Principles of Computer Systems Design Project Report

Tao Yu

***Abstract*—In this project, I extended HW4 with advanced features. Here I simulated the the architecture of Google File System which provides a very simple but efficient framework. As the entire GFS architecture if too complicated for this project, I designed and implemented a simplified version in this project and focused mainly on load balancing and security. Even though I did not care too much on features like consistency, my system turned to be reasonable on such aspects. In this project, I introduced fault tolerance strategy of GFS and designed a simple fault tolerance protocol based on it. Because of my tight schedule, I did not implement it in my project and due to the simple mechanism of my strategy, it may not be able to handle complex cases. Then I tested my code. This report is a comprehensive description of my system and what I have covered in this project.**

***Keywords*—load balancing, security, Google File System, consistency**

I. Introduction

DISTRIBUTED file system is one of the most popular things today used by lots of people from various background. Most people will have Dropbox, Google Drive, Sky Drive or some other client apps of distributed file systems installed on their desktop, laptop, cell phones or tablets. A key reason why they are so popular is that it makes our life simpler. Previously if we want to share something with others or use our own resources on different laptops, we need to bring a USB disk or send an email. With distributed file system, we can simple store the files on the cloud disk and log on our account to get the files on other computers to use our resources. Another key feature of distributed files system is security. Our data can easily get lost when the system is down. A good habit is back up the data on a portable hard drive. Today, you just need to store your files on the distributed file system and you don't need to worry about your data any more when your system is down. I think the popularity of apps like Google Drive and Dropbox is the key reason that we use the USB disk less and less.

Distributed file system changes how we work and makes our life much easier. However, design and implementation of a distributed file system is a challenging task. First, distributed file system is supposed to be used by multiple clients rather than one. So consistency is an important issue in distributed file system. Second, DFS is supposed to be fault tolerant. So replicas are usually used. So we must ensure that all the replicas are the same. Another big issue of DFS is that it usually needs to support quite a lot of clients. So performance is important. A good load balancing strategy is important.

In this project, I designed and implemented a simple DFS. My system is extended from HW4 with advanced features. Here I focused on load balancing and security. A DFS with all features is too complicated for this project. I will talk about famous designs of features not covered like fault tolerance.

II. Background

As a distributed file system, there are several aspects must be concerned: consistency, fault tolerance and security. Many famous frameworks like GFS, Amazon Dynamo provide good references. Here, I choose GFS as the main reference of this project. Actually, in this project, what I did is simply try to implement a simplified version of GFS. The architecture of GFS is shown in figure 1[1].

A. Consistency

Ideally, people expect the system follow a strict consistency which means a transaction issued first should be committed first and each transaction is viewed as atomic. This consistency is not possible in that due to network conditions, an early issued transaction can be delayed a long time and the server won't wait. Another relative strict consistency is sequence consistency which ensures that each transaction.

However, today's distributed systems are supposed to deal with millions of clients and petabytes of data a day. According to CAP theorem[2], we cannot expect a good consistence model and good availability at the same time. Therefore, except some financial distributed application which is very sensitive to order, most distributed applications use relaxed consistency model to achieve a higher availability. Distributed file system usually uses a relaxed consistency model as well. Typical distributed file system apps like Google Drive and Dropbox simply store the data on the local disk and synchronize in a certain period. To achieve even higher availability, the synchronization is not atomic as well. For example, GFS only guarantees that file namespace mutations (e.g, file creation) are atomic. I think this is due to GFS's master and chunk server design which will be discussed later in the load balancing.

B. Load balancing

Load balancing is another key issue in a distributed file system. Usually a load balancer is used to distribute requests to different servers. For file system, operations like ls should be supported, so simply distributing the requests makes such operations hard to support. What GFS does is using a single master design. As shown in figure 1, the master is responsible for storing the meat information of files and distributing request to different file. Rather than distributing the file itself. The master return the handles of the chunk server where the

