Principles of Computer System Design

Final Project

Xin Du
UFID: 18321309

Chen Yang
UFID: 96749584

1. Background

1.1 Introduction

We have come out the communication between one client and one sever (remote procedure call) in file system. Although using a single server is the simplest way to implement a service, such file system with only one single server is not stable. A variety of events can lead to the failure of a server instance[1]. Often one failure condition leads to another. Loss of power, hardware malfunction, operating system crashes, network partitions, and unexpected application behavior can all contribute to the failure of a server instance.

If the server shuts down suddenly while the client is writing a file, the file being written may end up being only partly updated. After the client requests this file from server again and read it, the client may find that this file begins with the new data, but ends with the previous data. If the server fails during the time that the client is writing a file, the data being written could be affected, even if the disk is perfect physically. When the client requests this file again and read it, the client may get nothing but messy code. The reason is that the contents of volatile memory are inside the error containment boundary and thus at risk of damage when the system fails. As a result, the faltering server may have accidentally corrupted the file data.

In this way, the most popular approach to address this problem is using mechanism of fault tolerance. Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components[2]. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively designed system in which even a small failure can cause total breakdown. A fault-tolerant[3] design enables a system to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails.

Within the scope of an individual system, fault tolerance can be achieved by anticipating exceptional conditions and building the system to cope with them, and, in general, aiming for self-stabilization so that the system converges towards an error-free state. However, if the consequences of a system failure are catastrophic, or the cost of

making it sufficiently reliable is very high, a better solution may be to use some form of duplication. In any case, if the consequence of a system failure is so catastrophic, the system must be able to use reversion to fall back to a safe mode. This is similar to roll-back recovery but can be a human action if humans are present in the loop.

In most cases, there are three kinds of methods to implement duplications of fault tolerance:
Replication[4]: Providing multiple identical instances of the same system or subsystem, directing tasks or requests to all of them in parallel, and choosing the correct result on the basis of a quorum;
Redundancy: Providing multiple identical instances of the same system and switching to one of the remaining instances in case of a failure (failover);
Diversity: Providing multiple different implementations of the same specification, and using them like replicated systems to cope with errors in a specific implementation.

1.2 Our Design

In our system, we choose the first approach - “replication” to design our system: using multiple replica servers for file data. The scheme is like N-modular redundancy, which is called Replicated State Machines.

Replicated State Machines[5] is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. The approach also provides a framework for understanding and designing replication management protocols. For replicated state machines to provide increased fault tolerance, the replicas should fail independently. Here we use the quorum approach to enhance the system. This approach defines read and write quorums Qr and Qw separately, and it sets Qr + Qw > Nreplicas and Qw = Nreplicas.

In our design, we set Qr = 3, 4 or 5, Qw = 5 and Nreplicas =5. It means that our file system has 5 replica servers for file data. When the client requests a “read” command from server, and there are at least 3 replicas agree on the data or witness value, then this “read” command can work successfully. When the client requests a “write” command from server, it should write to all 5 replica servers. Then this “write” command can work successfully.

In this project, we mainly address these four important problems:
(1) The repair mechanism should be done at the server side and transparent to the client.
(2) Data corruption in a single server-where data contents in a server has been be corrupted. We assume that data corruption errors are only tolerated if Qr >= 3.
(3) Server crash, restart-where one or more servers may fail, and a replacement server may start from a blank slate (i.e. without any file data).
(4) The system may block on a write request if there are unavailable servers, but should be able to respond to a read request as long as there are at least Qr servers available.

After solving these problems, our system became a robust, stable and fault-tolerant one client-multiple servers model rather than a fragile, mutable and non-fault-tolerant one client-one server model.

2. Implementation

We will describe our implementation according to those four problems that we mentioned above. The detailed thoughts and practical implementation will be included. We did not revise any codes on client side *(remote_tree.py)* in order to guarantee the transparent property for clients. We just do all the repair mechanism on server sides.
We created 7 servers: 1 mediator server 1 meta-server and 5 data-servers. All these servers are edited based on *simpleht.py*. The whole picture is shown as Fig.1. Client sends requests to the mediator-server which is responsible for retransmitting the requests to meta-server & data-servers and getting data from these servers to return results to clients.

