POCSD Project Technical Report

Kevin McGrath
12/3/21

1. Introduction and Problem Statement

This project is a client/server file system in which the blocks for a client are stored across N servers where N can be specified by the user when they initialize the client. This file system also implements RAID-5 which distributes both the data and parity blocks across multiple servers to evenly balance the load across the servers.

Through the redundancy implemented by RAID-5, this filesystem can handle corrupt blocks and fail-stops on a single server. By detecting these failures and handling them by using the parity blocks and XOR operations, the filesystem can update and recover blocks from a server that failed. Furthermore, there is a 'repair -server_ID' command that, once the server is back up and running, can recover all the blocks for that server and return the system to its usual functionality.

2. Design and Implementation

2.2. Memoryfs_client.py

In memoryfs_server.py I added command-line arguments "-sid" and "-cblk" to input the server_ID and corrupt block number, respectively. The server_ID isn't used for much other than to display the integer representing the server, but the corrupt block is important because it specifies which block to emulate decay for.

The main modification to the code from Design 3 was implementing the checksums and emulated decay. For checksums I used a dictionary (self.checksums{}) which I update on every Put() with the md5 checksum of the data. Then, in the Get() function I retrieve the checksum from the dictionary and check if the block number is equal to the corrupt block number. If it is, I overwrite the block data with corrupt data (0xFF for all the bytes), and then recalculate the checksum and store it in a variable. The "test" is then compared to the checksum stored in the dictionary and if they're not equal then a checksum error is returned by the server to the client. If they are equal, then the data is returned as usual.

```python
def Get(block_number):
  # Read block specified by block_number
  result = RawBlocks.block[block_number]
  test = RawBlocks.checksums.get(block_number)
  # Corrupt data if block_number = cblk
  if(block_number == CORRUPT_BLOCK_NUMBER):
    RawBlocks.block[block_number] = bytearray(b'\xFF') * BLOCK_SIZE
    test = CalcCheckSum(RawBlocks.block[block_number])

  # Validate checksum
  if test != RawBlocks.checksums.get(block_number):
    return CHECKSUM_ERROR

  return result

server.register_function(Get)

def Put(block_number, data):
  # Store data
  RawBlocks.block[block_number] = bytearray(data.data)
  # Compute and store a checksum for the data
  RawBlocks.checksums[block_number] = CalcCheckSum(RawBlocks.block[block_number])
  print('The checksum for block# ' + str(block_number) + ' is ' + str(RawBlocks.checksums[block_number]))
  return 0

server.register_function(Put)
```

Figure 1. Put() and Get() functions in memoryfs_server.py.

2.2. Memoryfs_client.py

The bulk of this project was implemented in the client and I'm going to explain the changes made in subsections corresponding to those described in the canvas assignment.

2.1.1. Handle N Block Servers

To handle N block servers, I simply store the port arguments into an array and then iterate through each server to initialize an array of servers with corresponding ports.

```python
# list to hold ServerProxy objects for all the servers
self.servers = []

# initialize clientID
if 0 <= args.cid < MAX_CLIENTS:
    self.clientID = args.cid
else:
    print('Must specify valid cid')
    quit()

# initialize XMLRPC client connection to raw block servers

# store number of servers argument
if 0 <= args.ns <= MAX_SERVERS:
    self.numServers = args.ns
else:
    print('Must specify valid number of servers')
    quit()

# list of the server ports provided as arguments
self.ports = [args.port0, args.port1, args.port2, args.port3, args.port4,
            args.port5, args.port6, args.port7]

# create a url for each server and use that to create various block servers
for i in range(0,self.numServers):
    if self.ports[i]:
        server_url = 'http://' + SERVER_ADDRESS + ':' + str(self.ports[i])
        self.servers.append(xmlrpc.client.ServerProxy(server_url, use_builtin_types=True))
        print(server_url)
    else:
        print('Must specify port number for each of the ' + str(self.numServers) + ' servers')
        quit()
```

Figure 2. Initializing N block servers based on command-line arguments.