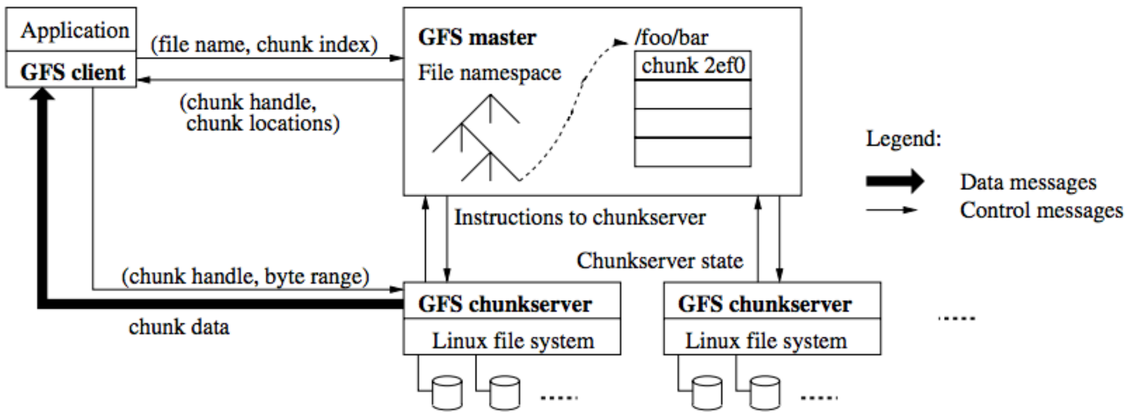


Fig. 1. Architecture of Google File System

file is stored or to be stored to the client. And the clines contact the chunk servers themselves to complete the operation. For simplicity, GFS uses chunk server and the master stores the chunk handles and chunk information. This single master framework makes load balancing easy to implement. For a newly created file, the master can select the chunk server based hash value of its filename which is similar to HW4 or the chunk server with minimum size. Enterprise may uses more complicated strategy which may involves some complex techniques like machine learning. This may be too complicated for this project.

C. Fault Tolerance

A fundamental fault tolerance method is to keep a backup. Every time the server is updated, make a new backup. However, this method is not a good choice in that this strategy unnecessarily backs up all the data including data that does not change every time. A popular way is to keep a basic backup of the server and log operations of the server. A potential problem of this strategy is that the log file may grow very large. A popular approach today is use differential file log. Initially, the server keeps a basic backup and a log file. When the log file grows to a certain size, the server update its back-up to the current state and clear all the log data. Another technique used in logging is write-ahead logging. Write-ahead logging is simply log before the transaction is applied. A possible situation is that the server fails during the transaction. The client may think the server has already committed the transaction. And when the server restarts and tries to recover the data, if the server logs after the transaction, this transaction won't be logged and may get lost. However, if the server adopts write-ahead logging, the transaction will be logged and won't get lost.

A further approach in fault tolerance is replication. In this strategy, replica servers are set up to deal with potential failures of the servers. Usually, the request is multicast to all these replica servers. When the original server is down, the replica server can serve the coming requests. So logging can keep the user data from getting lost. However, when the server is down, new coming request cannot be processed. With replication, requests can be processed by replica servers and the client won't be aware of the failure. Figure 2 shows how the replication strategy of GFS[1].

step 1: The client requests the server of the chunkserver which stores the key and replicas. If no one has the key, the master choose one replica at random.
step 2: The master replies the client the address of prime replica as well as addresses of secondary replicas.
step 3: Then client pushes the data to the all these servers. This can be done in any order.
step 4: Once all the replicas has acknowledged the data, the client sends a write request to the prime replica.
step 5: The prime replica multicasts the requests to all other replicas.
step 6: Upon receive the request, the replica finishes the operation and reply to the prime replica.
step 7: The prime replica replies to the client that the operation has been committed.

The work flow shows in figure 2 is the most basic phase. It is similar to the (to be continued) To support this basic phase, many complicated strategies need to be employed. For example in figure 1, we can see connections between master and chunk servers. This is heartbeat connect between the master and chunkserver. In this strategy, the master must know exactly which chunkservers are working. The GFS mechanism ensures that all the replicas are the same.

D. Security

Security is a growing concern of distributed system. Security involves interception, interruption, modification and fabrication. Providing complete security is very complicated, which is a great challenge to many big network enterprises. Nobody wants his or her data can be easily captured and decoded by other people. The most basic security strategy is encode the data at the sender and decrypt the data at the receiver. Here we cannot use some technique which every body can decode like Base64 in HW4. A widely used technique is SSL. SSL