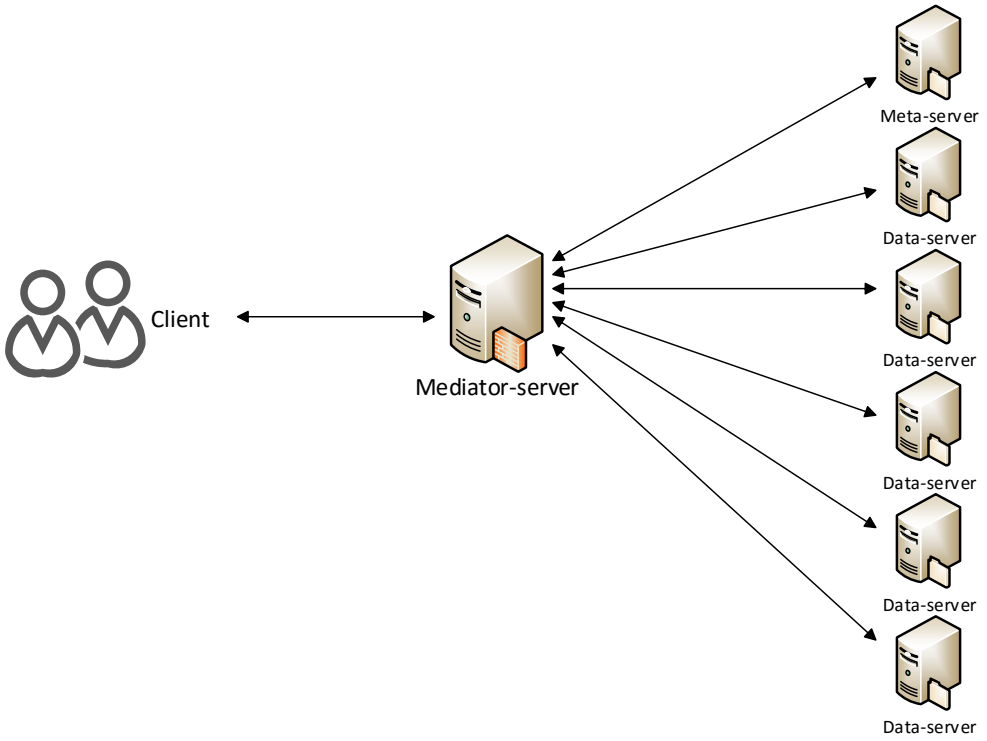

Fig.1 the system consists of 1 client 1 mediator-server, 1 meta-server and 5 data-servers

2.1 One Mediator Server

This is our main task in this project. It is a mediator between the client and all other servers. The client only connects with this mediator server. When the client wants to get data from server, it just asks data from the mediator server. The mediator will get the corresponding data from meta/data server, then pass it to the client. When the client wants to store data in server, it also gives data to the mediator server only. The mediator will put data to the corresponding meta/data server. These two features are achieved by two functions “put” and “get” in the mediator, which we will discuss later.

However, this mediator server is not only about data transfer. Several of the replica data servers may crash and then restart from a blank slate, or the data in data server may be corrupted. All four states and their relationship are shown as Fig.2. This transition diagram also shows our design of fault tolerance. Here, the mediator plays a significant role in masking corrupted errors and restoring data to data-servers. Thus, no matter what happens, the mediator will try its best to help the system maintain the normal state so that the client can get the most corrected data in most of times without fear.

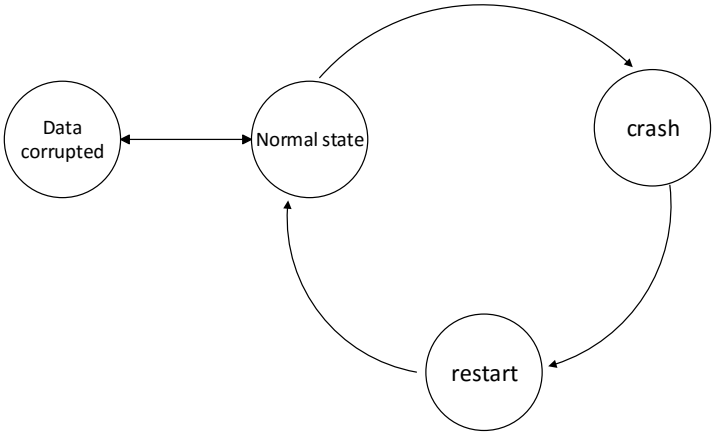

Fig.2 Transition diagram of four states in data-servers

2.1.1 "put" and "get" when Corrupted Data Happens in Data Server

To realize function of fault tolerance, we create a quorum approach in this mediator server. Here we combine the quorum approach and functions "put", "get".

