**Operating Systems and Networks**

# Docs++

*Simply put* - You need to implement a simplified, shared document system (similar in spirit to Google Docs) from scratch, with support for concurrency and access control.

Help Resource

## Deadline : 18 November 2025, 11:59 PM IST

## Doubts document links for Course Project

- Questions
- Answers

## Github Classroom

We will use GitHub classroom for the projects. The instructions for registering your team are given below. Please read all the instructions before proceeding.

1. One teammate has to create the team. They need to click on this link. This opens a page of registered teams.
2. When a team's member visits the link first, they need to create a new team using the team name. Nothing more, nothing less.
3. The remaining members now need to click on this link. This takes you to the list of registered teams. Join your team, which should have your team number as the name.
4. On the next screen, accept the assignment.

Please use the repo that gets created to work on your project. Any piece of work, code or artifact not present in the repo will not be considered for evaluation.

## Introduction

With the recent success of Zuhu, LangOS is determined to enter the competitive document collaboration market. You have been selected as part of their top engineering team to build the first version of their solution and present it to potential investors. The leadership has outlined their vision for the system, summarised below. Your task is to design and implement this system from the ground up, strictly following the provided specifications, within the next

month. The goal is a Christmas launch, so timely delivery is critical for this MVP (*no deadline extensions*). The outcome will impact both the company's future and your career (*with the added bonus of OSN grades, but that's just minor details*).

Good luck!

The system is composed of the following core components:

1. ***User Clients***:
   - Represent the users interacting with the system.
   - Provide the interface for performing operations on files (create, view, read, write, delete, etc.).
   - Multiple clients may run concurrently, and all must be able to interact with the system simultaneously.
2. ***Name Server***:
   - Acts as the central coordinator of the system.
   - Handles all communication between clients and storage servers.
   - Maintains the mapping between file names and their storage locations.
   - Ensures efficient and correct access to files across the system.
3. ***Storage Servers***:
   - Responsible for storing and retrieving file data.
   - Ensure durability, persistence, and efficient access to files.
   - Support concurrent access by multiple clients, including both reads and writes.

At any point, there would be a single instance of the Name Server running, to which multiple instances of Storage Servers and User Clients can connect. The User Clients and Storage Servers can disconnect and reconnect at any time, and the system should handle these events gracefully.

The event of Name Server failure is out of scope for this project. That is, if the Name Server goes down, the entire system is considered down and must be restarted.

## The File

Files are the fundamental units of data in the system, each uniquely identified by a name. Files are restricted to text data only. Every file consists of multiple sentences, and each sentence is made up of words. A sentence is defined as a sequence of words ending with a period (.), exclamation mark (!), or question mark (?). Words within a sentence are separated by spaces. This segmentation needs to be handled by the system, the user should be able to access the

file as a whole. For completeness and in interest of no ambiguity, a word is defined as a sequence of ASCII characters without spaces.

There is no imposed limit on file size or the total number of files, so the system must efficiently handle both small and large documents (which can variably grow after creation also).

Files support concurrent access for both reading and writing. However, when a user edits a sentence, that sentence is locked for editing by others until the operation is complete. This allows multiple users to view or edit the file simultaneously, but prevents simultaneous edits to the same sentence.

## [150] User Functionalities

The users (clients) must be able to perform the following operations:

1. **[10] View files**: User can view all files they have access to. They can view all files on the system, irrespective of the access using "-a" flag. "-l" flag should list files along with details like word count, character count, last access, owner, etc. Note, a combination of flags can be used lke "-al", which should list all the files with details.

```
VIEW  # Lists all files user has access to
VIEW -a # Lists all files on the system
VIEW -l # Lists all user-access files with details
VIEW -al # Lists all system files with details
```

2. **[10] Read a File**: Users can retrieve the contents of files stored within the system. This fundamental operation grants users access to the information they seek.

```
READ <filename> # Prints the content of the complete file
```

3. **[10] Create a File**: Users can create new files, allowing them to store and manage their data effectively.

```
CREATE <filename> # Creates an empty file with name <filename>
```

4. **[30] Write to a File**: Users can update the content of the file at a word level. This operation allows users to modify and append data to existing files.

```
WRITE <filename> <sentence_number>  # Locks the sentence for other users (if no
<word_index> <content> # Updates the sentence at <word_index> with <content>
.
.
.
```

```
<word_index> <content> # User can update the sentence multiple times
ETIRW # Relieves the sentence lock, allowing other users to finally write
```

Few important points to note here:

- After each WRITE completion, the sentence index update. So, care must be taken for ensuring concurrent WRITEs are handled correctly.
- The content may contain characters like period (.), exclamation mark (!), or question mark (?). The system should be able to recognise these sentence delimiters and create seperate sentences accordingly. (Please refer to examples given below, for more clarity).
    - Yes, every period (or question / exclamation mark) is a sentence delimiter, even if it is in the middle of a word like “e.g.” or “Umm... ackchually!”