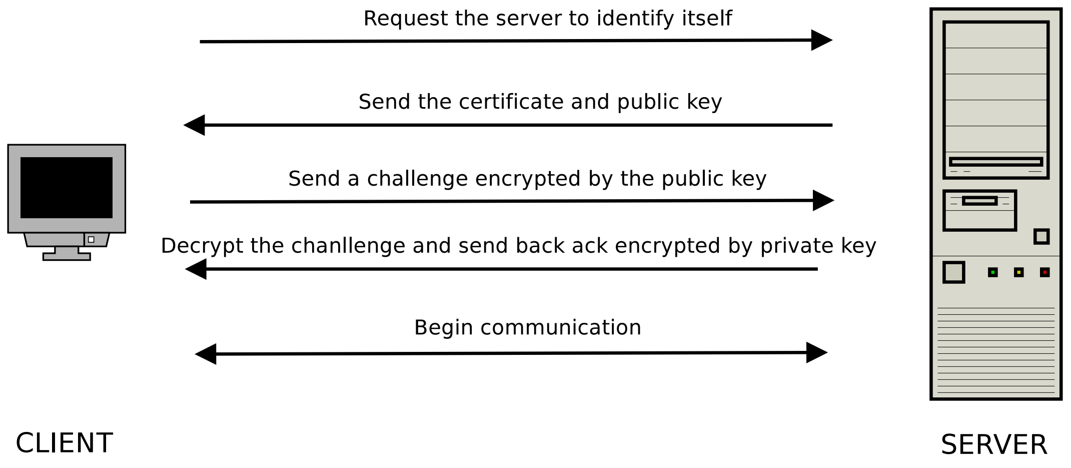

Request the server to identify itself
Send the certificate and public key
Send a challenge encrypted by the public key
Decrypt the chanllenge and send back ack encrypted by private key
Begin communication
CLIENT
SERVER

Fig. 3. How SSL works with private key and public key

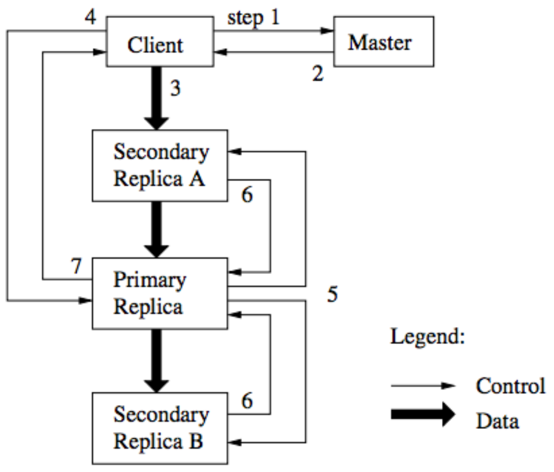


Fig. 2. Write Control and Data Flow

involves a private key and a public key. The mechanism of SSL is shown in figure 3.

III. Design & Implementation

A. Consistency

In HW3 and HW4, operations can be viewed as relatively atomic. Most operations are done by get and put. HW3 and HW4 use SimpleXMLRPCServer which blocks all other request when it is processing one request. In HW3 & 4, a wrapper class of hash table is used. Hash table is a synchronized data structure which allows only one thread to access the data structure. The only not atomic part is that when a new file is created, there are 2 put operations. One put the new create file in the hash table. Another one is append the filename in the hash table value with key '/'. This is not a big issue, These two put operation can simply be integrated into 1 put operation.

In my project, I simulated the GFS to use a single master framework. Generally, each update operation can be viewed as two put operation. 1 put update the meta data and the other update the contents. Therefore, similar to GFS, file namespace mutations are atomic in my system. This consistency is acceptable. So, no further changes are needed. No locks, no threads. Simply based on SimpleXMLRPCServer and hash table wrapper class. When the number of clients grows huge, we man want the SimpleXMLRPCServer to be able to server each request concurrently. This can be done using the ThreadingMixIn. If we change the server to the multithreading one, we may need to lock the data when doing updates. Some consistency models also say that the get operation should be locked as well to avoid certain conflicts. Here I suggests that lock the update is enough. According to the CAP theorem, we cannot expect a strict consistency model as well as good availability. In today's distributed systems, availability generally matters more than consistency.

B. Load balancing

Load balancing can be done through many ways. In HW4, what we do is simply use the hash function and store the file in a random server. Actually, the load balancing performance of this strategy is acceptable. However, there are some unreasonable problems of HW4 as a distributed system. First, the client should not know too much information about the servers. However, in HW4, the client knows exactly how many servers are there and addresses of all these servers. Therefore, when more servers are introduced, the client applications need to be updated as well. This is definitely a reasonable design.

In my project, I use a heuristic approach with help of the master. The mechanism is shown in figure 4. The master keep a record of all servers and size of each server. For each new create file, the master will select the server with minimum size. So the client application just need to know the address of the master. This method is much more reasonable.