(1) Put:

After client uses "put" function to put key and value to mediator server, mediator server needs to parse keys into two different categories: meta&list_node and data. The reason why we regard meta and list_node as one category is that: these two are both related to attributes of directory and file.

However, data is another kind of category since it is just contents of files. Key and value received by "put" function in mediator server are both instances rather than original string. Because of this change, in mediator server, we use key.data to get original string of key and use .split ("&&") [-1] to judge whether it is meta&list_node or data. If it belongs to meta or list_node, it will be retransmit to rpc0 which is meta server; if it belongs to data, it will be retransmitted to rpc1, rpc2, rpc3, rpc4, rpc5 which are five data servers. The format of retransmission is similar to client behavior. All the keys and values are not stored in mediator server and they are just be retransmitted to other servers.

(2) Get:

In a similar way, the key received by "get" function in mediator server is an instance rather than original string and we use key.data to get original string of key and

use .split(“&&”)[-1] to judge whether it is meta&list_node or data. If it belongs to meta or list_node, we get the corresponding value from meta server using key.data; if it belongs to data, we get the corresponding value from five data servers using key.data.

However, there exists a difference between these two categories. If the category is meta or list_node, after getting returned value from meta server, we return the result directly to client side for the client will use pickle.loads (res["value"].data) to get the true value; If the category is data, there are some extra work to do in order to achieve performance and fault tolerance: after getting returned values from five data servers, we need to use pickle.loads(res["value"].data) to get actual data from each of these data servers so that we can use these actual data to implement quorum approach. In the mediator server, these five returned data need to be compared.

To be specific, we import hashlib and hash these five returned data and then compare them. If the number of same hash results is equal or more than Qr, in other words, at least Qr replicas agree on the data or witness value, then this agreed data will be returned to client side and at the same time the corrupted data in corrupted data-servers will be replaced by the correct data by using put() again, which implemented in the mediator server. It will guarantee the consistency of all data servers.

2.1.2 Put and Get when crash happens in data-server

(1) Put:

The only influence on put when the server crash is that it refuses the request. We did not revise any code of put() in mediator server based on description on 2.1.1 for the requirement of this project is Qw= Nreplica, which means all the data-servers have to be written if the client sends request. The system will automatically throw errors if one or more servers are down. In this way, our policy is: if there exist one or more server s are down, the system will block a write request based on the requirement of this project. We only concern the situation where data-servers crash only after data has been successfully written into data-servers then crash happens.

(2) Get:

Here, we will use “try/except” to catch this connection exception and jump it into “except” to avoid system interruption caused by errors of connection refusing. The goal of this is that we will make sure client still can get data even though some data-servers crash.

The mediator server catches an exception when some of the data servers are down. This happens when we call the data servers’ get() function. In our project, if we find a crash in a server while reading data, we will set the corresponding data with a unique temporary value so that the mediator server can find it.

Ways to detect errors: We will get server copy of data from data servers, no matter whether it is down or not. That means: if a data-server is down, we will catch this

connection error and a unique, temporary value will be set as its returning data. If there are several servers down, their returned data will be set to different temporary values (here, we assume the client does not write these same temporary values that we set to data-servers). Then, we use the hash function to calculate the hash value of each returned value from data-servers (which also include those different temporary values if there are crashed servers). By comparing these data, we can get a statistic on these data. For example, we can get to know the common data (those unique and different temporary values cannot become this common data based on our assumption above) that most data servers have.

Ways to correct errors: Now that we know what are correct data and which are wrong. The next step is to write the correct answer to the servers that have corruption.

