Project Report

Principles of Computer Systems Design (EEE 5737)

Department of Electrical & Computer Engineering,
University of Florida, Gainesville, FL – 32608

ABHINANDAN JYOTHISHWARA
abhinandan@ufl.edu
UF ID: 89117194

AMOGH RAO
amoghrao@ufl.edu
UF ID: 13118639

Client-Server based Fault Tolerant Distributed FUSE File System

Abstract— **This paper describes our design and implementation of a client-server architecture on fuse file system with single client, single meta-server and multiple data-servers to achieve better performance and fault tolerance. Our implementation helps in providing fault tolerance in case of a power failure or data corruption on a single data server. The main focus of this report revolves around details about the implementation and performance evaluation of the designed architecture.**

Keywords—distributed file system; client-server model; FUSE; fault tolerance; redundancy

I. INTRODUCTION

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. A fault-tolerant design enables a system to continue its intended operation, possibly at a reduced quality level, rather than failing completely like it would in a naively designed system, when some part of the system fails. [1] This ability of the system to rebound from faults and still continue perform correct and consistently is called fault tolerance.

Redundancy is the most common approach used to achieve fault tolerance. Redundancy basically means duplication of critical components or functions of a system inorder to increase reliability of the system usually in the form of a backup or fail-safe or catering to improve the overall reliability of the entire system. [2]

There is a wide range of issues with the conventional filesystem when using the Client-Server model. These issues can range from power failures during transaction to data corruption due to component failure which can cause discrepancy in correctness and cause entire server to fail. These kind of errors not only affect the performance of the system but if data corruptions are not detected and corrected can cause failure of the entire system. Thus, there arises a need to detect, contain and rectify these errors. There are a number of techniques used in popular filesystems widely used today inorder to handle these issues. The techniques include redundancy – to store duplicate copies for reliability, checksum – to detect data corruption, backup recovery – to tolerate server crashes. We have implemented these techniques in our design to build a reliable distributed file system.

II. PROBLEM STATEMENT

The purpose of this project is to build a Client-Server based distributed FUSE file system which uses redundancy to achieve fault tolerance. Below mentioned are the required characteristics for the design implementation:

a. Single meta-server to store metadata of all the files and directory under the assumption that it is extremely reliable and never fails.
b. Multiple data-servers to store data as blocks in round-robin fashion such that a redundant copy of current server's data is stored in the next server to increase reliability.

    - Replica 1: [x % N, (x+1) % N, (x+2) % N, (x+3) % N ...] where x is the hash
    - Replica 2: [(x+1) % N, (x+2) % N, (x+3) % N, (x+4) % N ...]
    - The two replicas mean that the redundancy factor is 2

c. The data-server should have the capability to backup data in a persistent storage (hard disk) for later recovery on an event of server crash.
d. Data-server should tolerate crashes - where the server process is terminated and restarted at a later point in time. In such a scenario it should recover data from backup in persistent storage. If unsuccessful in doing that it should recover data from the redundant copy of data in the adjacent server.
e. It should be robust against data corruption. (Should be handled by verifying data using checksums)
f. It should be able to block writes when a server is down and should keep retrying the operation till it succeeds and the write call should not return till then. On the other hand reads should return successfully even if a single replica is available and the checksum verifies correctly

III. Design Approach and Implementation

A. *Deviations / Design choices:*

Hierarchical Structure:

The requirements of the projects was to host a hierarchical FS on a client while maintaining a flat name space on the server. While this would improve the performance of the kernels which involve only traversal followed by a read of the DS maintaining the information (meta or data), any operation which involves both traversal followed by a modification of the structure of the DS would suffer immensely i.e. the complexity would change from height/depth of the node in a hierarchical notion to number of elements in the file system in the flat name space implementation. E.g. consider deletion of a directory from a server maintaining the FS in a flat name space. Every element should in the whole DS (or hash table) should be looked at, determined whether it would appear in any subdirectory, and delete that entry.

The same problem follows for renaming a directory. Multiple files or entries from the hash table have to be removed, and for this, one would have to look at every entry in the hash table.

Unchanged FS kernels after blocking implementation:

The operating system calls, or FS kernels do not have any awareness of existence of redundant copy, or the existence of a check sum associated with each block of data. All these are handled by the two traversal functions (traverse / traverseparent) on the client which trigger the respective function on the server. And hide the existence of redundancy or error correction from the OS call. One can say these client side functions or APIs have been written such that they can be hosted on a proxy server with very little if not any change at all.

