Distributed File System

EEL5737 - Principles of Computer System Design

Sagnik Ghosh

December 5, 2020

1 Introduction

We have extended the Unix File System in a incremental approach to a distributed File System. In the phase I and phase II, we have implemented the layered design of File System with various Unix commands in python. In phase III, we introduced multiple client accessing the File System. We have used locking mechanism of the file server, so that it appears to the client that only it has access to it. The goal of the 3rd phase was to solve various race conditions.

2 Problem Statement

The goal of the phase IV of Design Assignment is to distribute load across multiple servers. Also introducing redundancy to achieve fault tolerance. The File System should continue to work when a server goes down or data block gets corrupted in a server.

3 Design and Implementation

There is 3 layer of the implementation of Distributed File System.

3.1 Shell

memoryfs_shell_rpc.py

This class represents the shell where the user communicate with the server. Initializes the file system, and provides the user with functionalities to invoke the commands.

ls : Listing components of current working directory
cd : change directory
cat : show content of a file
mkdir : create a directory

create : create a file
ln : create a link to the specified path
exit : to exit the shell and stop execution

3.2 Client

memoryfs_client.py

This provides the user the file system, hiding the implementation details of multiple servers. The blocks of the file system is internally distributed across multiple servers. The function InitializeMap maps the actual blocks to blocks across different servers and stores the mapping in a Python Dictionary called block_map. The dictionary is indexed on actual block number and the value is a tuple of server number, block number and parity server. The server number and block number in the tuple to the location the actual block is mapped to. The parity server is the corresponding parity server for the block.

Algorithm 1: Algorithm for predefined mapping of blocks

```
Result: Assigns Mapping for Blocks to block_map
$number\_of\_dataservers \leftarrow number\_of\_servers - 1$
$number\_of\_blocks\_per\_server \leftarrow$
  $TOTAL\_NUM\_BLOCKS/number\_of\_servers$
if $TOTAL\_NUM\_BLOCKS\%number\_of\_servers == 0$ then
    $number\_of\_blocks\_per\_server \leftarrow$
      $number\_of\_blocks\_per\_server + 1$
end
$parityserver \leftarrow number\_of\_dataservers$
for $block\_number\_begin$ in range
  $(0, TOTAL\_NUM\_BLOCKS, number\_of\_dataservers)$ do
    if $parityserver$ = -1 then
        $parityserver \leftarrow number\_of\_dataservers$
    end
    for $block\_number$ in range
      $(block\_number\_begin, block\_number\_begin +$
      $number\_of\_dataservers)$ do
        $server\_number \leftarrow block\_number\%number\_of\_dataservers$
        $block\_number\_on\_server \leftarrow$
          $\lfloor block\_number/number\_of\_dataservers \rfloor$
        if $server\_number \geq parityserver$ then
            $server\_number \leftarrow server\_number + 1$
        end
        $map\_tuple \leftarrow \{"id" : server\_number, "block\_number" :$
        $block\_number\_on\_server, "parity\_server" : parityserver\}$
        $block\_map[block\_number] \leftarrow map\_tuple$
    end
    $parityserver \leftarrow parityserver - 1$
end
```

The implementation of Get and Put which goes across RPC to server, is modified to handle a corrupted block or unavailability of a server. When server responds to a block as corrupted or refuses to connect, recover_Get is called to recover the data.

Algorithm 2: Algorithm for recover get

```
Result: Recovers a failed Get
$xorddata \leftarrow$ bytearray of zeros
$block\_number\_on\_server \leftarrow$ the block that failed/corrupted
$server\_number \leftarrow$ the server that is unavailable / has corrupted block
for $server\_iterator$ in range ($number\_of\_servers$) do
    if $server\_iterator = server\_number$ then
        continue
    end
    $existdata \leftarrow$
     $block\_server[server\_iterator].Get(block\_number\_on\_server)$
    for $i$ in range ($BLOCK\_SIZE$) do
        $xorddata[i] \leftarrow xorddata[i] \oplus existdata[i]$
    end
end
return xordata
```

3.3 Server

memoryfs_server.py

Along with block to store data, introduced a new data structure called checksum. Checksum array stores the checksum for corresponding data block. Server blocks are initialized with bytearray of zeros.

```python
def Put(block_number, data):
    RawBlocks.block[block_number] = data
    # store checksum only when it's not a corrupted block
    if block_number not in corrupted_blocks:
        RawBlocks.checksum[block_number] =
        hashlib.md5(data.data).digest()
    return 0

def Get(block_number):
    result = RawBlocks.block[block_number]
    # Server is initialized with bytearray of zeros,
    # no need to check checksum as data is not relevant
    if isinstance(result, bytearray):
        return result
    if RawBlocks.checksum[block_number] !=
    hashlib.md5(result.data).digest():
        return -1
    return RawBlocks.block[block_number]
```

When the data is of type bytearray, we need not do any check for match of checksum. The implementation of decay of a block is difficult, therefore, we used third argument in the command line for mentioning the corrupt block while starting the server. In case of the block which is gonna be corrupt, we are not storing the checksum for that.

