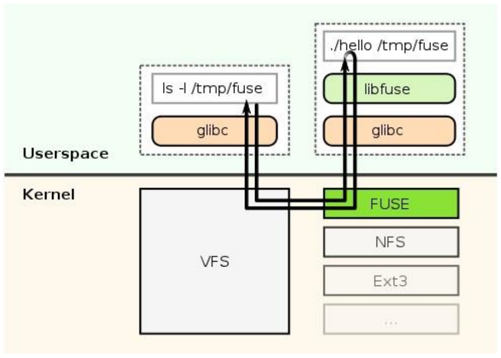

Fig. 1. the FUSE filesystem architecture

As the figure shows, The FUSE kernel module and the FUSE library communicate via a special file descriptor which is obtained by opening /dev/fuse. This file can be opened multiple times, and the obtained file descriptor is passed to the mount syscall, to match up the descriptor with the mounted filesystem.

However, there exists a very serious performance problem in the FUSE filesystem above, which is 4096 bytes writing limitation. This fixed size is designed base on the block size the kernel reading and writing and the limit only worked on architectures where the page size is less than or equal to this. When we combine FUSE with RPC and transmitting large files the problem emerged and is magnified [2]. The principle of RPC is illustrated as follows. An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters [3]. The remote server sends a response to the client, and the application continues its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server, such as an XHTTP call.

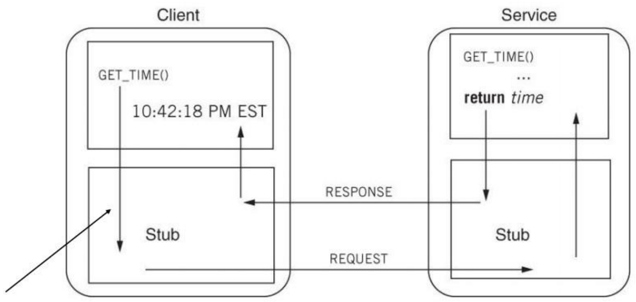

Fig. 2. The architecture of RPC

Then comes the FUSERPC we implemented in previous work, a new file system that provides a solution for the remote file system. The FUSERPC is the combination of fuse file system and the remote procedure call, which merges the advantage of both the two systems. The users are able to use the storage capacity in the remote sites from the service provider through the Internet. This enforces the modularity by clearly cutting the responsibility of the clients and servers. The clients deal with the organization of the file structures and the servers only store the data as clients instructed. In order to make everything works, a client needs to mount to the file system so that the FUSE can get called from the kernel when the client performs file operations. FUSE module fulfills the system calls and deals with the file structure, passing the marshaled data to the servers or retrieving data from servers.

II. Motivation

The above background provides an excellent platform for different users to implement their own systems. Our motivation is to overcome the bottleneck of slow transmission speed and ensure fault tolerance when server crash happens based on the architecture mentioned.

A. Overcome Bottleneck

The fuserpc enforces modularity by dividing different functionality to the servers and clients. However, this division also introduces the new communication overhead between the different modules, be more specific the overhead of network communication between clients and servers. When storing and retrieving files, all the data needs to traverse the communication links. So the users can observe sensible delay if the communication link is not used efficiently.

We took several experiments upon the file transfer under the localhost scenario and observed an interesting phenomenon. The time spent for a file transfer (copying a file to the mounted file system) is not linear to its size. In our experiment, a file of 300 KB needs to spend 5s to transfer, while for a file of 600 KB, the transfer time is about 15s rather than 10s. If the file size is increased to 900KB, the time spent on the transfer will grow further to about 29s. The transfer rate is quite low compares to the speed of localhost, which can be as high as 27.4 Gbits/sec (tested by iperf tool). What's more, the non-linear pattern of the time to the file size is not reasonable. Putting all these together we believe that there may exist some inefficient use of the communication link between the clients and servers.

We did a further experiment to test the amount of data flow across the communication link to confirm our guess. First we record the packets sent and received by the localhost are recorded. Then we start copying a file whose size is 1MB to the mounted file system. After finishing copying the file, we record the packets sent and received by the localhost again and compare with the number before the copy. Astonishingly, we found that the data followed across the link has increased to 198MB. Apparently, quite a lot of unnecessary I/O exits for the data transfer.

B. Fault Tolerance

Fault tolerance is an essential part to a reliable remote file system. When a remote file processing request is traversing network, it is not surprising that the message might get lost or delayed due to an unstable network environment, or worse that it suffers a bit flipping because of noise impact. Nevertheless, as multiple-server structure is implemented, it is of certain possibility that server would fail and become inaccessible.

Mechanisms dealing with faults, therefore, should be developed to make these inevitable faults tolerable. Generally, it should be ideal to somehow eliminate all errors that may lead to faults and build an absolutely stable system, totally getting rid of negative influence brought about by faults. However, it is impossible to swap out every noise existing in the network and no machine is able to operate perfectly without any failures. A more practical way is to construct a fault tolerant module so as to increase the opportunity that the system would still be operating properly when faced with faults.