Truncate functionality (for extra credit evaluation):

Since truncate can actually be achieved even without actual deletion of data, we choose to define truncate as simply updating the file size and shortening the last block to the required length if it is to be partially filled. Thus, there is a requirement of contacting at most 2 data servers, (one with the actual copy and the other with the redundant copy) which contain the last (potentially partially filled block). Thus, truncate may end up completion of execution even if a few data servers are down. Thus, the notion of previous version would be an empty file, following linux commands are executed:

```
echo “original_data” > t.txt
cat t.txt                    #original_data
echo “new_data” > t,txt      #if a fault occurs here,
cat t.txt
```

Depending on whether the fault occurs before or after an atomic truncate, completion, the following two outputs are possible

1) “”

   - No data is printed if data server cashes after truncate commits a change to the version numbers of the file in question (along with a conn refused from the write function)
2) “original_data”

   - Is printed if the data server crashes before the truncate is committed (along with a conn refused message from the truncate function)

Dealing with blocks on client mode vs server mode:

Before hosting the traversal functions with blocking on the server, if multiple blocks were needed by an OS read call, all of them could be accessed together as we were dealing with references only. Getting multiple blocks via a single RPC call amounts to minimization of number of RPC calls required for the read. This saves very little time as the majority contributor to the latency is the actual transfer of data and not the (RPC) function stack frame setup time. Thus, as the total amount of data to be communicated remains the same, we have not chosen to optimize the number of RPC calls, and rather spent more time on standardizing the abstraction of layers i.e. redundancy and fault correction from the FS kernels. Using the “cmd” in open and close modes which is elaborated in the future section.

Non-deterministic starting data server (for efficient load balancing along data-servers):

The suggested approach was to determine the starting data server based on a hash of the path. While this would enable achieving data balance, upon a rename, it is highly likely that the hash of the new path would give a different starting server. This would mean every data block of the (potentially very large) file being renamed has to be reorganized in every server. It is not intuitive to move data to achieve rename functionality. Thus we chose to store the starting data server as part of the meta data of every file (along with size and other meta data) on the meta server. All the functions make an RPC call to the meta for the size of the file, and thus fetching the starting data server number is hidden in terms of latency within this call. This data is used inorder to perform hashing described in the next section

Definition of Corruption:

For project:

RPC corrupt function accepts the following parameters to corrupt data with internal defaults:

def corrupt(self, path, bid=0, tred=False, cid=0, cchar='*')

1) Path, (no default)
2) Bid, (default 0) refers to the block id on that server
3) Cid , (defines the character number that need to be corrupted, default = 0)
4) Character to corrupt, (default is either * or +, i.e. if data itself is *, + will be used)
5) Tred, (default = false), will corrupt redundant copy present if True.

For extra credit:

Will corrupt all versions of the block in discussion.

B. *Design:*

The approach we took for our design of distributed filesystem was to make the client capable of performing all the computations and use server to just store the filesystem. The get and put functions are thus designed to get or update a directory (a complete node) or file (an element in the node) in the hierarchical FS. A node being a directory. All the writes and reads to the self.files and self.data of the filesystem is performed through communication (cmd) from client to respective servers (meta-server and N data-servers).

Starting with traverse and traverseparent functions which are capable of dealing with files and directories of a hierarchical FS, we chose to host them on the server along with the nested dictionaries. The return values of traverse and traverseparent are values and references respectively.

Returning value by reference loses its meaning in a client/server architecture. Everything should be performed as pass by value between the client and the server. The traverse and traverseparent have been modified such that is performs pass by value and pass by reference respectively within the server or the client. However, before returning the value from the server to the client, any reference is revisited, the data pickled into a string and value is returned instead of the reference.

Further, to make sure all modifications made to the value in the FS kernel, reflect in the server, extra steps need to be taken. This is the responsibility of each FS kernel. While both traversal functions have been discussed in the open mode i.e. fetching information from either the meta or the data server, these functions are equipped to operate in the close mode. Consider, the traverse/traverseparent fetching the metadata of a file or a directory from the meta server. The fetching of information would be done in the open mode. If the kernel which required this metadata modifies it, the kernel should call the traverse/traverseparent function in the close mode. This will translate to the server receiving the new data, and commenting to the copy on the server.

Implementation:

1. The client requests for information it wants to modify or read via the cmd message which will be discussed in the next section.
2. The server interprets the “cmd” message from the client, performs traversal (traverse or traverse_parent) on the appropriate nested dictionary, and returns data to the client.
3. The client on receiving the information (data or meta data) and uses it as required. If modification of the obtained info occurs in any given FS kernel, that kernel must re transmit the modified data to the server via the cmd message
4. The server then receives the modified data and updates its copy in the filesystem thus providing for “pass by value” like function call over network.