2.2.2 Distribute Blocks Across Multiple Servers for Put() and Get()

To implement RAID-5 I needed to map virtual blocks to physical data and parity blocks and followed the suggested implementation of creating two separate mapping functions.

For mapping to data blocks I handle everything pretty much the same as RAID-4 with taking the modulus to get the server and using division to get the block number for that server. The main difference is I check if the server I'm mapping to is greater than or equal to the server where I'm storing the parity for that block. If it is then I just need to add 1 to the server_ID and that will ensure the mapping for the rest of the blocks in that row will be correctly mapped as well.

Mapping the parity blocks was just about keeping the same mapping for the physical blocks as in the data mapping. I knew I had to make the server_ID translation so that the parity blocks would form a diagonal, so I basically use the physical block number instead of the virtual block number since that's what affects which server the parity block is stored on.

```python
### RAID5
def VirtualToPhysicalData(self, virtual_block):

    parityServer, parityBlock = self.VirtualToPhysicalParity(virtual_block)

    server_ID = virtual_block % (self.numServers-1)
    physical_block_number = virtual_block // (self.numServers-1)
    if server_ID >= parityServer:
        server_ID += 1

    logging.debug('Virtual Block ' + str(virtual_block) + ' mapped to DATA (Server ' + str(server_ID) + ', Block ' + str(physical_block_number) + ')')

    return server_ID, physical_block_number

def VirtualToPhysicalParity(self, virtual_block):

    physical_block_number = virtual_block // (self.numServers-1)
    server_ID = (self.numServers - 1) - (physical_block_number % (self.numServers))
    logging.debug('Virtual Block ' + str(virtual_block) + ' mapped to PARITY (Server ' + str(server_ID) + ', Block ' + str(physical_block_number) + ')')

    return server_ID, physical_block_number
```

Figure 3. Helper functions used to map virtual data and parity blocks to physical data and parity blocks.

The blocks are stored and retrieved with Put() and Get() like before, but now these functions use the helper functions to translate the virtual blocks to a server-block pair and call ServerPut() or ServerGet() which handle the extra logic of generating parity and handling failures when calling a server's Put() and Get() functions.

```python
## Put: interface to write a raw block of data to the block indexed by block number
## Blocks are padded with zeroes up to BLOCK_SIZE
def Put(self, virtual_block, block_data):

    logging.debug(
        'Put: block number ' + str(virtual_block) + ' len ' + str(len(block_data)) + '\n' + str(block_data.hex()))
    if len(block_data) > BLOCK_SIZE:
        logging.error('Put: Block larger than BLOCK_SIZE: ' + str(len(block_data)))
        quit()

    if virtual_block in range(0, TOTAL_NUM_BLOCKS):
        # ljust does the padding with zeros
        putdata = bytearray(block_data.ljust(BLOCK_SIZE, b'\x00'))
        # get physical server and block numbers from virtual block number
        dataServer, dataBlock = self.VirtualToPhysicalData(virtual_block)
        # put data in (server_ID, blocknumber)
        self.ServerPut(dataServer, dataBlock, virtual_block, putdata)

        return 0
    else:
        logging.error('Put: Block out of range: ' + str(virtual_block))
        quit()

## Get: interface to read a raw block of data from block indexed by block number
## Equivalent to the textbook's BLOCK_NUMBER_TO_BLOCK(b)
def Get(self, virtual_block):

    logging.debug('Get: ' + str(virtual_block))
    if virtual_block in range(0, TOTAL_NUM_BLOCKS):

        # Translate virtual
        server, block = self.VirtualToPhysicalData(virtual_block)

        return self.ServerGet(server, block)

    logging.error('Get: Block number larger than TOTAL_NUM_BLOCKS: ' + str(virtual_block))
    quit()
```

Figure 4. Modified Put() and Get() functions that take in a virtual block number, translate it to a server-block pair, and then call the ServerPut() and ServerGet() functions for that server-block pair.