In my implementation, I kept a hash table which stores the addresses of the servers and their size. This is only acceptable

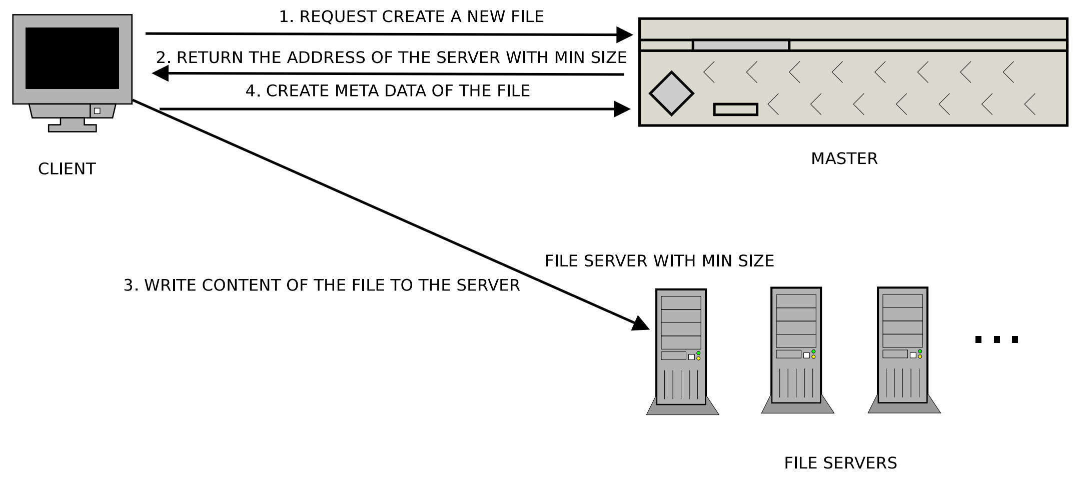

1. REQUEST CREATE A NEW FILE
2. RETURN THE ADDRESS OF THE SERVER WITH MIN SIZE
4. CREATE META DATA OF THE FILE
CLIENT
MASTER
FILE SERVER WITH MIN SIZE
3. WRITE CONTENT OF THE FILE TO THE SERVER
...
FILE SERVERS

Fig. 4. Mechanism of my load balancing approach

with a limited number of servers. For more servers, I suppose to use a min heap or priority queue to store these servers. This would largely reduce the complexity of selecting the server with minimum size. However, this is not good for searching a specific server. So it might be better to store the server information in a binary search tree or some more complicated data structure to reduce complexity. Every time when a server is set up, it will register at the master so that the master knows the existence of the newly set up server.

C. Fault Tolerance

Initially, I want to just use a simplified version with just the first 3 steps. However, this does not ensure that all the replicas are the same. With just the first 3 steps, the write operations to all the replicas may get interleaved with other write operations and the replicas might be different. However, the complete replication strategy of GFS is too complicated for this project, I could not implement a system like GFS. Other replication strategies might be simpler, but they do not fit well with the single master framework. For example, one possible approach would be like delegate the write to replica operations to the master. This strategy can easily make the master overloaded. The core In GFS, all the write operations are done by the client so it is not easy to ensure that all the replicas are the same. So the GFS has 7 steps of write control and data flow. Here, if a prime replica is selected and let all the following write operations done by the prime replica server, fewer steps are needed in the write control and data flow. However, this may make the prime replica servers easily get overloaded. So strategy of the GFS should be our first choice.

According to the GFS 7-step protocol, I suggests that a lazy approach can be used for replication. The master keeps a hash table which stores all the server. Each server has a short lease in the hash table. Each server keeps registering on the master to extend the lease by half of the original lease. The registering period of the lease is same as the lease extension, say half of the original lease.This lazy approach can detects which which chunkserver is alive and which chunkserver is dead. When a failed chunkserver restarts, it will register at the master again. Then the server will reply the restarted chunkserver with the address of an alive replica. Then the restarted server will ask the replica for its state and the working replica will send its state to the restarted worker. Then for each put(), follows the GFS 7-step protocol. For each get(), the master get the list of addresses and check one by one and return the address of the first-found alive chunkserver to the client. The client then contacts the chunkserver for data.

As time is limited, I did not cover replication in my project. What I did is simply use the logging strategy to logs the server's operations. When there are K operations in the log, the server updates the backup to the latest version and clear the previous log. When the server gets down, corresponding read or write would fail. When the server restarts, the server first loads the data from the backup and commit log one by one. Finally the chunkserver will recover to the state when it failed and continue working. In this project, we assume that the master is always working so that it will never fail.

D. Security

In this project, the user data must be protected from being easily captured and decrypted by others. In HW4, base64 is used to encrypt and decrypt the message, which actually provides no protection of the user data. So in this project, I use SSL to protect the data. SSL is short for Secure Sockets Layer which are cryptographic protocols designed to provide communication security over the Internet. SSL has a key pair: a private key and a public key. First, a private key is generated. Then a public key is generated according to the private key. The commands below show how to generate a 2KB key and a self-signed certificate using openssl. The second command will require you to fill in some information. A formal certificate needs to be authorized by certificate authority. For study purpose, a self-signed certificate is enough. For business purpose, a formal certificate is needed in that many software like browsers just recognize formal certificate.

```
openssl genrsa -des3 -out privkey.pem 2048
```

openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095

In my project, I generate the key and certificate using openssl. It is very easy to generate them using openssl which is installed in most unix systems. With OpenSSL, the key contains the public key information as well, so a public key doesn't need to be generated separately.I generated RSA key in this project. RSA key is based on large prime factors which is hard to decrypt. And then for simplicity, I use a self-signed certificate of the RSA key.

