DESIGN OF FAULT TOLERANT FUSE FILE SYSTEM

Govindraju, Madhusudan; Nair, Sanjay; Sanyal, Indraneel

University of Florida

ABSTRACT:

The project focusses on designing a fault tolerant FUSE file system. We are basically making the system fault tolerant by implementing NMR (N Modular Redundancy) where each operation on a file is replicated into N servers. The system is designed to be robust to single server failures. Quorum approach of file selection is incorporated in this project. The paper describes the functionality of data synchronization and consistency among all servers during a serve crash or file corruption.

Keywords: Fault Tolerance, NMR, quorum, data corruption, server refresh.

PROJECT BACKGROUND:

Fault tolerance of a system can be described as the system's resilience against faults and also provide desirable and consistent performance. One of the requirements of making the system fault tolerant is redundancy among the data-carrying servers. Redundancy is incorporated into the client server architecture where in the servers are replicated and store the same instances of data on all servers. By doing so, file corruption of server crashing will not affect the overall performance as the redundant servers will start functioning and accept requests from the client. When there are multiple servers functioning for a particular data entity the result is always fed into a voter which compares the results by the individual servers and chooses the value with maximum votes (quorum value) as the response for the client. Redundancy in most popular cases is implemented through TMR (Triple Modular Redundancy) or NMR (N Modular Redundancy).In our project, we have implemented redundancy using NMR. The number of servers will be specified by the user and our program and test cases will meet the requirement specified by the user.

The problem is quite interesting and difficult to solve. We have to develop and extend the FUSE filesystem which will be described in the following section. The system will have to be extended to include the quorum approach and communicating through the remote-procedure calls between client and server. If a server crashes, the client will not know about failing of server because the system repair mechanism is handled at the server side. Although the server can be repaired it starts with a clean slate and a mechanism should be incorporated to maintain data consistency among all servers. The data synchronization makes the project difficult in conceptual as well as in programming aspect.

INTRODUCTION TO FUSE FILESYSTEM:

Filesystem in Userspace abbreviated as FUSE is a special part of kernel in Linux that will allow users to develop their own file systems without changing the root privileges. There are virtual filesystems used in FUSE. The FUSE is itself is kernel and the filesystem used in user space.

FUSE allows users to make their own filesystems, modify filesystems, or use a special filesystem temporarily. FUSE can also be used to add extra features and abilities to a system or software. If users wish to access a storage unit with a exFAT/FAT64 filesystem, users can mount the filesystem in FUSE. However, FUSE may need some extra packages to be installed to run some filesystems or virtual filesystems.[1]. FUSE is open-source freeware that anyone may obtain and use. FUSE is stable, secure, and reliable. However, it is more efficient to use a "real" filesystem if possible. FUSE is compatible and works on Solaris, FreeBSD, Darwin, GNU/Hurd, OS X, Open Solaris, and others. Users can program their own filesystems and use them in FUSE without changing the Linux kernel. [1]