2.2.3 Implement the Logic to Deal with Failures

The first type of failure is a corrupt block which means the block decayed and the data there is no longer valid which I emulated with the -cblk command as previously mentioned. If the data read from a block is equal to CHECKSUM_ERROR, then I log that that block is corrupt and recover it using the self.RecoverBlock() function and return the recovered data.

The second type of failure is a fail-stop which occurs when a server is disconnected while the client is using it. I use try-except clauses to detect when a server crashes and after that I check if the server equals FAILED_SERVER so that I don't have to wait for the RPC to timeout every time when I already know that server is offline.

Handling a fail-stop in ServerGet() is straightforward – you read from the server in a try clause and if it's failed then in the except clause, I set the global FAILED_SERVER flag (default value is -1) to the integer value of the server, log the failure, and use self.RecoverBlock() to recover the data from the failed server.

Handling a fail-stop in ServerPut() is a bit more complicated since you also must worry about generating parity. First, I generate parity for the virtual block and then I attempt to store that parity in the try clause. If that fails then in the except clause, I set FAILED_SERVER = parityServer and log the failure. If it passes then I attempt to store the new block data and if that fails I set FAILED_SERVER = dataServer and then log the error.

It makes sense to check the parity first because if the parity is stored successfully but the Put() to the data block fails, I already generated the parity so I don't have to do anything else. If more than 1 server could fail at a time then I would have to change this implementation but since the parityServer and dataServer of a virtual block are always different and we're assuming only 1 server can fail at a time this is fine.

```python
def ServerPut(self, dataServer, dataBlock, virtual_block, block_data):

    global FAILED_SERVER

    # Map virtual block to it's parity block
    parityServer, parityBlock = self.VirtualToPhysicalParity(virtual_block)
    # ljust does the padding with zeros
    putdata = bytearray(block_data.ljust(BLOCK_SIZE, b'\x00'))
    if dataServer == FAILED_SERVER:
        logging.debug('FAILSTOP ON SERVER [' + str(FAILED_SERVER) + ']')
    # Complete a write during failstop by generating parity
        parity = self.GenerateParity(virtual_block, block_data)
        self.servers[parityServer].Put(parityBlock, parity)
    elif parityServer == FAILED_SERVER:
        logging.debug('FAILSTOP ON SERVER [' + str(FAILED_SERVER) + ']')
        self.servers[dataServer].Put(dataBlock, putdata)
    else:
        # generate parity for new block data
        parity = self.GenerateParity(virtual_block, block_data)
        # Handle failstop on parity put
        try:
            self.servers[parityServer].Put(parityBlock, parity)
        except ConnectionRefusedError:
            FAILED_SERVER = parityServer
            logging.debug('Failstop on server ' + str(FAILED_SERVER))
        # Handle failstop on data put
        try:
            # store new block data
            self.servers[dataServer].Put(dataBlock, putdata)
        except ConnectionRefusedError:
            FAILED_SERVER = dataServer
            logging.debug('Failstop on server ' + str(FAILED_SERVER))

        return 0

```

Figure 5. ServerPut() implementation.

```python
# Lower-level get for physical server and block numbers
def ServerGet(self, server, block):

    global FAILED_SERVER

    if server == FAILED_SERVER:
        logging.debug('FAILSTOP ON SERVER [' + str(FAILED_SERVER) + ']')
        data = self.RecoverBlock(server, block)
    else:
        # check for failstops and handle them appropriately
        try:
            data = self.servers[server].Get(block)
        except ConnectionRefusedError:
            FAILED_SERVER = server
            logging.debug('FAILSTOP ON SERVER [' + str(FAILED_SERVER) + ']')
            data = self.RecoverBlock(server, block)
    # handle corrupted blocks
    if data == CHECKSUM_ERROR:
        if FAILED_SERVER >= 0:
            logging.debug('Cannot recover corrupt block due to failstop on another server')
            logging.debug('CORRUPT BLOCK: Server = ' + str(server) + ' Block = ' + str(block))
        else:
            logging.debug('CORRUPT BLOCK [' + str(block) + '] ON SERVER [' + str(server) + ']')
            data = self.RecoverBlock(server, block)

```

Figure 6. ServerGet() implementation.