To integrate all these with the original SimpleHT, I need to change the SimpleXMLRPCServer to the SecureXMLRPCServer. I need to build the server upon HTTPS server rather than the HTTP server. HTTPS server is simply HTTP server which runs on SSL. There are some existing work equipping XMLRPC with SSL. SecureXMLRPCServer is basically based on SimpleXMLRPCServer. The main change is wrap the socket of the BaseHTTPServer with the key and certificate. The SecureXMLRPCServer has the has the same feature as SimpleXMLRPCServer that it will block other requests when processing one request. For the client part, we just need to change the url from "http://..." to "https://...".

E. Single Master

Master is the core part of my project. As mentioned above, master is responsible for load balancing and stores meta data of all the files. In the master I kept the put(), get(), read_file(), write_file() of SimpleHT and added getServer(), changeSize() and register(). In the master, we do not store the contents of the file. Instead, in the contents field, we store the address of the of chunckserver where the file is stored.

getServer() is a registered function which simply reply the request by the address of the chunk server with the minimum size. This function is usually called by the server when a new file is to be created.

changeSize() is another registered function called by the client. When some thing is pushed to the server by the client, the client will call this to update the size of each chunkserver at the master.

register() is a registered function called by the chunkserver. Every time a new chunkserver is set up, it will call the register function at the master to notify the master. When this method is called, the master will push the server into the hash table with an initial size 0 (here we assume that the chunk server is empty at the beginning).

F. Chunkserver

Here I just use the same name as the GFS to indicate that they are similar to chunk servers which store contents of the file. In fact, my system does not uses chunk numbers and any other chunk utilities at all. Similar to SimpleHT, each chunkserver stores a hash table. Here we don't need to store the meta information of the file. We just need to store the filename as a key and contents of the file as the value. So I just kept the put() and get(). Of course, this chunk server should be secured by SSL as well in that the client can directly contact the chunkserver. Chunkserver keeps all the functions of SimpleHT and has the same log() and commitLog() function as the master which has been discussed above. Here we assume that the master is always working. When a new chunkserver is set up, it will call the register() at the master to register itself at the master. The master then knows the existence of the chunkserver. In this project, the new chunkserver is assumed to have an initial size 0. In GFS, there are heart beat connections between the chunkservers and master. The heartbeat message may include meta information of the server so that the new chunkservers can join with pre-stored data. This heartbeat connection is mainly for fault tolerance features and is relatively complicated. Here as I did not cover too much about fault tolerance, I did not implement heart beat connection in this project.

G. Client

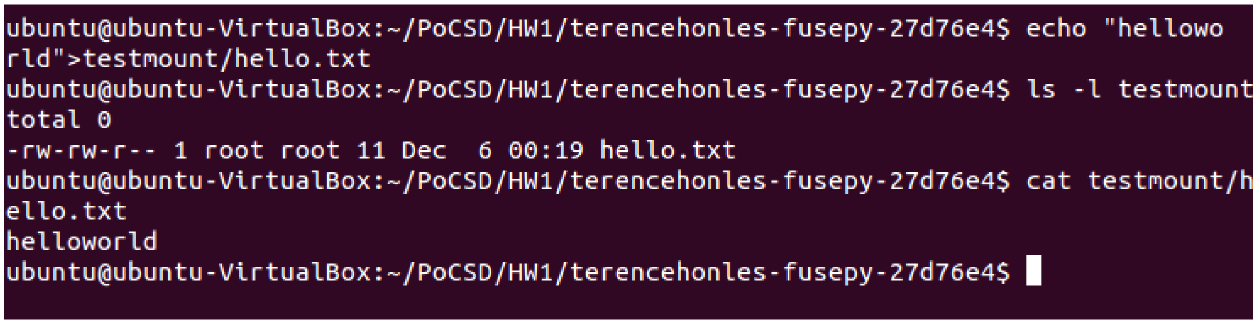
Client side apps need to be changed according to the single master framework. In this project, the client application should support the same operations as the HW3 & 4. Here what I changed is the wrapper class HtProxy so that we can keep the memory class unchanged. In the wrapper class, put() and get() are the main methods I changed.

