# Design/implementation

1. Implement artificial delay in memoryfs_server.py to model a server/network slow to respond

memoryfs_server.py

- Added CL arg. `-delayat` using argument parser.
- Added `request_count` variable
  - fixed scoping issue with `global`
- Modified Get and Put
  - increments request_count
  - checks if Nth request (request_count % delayat)=> sleep for 10 seconds

2. Implement at-least-once semantics for memoryfs_client.py

memoryfs_client.py

- Created a loop to retry self.block_server.Get, Put, and RSM requests
  - used try except else syntax
  - Exception *socket.timeout* => print SERVER_TIMED_OUT
  - Exception *ConnectionRefusedError* => print SERVER_DISCONNECTED
  - if any exception occured, it would sleep for RETRY_INTERVAL
  - if no exception occured the loop was broken

3. Implement an in-disk key/value store for memoryfs_server.py

memory_server.py

- Used argparser for CL args:
  - `-initdbm`: an integer flag
    - 1 => initializes zero blocks and writes to dbm
    - 0 => reads from dbm and puts into RawBlocks
  - `-dbmfile` : a string that names the file used by dbm in disk
- updated Put and RSM to update dbm if dbmfile specified

4. Implement a client-side cache for file data blocks read (Get()) in the Read() function in memoryfs_client.py

- Created a `block_cache` dictionary in Diskblocks

  - maps block_number to bytearray

- Created a `inode_cache` dictionary in Diskblocks

  - maps inode_number to gencnt

- Incremented gencnt in FileName Unlink and Write functions before StoreInode()

- In FileName.Read()

  - If the gencnt match
    - If the block is present in cache

            - Get blocks from the block_cache
            - print CACHE_HIT
        - If the block is not present in the cache, Get() from the server and store in the cache
    - If the gencnt do not match
        - invalidate all cache entries for this file inode
        - print CACHE_INVALIDATED

**concern** if there is a hard link, eg f1 => b0, b1 and f2 => b0 and f2 changes, and f2 is modified, shouldn't the cache be

## Testing Methods

- At-least-once behavior when timeouts and server disconnections occur

    - Timeouts:
        - started server with delayat
        - ensured client request recovered to shell after SERVER_TIMED_OUT
    - Server disconnections and persisting data across server restarts:
        - started server with dbm and client
        - created files/directories
        - stopped server
        - made client request
        - restarted server from dbm
        - ensured client request recovered to shell after SERVER_DISCONNECTED

## Assignment questions

**Q1)** In the code given to you, the Acquire() and Release() calls are placed around operations such as cat and append to ensure they run exclusively in one client at a time. What is one example of a race condition that can happen without the lock? Simulate a race condition in the code (comment out the lock Acquire()/Release() in the cat and append functions, and place sleep statement(s) strategically) to verify, and describe how you did it.

After commenting out the Acquire and Release statement for the append, I placed a sleep statement before self.FileObject.Write on line 211 of memory_shell_rpc.py. This allowed me to simulate reading the inode with 2 clients concurrently. Since they then wrote to the same offset, the overwrote one another which is a race condition as that isn't how append's intended functionality.

**Q2)** What happens when you don't store the data in disk using dbm on the server, and terminate/restart the server?

Without using dbm to sture data in disk, terminating/restarting the server loses all the files because they were stored in memory and not persistent storage.

**Q3)** What are the changes that were made to the Get() and Put() methods in the client, compared to the HW#3 version of the code?

Compared with the HW3 versions of the code, the Get() and Put() methods were modified to make xmlrpc calls to a server instead of modifying a diskblocks object directly with procedure calls.

**Q4)** At-least-once semantics may at some point give up and return (e.g. perhaps the server is down forever). How would you implement this in the code (you don't need to actually implement; just describe in words)

At-least-once semantics may choose to give up to prevent fate sharing when a server is down forever using either a timeout or request limit. Each request (Get, Put, RSM) can be modified so that the loop for retries doesn't continue if a server hasn't responded after a set time period or exceeds a set request limit.