

Design assignment 4 (class project)

[Start Assignment](#)

Due Friday by 11:59pm **Points** 100 **Submitting** a file upload
File Types pdf and zip **Available** Aug 23 at 12am - Dec 3 at 11:59pm 3 months

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.

What to turn in on Canvas

- A file Project.pdf with a comprehensive report on your design (see guidelines below for what to include/how to structure the report)
- A file Project.zip with:
 - A file memoryfs_client.py (see guidelines below for functionality and command-line arguments)
 - A file memoryfs_server.py
 - A file memoryfs_shell_rpc.py

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 your client/server file system of design assignment homework #3 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 you may store checksums for multiple data blocks in a single checksum block (e.g. a 128-byte block can store 128-bit MD5 checksums for eight blocks).

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 #3, 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.
Hint: 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()
- 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) Implement the repair procedure

memoryfs_server.py

The main changes you need to implement in the block server, compared to homework #3, are:

- 1) Expose the server_ID as command-line argument sid

- 2) Allocate extra blocks to store checksum information (a block can store multiple checksums - see above)
- 3) Store checksum on a Put()
- 4) Verify checksum on a Get() - if the stored checksum does not match the computed checksum, return an error
- 5) 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 -sid 0 -cblk 100
```

should set your server ID 0 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 -sid 0
```

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

memoryfs_shell_rpc.py

The main changes you need to implement in the shell, compared to homework #3, are:

- 1) A command-line argument "ns" specifying N (the number of servers).
- 2) Command-line arguments port0, port1, ... port7 specifying the port of the (up to 8) servers.

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

```
memoryfs_shell_rpc.py -ns 4 -port0 8000 -port1 8001 -port2 8002 -port3 8003 -nb 256 -bs 128 -is 16 -ni 16 -cid 0
```

Performance Analysis

[Required for EEL5737 students only; not required for EEL4736 students]: 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 #3), for at least two different block sizes and two different file sizes.

Report.pdf 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, for EEL-5737 students, 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