The most direct and effective solution to fault tolerance is redundancy. Taking advantage of multiple-server structure, files could be copied into different servers so that when the original piece becomes unavailable, the file system can fetch a backup file from another healthy server. Although it takes performance and memory space as trade-off, this simple mechanism adds little complexity to the system and does not bring in new faults. And because our goal is a fault tolerance remote file system but not a fault-free remote file system, it is not worthwhile to build a complex fault tolerance module in order of higher reliability, while in fact a complicated module may not necessarily lead to better performance because of its complexity.

III. System Architecture

The whole system architecture contains two parts: one is the construction of the client-server communication and the other is the fault tolerant design. Here we learn from previous famous and successful examples of Google File System (GFS) [4] and Amazon's highly available key-value store Dynamo [5]. From GFS, we draw on their architecture of storing and fetching data style in a client-server mode, and unified their master and chunk server functions into one server mode. To realize fault tolerance, we utilized the style of partitioning and replication of keys in Dynamo ring. The following will talk about the architecture in detail.

A. Client-Server Architecture

In GFS design, a cluster consists of two types of node: the Master node and a large number of Chunk servers. Each file is divided into fixed-size chunks and stored by Chunkservers. Each chunk has a unique label from the master node when created, and logical mappings to constituent chunks. Each chunk is also replicated several times throughout the network. The Master server stores all the metadata associated with the chunks, and all this metadata is kept concurrent by the Master server periodically receiving updates from each chunk server ("Heart-beat messages"). Here we design our own client-server architecture with the prototype of GFS, as is showed in figure 3.

The client contains a fuserpc module and a Proxy module. Within the client, the fuserpc realizes a series of functions such as getattr(), mkdir(), rename(), truncate(), write() and so on. Those functions enable the client to do the operations in the command line like ls, mkdir, mv, rm, ln, chmod etc. All the functions in fuserpc are actually achieved through two main system functions we customized in Proxy.

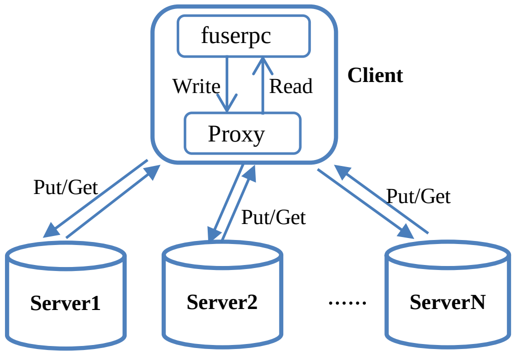

Fig. 3. The architecture of client-server mode

Typically, the Proxy will process those write file requests by dividing the file into fixed size chunks and choose to put the chunks on which server based on the hash function below.

$$S_{No.} = hash\big(pickle.dumps(fn) + str(chk_{No.})\big)\%(\#\ of\ servers) \quad (1)$$

$S_{No.}$ is the id of the server that would receive the put operation, fn is the path (filename), $chk_{No.}$ is the chunk number, $\#\ of\ servers$ is the number of servers and $hash()$ is to get the hash key of a string. With this function, we know not only to which server to put the chunk we want to write but also from which server to get the chunk of a specific file. The chunk size is chosen according to the block size of the FUSE filesystem, here we use 4096 byte as the most adaptive.

B. *Modified Dynamo Ring*

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability. Dynamo ring ensures high availability to satisfy user experience by balancing the tradeoff consider the cost, consistency, durability and performance. Below is the implementation principle of dynamo ring.

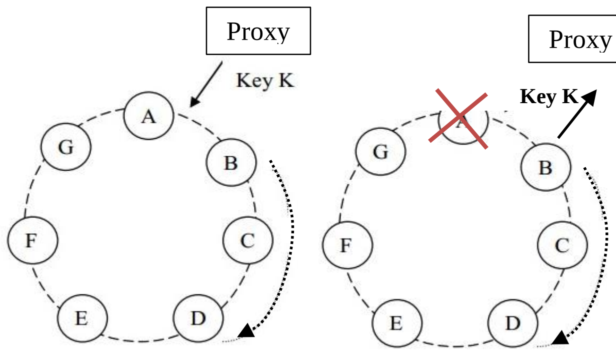

Fig. 4. The Principle of Dynamo Ring of Amazon

In this figure, A, B, C… G are all servers storing the chunks of data and metadata. When each server starts, it not only waits for the client to connect it but also works as a client to connect the rest servers. When the Proxy in the client put a key K (which may be a chunk) in server A, A will get the put request and stores the key. At the same time, A will wake up a thread called ducopy which would envoke putcopy() to put this key on the following several servers clockwise in the ring, in this example, B, C and D. These three servers will work when the primary server A crashes or meet with a temporary failure. Those systems enforcing high availability demand at least three backup servers to store the chunk, we simplify the process by just keeping one backup server. The backup mechanism could be described in the following method.

$$B_{No.} = \begin{cases} C_{No.} + 1 & C_{No.} = 1,2 \cdots N-1 \\ 1 & C_{No.} = N \end{cases} \quad (2)$$

$C_{No.}$ is the id of server that receives the request from the client, $B_{No.}$ is the id of server which performs the putcopy() to back up the chunk data. The detailed implementation will be discussed in the following section.