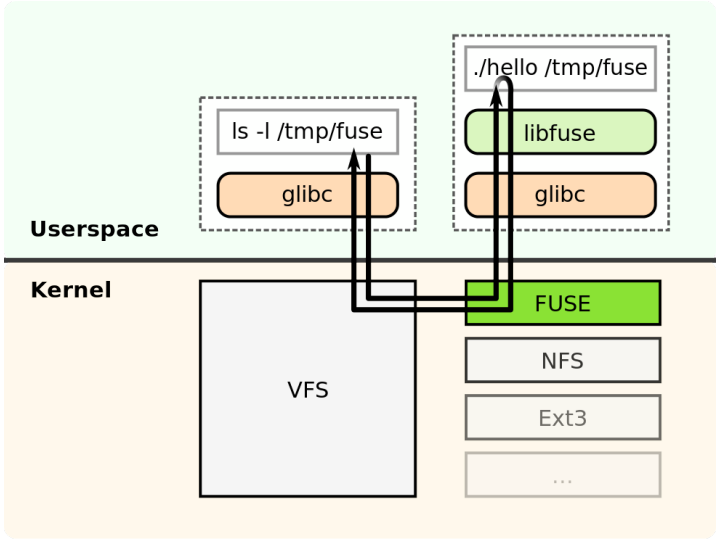

Figure 1: A flow-chart diagram showing how FUSE works

FUSE is a Linux module that acts as a “middle-man” or mediator between the FUSE filesystem and the Linux-kernel's VFS module. The VFS module can only be accessed by privileged users or processes. Since all users may access FUSE and FUSE may access VFS, that is how the permissions work to allow any user to use a FUSE filesystem. As for the FUSE filesystem itself, the code contains a structure variable that contains a pointer to functions in the FUSE filesystem code that respond to various actions that a process may try to use. In simple terms, the structure variable is a set of code that says a “read” syscall must have a particular set of parameters and is handled by a specific function defined in the FUSE filesystem's code.

N-MODULAR REDUNDANCY

N modular redundancy is fault tolerant mechanism that incorporates replication of a single module and votes on the output of each module. This is usually preferred for applications where reliability is the main concern and not performance or cost. In this mechanism, each module works individually, processes the requirement and produces an output. Although a few modules might stop working, because of the presence of extra modules an output is still available.

A voter is also part of this system which is responsible for providing the right output. The voter compares the output of all the modules and comes to an effective output based on the result that was obtained by maximum of servers. A basic TMR (N = 3 in NMR) is shown below:

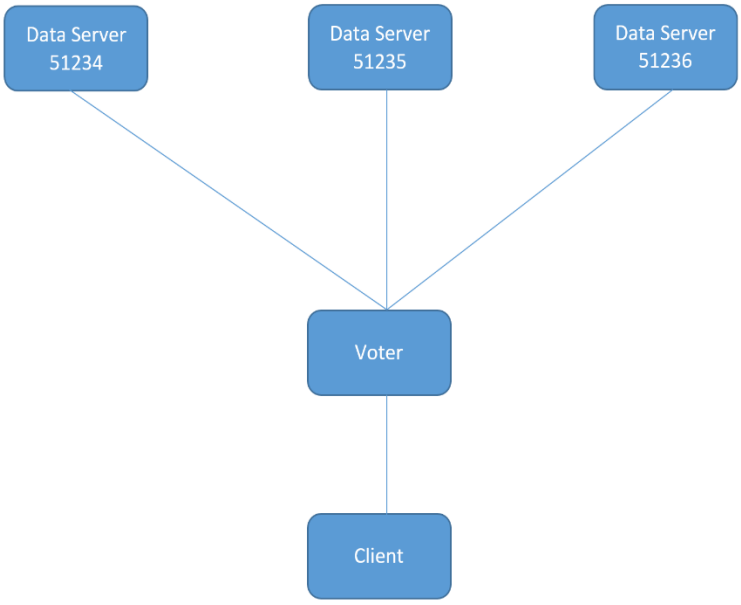

Figure 2: A basic TMR structure

There are a few disadvantages in this system. Firstly, the hardware overhead is extremely large. This ultimately affects the running and installation costs as well. The power requirement to all the modules is extremely large. The control logic involved in running an NMR system can get complex occasionally making it difficult in maintaining and debugging.

Although there are a few disadvantages as mentioned above, the main purpose of an NMR is to make the system reliable. It is usually used in space computing exposed to harsh environments and critical applications like bank transactions.

IMPLEMENTATION AND DESIGN:

Our implementation focuses on developing and extending the fuse file-system. We have added certain functions to meet the requirements.

A) SERVER CREATION AND ALLOCATION:

Our implementation involves creation of multiple instances of data servers running on multiple ports. It also receives the Qr and Qw values from the user, that denote the number of correct servers required and number of total servers respectively.

