PoCSD HW #5 Report

Introduction:

The goal of this project is to extend the file system we've been adding to all semester to support multiple servers with redundant block storage, emphasizing fault tolerance, data integrity, and performance. The project builds upon the foundations laid in previous design assignments, particularly the client/server file system developed in homework #4. The extended system aims to distribute and store data across multiple servers using the RAID-5 approach, providing load balancing, increased capacity, and fault tolerance.

Problem Statement:

The primary challenges include implementing RAID 5 functionality to tolerate a single failure and supporting recovery from two types of failures: corrupt block and server disconnection. The redundant block storage system should distribute data and parity information across servers, using checksums for error detection at the granularity of the specified block size.

For reads, the system should achieve load balancing by distributing requests across servers holding data for different blocks. Small reads of a single block should be served by a single server. Data integrity should be maintained using checksums, with the ability to correct errors using other servers' blocks and parity. Writes must update both data and parity blocks, following the RAID-5 approach to compute new parity. The system should be capable of tolerating a fail-stop failure of a single server, registering the failed server and continuing operations using the remaining non-faulty servers.

The project introduces a block caching mechanism, allowing for improved performance for a single client. Additionally, command-line arguments such as `-logcache`, `-startport`, and `-ns` are incorporated to enable cache logging, specify the starting port for the first server, and indicate the number of servers, respectively. The system also includes a repair procedure that regenerates blocks for a server that has crashed and is replaced by a new server with blank blocks. The repair process is initiated through a command in the shell.

In summary, this project extends a single-server file system to a distributed, fault-tolerant system with redundant block storage, leveraging RAID-5 principles and integrating features for performance optimization and recovery from failures. The rest of the report will provide an analysis of the design, implementation, and testing of the system.

Design and Implementation

Block.py

This file is where most the functionality of the system resides. Put() and Get() call on SinglePut() and SingleGet(). The cache is implemented in this file. The cache is initialized as an empty 2D matrix with the number of rows being the total number of blocks and the number of columns being the number of servers. A command-line input of '1' for logcache enables cache messages to be printed.

Put()

This is the main function called by all other files to send data to a server. The inputs to this function are the virtual block number and the data to be sent. The first step this function takes is to convert the

virtual block numbers to a physical block number, and which server it corresponds to (as well as the server for the parity bit).

In addition to sending data to the servers, this function also holds the responsibility of updating the parity block for each physical block row. To do this, first the function must retrieve the previous data block the new data will replace as well as the previous parity block. The function recognizes if either of the servers are disconnected or either data/parity blocks are corrupted and reconstructs the data using the other servers. This is done the same way as described in the Repair() function in Shell.py. Then, a new parity block is constructed using the old data.

First, the parity block is sent to its corresponding server using SinglePut() and then the new data block. If either of those SinglePuts() detect a server disconnection, a message is printed on the client's side to know which block caused the server disconnection.

Get()

This is the main function called by all other files to receive data from a server. The input to this function is just the virtual block number. Similar to Put(), the first step is to convert the virtual block to the physical block and server address. Then, all the server must do is call SingleGet() to retrieve the data. If SingleGet() returns the error code for a disconnected server or corrupt block, this function prints a message to alert the client of the issue and reconstructs the data using the XOR method described in Repair().

SinglePut()

Called by Put(). Takes in physical block number, server number, and data as input. First, this function checks if the input block number is within a valid range. Then, it used at-most-once semantics to attempt to call the servers' Put() function to store data in the respective server. The exception for this try:exception attempt is "ConnectionRefusedError", which is the error returned by the xmlrpc library when a server fails to connect. This data is then stored in the client's cache under the respective server and block number.