IV. Implementation

A. *Modularity and Bottleneck*

i. *Enforce modularity by separating the proxy and the memory*

One of the most significant design guideline that we follow is enforcing modularity. Each module only needs to care about its own part. The previous work on the fuserpc design did not pay attention to this guideline. As shown in the architecture, we separate the fuserpc into two modules, the fuse module and the rpc module. The new fuserpc4.py is the fuse modules. Most of its functions remain the same as the previous work. Some function like read(), write(), rename() and etc are modified to tune the performance. The detail implementation will be discussed in the following. The fuse module does not care about the remote accessing or fault tolerance. When it finishes its work, it passes the value to the proxy module. The proxy module on the other side, only responsible for data accessing, it deals with the detailed data marshal and un-marshal and makes use of the fault tolerance mechanism to provide the fuse module a reliable data load and store service.

ii. *detailed bottleneck investigation*

Experiment shows that the whole file is not written into the file system in a single time. A file write consists of several write(self, path, data, offset, fh) calls, each call only pass in a block of data, which is fixed to 4,096KB for the fusepy system. The same situation also happens on the read. The only difference is that the size of read(self, path, size, offset, fh) call is much larger that 4,096KB. For the first read the size is 65,536KB and for the rest reads the size is 131,072KB. This appending writes pattern works well and helps improving the efficiency for the normal Linux file system design on the hard disk, however, for fuserpc this pattern becomes a bottleneck. Because each write, it will first retrieve the file then change the metadata and append 4KB data and finally store the file back. When storing a 1MB file, it will traverse the communication link 512 times, most of which is unnecessary.

iii. Optimization on the read and write

When writing the file, the metadata should be changed accordingly. However, there’s almost no need for us to read the data back. We need find a way, which is able to store the data modify the metadata but do not fetch the previously stored data in order to sharpen the performance. The solution we used is to separate the metadata and strip the data to piece of 4096 KB. So in our implementation what we do is to assign unique a key metadata as well as each data block. So the metadata no longer bounded with the data and can be fetch separately. This more fine-grained data and metadata management helps to better organize the files by minimize the unnecessary communication I/Os. For each write, the newly added block is assigned a new key and then store to the remote hash table. The previous stored data will no longer be retrieved, only the metadata is fetched and modified.

File read on the other side also needs to be adjusted correspondingly. Before fetching the data, we need to have the detailed information of the file. How big the file is and what the block keys are. The information is stored at the metadata of the file. So the first is to fetch the metadata of the file. After reading the metadata, the size of the file is read and can be used to reconstruct the block keys. After retrieving the data by using the block keys, the data of the file is then concatenated together according to the sequence.

iv. Dealing with non-4096 pieces

Although most write and read has an input offset that is integer multiples of 4096, there’re still some exceptions. Especially when we append a string at the end of a file. In this case, the last block piece data may not contain fewer than 4096KB of data. When writing to the end of the file, the write() will first be called to fill the last block and then create new data blocks for the rest of the appending data. Different from writing data to a new block, filling the block require us to identify where the block is and then read the block back and fill it. One of the issues that we need to pay attention to here is that, the offset offered to the write may point to the middle of a block, so it is very important for us to identify this situation before we go on doing more operations. The following pseudo code gives a detailed description of the optimization on the read and write.

v. Pseudo code

```
procedure write(self, path, data, offset, fh):
    remain = offset % 4096
    fetch the metadata
    if remain is 0  // new block
        assign a new key to the block
        store the block back
        change the size in the metadata
    else
        locate the offset block
        fetch the offset block
        append new data to the block
    store the modified metadata
    return length of data

procedure read(read(self, path, size, offset, fh):
    value = “”
    remain = offset % 4096
    if remain is not 0
        locate the block
        fetch the block
        append corresponding part of block to value
        size = size - remain
    while size >= 4096
        locate the block
        fetch the block
        append fetched data to value
        size = size – 4096
    if size > 0
        locate the block
        fetch the block
        append corresponding part of block to value
    return value
```

B. Fault Tolerance

As mentioned above in the architecture part, data transferred to the server are duplicated into its neighbor server in round-robin pattern. Because we are expecting the entire fault tolerance module to be transparent to the caller of file system, data replication is handled by server instead of client, and for the same reason, we takes advantage of multithread programing method to design this implementation.