My RecoverBlock() function is really simple since I realized whether the block you're recovering is parity or data, you still end up doing an XOR of every server except the one you're recovering for.

For GenerateParity() I just translate a virtual block into (server, block) pairs for the data and parity blocks and then XOR the old data with the new data and XOR the result of that with the old parity to generate the new parity which I then return since I handle the actual storing of the parity in ServerPut().

```
## Recovers data for a specific (server, block) pair
def RecoverBlock(self, server, block_number):

    recovered = bytearray(BLOCK_SIZE)

    logging.debug('Recovering Server [' + str(server) + ']  Block [' + str(block_number) + ']')

    # XOR other servers
    for i in range(0, self.numServers):
        # Don't include server we're recovering for
        if i != server:
            block = bytearray(self.ServerGet(i, block_number))
            recovered = bytearray(np.bitwise_xor(recovered, block))

    logging.debug('Recovered: ' + str(recovered.hex()))

    return recovered

## Generate parity for new data
def GenerateParity(self, virtual_block, newData):

    # Translate virtual
    dataServer, dataBlock = self.VirtualToPhysicalData(virtual_block)
    parityServer, parityBlock = self.VirtualToPhysicalParity(virtual_block)
    # read old data block
    oldData = self.ServerGet(dataServer, dataBlock)
    # read parity block
    oldParity = self.ServerGet(parityServer, parityBlock)
    # pad new data
    newData = bytearray(newData.ljust(BLOCK_SIZE, b'\x00'))
    # XOR new data with old data
    dataXOR = bytearray(np.bitwise_xor(oldData, newData))
    # XOR result of previous XOR with the parity block to get the new parity
    newParity = bytearray(np.bitwise_xor(dataXOR, oldParity))
    # store the newly geneated parity block
    newParity = bytearray(newParity.ljust(BLOCK_SIZE, b'\x00'))

    return newParity
```

Figure 7. RecoverBlock() and GenerateParity() functions used in ServerGet() and ServerPut().

2.2.4 Implement the Repair Procedure

The last major component of the client is the Repair() function which reconnects to the specified server that had previously failed and recovers all the blocks. For this function I reset the FAILED_SERVER flag to -1 so that the rest of my system knows there isn't a failure stop anymore, and then read the total number of blocks from the server I'm recovering and use that in my loop range(). I could've also just assumed that all servers would be initialized with the same size, but I wanted to add this extra functionality. Within the loop I iterate through all the blocks of the server, recover the block data for that server, and store it in the server.

```python
# Repair: reconnects to server_ID, and regenerates all blocks for server_ID using data from the other servers in the array
def Repair(self, server_ID):

    global FAILED_SERVER

    # Reconnect to server [server_ID]
    server_url = 'http://' + SERVER_ADDRESS + ':' + str(self.ports[server_ID])
    self.servers[server_ID] = (xmlrpc.client.ServerProxy(server_url, use_builtin_types=True))
    logging.debug('Reconnected server [' + str(server_ID) + '] to port [' + str(self.ports[server_ID]) + ']')
    # Reset FAILED_SERVER to -1
    FAILED_SERVER = -1
    # Get the size of the server
    SERVER_SIZE = self.servers[server_ID].GetServerSize()
    # Regenerate all blocks for server [server_ID]
    for i in range(0, SERVER_SIZE):
        recovered_block_data = self.RecoverBlock(server_ID, i)
        self.servers[server_ID].Put(i, recovered_block_data)
        # logging.debug('Recovered block [' + str(i) + ']')
        # logging.debug(recovered_block_data.hex())
```

Figure 8. Repair function in the client that reconnects to server and recovers data.