2.1.3 Restart when crash happens in data-server

In our project, we provide a way to remotely terminate the server(terminate() in data-servers) and also a way to restart the server(restart0() in data-servers to make self.data={} in data-servers). After some data servers restart, the next thing we need to do is back up data. At the beginning, we wanted to implement restoring part in data-server side, however, we found each data-server is independent and does not know other servers port and we do not provide any information about other data-servers' port to the restarted data-server. So, finally, we implemented the procedure(restart0()) of restoring data for restarted data-servers in mediator server for only the mediator server knows all the data-servers' port and can know which data servers are working correctly which are not according to the result of calling data-servers' print_content() and refill data to restarted data-servers. So, after call restart0() in data-server(make self.data={}) we need to call restart0() in mediator server(make data-server's self.data full) so that data-server can start from a blank slate.

Thus, the procedure is like this: when the control server get a request of restart, the only thing it knows is that some servers have no data (self.data = {}) and they need to retrieve the data from the other servers. Our method to do this is: we call print_content() of all data-servers to see if one or more results are not equal to {}. If there exists result not equal to {}, we iterate all the keys in this result and call get() in mediator server because the get() function in the mediator server can recover the missing or wrong data for all data-servers. We take advantage of get() to help us achieve backing up information for restarted data-servers.

2.2 One Meta Server

Since there is only one meta-server, we created this server based on an assumption that the meta-server will never have any faults or failures. The meta-server contains just a simple hash table which stores directories and files metadata: *meta* and *list_node*.

2.3 Five Data Servers

These 5 data servers are replica servers for file data. Just like meta-server, they are simple hash table which store a copy of each file block.

The five data servers works independently. Only mediator server knows who are working. Data servers cannot see each other. The benefits of this design is the scalability and independent. You can remove or add data server anytime.

The data server provide a “Put” API to the mediator server. Some when the mediator server want to retrieve the data. It will call the data servers’ API. Besides the “Put”, we also add terminate, restart, list content, corruption functions on data server for debugging and testing.

3. Testing Mechanisms

After developing the program, next step is to test the program to find whether the program functions and produced outputs are as desired as we thought. Our main testing task is to test whether the quorum approach works well. Here we add three functions to our replica data-servers. When we need to test the program, we just use test file to call these functions to see if the results are what we want.

(1) def list_contents()
When we call this function, it should return a list of keys corresponding to values stored on the data server. This function helps to check if the test works normally.

(2) def corrupt(key)
When we put the key and call this function, it corrupts the value corresponding to the key in the data-server. Here we use some strings like “This is a corrupted data” instead of the correct data. If we want to corrupt data in several data-servers, we use different strings to avoid the wrong number of same returned data calculated in control-server. Please note, in our test file, we default you have already created “a.txt” and this test file calls data-server’s corrupt (key) with key is “/a.txt&&data” in default.

(3) def terminate()
When we call this function, it should shut down the data server [6]. Here we set a global variable “quit”=0 and a judgment statement in the original data server using a while loop to monitor the change of “quit” value : when function terminate() is not called, which means it is in the normal operating state, “quit” will maintain to be 0, so the server will continue dealing with requests(file_server.handle_request()); when function terminate() is called, it will change this global variable “quit” to “1”, then the execution will break out of the while loop, and print “data_server n is crash”. This procedure then can simulate the crash of server.

(4) def restart0():
When we call this function, it simulates “restarting the corresponding data server”. It wipes all the data in the data server, though it does not shut down and restart the server actually.

Here is one example to test our program. First open all the servers and client, and run our program correctly (see Fig.3). After putting some data into server, we can begin our test. When *test_corrupted.py* calls corrupt(key) and list_contents(), it should print a list of keys corresponding to values stored on the data server (see Fig.4). We should note that the value corresponding to the key in corrupt(key) is “This is a corrupted data”. Then we run *test_crash.py* to call terminat(), which shuts down one of the replica data servers. When we continue to run the code, it will return an error, which is “connection refused” (see Fig.5). In *rpc1.py*’s terminal, it will print that “data_server1 crashes” (see Fig.6). We can also call restart0() to restart one of the replica data servers and make it to the blank slate. In *test_restart.py*’s terminal, it will print the data in server before restarting, data in server after restarting without restoring data, and data in server after restarting with restoring data, respectively (see Fig.7).

```
chenyang@ubuntu:~/fusepy$ cd fusemount
chenyang@ubuntu:~/fusepy/fusemount$ echo "1111">a.txt
chenyang@ubuntu:~/fusepy/fusemount$ cat a.txt
1111
chenyang@ubuntu:~/fusepy/fusemount$ mkdir a
chenyang@ubuntu:~/fusepy/fusemount$ ls
a  a.txt
chenyang@ubuntu:~/fusepy/fusemount$
```

