ECE 599/CS 579 PoCSD
Project assignment

1. Introduction and background

The goal of this design and implementation assignment is to assess your understanding of the application of design principles discussed in class to implement new functionality to improve the performance of the client/service file system of assignment #4. Specifically, you will significantly extend the file system to support distributed storage and fault tolerance following an approach conceptually similar to RAID (Redundand Array of Inexpensive Disks), but applied to our client/server file system. This project is intended only for the students enrolled in the graduate sections of the class.

2. Code overview

Your starting point for this assignment is the solution codebase for assignment #4 that has been released to you.

3. Design requirements

In the last design assignment, you built a client/server based file system where clients accessed data blocks stored at a server. In this final project, you will extend your file system to multiple servers with support for redundant block storage. The project will expose you to practical issues in the design of systems including: client/service, networking, and fault tolerance.

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.

Sor this design, assume there is only one client in the system. This will simplify the design and allow you to focus on the server distribution aspect of the project.

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 must use the other server's blocks and parity to correct the error such that, from the file system's perspective, the error is masked

For writes:

- Writes 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 must flag the server that did not respond as failed, and continue future operations (reads/writes) using only the remaining, non-faulty servers
- Your design must 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

Repairs:

- 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 works 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.

For full credit:

- Make sure your implementation prints out the following status messages to the screen (exactly these strings):
    - CORRUPTED_BLOCK block_number
    - SERVER_DISCONNECTED operation block_number

4. Implementation

block.py:
The main changes you need to implement in the client, compared to assignment #4, are:

- Implement the logic to handle N block_servers, instead of a single server, each with its own endpoint (see below - these are initialized by block.py)
- Implement the logic to distribute blocks across multiple servers for Put() and Get(). This should follow the RAID-5 approach.
- 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())

- When a corrupt block is accessed, or when a server is detected to be disconnected, print out the following messages, respectively, where block_number is the virtual block number of a corrupted/unavailable block, and operation is PUT, GET:
    - CORRUPTED_BLOCK block_number
    - SERVER_DISCONNECTED operation block_number
- Examples:
    - CORRUPTED_BLOCK 100
    - SERVER_DISCONNECTED PUT 3
- Modify the RSM() function to always return success. Because we are assuming there is only one client in the system, and to simplify the design, this modification is safe. Essentially, the RSM() function does not need to contact the server since the single-writer is guaranteed by the specification of a single client only. This simplifies the design significantly as you don't need to worry about enforcing before-or-after atomicity across multiple servers when failures occur
- Implement the repair procedure

blockserver.py:
The main changes you need to implement in the server, compared to assignment #4, are:

- Allocate another data structure to store checksum information
- Compute and store checksum on a Put()
- Verify checksum on Get() - if the stored checksum does not match the computed checksum, return an error
- Implement an emulation of block decay
    - To do this, expose a block number to be correupted with an emulated decay as an optional command-line argument "cblk"
        - on each server, the -cblk argument refers to the *physical block number in that server*
    - 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:
      python3 blockserver.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:
      python3 blockserver.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*!

fsmain.py:
The main changes you need to implement in command-line parsing, compared to assignment #4, are:

- A command-line argument "ns" specifying N (the number of servers)
- A command-line argument "startport" specifying the port of the first server (i.e. server id 0); the remaining servers must be listening to ports in consecutive numbers, i.e. startport+1, startport+2, ... , startport+N-1
- The -nb size on the client command line must be the total number of useable blocks in your system, e.g. 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))

For example, if you have N=5 servers running on localhost (see above how to specify server ports), these commands will start the servers and the client. The total number of useable blocks by the client will be 1024:

```
python3 blockserver.py -bs 128 -nb 256 -port 8000
python3 blockserver.py -bs 128 -nb 256 -port 8001
python3 blockserver.py -bs 128 -nb 256 -port 8002
python3 blockserver.py -bs 128 -nb 256 -port 8003
python3 blockserver.py -bs 128 -nb 256 -port 8004
python3 fsmain.py -ns 5 -startport 8000 -nb 1024 -bs 128 -is 16 -ni 16 -cid 0
```

shell.py:
The main change you need to implement in the shell, compared to assignment #4, is the implementation of a repair command, which reconstructs the data for a server that has failed and has been restarted with empty blocks. The argument given to the repair command is the ID of the server to be repaired, starting from 0. For example, with the 4-server configuration above:

```
% repair 0      <- this repairs the contents of server 0 (listening on port 8000)
% repair 3      <- this repairs the contents of server 3 (listening on port 8003)
```

Hints:

- To preserve existing abstractions and leverage modularity, you can rename the Get() and Put() methods (e.g. SingleGet(), SinglePut()) and build your new RAID logic in new Get(), Put(), methods.
- 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:
    - implement checksum handling on the server-side, and return error when checksums don't match (-cblk)
    - implement abstraction layer for Get(), Put() on client side (see above) and detect/print CORRUPTED_BLOCK
    - implement support to connect to multiple servers on the client side (hint: you can use a dict for block_server)
    - implement RAID-1, where data is Put() to all servers, and Get() from any server
    - expand it to support detection when a server is disconnected and failover to another server (use at-most-once, see below)
    - expand it to support detection of a checksum error and failover to another server
    - expand to RAID-4, where one server is the parity server
    - expand to RAID-5, where the parity is distributed
    - implement the repair procedure
- Differently from assignment #4, you must use at-most-once semantics to detect that a server that is disconnected - i.e. fail fast, without timeout/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()

- 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 A, B, and C are byte arrays of BLOCK_SIZE:

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

5. Performance analysis

Conduct a performance analysis of your design, and quantitatively compare 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, using files that have at least 2*N blocks in size, where N is the number of servers.

6. What to turn in

Upload to Canvas:

- A file Project.pdf with your final project report, which is a longer technical report-style document (6-8 pages) 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
    - Summarize the performance analysis described in Section 5
  - 4) Test and reproducibility
    - Describe how you tested your design
    - I will also test your design, so make sure your code follows the specification of the command-line arguments as above
      - To be on the safe side, copy and past instructions on how you ran your code for your tests
  - 5) Conclusions
- A file Project.zip with:
  - Your modified block.py, blockserver.py, fsmain.py, and shell.py