In the program, we alter simpleht.py by adding a putcopy() function and one thread class to realize the mechanism. Its logic is shown as the following pseudo-code:

```
Procedure put(key, value, ttl):
   keydata ← Binary_to_data(key)
   valuedata ← Binary_to_data(value)
   file_exist_time ← ttl + currentTime()
   Lock
   Store_data_toHashtable(keydata, (valuedata, file_exist_time))
   Unlock
   Begin_dupThread(key, value, ttl, NS_addr)
return

Procedure putcopy(key, value, file_exist_time):
   keydata ← Binary_to_data(key)
   valuedata ← Binary_to_data(value)
   Lock
Store_data_toHashtable(keydata, (valuedata, file_exist_time))
```

```
  Unlock
return

ThreadClass dupThread(ID, Name, key ,value, file_exist_time,
NS_addr):
Procedure duput(key, value, file_exist_time, NS_addr):
Xmlrpc_connect(NS_addr)
remote_call_putcopy(key, value, file_exist_time)
```

The procedure put() is the original stroring function in simpleht.py. To involve data replication in the server, a thread invoker is added to this function so that every time data is being pushed into server, it will awake a new thread called dupThread to transmit the data to its neighbor server. Note that it is necessary to create another function to deal with data replicating, because if the server operates replica storing in put(), it would futher invoke a thread to call put() on its neighbor server, and this remote process invoking will transferring along servers round and round, leading to a deadloop and never returning to caller. That's why a putcopy() function takes place to handle replica storing instead of put(). An awaken thread will then notify a function called duput() with key, value, and existing period of data and the network address of the neighbor server to invoke the remote process putcopy(), where the ip address is obtained when establishing the server from command line, and putcopy() would take the last step to store the backup data, accomplishing the entire data replication mechanism.

The figure below illustrates the progress of server-handled data replication more comprehensibly. Note that it is feasible to implement a serial pattern of replica transferring by invoking remote putcopy() function inside put() directly. However, thread-based parallelized replica transmission is more reasonable since it will not influence the performance of file I/O module.

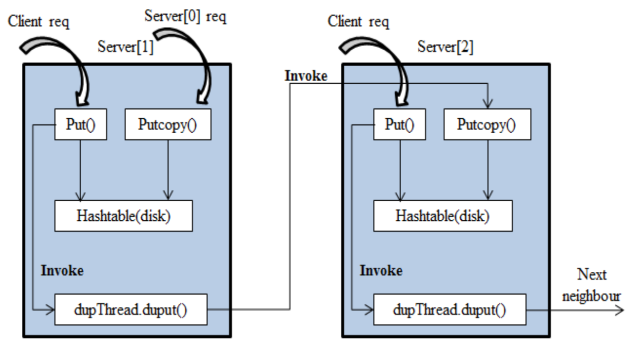

Fig. 6. Progress of data replicating

In the caller side, we input server ip addresses in the same order with that of server neighbors when mounting the file system, so that the system is able to locate backup servers correctly. In detail, the local file system will initially try to fetch file chunks from original server. If failure occurs, it turns to backup server to obtain the target chunk by adding one to the pointer of server address or returning to the first ip when it tried to connect the last server.

However, one backup does not provide satisfying performance. Assume that with probability P(F) the file system may fail to fetch a data chunk from one server. Combined with one piece of backup, the probability of failure of data retrieving is P(F)^2. This number is in fact insufficient to establish a reliable server because a file could be of huge size with hundreds of chunks. Every time when we are fetching a huge file we must pulls back every chunk data, and that produces really low success rate. One solution is to trade space off by creating more replicas to improve the reliability. Our choice is spending delay time to achieve higher rate of fault tolerance, retrying to fetch data when receiving an error for limited rounds.

Retrying works more efficiently. Consider that we retry maxRound times. And for producing more replicas, we set the number as N. If a file is of R megabytes size and each chunk is as large as C kilobytes, the rate that the file system could fetch the entire file successfully will respectively be:

$$R(origin) = (1 - P(F)^2)^{\frac{1024R}{C}} \quad (3)$$

$$R(retry) = (1 - P(F)^{2maxRound})^{\frac{1024R}{C}} \quad (4)$$

$$R(replica) = (1 - P(F)^N)^{\frac{1024R}{C}} \quad (5)$$

Obviously, retrying method upgrades the rate most significantly in terms that one more time of retrying equals two more replicas, not to mention that increased number of replicas will lead to increased number of servers and data transferring.

In pseudo-code, this mechanism is implemented as:

```
Procedure get(key):
While(currentRound < maxRound) do:
   Try: xmlrpc.connect(originalServer)
      Remote_call_get(key)
         return
   If fail:
      Try: xmlrpc.connect(backupServer)
            Remote_call_get(key)
            return
      If fail: currentRound++
```

If after maxRound times of refetching the failure still occurs, we mark it as untolerated failure and raise an error.