Fig.3 normal state of file system

```
chenyang@ubuntu:~/fusepy$ python clienttest.py
[['/&&data', ["S''\np0\n.", <DateTime '20151205T21:42:54' at 7f474de4a5f0>]], ['
/a&&data', ["S''\np0\n.", <DateTime '20151205T21:43:47' at 7f474de4ab00>]], ['/a
.txt&&data', ["S'1111\\n'\np0\n.", <DateTime '20151205T21:43:33' at 7f474de4abd8
>]]]
the value of a.txt
["S'This is a corrupted data'\np0\n.", <DateTime '20151205T21:43:33' at 7f474de4
ae60>]
[['/&&data', ["S''\np0\n.", <DateTime '20151205T21:42:54' at 7f474d9a6b48>]], ['
/a&&data', ["S''\np0\n.", <DateTime '20151205T21:43:47' at 7f474d9a6bd8>]], ['/a
.txt&&data', ["S'This is a corrupted data'\np0\n.", <DateTime '20151205T21:43:33
' at 7f474d9a6cf8>]]]
```

Fig.4 the original stored data and corrupted data (see red line)

```
  File "/usr/lib/python2.7/httplib.py", line 835, in _send_output
    self.send(msg)
  File "/usr/lib/python2.7/httplib.py", line 797, in send
    self.connect()
  File "/usr/lib/python2.7/httplib.py", line 778, in connect
    self.timeout, self.source_address)
  File "/usr/lib/python2.7/socket.py", line 571, in create_connection
    raise err
socket.error: [Errno 111] Connection refused
chenyang@ubuntu:~/fusepy$
```

