

Design Report
Programming Assignment 2: File Sharing Server

**In Fulfillment
of the Requirements
of COEN 346**

by
Justin Pasztor ID: 40281784
Sai Hanish Pagadala ID: 40264290

Concordia University
November 2025

Data Structures and Functions

The UML diagram in Figure 1 shows the main classes that were modified while completing this assignment. The legend shows which variables or functions that were added (highlighted in yellow) or modified (highlighted in pink). The modifications and additions will be described below as follows:

FEntry

- Public Methods:
 - **setFirstBlock(short firstBlock)**: setter function for modifying which data block a file begins with.

FNode

- Public Methods:
 - **getBlockIndex()**: returns the block index of an FNode.
 - **getNext()**: returns the index of the next block of data in the file.
 - **setNext(int next)**: setter function for modifying the next block of data in the file.

FileSystemManager

- Fields:
 - **fentryTable**: array of file entries (FEntry) that holds metadata for all files.
 - **fnodesTable**: array of FNodes acting as a linked list to build the block chains for each file's content.
 - **rwLock**: lock that enables multiple simultaneous reader access and exclusive writer access.
 - **readLock**: reader-side lock used for readFile() and listFiles().
 - **writeLock**: write-side lock used for mutual exclusion during critical section of creating, writing, and deleting files.
 - **FENTRY_BYTES**: fixed number of bytes required to store one FEntry on the disk.
 - **FNODE_BYTES**: fixed number of bytes required to store one FNode on the disk.
 - **fentryRegionBytes**: variable to store the computed size of the region reserved for all FEntries on the disk.
 - **fnodeRegionBytes**: variable to store the computed size of the region reserved for all FEntries on the disk.
 - **metadataBytes**: variable that stores the total size of all combined metadata regions.
 - **firstDataBlockIndex**: Offset in the disk where the first actual data block begins.
 -
- Methods:
 - Private Methods:

- **computeMetaDataBytes ()**: calculates the sizes of FEntry/FNode regions and initializes the **metadataBytes** and **firstDataBlockIndex** variables.
- **findFile(String name)**: searches **fentryTable** for a file based on the passed file name, otherwise returns -1 if not found.
- **findFreeSlot()**: searches for a free slot in **fentryTable** for creating a new file.
- **offsetOffFNode(int j)**: returns the exact byte offset on disk where the j-th FNode is stored.
- **blocksNeeded(int n)**: calculates how many data blocks are required to store n-bytes of user content.
- **findFreeDataBlocks(int n)**: finds n free data blocks and returns their indices (used during file writing).
- **findFreeNodeIndecies(int n)**: helper function to find n free FNodes to build the linked block chain.
- **rebuildFreeBlocks()**: function to rebuild the **freeBlockList** when loading an existing filesystem.
- **loadMetaData()**: loads all FEntries and FNodes from disk into memory at startup
- **saveMetaData()**: writes all FEntries and FNodes from memory back to the disk after updates.

- Public Methods:

- **FileSystemManager(String filename, int totalSize)**: constructor that initializes the metadata tables, computes region sizes, loads existing disk metadata or formats a new filesystem.
- **createFile(String fileName)**: creates a new file on the disk with name **filename**
- **writeFile(String filename, byte[] contents)**: writes **contents** into the file specified by **filename**.
- **readFile(String filename)**: reads file contents based on the file specified by **filename**.
- **deleteFile(String filename)**: deletes the file specified by **filename**.
- **listFiles()**: lists all existing files on the disk in the console.

FileServer:

- Private Methods:
 - **handleClient(Socket clientSocket)** : Runs all commands (on its own thread) that were sent by a connected client.
- Public Methods:
 - **FileServer(int port, String fileSystemName, int totalSize)** : Initializes a **FileSystemManager** using the **fileSystemName** and **totalSize** parameters and sets up the server to accept client connections.

- **start()** : opens a server socket that begins server execution and continuously accepts new client instances by creating a thread for each client. Each client runs their own **handleClient**.