> *Hint*: For resolving concurrent read-write issues, you may write to a temporary swap file initially, and move the contents to the final file once all updates are complete. You may also consider using locks, semaphores, some algorithmic approach, etc.

1. **[15] Undo Change**: Users can rever the last changes made to a file.

```
UNDO <filename> # Reverts the last change made to the file
```

Note: The undo-es are file specific, and not user specific. So, if user A makes a change, and user B wants to undo it, user B can also do it. The undo history is maintained by the storage server.

1. **[10] Get Additional Information**: Users can access a wealth of supplementary information about specific files. This includes details such as file size, access rights, timestamps, and other metadata, providing users with comprehensive insights into the files they interact with.

```
INFO <filename> # Display details in any convenient format, just that all abo
```

2. **[10] Delete a File**: Owners should be able to remove files from the system when they are no longer needed, contributing to efficient space management. All data like user access should be accordingly updated to reflect this change.

```
DELETE <filename> # Deletes the file <filename>
```

3. **[15] Stream Content**: The client establishes direct connection with the Storage Server and fetches & displays the content word-by-word with a delay of 0.1 seconds between each word. This simulates a streaming effect, allowing users to experience the content in a dynamic manner.

```
STREAM <filename> # Streams the content of the file word by word with a delay
```

Note: If the storage server goes down mid-streaming, an appropriate error message should be displayed to the user.

1. **[10] List Users**: Users can view a list of all users registered in the system.

   ```
   LIST # Lists all users in the system
   ```

2. **[15] Access**: The creator (owner) of the file can provide access to other users. The owner can provide read or write access. The owner can also remove access from other users. The owner always has both read and write access.

   ```
   ADDACCESS -R <filename> <username>  # Adds read access to the user
   ADDACCESS -W <filename> <username>  # Adds write (and read) access to the use
   REMACCESS <filename> <username>  # Removes all access
   ```

3. **[15] Executable File**: Users (with read access) can “execute” the file. Execute, here, means executing the file content as shell commands. The output of the command should be displayed to the user.

   ```
   EXEC <filename> # Executes the file content as shell commands
   ```

Note: The execution must happen on the name server; and the outputs as is should be piped to the client interface

## [40] System Requirements

The system must support the following requirements:

1. **[10] Data Persistence**: All files and their associated metadata (like access control lists) must be stored persistently. This ensures that data remains intact and accessible even after Storage Servers restart or fail.

2. **[5] Access Control**: The system must enforce access control policies, ensuring that only authorized users can read or write to files based on the permissions set by the file owner.

3. **[5] Logging**: Implement a logging mechanism where the NM and SS records every request, acknowledgment and response. Additionally, the NM should display (print in terminal) relevant messages indicating the status and outcome of each operation. This bookkeeping ensures traceability and aids in debugging and system monitoring. Each entry should include relevant information such as timestamps, IP, port, usernames and other important operation details crucial for diagnosing and troubleshooting issues.

4. **[5] Error Handling**: The system must provide clear and informative error messages for all sorts of expected / unexpected failures, including interactions between clients, Name Server (NM), and Storage Servers (SS). Define a comprehensive set of error codes to cover scenarios such as unauthorized access, file not found, resource contention (e.g., file locked for writing) and system failures. These error codes should be universal throughout the system.

5. **[15] Efficient Search**: The Name Server should implement efficient search algorithms to quickly locate files based on their names or other metadata, minimizing latency in file access operations. Furthermore, caching should be implemented for recent searches to expedite subsequent requests for the same data.

> Note: An approach faster than O(N) time complexity is expected here. Efficient data structures like Tries, Hashmaps, etc. can be used.

# [10] Specifications

## 1. Initialisation

1. **Name Server (NM)**: The first step is to initialize the Naming Server, which serves as the central coordination point in the NFS. It is responsible for managing the essential information about file locations and content.

> Note: The IP address and port of the Naming Server can be assumed to be known publicly so that it can be provided to Clients and Storage servers while registering.

1. **Storage Server (SS)**: Each Storage Server is responsible for physically storing the files and interacting with the Naming Server.Upon initialization, the SS sends vital details about its existence to the Naming Server. This information includes: IP address, port for NM connection, port for client connection and a list of files on it.

2. **Client**: Clients on initialisation should ask the user for their username (for file accesses) and pass this information along with its IP, NM port and SS port to the Name Server.