Fig.5 terminate() will make server refuse dealing with requests

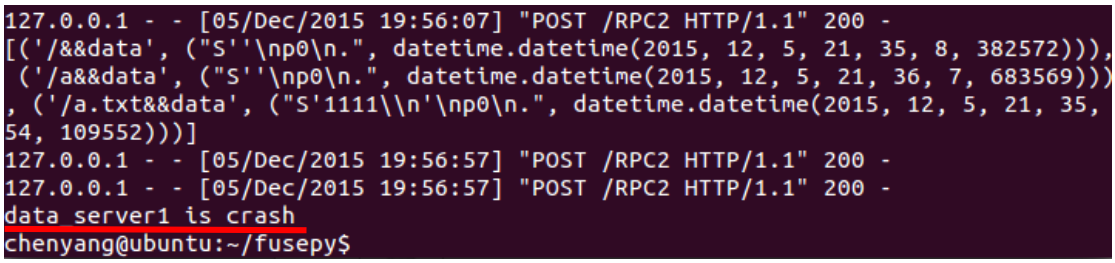

Fig.6 terminate() will make data-server crash

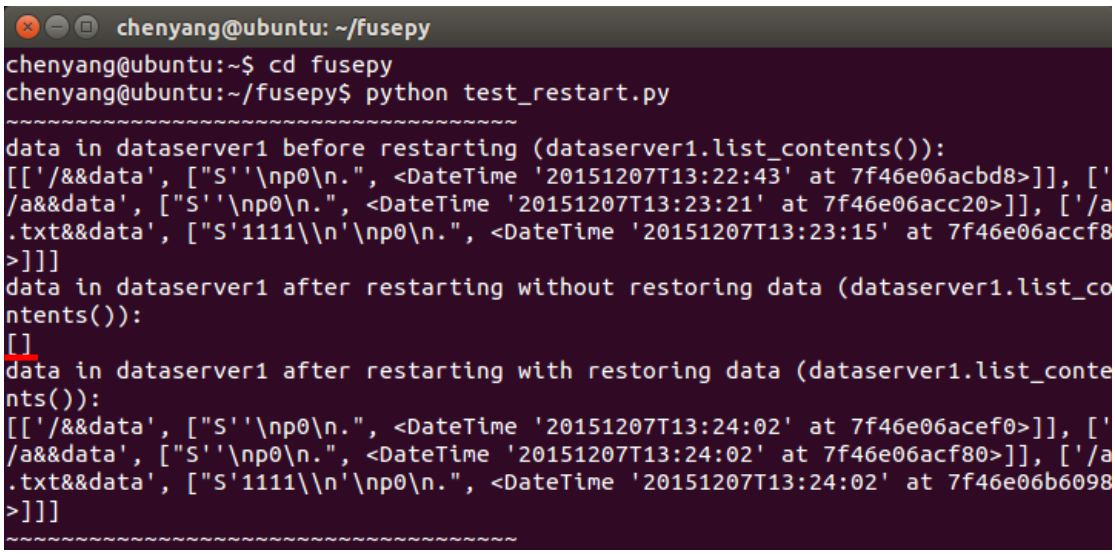

Fig. 7 reatart0() will make data in data-server empty

4. Evaluation Mechanisms

We use two tables, one summarizes all the situations that a file system with one single client and one single server may meet, and another table summarizes all the situations that a file system with one single client and replica data-servers may meet.

When there is single client and one single data-server, shown as Table.1:

$Q_r = 1, Q_w = 1$		
Number of Corruption	0	1
Result of Write	√	√
Result of Read	√	×
Number of Crash	0	1
Result of Write	√	×
Result of Read	√	×

Table.1 result of read and write of one single client and one single data-server

When there is single client and five replica data-servers, we change Qr to 5, 4, 3, 2 and 1. The results of read and write are as Table.2, Table.3, Table.4 Table.5 and Table.6:

$Q_r = 5, Q_w = 5$						
Number of Corruption	0	1	2	3	4	5
Result of Write	√	√	√	√	√	√
Result of Read	√	×	×	×	×	×
Number of Crash	0	1	2	3	4	5
Result of Write	√	×	×	×	×	×
Result of Read	√	×	×	×	×	×

Table.2 result of read and write of single client and five replica data-servers with Qr=5, Qw=5

$Q_r = 4, Q_w = 5$						
Number of Corruption	0	1	2	3	4	5
Result of Write	√	√	√	√	√	√
Result of Read	√	√	×	×	×	×
Number of Crash	0	1	2	3	4	5
Result of Write	√	×	×	×	×	×
Result of Read	√	√	×	×	×	×

Table.3 result of read and write of single client and five replica data-servers with Qr=4, Qw=5

$Q_r = 3, Q_w = 5$						
Number of Corruption	0	1	2	3	4	5
Result of Write	√	√	√	√	√	√
Result of Read	√	√	√	×	×	×
Number of Crash	0	1	2	3	4	5
Result of Write	√	×	×	×	×	×
Result of Read	√	√	√	×	×	×

Table.4 result of read and write of single client and five replica data-servers with Qr=3, Qw=5

$Q_r = 2, Q_w = 5$						
Number of Corruption	0	1	2	3	4	5
Result of Write	√	√	√	√	√	√
Result of Read	√	√	√	√	×	×
Number of Crash	0	1	2	3	4	5
Result of Write	√	×	×	×	×	×
Result of Read	√	√	√	√	×	×

Table.5 result of read and write of single client and five replica data-servers with Qr=2, Qw=5

$Q_r = 1, Q_w = 5$						
Number of Corruption	0	1	2	3	4	5
Result of Write	√	√	√	√	√	√
Result of Read	√	√	√	√	√	×
Number of Crash	0	1	2	3	4	5
Result of Write	√	×	×	×	×	×

Result of Read	√	√	√	√	√	×

Table.6 result of read and write of single client and five replica data-servers with Qr=1, Qw=5

As you can see from above tables, when the system is single client and one single data-server, as long as there is one data-server has corrupted data or crashes, the client cannot get the desired results. However, when the system is single client and five replica data-servers with the fixed Qw and different Qr, the probability for client to get the desired results is different. When Qr=5, Qw=5, the result is the same as system with single client and one single data-server; when Qr decreases, the system can tolerant corrupted data and server-crash, and client can still get results from server. So, as you can see, this system improves the performance a lot compared the previous approach.

5. Evaluation

In this project, we have developed a FUSE filesystem that can support advanced features: server replication for performance and fault tolerance. The tables made above have demonstrated that when comparing to system with only one single server, our system which has replica data servers showed much better fault tolerance. Also the configurable ability of Qr and servers' ports make our file system more convenient to our client.

