Design/implementation

Your goal in the project is to distribute and store data across multiple data servers to: 1) reduce their load (i.e., distributing requests across servers holding replicas), 2) provide increased aggregate capacity, and 3) increase fault tolerance.

Your redundant block storage should follow the general approach described for RAID-5 in class. You can use the client/server file system of design assignment homework #4 solution as a starting point for this project. Use integers to identify your servers, and configure the system so that there is a total of N servers - your design must work with at least N=4 and up to N=8 servers (the file servers may run on the same computer).

As part of the design, your system must distribute data and parity information across servers, at the granularity of the block size specified in your configuration file. Your design must also store block checksum information to allow for you to detect single-block errors. You may use 128-bit MD5 as checksums, and store checksums in a dedicated data structure in the server.

For reads: when there are no failures, your design should allow for load-balancing, distributing requests across the servers holding data for different blocks – i.e., for a large file consisting of B blocks, you should expect to have on average B/N requests handled by each server. "Small" reads of a single block should be able to be served by a single server. Data integrity should be detected on reads by first using checksums, without accessing a parity server. If data in a single server/block is detected as corrupt, you should use the other server's blocks and parity to correct the error.

For writes: writes should attempt to update *both* the data and parity block. Follow the approach described in class to compute the new parity from old data, old parity, and new data. If acknowledgments from both data and parity blocks return successfully, your write can complete successfully and the client returns. If only one acknowledgment is received, your system should register the server that did not respond as failed, and continue future operations (reads/writes) using only the remaining, non-faulty servers. Your design should be able to tolerate a fail-stop failure of a single server. Note that if a server has failed, you won't be able to write new blocks to it - but writes can still complete successfully by using the remaining disks - again, recall how RAID-5 uses XOR of the old data, new data and parity block to generate a new parity block.

Repair: you must also implement a simple process of repairing/recovery when a server that crashed is replaced by a new server with blank blocks. The repair procedure should work as follows: in the shell, when you type the command "repair server_ID", the client locks access to the disk, reconnects to server_ID, and regenerates all blocks for server_ID using data from the other servers in the array.

memoryfs_client.py

The main changes you need to implement in the client, compared to homework #4, are:

1) Implement the logic to handle N block_servers, instead of a single server, each with its own endpoint (see below - these are initialized by memoryfs_shell_rpc.py)

2) Implement the logic to distribute blocks across multiple servers for Put() and Get(). This should follow the RAID-5 approach.

3) Implement the logic to deal with the two types of failures: corrupt block (Get() returns an error), and fail-stop (server is down and does not respond to a Put() or Get())

4) When a corrupt block or server disconnected, print out the following messages, respectively, where block_number is the corrupted blocks' number, operation is PUT, GET, or RSM, and serverid is the disconnected server:
CORRUPTED_BLOCK block_number
SERVER_DISCONNECTED operation serverid
Examples:
CORRUPTED_BLOCK 100
SERVER_DISCONNECTED PUT 3

5) Implement the logic to support RSM() by using a *single server* as a special case (use server id 0). This means that all RSM() requests will hit the single server 0, and failure of this server will cause failure of your system. This is ok - bootstrapping RSM across distributed blocks is more complex (can you guess why?), and we will not get into this complexity in the project.

6) Implement the repair procedure

memoryfs_server.py

The main changes you need to implement in the block server, compared to homework #4, are:

1) Allocate another data structure to store checksum information (a block can store multiple checksums - see above)

2) Store checksum on a Put()

3) Verify checksum on Get() and RSM() - if the stored checksum does not match the computed checksum, return an error

4) Expose a block number to be damaged with an emulated decay as an optional command-line argument "cblk". This block, on a Get(), returns an error, emulating corrupted data (the likelihood of an actual corrupted in-memory data block is so small that this needs to be emulated).

For example:

python memoryfs_server.py -bs 128 -nb 256 -port 8000 -cblk 100

should set your server to listen on port 8000, and any Get() for block number 100 should return a checksum error, whereas:

python memoryfs_server.py -bs 128 -nb 256 -port 8000

listens on port 8000, and no Get() will return a checksum error

Make sure your client prints out "CORRUPTED_BLOCK " followed by the block number when the server responds with a checksum error

memoryfs_shell_rpc.py

The main changes you need to implement in the shell, compared to homework #4, are:

1) A command-line argument "ns" specifying N (the number of servers).

2) Command-line argument startport specifying the port of the first server (i.e. server id 0); the remaining servers must be listening to ports startport+1, startport+2, ... , startport+N-1

For example, if you have four servers running on localhost (see above how to specify server ports):

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

Hints

- to preserve existing abstraction and leverage modularity, you can rename the Get(), Put(), and RSM() methods (e.g. SingleGet(), SinglePut(), SingleRSM()) and build your new RAID logic in new Get(), Put(), and RSM() methods. Note that for RSM() it should return RSM_UNLOCKED only if *all* disks in the RAID set return RSM_UNLOCKED

- once again, develop in small steps and test them thoroughly before moving to the next. This will also ensure you get partial credit if you don't finish the entire project. A suggested sequence is:
  1) implement checksum handling on the server-side, and return error when checksums don't match (-cblk)
  2) implement abstraction layer for Get(), Put(), RSM() on client side (see above) and detect/print CORRUPTED_BLOCK
  3) implement support to connect to multiple servers on the client side (hint: you can use a dict for block_server)
  4) implement RAID-0, where data is Put() to all servers, RSM() to server 0, and Get() from any server
  5) expand it to support detection when a server is disconnected and failover to another server (use at-most-once, see below)
  6) expand it to support detection of a checksum error and failover to another server
  7) expand to RAID-4, where one server is the parity server

8) expand to RAID-5, where the parity is distributed
9) implement the repair procedure

- unlike in HW#4, you should switch to at-most-once for detecting a server that is disconnected, i.e. fail fast without wait/retry

- come up with a function that maps a virtual block number to a (server,physical_block_number) to help implement the RAID-ified Get(), and a virtual block number to a (server,physical_block_number) for parity to help implement the RAID-ified Put()

- on each server, the -cblk argument refers to the physical block number in that server

- the -nb size on the client should be the total number of useable blocks in your system. for example, if you have RAID-5 with N=5 servers, and each server has -nb 256, on the client-side you should use -nb 1024 (i.e. 256*(5-1))

- You can use the ^ operator to implement xor in Python. The trick is, you need to do byte-by-byte in your byte array. For instance, to compute C = A xor B when these are byte arrays:

```
for i in range(BLOCK_SIZE):
  C[i] = A[i] ^ B[i]
```

Performance Analysis

[conduct a performance analysis of your design, comparing quantitatively the average load (i.e. number of requests handled per server), of your design compared to the baseline case of single-server (homework #4), for at least two different block sizes and two different file sizes.

Final report guidelines

For the final report, it is expected to be a longer technical report-style document than in previous assignments, including the following sections:

1) Introduction and problem statement
Describe in your own words what your project accomplishes, and motivate the decisions made

2) Design and implementation
Include a detailed description of your design and how you decided to implement everything from the corruption of data, virtual-to-physical block translation, and handling failures. Use subsections for highlighting the major changes for each of the python programs.

3) Evaluation
Describe how you tested your program and, how you evaluated the load distribution.

4) Reproducibility
Describe step by step instructions for how to use and run your file system and if possible, include how to run the tests you used to verify your code.

5) Conclusions