B) ASSIGNING SERVER STATUS:

We have defined three states for the servers. They are shown below:

STATUS	STATE OF SERVER
State-0	Functional Server
State-1	Crashed Server or Connection Error
State-2	Corrupted Data Server

Table 1: Server States

The states are used to monitor all servers and take appropriate action based on the states. State-1and State-2 results in triggering the functionality (refresh function), that is responsible for maintaining data consistency among all servers. The implementation of this function is explained below in the voting mechanism

C) SERVER-VOTING MECHANISM:

There are multiple ways in which voting can be implemented. We have incorporated a linear comparison among all servers. Upon a read request, all the servers will give its own version of the response. The voting function compares each response by the server with the responses of all remaining servers. Based on the comparison, we keep a count on the number of servers that are giving the right response and the ones that are not. This count has its own version for each individual server upon comparison with others. Based on the counts received we assign a particular status to that server as described below:

- Case 1: If the number of servers with the correct data is greater than the corrupted/crashed server (wrong data): In this case, it will take the corresponding correct data from the server and give the output to the client.
- Case 2: If number of servers with the correct data is lesser than the corrupted/crashed server (wrong data): In this case, it will change the server status to 2 and proceed to the next server (in case there is a server connection error we set the server status to 1 and proceed to the next server).

Here it is to be noted that server state involving crashed server is not involved in reading data as it would always lead to an exception. Hence the comparison takes place among servers that are online.

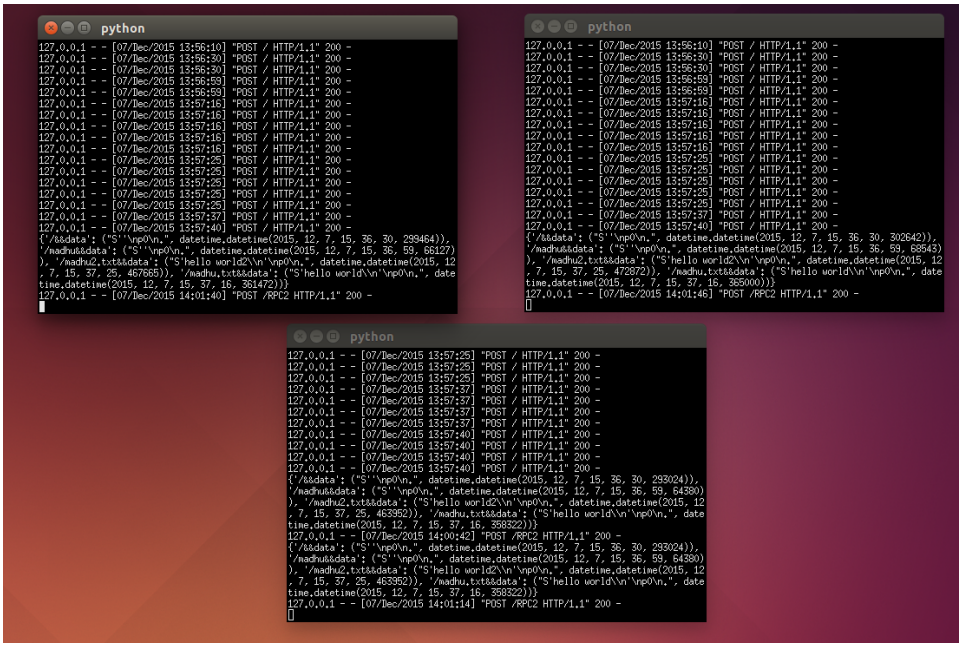

Figure 3: All three servers showing same content on a read

D) WRITING DATA TO THE SERVER:

Writing data is also an important functionality as all the servers that are online need to be written into at the same time in order to maintain data consistency and replication. Data is written into all the servers with status 0. If the function comes across a server with status 1 or status 2, it recognizes the requirement of copying data from the servers that are online into servers that are crashed or corrupted. This is done by calling another functionality that is responsible for maintaining data consistency among all the servers. This function called refresh() is explained in the following section.

E) SERVER REFRESH:

As explained above, when a server is in the crashed/corrupted state the *refresh()* function is called. This function takes in information about the crashed/corrupted server and the server that is functioning normally as parameters. Based on the information provided it triggers a function on the server that is responsible for copying all the data from the correct server to the crashed/corrupted server. The functionality implemented on the server is *load_data()* which will be explained later. This function is called in two different ways based on the server status as mentioned below:

- Case-1: When server status is 1: In this case, the *put()* function will call the *refresh()* function. If the server is back online then it will go ahead and load data from a correct server and the server status variable will be changed to 0, otherwise the server status will remain 1.