1. Normal state

Normal state is the system can be written and read:

```
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ echo "1234">a.txt
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ cat a.txt
1234
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ echo "5678">a.txt
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ cat a.txt
5678
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$
```

2. Corruption:

With Qr=3, Qw=5:

Data in two data-servers corrupted:

```
wenchao@wenchao-K46CB:~/Desktop$ python test_corrupted.py
++++++++++++++++++++++++++++++++++++
NOTE: please make sure you have created a 'a.txt'or writen something in 'a.txt'
file in fusemount before run this test_corrupted.py
++++++++++++++++++++++++++++++++++++
data in data_sever1 before corrupted (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f6ef2247bd8>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:29:32' at 7f6ef2247c
20>]]]
data in data_sever1 after corrupted (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f6ef2247b90>]], ['
/a.txt&&data', ["S'This is a corrupted data1'\np0\n.", <DateTime '20151207T01:29
:32' at 7f6ef2247e60>]]]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
data in data_sever2 before corrupted (dataserver2.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f6ef2252050>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:29:32' at 7f6ef22520
e0>]]]
data in data_sever2 after corrupted (dataserver2.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f6ef2247c20>]], ['
/a.txt&&data', ["S'This is a corrupted data2'\np0\n.", <DateTime '20151207T01:29
:32' at 7f6ef2252200>]]]
wenchao@wenchao-K46CB:~/Desktop$
```

The result is system can still read successfully:

```
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ cat a.txt
1234
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$
```

Data in three data-servers corrupted:

```
wenchao@wenchao-K46CB:~/Desktop$ python test_corrupted.py
++++++++++++++++++++++++++++++++++
NOTE: please make sure you have created a 'a.txt'or writen something in 'a.txt'
file in fusemount before run this test_corrupted.py
++++++++++++++++++++++++++++++++++
data in data_sever1 before corrupted (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f45aeea1bd8>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:30:50' at 7f45aeea1c
20>]]]
data in data_sever1 after corrupted (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f45aeea1b90>]], ['
/a.txt&&data', ["S'This is a corrupted data1'\np0\n.", <DateTime '20151207T01:30
:50' at 7f45aeea1e60>]]]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
data in data_sever2 before corrupted (dataserver2.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f45aeeab050>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:30:50' at 7f45aeeab0
e0>]]]
data in data_sever2 after corrupted (dataserver2.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f45aeea1c20>]], ['
/a.txt&&data', ["S'This is a corrupted data2'\np0\n.", <DateTime '20151207T01:30
:50' at 7f45aeeab200>]]]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
data in data_sever3 before corrupted (dataserver3.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f45aeea1f80>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:30:50' at 7f45aeea1b
d8>]]]
data in data_sever3 after corrupted (dataserver3.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:29:23' at 7f45aeeab0e0>]], ['
/a.txt&&data', ["S'This is a corrupted data3'\np0\n.", <DateTime '20151207T01:30
:50' at 7f45aeeab248>]]]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
wenchao@wenchao-K46CB:~/Desktop$
```

The result is the system cannot read:

```
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ cat a.txt
cat: a.txt: Bad address
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$
```

3. Crash:

With Qr=3, Qw=5:

Two data-servers crash:

```
wenchao@wenchao-K46CB:~/Desktop$ python test_crash.py
data in dataserver1 before crashing (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:23:21' at 7fd5dfa2bbd8>]], ['
/a.txt&&data', ["S'5678\\n'\np0\n.", <DateTime '20151207T01:23:47' at 7fd5dfa2bc
20>]]]
server1 crashes
data in dataserver2 before crashing (dataserver2.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:23:21' at 7fd5dfa36170>]], ['
/a.txt&&data', ["S'5678\\n'\np0\n.", <DateTime '20151207T01:23:47' at 7fd5dfa362
00>]]]
server2 crashes
```

The result is: can read but cannot write any more.

```
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ echo "9012">a.txt
bash: a.txt: Bad address
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ cat a.txt
5678
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$
```