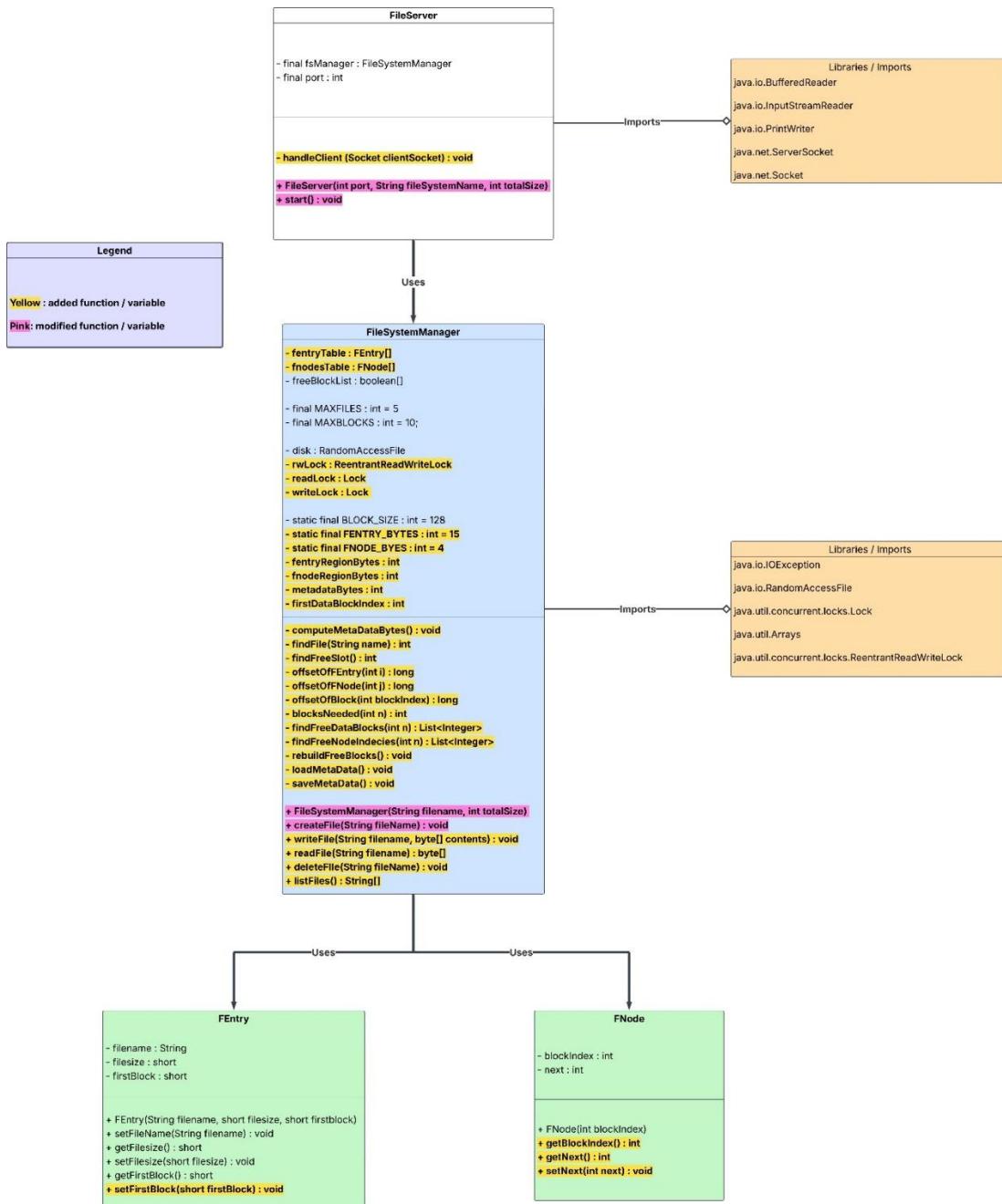


Figure 1: UML Diagram

Algorithms

File System Layout and Initialization

FileSystemManager opens (or creates) the backing disk file using RandomAccessFile in read–write mode and sets its length to the specified total size. It then computes how many bytes are required for the FEntry and FNode arrays and from that derives metadataBytes and firstDataBlockIndex. Metadata always resides at the beginning of the file, and data blocks start at block index firstDataBlockIndex.

The manager allocates in-memory arrays for FEntry, FNode, and freeBlockList, then calls loadMetaData() to populate them from disk if metadata already exists. Finally, it calls rebuildFreeBlocks() to reconstruct the free block bitmap from the metadata.

Following are the implementations of the required functions:

Create

Method: void createFile(String fileName) throws Exception

Algorithm:

1. Validate the filename (non-null, non-empty, within the allowed length).
2. Acquire the write lock.
3. Use findFile to check if the file already exists; if so, throw an exception.
4. Use findFreeSlot to find a free position on the FEntry table; if none is available, throw an exception.
5. Create a new FEntry with size 0 and firstBlock set to -1.
6. Store the new entry in the FEntry table and call saveMetaData() to persist the change.
7. Release the write lock.