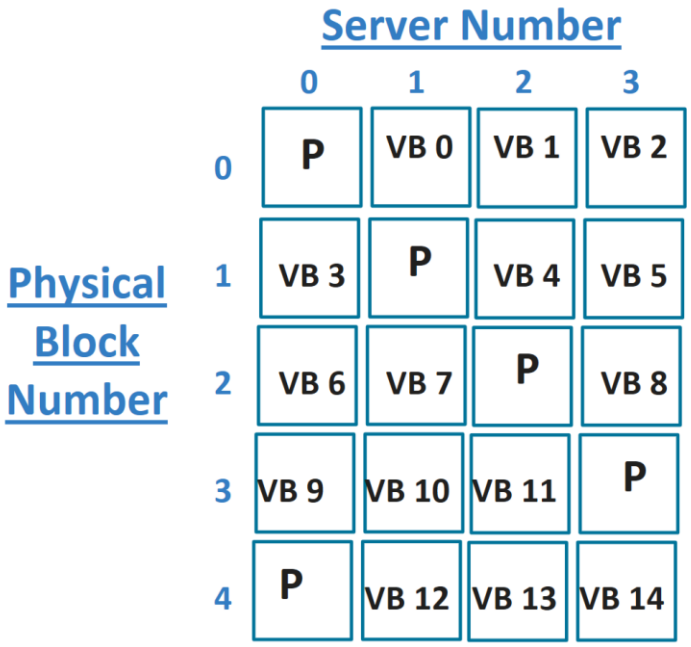
Since the client is writing to the server, it should update the block in the RSM server which denotes which client wrote last. To do this, we must again translate the virtual block number to a physical block number and server, but for the block which stores the last writer. At-most-once semantics are used again to Put() this value into the server.

SingleGet()

Called by Get(). Takes in physical block number and server number as input. Returns error code for server disconnect or data corruption if it occurs. This function also checks if the input block number is within a valid range, but it also checks if there's data in the cache for the respective server and block number. If so, this value is returned. Otherwise, at-most-once semantics are used to retrieve data from the block number of the given server and either the data is returned or an error code for server disconnection or data corruption.

VirtualToPhysical()

This function converts virtual block numbers to physical block numbers and the server number, as well as the server for the parity block. This is achieved using the floor divide and modulo python operators. The parity blocks are distributed sequentially for each block number. For a system with 4 servers, the parity blocks are distributed like so, where P is a parity block and VB # is Virtual Block #:



RSM()

RSM is similar to the previous homeworks, with the exception that locks are stored in the penultimate block of the last server and at-most-once semantics are used.

CheckAndInvalidateCache()

CheckAndInvalidateCache is similar to the previous homework, with the exception that the last writer block is calculated as the total number of blocks over the number of servers – 2, and the method of initializing the cache is slightly different due to it being a 2D matrix of defined size.

Blockserver.py

This file implements the functions servers use to interface with the clients. In addition to the altered functions from the previous homeworks, this file now also allows for the -cblk command argument, which corrupts the block number passed in when starting the server.

Put()

The Put() function in blockserver handles Put requests sent to a server. On each call of this function, the data is stored in the server's block number that's passed to the function, along with a checksum which is generated and stored in a dict. For each checksum, the MD5 checksum generator (from the hashlib library) is initialized, generated, and stored.

If cblk is set to a value and the block number is equal to that value, the first five bites of the data are replaced with random information to corrupt the data. The checksum stays the same, however, so when the checksum is recalculated again on a Get(), it'll detect the corruption.

Get()

The Get() function also reinitializes the checksum object, recalculates the checksum based on the data being retrieved, and compares that checksum to the stored checksum for the data block. If the two checksums are not equal, data corruption has occurred, and instead of returning data, the function returns a CORRUPTED BLOCK error code.

Before comparing checksums, the function first determines if a checksum is stored for the requested data block. This functionality exists because at times, clients request blocks from the server before they are written to.

Shell.py

The only change made to shell.py is the addition of the repair() command.

Repair()

When called, the repair() command first considers whether the input server number is a valid integer or within a valid range. Then, the function iterates through each block within the total number of blocks for each server, divided by the number of servers. For example, if the clients' total number of blcoks is 256, the servers hold 64 blocks each.

On each iteration, the function reconstructs the block of the missing server by XORing the data from the corresponding blocks of the remaining servers and stores the reconstructed block into the new server. This process isn't instant, so a progress bar was added to show the process of restoring the server.

Fsmain.py

The only difference in this file is the inclusion of the additional command arguments logcache, startport, numservers when starting the client.

Fsconfig.py

This file initializes the additional command line arguments mentioned above. The number of servers is assumed to be 4 if no argument is provided, the start port for the servers is assumed to be 8000, and logcache is assumed to be disabled.

Evaluation

We tested our program thoroughly after each implementation of new features. When making changes to block.py, we ran a series of commands on each level of RAID that we were able to implement. For all levels of RAID, we first confirmed that the file system was able to abstract the block level away and function as it previously did.