2.3. Memoryfs_shell_rpc.py

The only real changes to the shell were adding the command-line arguments for the numbers of servers and specifying ports, as well as the "repair" command for the interpreter and the corresponding function which calls a repair function in the client.

```python
ap.add_argument('-ns', '-ns', type=int, help='an integer value')
ap.add_argument('-port0', '--port0', type=int, help='an integer value')
ap.add_argument('-port1', '--port1', type=int, help='an integer value')
ap.add_argument('-port2', '--port2', type=int, help='an integer value')
ap.add_argument('-port3', '--port3', type=int, help='an integer value')
ap.add_argument('-port4', '--port4', type=int, help='an integer value')
ap.add_argument('-port5', '--port5', type=int, help='an integer value')
ap.add_argument('-port6', '--port6', type=int, help='an integer value')
ap.add_argument('-port7', '--port7', type=int, help='an integer value')

ap.add_argument('-nb', '--total_num_blocks', type=int, help='an integer value')
ap.add_argument('-bs', '--block_size', type=int, help='an integer value')
ap.add_argument('-ni', '--max_num_inodes', type=int, help='an integer value')
ap.add_argument('-is', '--inode_size', type=int, help='an integer value')
ap.add_argument('-cid', '--cid', type=int, help='an integer value')

```

Figure 9. Command-line arguments for the number of servers and the ports.

```python
def Interpreter(self):
  while (True):
    command = input("[cwd=" + str(self.cwd) + "]:")
    splitcmd = command.split()
    if len(splitcmd) == 0:
      continue
    elif splitcmd[0] == "repair":
      if len(splitcmd) != 2:
        print("Error: repair requires one argument")
      else:
        self.repair(splitcmd[1])
    elif splitcmd[0] == "cd":
      if len(splitcmd) != 2:
        print ("Error: cd requires one argument")
      else:
        #self.FileObject.RawBlocks.Acquire()
        self.cd(splitcmd[1])
        #self.FileObject.RawBlocks.Release()
```

```python
# implements repair
def repair(self, server_ID):
  try:
    server_ID = int(server_ID)
  except ValueError:
    print('Error: ' + server_ID + ' not a valid Integer')
    return -1
  logging.info('Initiating repair on server ' + str(server_ID) + '...')
  self.FileObject.RawBlocks.Repair(int(server_ID))
  logging.info('Repair complete!')
```

Figure 10. Interpreter command for repair function and the shell repair function which does some error checking and calls a client-side repair function.

3. Evaluation

3.1. Testing

To test my program, I ran N = 4 to N = 8 servers and would create some files and directories by copying and pasting commands from a text document, crash one of the servers, ls and cat a few times and check the log to see if the fail-stop was detected and handled correctly, then repair the server and check the log again to make sure everything worked as normal and that I recovered all the blocks for that server. I then repeated that process for all the servers I was running and made sure to use as many commands as possible (ln, cat, append, cd, chdir, mkdir, create).

After testing the fail-stops I moved on to testing corrupt blocks by starting a server with the "-cblk" command included and then either doing some operations or just calling "showblock block#" and checking the log file to make sure the corruption was detected and that the block was recovered.

3.2. Evaluation

I added a "showload" command to my shell which will display the load on each server as well as the average. Screenshots of this command being used can be seen below. I created a text file that makes 8 directories in the root inode and then in each directory creates a file and appends 210 bytes. To meet the requirements for testing with multiple block sizes and file sizes I made another text file that creates 2 files in each directory and appends 210 bytes to each file.

```
[cwd=8]:showload
# Requests = 1214
[cwd=8]:
```

```
Server [0] requests = 221
Server [1] requests = 129
Server [2] requests = 361
Server [3] requests = 491
Average Load (requests/server) = 300
[cwd=8]:
```