V. Evaluation

In this section, we will talk about how we carry out the experiments and what arguments we are testing in detail. In general, we evaluate the whole system based on three fundamental indicators including the reading/writing speed, the availability of the servers and the overall performance improvement we achieved.

A. *Reading/Writing Speed*

i. *Nonlinear property elimination*

First we did experiment on measuring the time elapsed before and after our optimization on the reading and writing byte limits. Below is a figure showing this relationship when we copy a file from local host to the FUSE mounted directory.

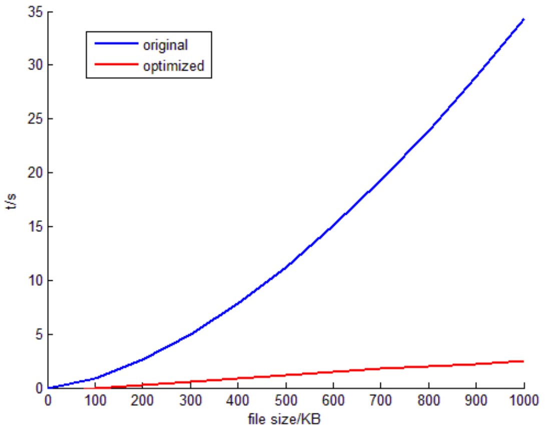

Fig. 7. Comparison between time cost before and after optimization

In this picture, the blue line is the time cost of the original FUSERPC system and the red line is the time cost after optimized. We can easily see that before the optimization the time cost on writing the file to the mounted directory is not linear to the size of the file. This is due to some inefficient use of the communication link between the clients and servers. After the optimization, the red line has showed a very close to linear shape and this is the best evidence that we had eliminate the inefficient part to the minimum and pushed the writing and reading mode to the optimal.

We further measured some more data and created the table below which shows the exact time before and after the optimization. In the improvement column, the time reduction rate in percentage strongly highlight the achievement we obtain.

TABLE I
TIME ELAPSED BEFORE AND AFTER OPTIMIZATION

File Size (KB)	Time before optimization (ms)	Time after optimization (ms)	Improvemen (%)
300	4921	605	87.8
600	15052	1474	90.2
900	28949	2248	92.2
1200	46043	2927	93.6
1500	65990	3510	94.7
1800	88552	3998	95.5
2100	113549	4391	96.1

ii. Monte Carlo Simulation

In order to make the experiment more accurate and meaningful, we implemented Monte Carlo Methods to simulate the real situations of accessing a file system. Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results; i.e., by running simulations many times over in order to calculate those same probabilities heuristically just like actually playing and recording your results in a real casino situation. They are most suited to be applied when it is impossible to obtain a closed-form expression or infeasible to apply a deterministic algorithm [8].

Here we use fixed-size files and run Monte Carlo Methods to generate samples from a probability distribution as the following figure demonstrates.

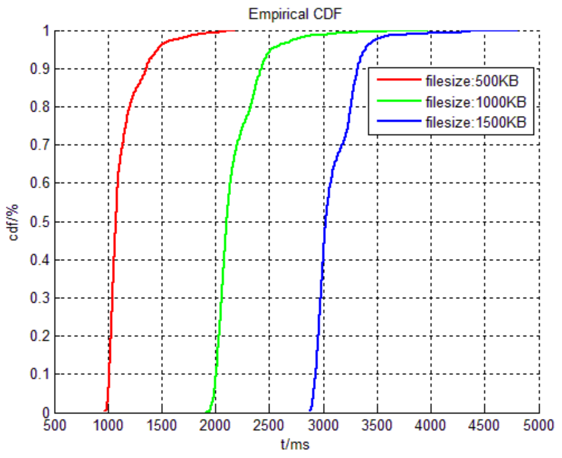

Fig. 8. Monte Carlo Simulation of fixed-size file writing

In this figure, we did a “cp” operation from the local host to the mounted FUSE directory with three different fixed-size files which are 500KB, 1000KB and 1500KB. In the shell, we run each fixed-size file for 1000 iterations randomly and printed out the time elapsed to the output txt file. After calculation of the CDF in MATLAB [9], we got the cumulative distribution of the time cost of writing a file to the server. The red line shows writing a 500KB file, our new system architecture costs about 1300 millisecond with the probability of 0.9. The green line shows a similarity of writing a 1000KB file using 2400 millisecond in 90% of the conditions. The blue line is showing of writing a 1500KB file in 3300 millisecond.