In put(), the client first tries to get the meta data from the server with the corresponding key. If the key exists in the hash table, then the clients can get the value of that key along with the corresponding address of the chunkserver. Then the client establish a xmlrpc connection with the chunkserver of that address. Then the client pushes the key, contents of the file, and ttl to the chunkserver. Then the changeSize() of the master is called by the client to change size of corresponding chunkserver. Finally the server pushes the meta information of the file along with address of the chunkserver where the file is stored to the master. If the key does not exist in the hash table, call getServer() to get the chunkserver where the file is to be stored. The set up connection with that server. Following operations are the same as the case where the key exists in the hash table. The pseudo code of this function shows as below.

```
**if** $get(key) \neq none$ **then**
    $chunkserver \leftarrow get(key)[''contents'']$
**else**
    $chunkserver \leftarrow getServer()$
**end if**
$rpc(chunkserver).put(val[''contents''])$
$val[''contents''] = chunkserver$
$changeSize(key, chunkserver, val[''st_size''])$
$put(val)$
```

In get(), we first get the value of the corresponding filename from the master. Then we get the address of the chunkserver storing this file now. Then the client sets up xmlrpc connection with the chunkserver and get contents of the file. Then we replace the value in contents field of the value which is actually address of the corresponding server with the real contents of that file and return them to the memory class. The pseudo code of get() shows as below:

```
**if** $get(key) \neq none$ **then**
    $val \leftarrow get(key)$
```



```
ubuntu@ubuntu-VirtualBox:~/PoCSD/HW1/terencehonles-fusepy-27d76e4$ echo "hellowo
rld">testmount/hello.txt
ubuntu@ubuntu-VirtualBox:~/PoCSD/HW1/terencehonles-fusepy-27d76e4$ ls -l testmount
total 0
-rw-rw-r-- 1 root root 11 Dec  6 00:19 hello.txt
ubuntu@ubuntu-VirtualBox:~/PoCSD/HW1/terencehonles-fusepy-27d76e4$ cat testmount/h
ello.txt
helloworld
ubuntu@ubuntu-VirtualBox:~/PoCSD/HW1/terencehonles-fusepy-27d76e4$
```

Fig. 5. capture of basic linux commands

```
        $chunkserver \leftarrow val["contents"]$
        $contents \leftarrow rpc(chunkserver).get(key)$
        $val['contents'] = contents$
    else
        $raiseerror$
    end if
    return $val$
```

With changes in the wrapper class, the memory class can keep unchanged as the interface are kept the same and all the extra work is done in the wrapper functions.

H. New Server Join

As a scalable distributed system, it is very common that new servers join this system. In this case, the servers are chunkservers. In this case, nobody will be willing to change the code to add new servers. So automatic update must be employed by the system.

When a new server comes into the system, it contacts the master directly whose address has been pre-stored in the chunkserver. The chunkserver first checks the data and remove expired entries and then calls the register() of the master and the master will first checks the data and removes the expired entries and then put the server name as the key and its size as the value in the servers hash table. Then the server is available for storage.

I. Server Recovery

Although replication, which is the core part of fault tolerance, is not covered in this project, logging and backup is still used. Normally in this project, it is assumed that servers are always working. When some servers fail, corresponding read and write operations fail as well. The worst case is that when the master fails, the entire system will stops working. Here we assume that the master will never fail. Even so, as far as I am concerned, the user's data should be protected from getting lost. Therefore, I used backup in this project and logs the update operations which has been discussed above. Here when a server fails and then restarts, it first tries to loads the backup. Then the server reads the log one by one. The log() function mainly logs the put function which update the user data. Then the server conducts these operations one by one updating the user data to the state when the server fails

IV. TEST

A. Basic Test

Basic test simply tests the representative Unix commands to make sure that the code is working. Here I test the following commands:

echo ”hello worlds”>testmount/hello.txt
ls -l testmount
cat testmount/hello.txt

Figure 5 shows that these 3 commands work exactly like the local file system. Other linux commands are similar. This system can function properly.

B. Multi-client Test

In this test, what I want to do is open 10 clients. Each client runs a script which keeps inserting a line into a file. These 1- clients writing to the same file called hello.txt. Each client writes 5 lines. Then check the file whether each write is atomic. Here echo command does not append a new line at the end of the file. The linux see command support such operations. An example shows as below:

sed -i ’$a this is line 2 without redirection’ filename

Result shows there are 50 lines one after another in the file. This indicate these write operations are atomic.