- Case-2: When server status is 2: In this case, the *put()* function will call the *refresh()* function and the corrupted data will get overwritten by the corrected data from the other servers through the *load_data()* function.

F) LOAD DATA:

This function basically copies one server from another. This function retrieves the dictionary contents form the correct server and copies on to the corrupted/crashed server that requires the new data.

TESTING MACHANISMS:

Test Cases:

A set of basic checks were performed to test the robustness and functionality of the design. These are mentioned below:

1. *Consistency of Data writing:*
   When a data is written, the data is checked on all three servers using the *get_content()* on each of the server objects.
2. *Consistency during data reading:*
   The lookup for a particular file was done using its filename as the key. Before reading, it compares the data and the metadata for the accessed filename.
3. *Server goes down:*
   When there are several servers running that has the file system stored there is a good probability that some of them might go down. The server can be terminated using *terminate()* function. This is where the server loses all the data.

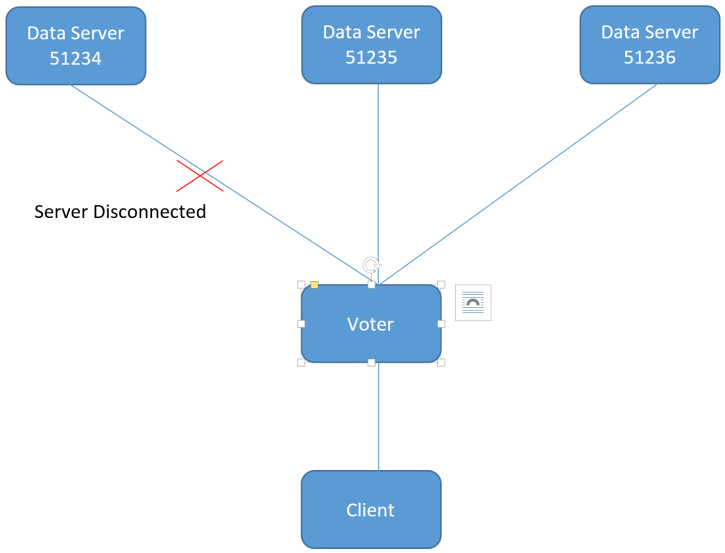

Figure 4: One server down and disconnected

4. Server back online:
   When the server is back online we need to check the data consistency. The contents can be printed using *get_content()* function. When the server has restarted before encountering a connection error in the file system, a key error is raised. This is because there is no data inside the server. Here the server status is set to 1.

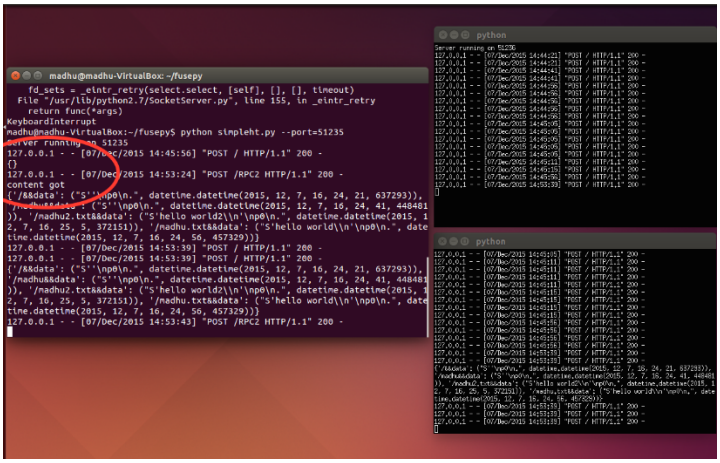

Figure 5: Server restarted with empty data and after refresh data is copied.

5. Corruption of data
   Using the function *list_contents()* we can get a set of keys that are already stored in the server. Using one of the keys present we will corrupt the value held by that key. By doing so, we are able to corrupt the data in one server while all other servers have the correct values.