B. Fault tolerance

Referring to formulas in the chapter Implementation, the possibility that file system would operate properly without fault-tolerance module is:

$$R(no\ FT) = (1 - P(F))^{\frac{1024R}{C}} \qquad (6)$$

Note that simpleht.py will randomly raise an error with a rate of 10% when fetching data. Hence for the original design of single-server file system, the probability that we can successfully retrieve a file of 500KB size is 0.00019%, if the file is divided into 4 kilobytes of chunks. This result means that without replica, it is nearly impossible to retrieve a 500KB-size file successfully from a single server. Even though we could establish multiple servers to distribute file chunks into different “disks”, it is helpless to the reliability of the system because a failure of any of these fault-intolerable servers would fail the entire system. Assume we have K servers and assume that files are distributed to these servers in uniform pattern, we can obtain the rate it could fetch a file successfully by:

$$R(multiServer) = \left(\left(1 - P(F)\right)^{\frac{1024R}{CK}}\right)^{K} = \left(1 - P(F)\right)^{\frac{1024R}{C}} \quad (7)$$

Obviously, result is the same with single server.

After implementing fault-tolerance module, single chunk failure rate becomes $P'(F) = P(F)^2$ because each chunk is now recorded in two distinct servers and only when two servers response error at the same time would fetching behavior of caller fail. With naïve fault-tolerance implementation, the caller can obtain a 500KB large file at a rate of 28.47%. This rate will decrease exponentially while file size increases. When we are requiring a 2MB large file, the rate falls to an unacceptable 0.66%.

Improved fault-tolerance combined with retrying mechanism will enhance the reliability significantly. Retrying two times can upgrade the success fetch rate of 500KB file to 98.76%, and 2MB file 95.01%. And if we increase retry time to more than three times, file size will have little influence on the reliability. Even though we are retrieving a 2GB large file, with 4 times of retrying, the fail rate is 99.48%, still above a reliable 99%.

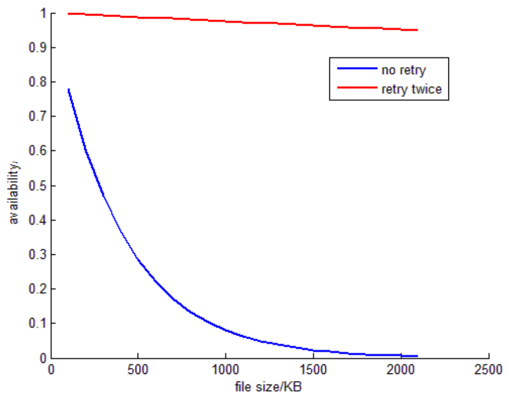

Fig. 9. The comparison of availability between no retry and retry twice

In the figure above, we compared the availability of the servers when there is no retry and retry twice. The blue line has clearly showed that in the original system, the servers could hardly work when the file size is a little larger, for example, consider a 1000KB size file, we could only get accessed with a probability of less than 10 percentage. For customers, this will be thought as a system down. However, after the retry mechanism is implemented, with only twice retries, the availability is apparently higher and very close to 100 percentage.

In figure. 10, we use three sized files to test the relationship between the number of retries and the availability. The three colored lines are very near to each other, but the whole trend is distinct. Within 3 times of retry, the availability will reach 1 for all of the 500KB, 100KB and 1500KB files.

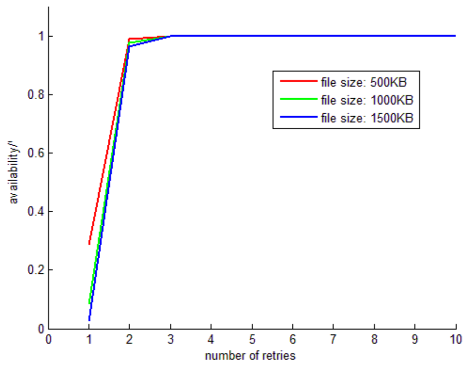

Fig. 10. The relationship between availability and retry times

Nevertheless, data replication is able to deal with server down as it is possible for caller to fetch file data from backup servers. Because backups are stored in a round-robin pattern, the file system can at most tolerate N/2 servers down when N is even or (N-1)/2 servers down when N is odd. The following figure illustrate in what case server down is tolerable.

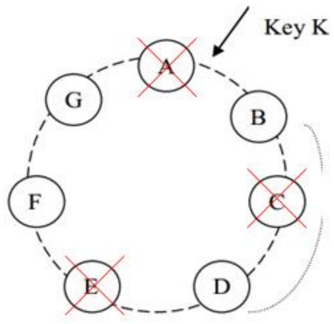

Fig. 11. Tolerable situation

VI. Related Works

In this section, we will talk about several very successful and practical file system that are being widely used. They are Google File System (GFS), Hadoop Distributed File System (HDFS), Network File System (NFS) and some existing problems.

The famous Google File System is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. While sharing many of the same goals as previous distributed file systems, the design has been driven by observations of application workloads and technological environment, both current and anticipated that reflect a marked departure from some earlier file system assumptions. It is widely deployed within Google as the storage platform for the generation and processing of data used by the service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In GFS, each file is divided into fixed-size chunks storing in the chunk servers with a unique label assigned by the master node on the file chunk [4]. The master node involves only in metadata operation so the client interacts directly with chunk servers for most of the time. This design reduces the latency of data transmission to a large degree but brings high failure rate of individual nodes and the subsequent data loss. Algorithms and mechanisms involving fault tolerant are highly needed.