1. **Block Size = 256 bytes, 8 files with 210 bytes each**

```
[cwd=8]:showload
# Requests = 1830
[cwd=8]:
```

```
Server [0] requests = 314
Server [1] requests = 186
Server [2] requests = 717
Server [3] requests = 689
Average Load (requests/server) = 476
[cwd=8]:
```

2. **Block Size = 256 bytes, 16 files with 210 bytes each**

```
[cwd=8]:showload
# Requests = 1447
[cwd=8]:
```

```
[cwd=8]:showload
Server [0] requests = 160
Server [1] requests = 321
Server [2] requests = 571
Server [3] requests = 434
Average Load (requests/server) = 371
[cwd=8]:
```

3. **Block Size = 128 Bytes, 8 files with 210 bytes each**

```
[cwd=8]:showload
# Requests = 2467
[cwd=8]:
```

```
[cwd=8]:showload
Server [0] requests = 276
Server [1] requests = 498
Server [2] requests = 1241
Server [3] requests = 627
Average Load (requests/server) = 660
[cwd=8]:
```

4. **Block Size = 128 Bytes, 16 files with 210 bytes each**

As you can see from the screenshots, my filesystem succesfully balances the load fairly evenly between the servers. Although the virtual blocks are theoretically mapped completely evenly, in the actual implementation some servers will have higher loads than others because they contain the root inode or multiple directory blocks. Regardless of this, the load for even the most used server is still much less than that of a single-server file system.

4. Reproducibility

1. To run the filesystem, you first need to start running a minimum of 4 servers. You can run N servers (MAX N = 8) by running these commands for each server you'd like in separate terminals:

   python memoryfs_server.py -bs 128 -nb 256 -port 8000 -sid 0
   python memoryfs_server.py -bs 128 -nb 256 -port 8001 -sid 1

python memoryfs_server.py -bs 128 -nb 256 -port 8002 -sid 2
...
python memoryfs_server.py -bs 128 -nb 256 -port XXXX -sid N

2. Now that you have your servers up and running you need to run the shell with this command:
   memoryfs_shell_rpc.py -ns 4 -port0 8000 -port1 8001 -port2 8002 -port3 8003 -nb 768 -bs 128 -is 32 -ni 32 -cid 0

Note: The total number of blocks for the filesystem (which you specify in command 2.) is equal to the number of blocks in each server multiplied by one less than the total number of servers, since 1/4th of the blocks are parity blocks. For example: 768 = 256 blocks/server * (4-1) servers.

3. To test the emulated decay, you can modify a command from 1. for one of the servers to include –“cblk block_number” to specify which block on that server is corrupt. You can then open the log file and search for “CORRUPT” and there will be statements showing a checksum error was detected and the block was recovered successfully.

4. To test the fail-stop functionality, you can simply ctrl+c one of your server terminals and then perform some shell operations like making directories and creating/appending to files, and then check the log file and search “FAIL-STOP” to find if your command resulted in the fail-stop being tolerated. I also like to create some files/directories before I cause a fail-stop just to make sure that I’m able to recover both new and old data.

5. After you’ve tested the fail-stop toleration you can go back to the terminal you ctrl+c’d and re-run that server with the same command you used before (must use same port). Then you can run the command “repair server_ID” in the shell and the log file will spit out each block that it’s recovered for the repaired server. After this your filesystem will be back to normal and you can even try causing a fail-stop on a different server.

5. Conclusion

Through this project I learned about client/service, networking, and fault tolerance as well as best practices for testing a complex system and debugging non-trivial errors that couldn’t be accurately traced back as they were occurring in the server.

Another big takeaway from this project was applying virtualization by mapping virtual blocks to physical block and making my system modular by using generic functions for each component I needed. This made my code a lot easier to understand and helped a ton with finding errors.

All in all, this project reinforced the key concepts we’ve been covering all semester and was a challenging yet enjoyable learning experience.