C. Load Balancing Test

In load balancing test, I opened one client and kept writing to different files. Here I write a script which kept echoing ’hello world’ to different files for 10 times. Here there are 3 chunkservers, . On all these 3 terminals there were log information which indicated that files are created. This test showed that content so these files are distributed to different servers which showed that the load balancing was at least working. Here 3 chunkservers and 30 files are too small to test the load balancing. So to evaluate the load balancing strategy, I used 8 chunservers and 200 files. Table I shows the sizes of each table after the 200 echo commands. The content of each file is ’hello’

From table I, it is obviously that the data is uniformly distributed to each server. In HW4, files are randomly distributed to each server as. Normally, when there are lots of files

TABLE I. SIZE OF EACH CHUNKSERVER

chunkserver	size
1	150
2	150
3	150
4	150
5	150
6	150
7	150
8	150

created, the files are uniformly distributed to each servers based on number of files rather than size of the server. This load balancing strategy guarantees that contents of a new create file is stored in the server with the smallest size. The distribution of HW4 can only get uniform when there are lots of files. In this test case, the contents of each file is the same for simplicity. However, if the contents of each server is very different, the servers will get unbalanced. For example if there are 9 files and 3 servers. One file is 10 MB and others are 1MB. If so, ideally HW4 will distribute 3 files to a server. The size of each setver will be 12MB, 3MB, 3MB. The balance result of my code would be 10MB, 4MB, 4MB, which is better than the random distribution in HW4. Therefore, my load balancing algorithm is more reliable than random distribution.

D. Logging & Recovery Test

This test mainly tests two aspects: 1) whether the system keeps the backup and logs as expected; 2) whether the system can recover the user data based on backup and logs.

For simplicity, I updates the backup and clear the logs every 5 operations. I tested 7 sed commands to the same file. Then I checked the backup and log file. Here there were just two log operations and the backup was not empty. This indicated that the logging and backup are working properly.

Then I shut down the chunkserver and restarted that chunkserver. Then I checked the contents of the file. Result showed that the content was the as before. This indicated that the recovery of chunkserver is working.

E. Security Test

Security is another major aspect I implemented in this project. It is not easy to test this part as SSL is wrapped in the socket below the SecureXMLRPCServer. Then I found a method. I found the linux command tcpdump which can capture packets on corresponding packets. First I called the put function of the SimpleHT. Here for simplicity I used a string key and string value which requires no serialization and I did not encode them with Base64 like HW4. Figure 6 shows the capture of corresponding packet. Here we can see clearly what function was called and parameter value. It is easy to conjecture that with Binary() in HW4, the only difference is that these parameters would be encoded by Base64 which can be easily decoded. I think experienced hackers can easily identify that the data is encoded by Base64 and decode the content immediately. So using Binary() is actually just the same as what is shown in figure 6, providing no protection of the user data.

Then I tried the put function of my chunkserver. I used the same key and value. Figure 7 shows the capture of the encrypted capture. Here as the data is encrypted by the public key. Generally the data encrypted by the public key only be decrypted by the private key and the private key can hardly be derived from the public key. So even though other servers may have the public key, the cannot decrypt the data. So generally only the sever can decrypt the data. From figure 7, we can see that it is not human-readable at all. There are some string like 'University of Florida' @and 'gmail.com'. These string are information of the certificate that I used when generating the self-signed certificate.

V. CONCLUSION & FURTHER WORK

A. Conclusion

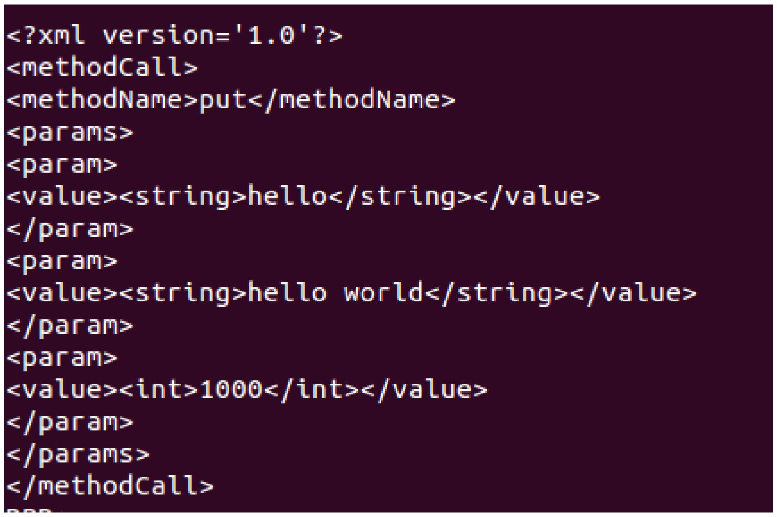
Result shows that this system functions well supporting same linux commands as HW4, can supports multiple clients and the message is secured with SSL. The data is distributed to different servers and the variation of sizes of each server is acceptable. The main advantage of this framework is that the master distributes lots of work to among different clients so that the load of the master is reduced. I all these write work is done by the master, even thought master is supposed to be a very powerful workstation, it still can easily get overloaded. However, as a distributed file system, this is not enough. Further work is needed which is discussed below.