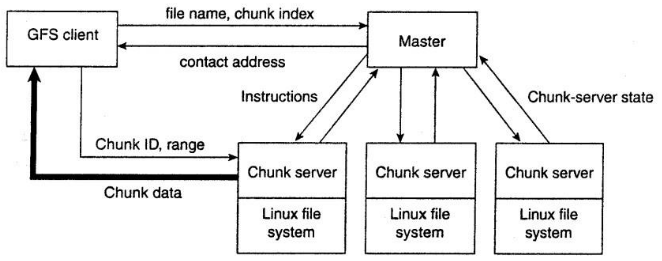

Fig. 12 The working principle of GFS

HDFS is famous for highly fault-tolerant and is designed to be deployed on low-cost hardware. It has master/slave architecture with a single NameNode, a master server and a number of DataNodes in one cluster. The NameNode makes all decisions regarding replication of blocks by periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly [5]. The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size.

Next is the reliability problem at massive scale as one of the biggest challenges faced at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems. The design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience [7]. Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

The Network File System (NFS), designed by Sun Microsystems, Inc. in the 1980s, is a client/service application that provides shared file storage for clients across a network. An NFS client grafts a remote file system onto the client's local file system name space and makes it behave like a local UNIX file system. Multiple clients can mount the same remote file system so that users can share files [10]. The NFS is probably the most prominent network service using RPC. It allows you to access files on remote hosts in exactly the same way you would access local files. A mixture of kernel support and user-space daemons on the client side, along with an NFS server on the server side, makes this possible. This file access is completely transparent to the client and works across a variety of server and host architectures.

NFS offers a number of useful features. Data accessed by all users can be kept on a central host, with clients mounting this directory at boot time. For example, you can keep all user accounts on one host and have all hosts on your network mount /home from that host. Administrative data can be kept on a single host. There is no need to use RPC to install the same stupid file on 20 different machines.

Another very important topic that we emphasized a lot in this paper is the writing and reading number of byte limit challenge. In terms of this area, there are still some people trying to overcome this performance bottleneck by writing C code to make the maximum size configurable. The C file has to be placed under the fuse directory and be called in a specific condition. Normal writes which go through the page cache are still limited to a page sized chunk per request [2]. Not only the modularity is destroyed in this method and potential hazard latent, the main drawback of this implementation is also the low execution speed. The minor improvement of calling C to solve the limitation problem is unacceptable.

VII. Conclusion

In original design, fuserpc file system will establish a connection between local file system and remote file server. It does not apply chunk-based data fetching method, meaning that whatever sizes of part of the file the caller is requiring from server, the file system will fetch the entire file back. What's worse, the original structure of write() funcion in fuserpc leads to a recursive writing pattern. The system is able to handle only 4096 bytes of data updating, and after finishing the former chunk, it will pull the whole stored file back to append the next 4096 bytes of data to it. So, with 4KB large file, file system writes 4KB, and with 8KB large file, the system has to transmit 12KB data to server, not including the flow of reading file back yet. The amount of data the file system will handle when writing will grow exponentially when the size of file increases. Assume we are storing N kilobytes file, the total amount of data the system will have to write is:

$$Total\, write = \frac{4+N}{2} * \frac{N}{4} = \frac{N}{2} + \frac{N^2}{8} \quad (8)$$

This procedure will degrade the performance of file system significantly.

Nevertheless, original fuserpc design does not support fault tolerance. Because file system has to read file back to push new chunk to it recursively, it is of great possible that not only reading is not vulnerable to error, but also writing will raise errors as well in the initial implementation of fuserpc.

Our implementation is aimed at resolving the two issues at the same time. Instead of writing file recursively, we now provide each 4096 bytes chunk of the entire file with a unique key, and separate chunks into different servers. Every time when we want to conduct some operations on some certain chunks, we are only retrieving the target chunks but not the entire file, thus reducing the data flow when reading and writing and decreasing the cost of data transmission. Moreover, to improve the reliability of the file system, we apply a fault tolerance module to simpleht.py, namely the server module, and a data refetching module to our local file system module.

The experiment in the above evaluation chapter has proved that our chunk-based file system implementation upgrades the performance in a great deal. In the original design, it will take 90s or more time to copy a megabytes size of file to the mounted directory. After refining the elapse time decreases to only 3s on average. And the retrying combined fault tolerance module increases the reliability so that when then number of retry is 4 times, the size of files can influence little to the availability of the whole system.