Write

Method: void writeFile(String filename, byte[] contents) throws Exception

Algorithm:

1. Use findFile to locate the file; if not found, throw an exception.
2. Acquire the write lock.

3. If contents have length 0, clear the existing block chain for this file by travelling through its FNodes, zeroing/emptying relative data blocks, marking blocks free, cancelling FNodes, and resetting the FEntry size and first block to zero values. Save metadata and return.
4. Compute the number of required blocks from contents length.
5. Use `findFreeDataBlocks` and `findFreeNodeIndices` to ensure enough free blocks and FNodes are available; if not, throw an exception.
6. In a try block, construct a new chain of FNodes and data blocks:
 - For each required block, pick a free node index and a free block index.
 - Seek to the data block offset in the disk, write the corresponding slice of the contents.
 - Create a new FNode, link it to the previous one, and mark the block as used.
7. After finishing writing to the new chain, travel the old chain (if any) and zero out old blocks, free them, and clear the old FNodes.
8. Update the FEntry with the new head FNode index and the new file size, then call `saveMetaData()`.
9. In the catch block, roll back partial allocations by clearing any newly assigned FNodes and blocks, restoring the previous data.
10. Release the write lock.

Read

Method: `byte[] readFile(String filename) throws Exception`

Algorithm:

1. Use `findFile` to locate the file; if not found, throw an exception.
2. Acquire the read lock.
3. If the stored file size is zero or the first block is -1, return an empty byte array.
4. Allocate an output array of length equal to the file size.
5. Starting from the FEntry's firstBlock index, travel the linked FNode chain:
 - For each node, compute the block offset, determine how many bytes remain to be read, and read up to `BLOCK_SIZE` bytes into the output array.
 - Move to the next FNode until all bytes have been read.
6. Release the read lock and return the output array.

The server converts this byte array to a String and sends content to the client. Example: [104, 101, 108, 108, 111] will be converted to “hello”.

Delete

Method: void deleteFile(String fileName) throws Exception

Algorithm:

1. Use findFile to locate the file; if not found, throw an exception.
2. Acquire the write lock.
3. Traverse the FNode chain starting from the file’s firstBlock index. For each node:
 - Zero out the corresponding data block in the disk.
 - Mark the block as free in freeBlockList.
 - Clear the FNode entry from the FNode table.
4. Remove the FEntry for the file by setting its table slot to null.
5. Call saveMetaData() to save the deletion.
6. Release the write lock.

This both removes metadata and overwrites file data to prevent reuse of old contents.

List

Method: String[] listFiles()

Algorithm:

1. Acquire the read lock.
2. Iterate over the FEntry table and collect the filenames of all non-null entries.
3. Convert the collection to a String array and return it.
4. Release the read lock.

The server layer outputs the result either as NO_FILES or as a list of names separated with commas.

Multi-Threading

The server uses a one-thread-per-client model. Each incoming socket connection is handled by a Java thread. The threads are independent and share no client-specific data. All shared data is accessed through FileSystemManager.

FileSystemManager protects its internal shared data using a single ReentrantReadWriteLock. The following policy is used:

- Read operations (readFile, listFiles) acquire the read lock, allowing multiple readers.
- Write operations (createFile, writeFile, deleteFile) acquire the write lock, ensuring exclusive access.

All public methods in FileSystemManager have a straightforward locking pattern: they acquire either the read lock or the write lock at the beginning and release it in a “finally” block. No function holds both locks at the same time, and there is only a single lock object, so there is no risk of deadlock due to inconsistent lock ordering.

Rationale

The chosen design closely matches the assignment requirements by using fixed-size arrays for file entries and file nodes. This design uses one-thread-per-client server model.

A single reader–writer lock contained in FileSystemManager provides synchronization model. It applies to mutual exclusion between writers and readers while allowing concurrent reading. Usage of the lock in FileSystemManager prevents accidental unsynchronized access to shared data.

The writeFile method is organized to only replace the old FNode chain after the new chain has been fully allocated and written. If an error occurs during allocation or writing, the method returns the file system to its previous consistent data by cleaning up partially allocated structures. Delete and create operations follow similar careful patterns, ensuring that metadata and data remain consistent even in the presence of errors.

The design can be extended by increasing MAXFILES and MAXBLOCKS, adding new server commands, or introducing additional metadata. The separation between FileServer and FileSystemManager allows to change the procedure without modifying the basic file system logic.