The message “cmd” is as below:
cmd={'function':'traverse','path':path,'tdata':tdata, ‘handle':handle, 'update':update, 'bid':SBid, ‘tred’:tred}

function = string, choose traversal method, : traverse / traverseparent
path = string, path
tdata = Boolean, True => refer self.data, False => self.files
handle = string, type of return value needed “open” => information, “close” => success
update = string, information (from or to the server)
bid = integer, block id of the file (used to perform hashing for multiple dataservers, refer to entire file if bid<0)
tred=Boolean, choose between original copy or redundant copy (if tdata=True)

cmd examples with and without modification for p[stsize updation] in meta

Hashing:

SBid = bid//self.num_D
SerN = ((starting_d+bid)%self.num_D))

bid = block id of the file
SBid = block id on the specific dataserver
SerN = Server ID
Self.num_D = number of data servers
Starting_d=data server to store the first block of the file in.

IV. TESTING MECHANISM

Please find embedded below, the master test script. The “echo” commands indicate what the nature of testing is. Various hierarchical structure tests as well as write and truncate Stress Tests have been performed. Data balance,corruption detection and crash detection have been tested manually.

Main test script:

```
#!/bin/bash

echo "preping mounted dir test"
rm -rf ./tttt/*
cp hierarTest.sh ./tttt/

echo "running hierarchical tests on mounted dir"
cd tttt/
./hierarTest.sh > hres.txt
cd ..

echo "preping ref dir test"
rm -rf ./refdir/*
cp hierarTest.sh ./refdir/

echo " running hierarchical tests on ref dir"
cd refdir
./hierarTest.sh > hres.txt
cd ..

echo "comparing outputs"
cp ./tttt/hres.txt ./hres.txt
cp ./refdir/hres.txt ./href.txt
diff -q hres.txt href.txt

echo "running write stress test"
 ./bruntest_write.sh
 mv error.txt error_write.txt

echo "running truncate stress test"
 ./bruntest_truncate.sh
 mv error.txt error_truncate.txt
```

Write Stress Test: (bruntest_write.sh)

“dd if=s.txt of=t.txt seek=17 conv=notrunc oflag=seek_bytes status=none”

Above is a Linux command which triggers a write OS call with offset 17. The amount of data written is governed by the contents of input file “if”. Execution of this command with

1) Initial file size of target t.txt with content “abcdef…”
2) Amount of data written or size of input file with contents “12345…”
3) Offset used to start writing in the target file

enabled us to perform stress testing.

```
#!/bin/bash
oneWrite()
{
   c=0
   str=""
   while [ $c -lt $1 ]
   do
      for letter in {a..z} ;
      do
         if [ $c -eq $1 ] ; then
            break;
         fi
         ((c++))
         str=$str$letter
      done
   done
   rm t.txt 2>/dev/null
   echo -n $str > t.txt
#   echo $str written to t.txt
   c=0
   str=""
   while [ $c -lt $2 ]
   do
      for letter in {0..9} ;
      do
         if [ $c -eq $2 ] ; then
            break;
         fi
         ((c++))
         str=$str$letter
      done
   done
   rm s.txt 2>/dev/null
   echo -n $str > s.txt
#   echo $str written to s.txt
   dd if=s.txt of=t.txt seek=$3 conv=notrunc oflag=seek_bytes status=none
#   echo s.txt copied to t.txt at $3
}
echo "" > error.txt
```

```
for x in {0..30} ;
do
    echo x is $x
    for y in {0..60} ;
    do
        for z in {0..60} ;
        do
            oneWrite $x $y $z
            cd tttt/
#             echo directory changed to tttt/
            oneWrite $x $y $z
            diff -q t.txt ../t.txt
            flag=$?
            if [ $flag -ne 0 ] ; then
                echo "Error : ${x},${y},${z}" >> ../error.txt
                echo fail at $x $y $z
            fi
            cd ..
        done
    done
done
```

Truncate Stress Test:

Parameterization of initial file size and target file size with the command,

“truncate -s N file.txt “

enabled us to perform stress testing of truncate.

V. TASKS PERFORMED

Task Name	**Member Responsible**
Redundant copy storage in adjacent data server	Amogh
Data store and recovery from backup	Amogh
Checksum integration and corruption test and debugging	Abhinandan and Amogh
Blocking writes when server unavailable	Abhinandan
Enabling read from redundant storage when server unavailable	Abhinandan
Testing and debugging by writing test scripts	Abhinandan and Amogh
Extra credit work : Write atomicity by maintaining version	Abhinandan
Extra credit work testing and debugging	Amogh

VI. REFERENCES

[1] https://en.wikipedia.org/wiki/Fault_tolerance

[2] https://en.wikipedia.org/wiki/Redundancy_(engineering)

[3] Peric, D.; Bocek, T.; Hecht, F.V.; Hausheer, D.; Stiller, B., "The Design and Evaluation of a Distributed Reliable File System," Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on, vol., no., pp.348,353, 8-11 Dec. 2009.

[4] Saltzer, Jerome H., and M. Frans Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.