Christopher Brant
Nick Poindexter
EEL 5737
Due 12/4/19

Final Project Design, Testing, and Analysis

Design and Implementation

Our design for the overall structure of our code falls into a relatively simple layered structure in our raidcontroller.py file.

Raidcontroller.py Overview

The raidcontroller.py layer sits between the Block Layer and the Server stubs of all the servers which store the data. In general, the design utilizes virtual blocks and virtual inodes to keep track of data throughout the system. Each virtual block object holds the virtual block number, the server where the data can be found, the respective "real" block number on the server that holds the data, the virtual block number of the parity block, and whether the block is valid. The virtual inodes handle the same way normal inodes do, but instead of being stored on the servers themselves, the virtual inode table is within the raid controller layer.

The raidcontroller.py layer has two types of functions: the type that interface with the layer above it, and functions that make calls to the data servers. The layer keeps track of the virtual blocks, virtual inodes, current servers, and their states.

virt_functions

Virt functions are create for get_valid_data_block, free_data_block, update_data_block, and get_data_block. All virt functions utilize a try-catch block that will catch if a server failure occurs during a request to a server, and update the status of the server in the server state table. The virt functions also handle serializing and deserializing of data through the use of pickle.dumps and pickle.loads.

Internal Helper Functions

- *Xor_strings* - Xor's two strings together by converting them to byte arrays, xoring them together, and then returning a string.
- *Data_to_checksum* - Returns the appended checksum to the passed in data block.
- *Checksum_to_data* - Returns the data block if the checksum is what it's supposed to, returns a failure otherwise.
- *Get_fixed_data_block* - Returns the data on a failed server by fetching the data and parity information on the other servers and xoring them together.
- *getNumServerFailures* - Returns the number of server failures by parsing the server state table.

Christopher Brant
Nick Poindexter
EEL 5737
Due 12/4/19

Block Layer-Interfacing Functions

- *Inode_number_to_inode* - Returns an Inode by indexing the vnode table with the given inode number.
- *Get_data_block* - If there is less than one failure server, this function just calls get_virt_data_block, else this function calls get_fixed_data_block.
- *Get_valid_data_block* - Creates a valid data block by calling valid_virt_block, updating the valid bit and server block number of a virtual block, and creating a valid parity block to hold the parity information.
- *Free_data_block* - Retrieves the old parity and old data block information. Updates the parity block by xoring the old parity information with the old data block information. Then, it frees the data block on the server.
- *Update_data_block* - If all the servers are up, the function just updates the data block and parity blocks, along with their respective checksum information. If the server that is trying to be updated is down, then the function retrieves the data from all the other servers and xors them together with the new data being written, eventually writing the updated parity to the server.
- *Update_inode_table* - Updates the virtual inode table.
- *Status* - Displays the current contents of the vnode table, vblock table, and showing their contents by reading from the servers.

User-Interface Shell

The shell is created using python's cmd package, where the required functions are each registered as command shell functions. This is done in the Filesystem.py layer.

Running the Filesystem

To run the filesystem, configure the number of desired servers and the desired delay length during each read/write in the config.py python file. Then run the backchannel.py file to bring up the servers. Once the servers are up, run the Filesystem.py python file to bring up the command shell. To break this down stepwise would be to open two terminals, in one of them, run "python backChannel.py <number of servers>", and then in the other, run "python FileSystem.py". Then once those are open, a user can run the same commands in the command shell they would run in the FileSystem.py script.

Testing Procedure

Christopher Brant
Nick Poindexter
EEL 5737
Due 12/4/19

The way a server failure is created is using the backchannel.py script. Once a server is selected from the command prompt, the server's state is changed to false. Therefore, all requests made to those servers from our application will return a server state of false, indicating a server failure. With the current server setup, if a shell is just closed, the exception sent back causes the raidcontroller.py to fail in a way python cannot handle due to its limitations. Therefore, the above mentioned testing technique must be used. The series of commands that we used within our interactive shell to verify our functionality is listed below:

$ mkdir /A
$ mkdir /B
$ create /A/1.txt
$ create /B/2.txt
$ write /A/1.txt POCSD 0
$ write /B/2.txt LongTestWord 0
$ status
$ read /A/1.txt 0 5
$ read /B/2.txt 4 4
$ mkdir /A/C
$ mv /A/1.txt /A/C
$ rm /B/2.txt
$ status
---- at this point we enter "0" into the terminal running backChannel.py to "corrupt" server 0 such that it will fail and stop upon the next command it receives ----
$ mv /A/C/1.txt /A
$ status
$ read /A/1.txt
$ write /A/1.txt hey 0
$ read /A/1.txt
$ rm /A/1.txt
$ status

Tests and Performance Analysis

The process of testing that we chose to step through was to yet again replicate the series of commands at the top level, in Filesystem.py, that were used to test our previous homework implementations. The tests that were completed were a series of attempts of stepping through each individual action at the top level within our interactive shell and printing the status of the entire file system as a way of checking to ensure

everything is working as intended. It proved to be much simpler to just test our code by running the full system script so that if any issues were to happen, they would become obviously apparent. Once our interactive shell was also implemented, we did the same stepwise testing in which we would take down a server and make sure that everything necessary still ran as expected.

A performance analysis of this design would include measures of requests per server based on what command is being entered, in multiple different scenarios. Those scenarios would be whether all servers are up, if one server has failed, and when compared to the single server solution of HW4. Throughout this analysis we refer to the number of servers simply as N, and the same basic configuration is used for HW4 and our RAID-5 implementation. Specifically, our RAID-5 implementation, both for the non-failure scenario and failure scenario, worked out to have only four commands which caused requests to be sent to a server. Those commands are write, read, rm, and status. Subsequently, we will only compare the functions that we can model with a function of average requests per server. Status is a special command in our case, as it only calls requests to a server if that server contains a block that is currently allocated. So it cannot be represented by a function other than (total blocks allocated / N). This is the case for both failure and non-failure scenarios, except in a failure scenario there will be more requests when reading from blocks on the failed server.

We know that in HW4's single server configuration, write completes 8 requests per block, read completes 7 requests per block, rm completes (for some reason) 30 requests to remove a file with only one block. In the non-failure scenario, the write command completes (4/N) requests per block that needs to be written, the read command completes (1/N) requests per block that needs to be read, and the rm command completes (2/N) requests per block that is being removed in a file. In the failure scenario, write completes [(N-1) + 1] / N requests per block to write, read completes (N-1) / N requests per block to read, and rm completes 1 request per block of a file to remove.

Based on these functions, the amount of requests per server become drastically reduced overall in a RAID-5 system without failure. We see a reduction from a single server setup as well, even when there is a failure in the RAID-5 system. The only outlier is the status function as it can call a read for every single block that has been allocated in the filesystem, and if the filesystem is near capacity, that would be a very large amount of reads. Overall, as expected, we do see a large reduction in average requests per server for the commands that are affected by this different implementation.