B. Further Work

Further work comes in 2 main directions. First, replication is needed. The GFS 7-step work flow need to be implemented. Besides chunkservers, master may also need replication. It is too complicated to implement all these features. It's google's work. Second, I think the data can be stored in a key-value store rather than a hash table in the memory. With more users and data, hash table in main memory can hardly satisfy our requirements. A simple way is just store the data in a key-value store like MongoDB which provides similar interfaces as the hash table.

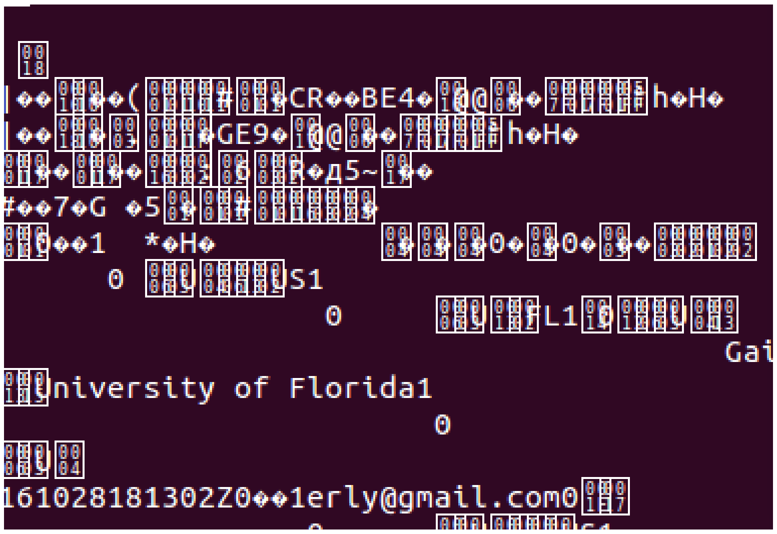
Another further work direction may be concerned with scalability. In GFS, the heatbeat message is much more complicated than my simple register function as well. The heartbeat message in GFS may include information of the server and some information about file stored on that server. Such work can possibly break the assumption that the new coming server is initially empty. This can make our system more scalable. Another thing is that in GFS, replication is not in server level, it is in chunk level. This loop coupling design may result in a higher availability. However, this also makes consistency protocol and recovery more complicated.

User experience is a direction of further work as well. in this project, the write or read operations will directly write remote servers or read from remote servers. This will be



```
<?xml version='1.0'?>
<methodCall>
<methodName>put</methodName>
<params>
<param>
<value><string>hello</string></value>
</param>
<param>
<value><string>hello world</string></value>
</param>
<param>
<value><int>1000</int></value>
</param>
</params>
</methodCall>
```

Fig. 6. Package captured with no encryption



```
��(#CR��BE4�@@��h�H�
��GE9�@@��h�H�
���R�д5~�
#��7�G �5#
��1  *�H�                 �0�0��
    0 US1
                0     FL1
                                        Gai
University of Florida1
                           0

161028181302Z0��1erly@gmail.com0
```

Fig. 7. Package capture with encryption

relatively slow compared to our own disk. The user will feel annoying for the long delay. What Google Drive or Dropbox does is that you write in the main memory or you own disk. The synchronization is running at the background. The synchronization periodically checks the last modified time with the last modified time of the file on the server and updates if necessary. The update follows the consistency protocol and fault tolerance design.

There are other further work directions. For example, this system can be extended to support different users that each user can just see files belongs to them. Another possible concern is that we should not use a pair of keys all the time. As time goes by, the possibility that the key gets decodes grows. Therefore we can use session keys instead to achieve better security.

APPENDIX A
HOW TO RUN THE SYSTEM

- run master.py in the master documents like

  python master.py
- run chunk.py in documents chunk1, chunk2, ... respectively like

  python chunk.py #port_number
- run the client similar as before like

  python client.py #mountpoint
- open another terminal, enter commands to test the system.

REFERENCES

[1] Sanjay Ghemawat , Howard Gobioff , Shun-Tak Leung, The Google file system, Proceedings of the nineteenth ACM symposium on Operating

systems principles, October 19-22, 2003, Bolton Landing, NY, USA [doi¿10.1145/945445.945450]

[2] Nancy Lynch and Seth Gilbert, ?Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services?, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59.