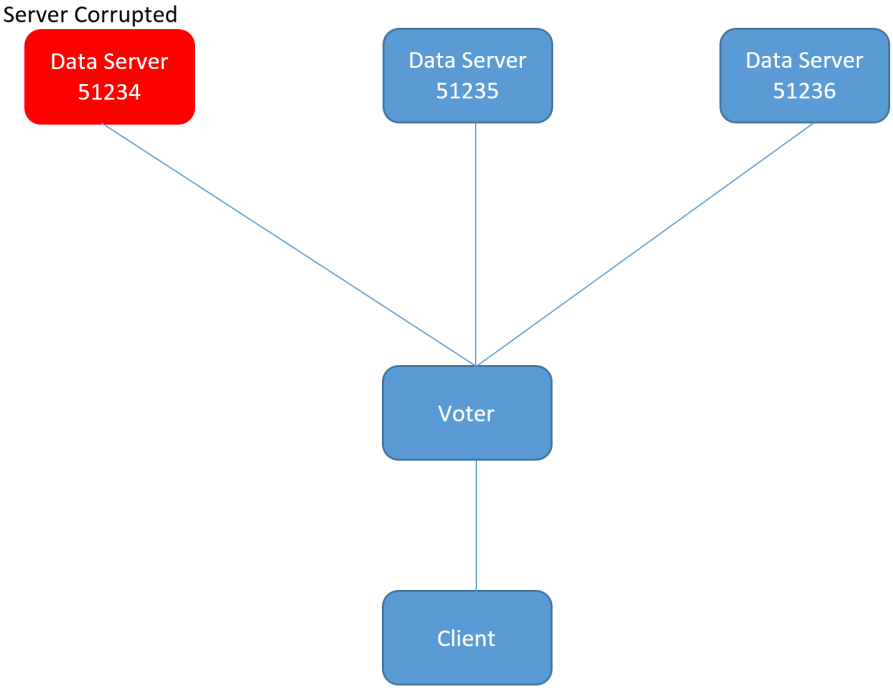

Figure 6: Server corruption in one server in a TMR system

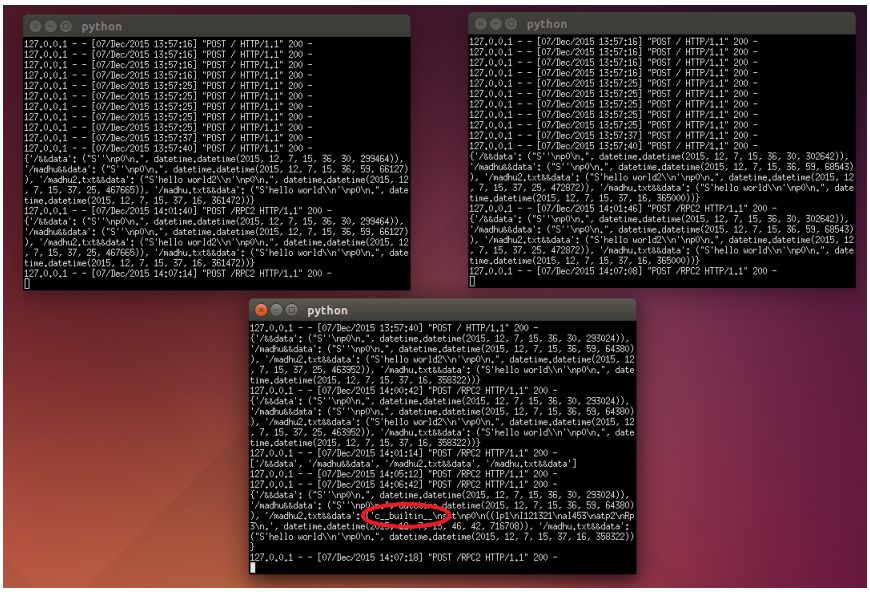

Figure 7: One server data corrupted

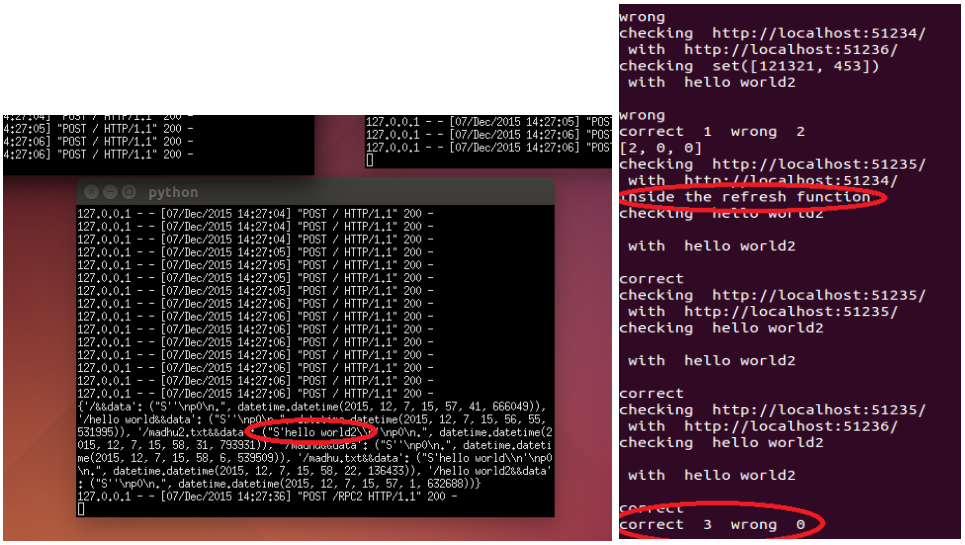

Figure 8: Data refreshed in the server post corruption

Testing Samples:

A) Accessing server directly

The contents of the server can be viewed by opening terminal and python interpreter

python

import xmlrpclib

from xmlrpclib import Binary

s = xmlrpclib.ServerProxy('http://localhost:51234)

s.get_content() #this lists the contents in the server

B) Corruption of Data in the file System:

Open a new terminal, keeping the file system running and corrupt the data by proxying into one of the servers

python

import xmlrpclib

from xmlrpclib import Binary

s = xmlrpclib.ServerProxy('http://localhost:51234)

s.get_content() #this lists the contents in the particular server

s.list_content() # returns a list of keys present in the server

s.corrupt('key_value')# use one of the keys present in the list. This would corrupt the value

C) Bringing down a server:

Open a new terminal, keeping the data servers running and terminate one of the servers by proxying into that server.

python

import xmlrpclib

from xmlrpclib import Binary

s = xmlrpclib.ServerProxy('http://localhost:51234)

s.get_content() #this lists the contents in the particular server

s.terminate() #shuts down the server

POTENTIAL ISSUES:

As the number of servers increase performance can be an issue. Currently in this implementation the refresh function is called only in the next iteration after the server experiences a crash or a data corruption. This can be resolved by a constant refresh mechanism that is spawned in a separate thread.

EVALUATION

The project implements the desired requirements of the user and gives us the necessary redundancy without sacrificing the performance. We have tested and evaluated our design using various test cases and also the snapshots of some of the test cases results are included in the report. It is able to replicate and distribute the files among all the servers. It restores corrupt data during access as well as refresh. It also loads the data after a server recovers from a crash. At all possible instances each server functions independently and does not interact with other servers thus avoiding a chances of correlated failures. Currently the server refresh functionality is sequential in nature. Performance of the module can be greatly enhanced if the refresh functionality can execute as a parallel continuous monitoring function that detects anomalies in the server data consistency and initiate repair independently.

REFERENCE:

[1] http://www.linux.org/threads/fuse.6211/

[2] http://www.ibm.com/developerworks/library/l-fuse/

[3] J. von Neumann, "Probabilistic logics," in Automata Studies. Princeton, N. J.: Princeton Univ. Press, 1956.

[4] Principles of Computer System Design- Part-2- Jerome Saltzer, M.Frans Kaashoek, Massachusetts Institute of Technology.