For RAID 0, we checked that the number of available blocks was equal to the product of the block size of each server multiplied by the block size of each server. We also checked that data was being distributed equally between each of the blocks. Using showblock().

For RAID 1, the intention was for all puts and gets to be mirrored across all disks. To confirm this, Get()s pulled data from all servers and did a check that the values were identical before returning.

For RAID 4, similar to RAID 0, we confirmed that the blocks were written evenly across. We also checked that the parity data was being written to the designated to the correct server. This was done by printing the calculated parity server on each operation.

For RAID 5, we also checked that the servers were being written evenly across. What was different for RAID 5 compared to 4 was that we needed to confirm the diagonal pattern of the parity serves. This was also done with print statements.

The final test for the RAID5 was whether it could mask failures in the file system. This was done in Get(), where if a server disconnected or had a corrupt block, SingleGet() would fail and we could automatically return the correct data when creating, appending, and cat'ing files.

We also had to test if the clients could detect disconnected servers and corrupted blocks. To do this, we simply started a file system using different numbers of servers, from n = 4 to 8. We then disconnected servers to see if the client could detect them. We tested this with servers in several different positions to ensure that it would not, for example, always neglect the 3rd server.

To check that the file system could detect corrupt blocks, we followed a similar process. Using two clients, and a server with a corrupt block, we tested that the servers could correctly detect the corrupt block and notify the user. This was done with -cblk on different servers as well as on different blocks.

With both of these, we also tested that if a new server was loaded (without cblk), then the repair function would then return all data properly without any warnings.

We also checked that the checksums were indeed incorrect for corrupted blocks.

We conducted these tests on the same system multiple times to see that it wouldn't fail at any point.

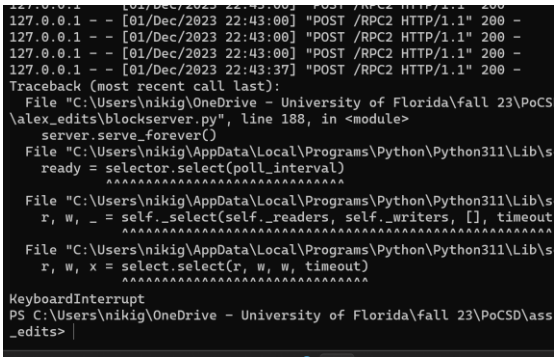
<u>Below is a demonstration of the file system in action:</u>

- First we start by starting servers 0-7 and clients 0-1

Client 0	Client 1
	RSM SERVER: 7
1	[cwd=0]%create f1
RSM SERVER: 7	[cwd=0]%create f2
[cwd=0]%append f1 1	[cwd=0]%
Successfully appended 1 bytes.	[cwd=0]%
[cwd=0]%append f2 2	2 [cwd=0]%
Successfully appended 1 bytes.	[cwd=0]%
[cwd=0]%	[cwd=0]%cat f2
[cwd=0]% 3	2
[cwd=0]%	[cwd=0]%cat f2
[cwd=0]%	SERVER DISCONNECTED GET 9
[cwd=0]% 4	2
[cwd=0]%repair 3	[cwd=0]%
Repairing block: 31 of 31	[cwd=0]% 5
[cwd=0]%	[cwd=0]%cat f2
[cwd=0]%	2
[cwd=0]% 6	[cwd=0]%repair 3
[cwd=0]%	Repairing block: 31 of 31
[cwd=0]%cat f2	[cwd=0]%
CORRUPTED_BLOCK 9	[cwd=0]%
2	[cwd=0]%
[cwd=0]%append f2 345	[cwd=0]% 7
CORRUPTED_BLOCK 9	[cwd=0]%
CORRUPTED_BLOCK 9	[cwd=0]%
Successfully appended 3 bytes.	[cwd=0]%
[cwd=0]%	[cwd=0]%cat f2
[cwd=0]%	CORRUPTED_BLOCK 9
[cwd=0]% 8	2345
[cwd=0]%	[cwd=0]%repair 3
[cwd=0]%	Repairing block: 31 of 31
[cwd=0]%cat f2	[cwd=0]%
2345	9
[cwd=0]%	

- Step 1: Client 1 creates 2 files
- Step 2: Client 0 appends to those files
- Step 3: Client 1 reads from those files and sees the changes

Then, the server is disconnected:



- Step 4: Client 1 attempts to read again, returns server disconnect error
- Step 5: The server is restarted and client 0 repairs the server
- Step 6: Client 1 reads from the server to show that it was successfully repaired. The server is then started again with a corrupted block and repaired by client 1:

```
_edits> python blockserver.py -port 8003 -nb 64 -bs 128 -cblk 1
Running block server with nb=64, bs=128 on port 8003
```

- Step 7: Client 0 attempts to read, but finds the block is corrupted. Client 0 writes to the block anyways.
- Step 8: Client 1 reads from that block and still sees the updated information, despite that block being corrupted. Client 1 repairs the server.
- Step 9: Client 0 reads from the server and receives no errors.