Based on these observations, we conclude that the number of dataflow I/O contributes most to the bottleneck of performance of file system. Even though with our LAN and localhost network, the remote reading and writing still occupies a great deal of time, not to mention transferring with narrower bandwidth in inter-network environment. In order to enhance the performance of entire file system, the most significant issue is to reduce dataflow I/O so that it could functions correctly with least number of I/O. And fault tolerance plays a vital role in maintaining the availability of servers. 10% of error rate seems low, but when file size grows up, this little weak point will become fatal problem. Sometimes we have to make trade-offs to achieve a satisfying application, comparing complexity, performance, reliability, cost and so on. The issue is we should keep in mind what is exactly the goal we are chasing.

VIII. Future Work

We believed that our revised fuserpc are sufficiently complete to flat file system tasks with high performance and high availability. Extensive tests are conducted on the same machine client / server scenario and LAN separate client / server scenario (client and servers are on different virtual machines but on the same host machine). As shown in the previous sections, it provides very high availability and performance. We believe it can be useful in some scenarios.

To be a more productive file system, it should be able to traverse the public network, where we can experience a harsher environment than the simple local area network like larger delays and frame lost. This requires us to push more on the efficiency pursuing. Another problem on the public network is that communication link on the public link is less secured than that of the local area networks. Plain text transmission on the untrusted network may have some potential security problems. It is possible that the data transferred may be intercepted, damaged or falsified by the unwanted third part. Some protection mechanism can be utilized like RSA or DES encryption.

Load balancing can also help to release the server burden. In our implementation we use a simple hash function provided by python to determine the server to perform the data store. Experience from seasoned database engineers and scientists tell us that it is impossible to rely a simple hash function to perform uniformed data distribution. It is inevitable that some of the servers will receive more load than the others. When one or more servers go down, the backup servers are sure to experience sudden burst of data flow. A guided data flow by a loader balancer will help more to alleviate the situation and provide better service to the customers.

To push a more reliable network file load and store, a MD5 check can be applied to check whether the truck fetched is the complete truck and whether the file concatenated together is a complete file. The mechanism can help to deliver a higher level of faults check although a little overhead may be introduced to the system.

To be more compatible service with the modern UNIX file system, we can add more levels in the system. Careful consideration should be taken to the directory structure design. A simple list will obvious be inefficient for complex tree like multi-level architecture and might be the new bottleneck.

Acknowledgment

We would like to thank our lecturer Dr. Figueiredo who gave us clear and detailed descriptions on the platform of operating systems during the course lectures. This unique project he provides for us improved our practical ability on independent research. We would also thank Tony Pierre, the energetic guy, who also helped a lot for the PoCSD course. Thank you.

APPENDIX

i. Bash test usage

This is introduction to the run_me.sh script that is used to test our fuserpc implementation. The test script has several flag that can be used for different test purpose.

-p is used to set the port that the first server will listen to, the following server will listen to the next port in nature order. The default port is set to 2222. In this case the next started server will listen to 2223 and the next will listen to 2224 and so on so forth.

-n is used to indicate how many servers that should be set up. The default number of servers is set to 3. For the fault tolerance design, n is not allow to set to any numbers that is smaller than 2.

-m is used for manual tests. With is flag on, the script will set up the servers according to the -p and -n flag and mounted to the whole file system. Then you can use command line or a gui to access the file system.

-k this flag will kill all the servers listening to the port according to the -n -p flag

-h show help information

ii. example use case

```
./run_me.sh -n 4 -p 3000
```

This will start 4 servers listening to 3000 3001 3002 and 3003. The auto test will create several files ranging from 10K-100K, 100K-1000K. Script will automatically do copy, paste, symbolic links, head, tail and so on.

```
./run_me.sh -n 4 -p 3000 -m
```

This will automatically deploy and mounted the system, but will not do any test. You can use command line or GUI to interact with the file system.

```
./run_me.sh -n 4 -p 3000 -k
```

This will clear the server deployment, but before doing this you have to manually umount the file system.

REFERENCES

[1] Eggert, Paul R., and Douglas Stott Parker Jr. "File Systems in User Space." USENIX Winter. 1993.

[2] *FUSE maximum write size*

[3] Birrell, Andrew D., and Bruce Jay Nelson. "Implementing remote procedure calls." ACM Transactions on Computer Systems (TOCS) 2.1 (1984): 39-59.

[4] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS Operating Systems Review. Vol. 37. No. 5. ACM, 2003.

[5] Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." (2007).

[6] Tanenbaum, Andrew S., and Maarten Van Steen. Distributed systems. Vol. 2. Prentice Hall, 2002.

[7] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." SOSP. Vol. 7. 2007.

[8] Kalos, Malvin H., and Paula A. Whitlock. Monte Carlo methods. John Wiley & Sons, 2008.

[9] Shigley, Joseph Edward, et al. Mechanical engineering design. Vol. 89. New York: McGraw-Hill, 1989.

[10] Hartman, John H., and John K. Ousterhout. The Zebra striped network file system. Vol. 27. No. 5. ACM, 1994.