Three data-servers crash:

```
wenchao@wenchao-K46CB:~/Desktop$ python test_crash.py
data in dataserver1 before crashing (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:26:18' at 7ff3fa02dbd8>]], ['
/a.txt&&data', ["S'5678\\n'\np0\n.", <DateTime '20151207T01:26:51' at 7ff3fa02dc
20>]]]
server1 crashes
data in dataserver2 before crashing (dataserver2.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:26:18' at 7ff3fa038170>]], ['
/a.txt&&data', ["S'5678\\n'\np0\n.", <DateTime '20151207T01:26:51' at 7ff3fa0382
00>]]]
server2 crashes
data in dataserver3 before crashing (dataserver3.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:26:18' at 7ff3fa02dd88>]], ['
/a.txt&&data', ["S'5678\\n'\np0\n.", <DateTime '20151207T01:26:51' at 7ff3fa02dd
d0>]]]
server3 crashes
```

The result is: cannot write and read any more.

```
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ echo "9012">a.txt
bash: a.txt: Bad address
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$ cat a.txt
cat: a.txt: Bad address
wenchao@wenchao-K46CB:~/Desktop/fusepy/fusemount$
```

4. Restart:

The data in data-server before restart and after restart:

```
wenchao@wenchao-K46CB:~/Desktop$ python test_restart.py
data in dataserver1 before restarting (dataserver1.list_contents()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:40:51' at 7f5faa4e3bd8>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:41:00' at 7f5faa4e3c
20>]]]
data in dataserver1 after restarting without restoring data (dataserver1.list_co
ntents()):
[]
data in dataserver1 after restarting with restoring data (dataserver1.list_conte
nts()):
[['/&&data', ["S''\np0\n.", <DateTime '20151207T01:41:33' at 7f5faa4e3e18>]], ['
/a.txt&&data', ["S'1234\\n'\np0\n.", <DateTime '20151207T01:41:33' at 7f5faa4e3e
a8>]]]
wenchao@wenchao-K46CB:~/Desktop$
```

After one data-server restarts, all data-servers have the same data.

```
127.0.0.1 - - [07/Dec/2015 00:03:04] "POST /RPC2 HTTP/1.1" 200 -
the number of agreed data:
5
127.0.0.1 - - [07/Dec/2015 00:03:04] "POST /RPC2 HTTP/1.1" 200 -
```

6. Potential Issues

Our file system has been introduced here so far. It shows the huge benefits of using replica data server when comparing with one single data server. However, there are still some potential issues that worth attention.

(1) In the design requirement, the code should be generic to accommodate any number of data-servers and the Qr and Qw values should be both configurable. However, in our design, we can only configure Qr's value and make the system run normally. If we change the number of data-servers, which is also the value of Qw, the whole system will not work correctly. So this problem makes our system less flexible.

(2) Since we did not write down the function to truly restart the data server, instead, we just wipe all the data on the data server to simulate function "restart". Our program may not be able to handle the situation when data server actually crashes and then restarts.

(3) Theoretically, when some of the data servers crash, mediator server will notice this and do not let these data server get involved with the quorum approach. However, we have not realize this function. Instead, we only put content "blank1" "blank2"… "blank5" to cover the original value in those crashed data servers. Though this will not affect the result of the quorum comparison, it is still a potential issue.

7. References

[1] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment", IEEE TRANSACTIONS ON COMPUTERS, VOL. 39, NO. 4, APRIL 1990, pp. 447-459.
[2] M. Satyanarayanan, "Distributed File Systems", Huygens Systems Research Laboratory, Universiteit Twente, Enschede
[3] FRED B. SCHNEIDER, "Implementing Fault-Tolerant Services Using the State Machine", ACM Computing Surveys, Vol. 22, No. 4, December 1990, pp. 300-319.
[4] Anupam Bhide, Elmootazbellah N. Elnozahy, Stephen P. Morgan, "Implicit Replication in a Network File Server", IEEE, pp. 85-90.
[5] Ylkang Xu, K. Vahalia, Jiang, Gupta, Tlelnic, "File Server System Using File System Storage, Data Movers, And An Exchange Of Meta Data Among Data Movers For File Locking And Direct Access To Shared File Systems", United States Patent
[6] "Remotely exit a XMLRPC Server cleanly (Python recipe)",
http://code.activestate.com/recipes/114579-remotely-exit-a-xmlrpc-server-cleanly/