In a separate test, logcache is set to 1:

```
[cwd=0]%append f1 1
CACHE_MISS 20
CACHE_INVALIDATED
CACHE_MISS 1
CACHE_MISS 1
CACHE_MISS 2
CACHE_MISS 1
CACHE_MISS 1
CACHE_MISS 1
CACHE_MISS 0
CACHE_MISS 0
CACHE_MISS 0
CACHE_MISS 0
CACHE_MISS 0
CACHE_WRITE_THROUGH 0
CACHE_WRITE_THROUGH 0
CACHE_MISS 2
CACHE_MISS 2
CACHE_MISS 2
CACHE_WRITE_THROUGH 2
CACHE_WRITE_THROUGH 2
CACHE_MISS 1
CACHE_MISS 1
CACHE_MISS 1
CACHE_WRITE_THROUGH 1
CACHE_WRITE_THROUGH 1
Successfully appended 1 bytes.
CACHE_WRITE_THROUGH 63
[cwd=0]%
```

Reproducibility

Exact commands we used for a system with 4 servers and 2 clients:

Servers 0-3:

python blockserver.py -port 8000 -nb 64 -bs 128 # graders might need to change python to python3

python blockserver.py -port 8001 -nb 64 -bs 128

python blockserver.py -port 8002 -nb 64 -bs 128

python blockserver.py -port 8003 -nb 64 -bs 128

Clients 0-1:

python fsmain.py -ns 4 -startport 8000 -nb 256 -bs 128 -is 16 -ni 16 -cid 0 -logcache 0

python fsmain.py -ns 4 -startport 8000 -nb 256 -bs 128 -is 16 -ni 16 -cid 1 -logcache 0

General commands you can use

To use the file system, the user must first initialize n servers using the command in n separate terminals:

python3 blockserver.py -nb -bs -port -cblk

Where -nb, -bs and –port are followed by integer values. -nb, -cblk and -bs can be undefined, in which case they will take on default values. n must be at least 4 and no more than 8. Each server must have an equal number of blocks and must have sequential ports. We generally numbered the ports starting from 8000 up to 8007.

Next, the user must start the file system. This is done using the command:

python3 fsmain.py -ns -startport -nb -bs -ni -is -cid -logcache

Similar to the previous part, the aguments should be followed by integer values. Additionally, all but

-startport and -cid can be left undefined so that they take default values. Leaving these two arguments undefined can lead to problems when using multiple clients and servers.

To reproduce the tests we did, the user can create multiple combinations of the above and use create(), append(), cat(), and mkdir() to demonstrate the basic functionality and correctness of the file system. AS described in the pervious section, user may use showblock() to confirm the correct raw data in each block.

These tests can then be completed using the -cblk argument in blockserver.py to test the detection and masking of corrupted blocks.

Separately, the user can test for the detection of a disconnected server by closing out one of the server terminals and checking if the filesystem can mask the failure.

The repair() function can be tested after closing a server, which is done after either simulating a disconnected server or closing out a corrupted server. After closing the server, a new server with the same parameters must be run using the command above. The repair() function

Conclusions

Implementing the RAID 5 file system, complete with error masking and repair procedures was a great challenge but proved to be very useful in demonstrating the concepts behind creating a redundant and failure-tolerant file system. It showed us the importance of being able to locate the failure with checksums in addition to using error correction codes to mask and/or repair the failure. The project also

demonstrated how either of the aforementioned strategies, error correction codes and checksums, are unable to deal with failures on their own. For example, using an ECC would not be useful without the checksum as we would not know which block needed to be corrected. Likewise, a checksum on its own cannot repair a failure after locating it.

The most challenging aspects of the project were the proper data storage patterns of each of the RAID levels, ensuring that the RAID levels all worked properly, and using the hashlib to check for errors in corrupted blocks.