4 Evaluation

I ran the program using 5 servers to evaluate the performance. Parameters used for testing.

```
BLOCK_SIZE = 64
INODE_SIZE = 64
MAX_INODE_BLOCK_NUMBERS = (INODE_SIZE - 8) // 4 = 14
```

Below we represent the allocation of blocks across various servers. We create a file, keep appending data to it. Initially we append 9 bytes, then we append 73, 59, 117, 173, 105 and 290 bytes respectively.

File Size is in bytes, followed by number of blocks and distributed across separate servers.

File Size	Total Blocks	Server 0	Server 1	Server 2	Server 3	Server 4
9	1	0	0	1	0	0
82	2	0	0	1	0	1
141	3	1	0	1	0	1
258	5	1	1	1	1	1
431	7	2	1	1	1	2
536	9	2	1	2	2	2
826	13	2	2	3	3	3

We can as we keep appending the data, the file size increase, so the file system client keeps allocating new blocks to it. And these new blocks distributed across all the servers. So the loads keeps getting distributed evenly. The server numbers can be seen on a cat command on the log.

Next we represent the actual block number and their mapping to server and block number in the respective servers. As we see from the table (on next page) below, the virtual block number allocated are in sequence, but these are distributed across the 5 actual servers working underneath.

5 Reproducability

We ran the 5 servers on separate terminals with the following commands. The structure of the command line inputs: python3 [file_name] [port_number] [corrupted_block_number]

Block Number	Server Number	Block Number on Server
26	2	6
27	4	6
28	0	7
29	1	7
30	3	7
31	4	7
32	0	8
33	2	8
34	3	8
35	4	8
36	1	9
37	2	9
38	3	9

```
python3 memoryfs_server.py 8000
python3 memoryfs_server.py 8001
python3 memoryfs_server.py 8002
python3 memoryfs_server.py 8003
python3 memoryfs_server.py 8004 2
```

Now run the shell for memoryfs with command

```
python3 memoryfs_shell_rpc.py 5 localhost:8000 localhost:8001
localhost:8002 localhost:8003 localhost:8004
```

Now we create a file and append content to it.

```
[cwd=0]:create Sagnik.txt
[cwd=0]:append Sagnik.txt asdasd123
Successfully appended 9 bytes.
[cwd=0]:cat Sagnik.txt
asdasd123
```

We can see the error message in log.

```
INFO:root:Getting Blocks:  block_number 11 server number 4 block no in server 2
ERROR:root:Get: Corrupted Block 11
```

But we see the content and the file system continues to work fine. Now we stop the server hosted on port 8004. And continue executing the following commands.

```
[cwd=0]:mkdir Bob
[cwd=0]:ls
[3]:./
[1]:Sagnik.txt
[1]:Bob/
```

```
[cwd=0]:cd Bob
[cwd=2]:ls
[1]:./
[3]:../
[cwd=2]:create Kalpak.txt
[cwd=2]:ls
[2]:./
[3]:../
[1]:Kalpak.txt
[cwd=2]:append Kalpak.txt sakjdhakdsbl1233123
Successfully appended 19 bytes.
[cwd=2]:cat Kalpak.txt
sakjdhakdsbl1233123
[cwd=2]:create Amit.txt
[cwd=2]:ls
[3]:./
[3]:../
[1]:Kalpak.txt
[1]:Amit.txt
[cwd=2]:append Amit.txt sdlajal312
Successfully appended 10 bytes.
[cwd=2]:ls
[3]:./
[3]:../
[1]:Kalpak.txt
[1]:Amit.txt
[cwd=2]:cat Amit.txt
sdlajal312
```

Now we see a few interesting log messages, which tells us that the client failed to communicate to server 4, which is hosted on port 8004.

```
DEBUG:root:InodeNumberToInode: 4
DEBUG:root:Get: 7
INFO:root:Getting Blocks:  block_number 7 server number 4 block no in server 1
INFO:root:Get Failed for server: 4
DEBUG:root:InodeNumberToInode : inode_number 4 raw_block_number: 7 slice start: 0 end: 16
DEBUG:root:tempinode: 0000000a000100010000000e00000000
DEBUG:root:Read: file_inode_number: 4, offset: 0, count: 128
DEBUG:root:InodeNumberToInode: 4
DEBUG:root:Get: 7
INFO:root:Getting Blocks:  block_number 7 server number 4 block no in server 1
INFO:root:Get Failed for server: 4
```

But we see the file system still continues to execute even when a server is not working.

6 Conclusions

Even though the final requirement of designing a UNIX file system distributed across various servers looks daunting but incremental design approach through design phases made it impossibly easy. This proves the fundamental design principle of layering and modularity. The first two phases were focused on making the file system work with a single user using shell for interaction. The next step focused on RPC calls and distributing the file system across client and server. The final step was to virtualize the data blocks in the client to data blocks distributed across different servers. The parity blocks which is also distributed, applied redundancy in data, when a server is unavailable or a block is corrupted, we can use the parity to reconstruct the data. Thus, we could made the file server fault tolerant.