## 2. Name Server

- **Storing Storage Server data**: One of the fundamental functions of the NM is to serve as the central repository for critical information provided by Storage Servers (SS) upon connection. This information is maintained by NM, to later direct data requests to appropriate storage server. As mentioned in specification 2, these lookups need to be efficient.
- **Client task feedback**: Upon completion of tasks initiated by clients, the NM plays a pivotal role in providing timely and relevant feedback to the requesting clients. This is really

important in real-systems where client response latency is pivotal.

### 3. Storage Servers

The Storage Servers are equipped with the following functionalities:

- **Adding new storage servers**: New Storage Servers (i.e., which begin running after the initial initialisation phase) have the capability to dynamically add their entries to the NM at any point during execution. This flexibility ensures that the system can adapt to changes and scaling requirements seamlessly. The initialisation process at the storage server side follows the same protocol as described in Specification 1.
- **Commands Issued by NM**: The Name Server can issue specific commands to the Storage Servers, such as creating, editing or deleting files. The Storage Servers are responsible for executing these commands as directed by the NM.
- **Client Interactions**: Some operations require the client to establish direct connection with the storage server. The storage server is expected to facilitate these interactions as needed.

### 4. Client

Whenever a client boots up, it asks the user for their username. This username is then used for all file access control operations. The system should ensure that users can only perform actions on files they have permissions for, based on their username. This username is relayed to the NM, which stores it along with the client information until the client disconnects.

Clients initiate communication with the NM to interact with the system. Here's how this interaction unfolds:

- Any file access request from the client is first sent to the NM, which locates the corresponding Storage Server hosting that file (one of many), using its locally stored information.
- Depending on the type of operation requested by the client, the NM may either handle the request as a middleman or facilitate direct communication between the client and the appropriate Storage Server. The operations can be broadly categorized as follows:
    - **Reading, Writing, Streaming** : The NM identifies the correct Storage Server and returns the precise IP address and client port for that SS to the client. Subsequently, the client directly communicates with the designated SS. This direct communication is established, and the client continuously receives information packets from the SS until a predefined "STOP" packet is sent or a specified condition for task completion is met. The "STOP" packet serves as a signal to conclude the operation.
    - **Listing files, Basic Info and Access Control** : The NM handles these requests directly. It processes the client's request and retrieves the necessary information from its local

storage. Once the information is gathered, the NM sends it back to the client, providing the requested details without involving any Storage Server.

- **Creating and Deleting Files** : The NM determines the appropriate SS and forwards the request to the appropriate SS for execution. The SS processes the request and performs the specified action, such as creating / deleting the file. After successful execution, the SS sends an acknowledgment (ACK) to the NM to confirm task completion. The NM, in turn, conveys this information back to the client, providing feedback on the task's status.
- **Execute** : The NM requests for information from SS, but the main processing and communication is handled by the NM directly. The NM executes the commands contained within the file and captures the output. This output is then relayed back to the client, providing them with the results of the executed commands.

## [50] Bonus Functionalities (Optional)

- **[10] Hierarchical Folder Structure**: Allow users to create folders and subfolders to organize files. Users should be able to navigate through this hierarchy when performing file operations. Some associated commands that are expected to be implemented are:

```
CREATEFOLDER <foldername> # Creates a new folder
MOVE <filename> <foldername> # Moves the file to the specified folder
VIEWFOLDER <foldername> # Lists all files in the specified folder
```

- **[15] Checkpoints**: Implement a checkpointing mechanism that allows users to save the state of a file at specific points in time. Users should be able to revert to these checkpoints if needed. The following commands are expected to be implemented:

```
CHECKPOINT <filename> <checkpoint_tag> # Creates a checkpoint with the given tag
VIEWCHECKPOINT <filename> <checkpoint_tag> # Views the content of the specified
REVERT <filename> <checkpoint_tag> # Reverts the file to the specified checkpoin
LISTCHECKPOINTS <filename> # Lists all checkpoints for the specified file
```

- **[5] Requesting Access**: Users can request access to files they do not own. The owner of the file can then approve or deny these requests. There is no need of a push-notification mechanism, a simple storing of requests and an owner-side feature to view and approve/reject requests is sufficient.
- **[15] Fault Tolerance**: To ensure the robustness and reliability of the system, the following fault tolerance and data replication strategies need to be implemented:

- **Replication** : Implement a replication strategy for data stored within the system. This strategy involves duplicating every file and folder in an SS in another SS. In the event of an SS failure, the NM should be able to retrieve the requested data from one of the replicated stores. Every write command should be duplicated asynchronously across all replicated stores. The NM does not wait for acknowledgment but ensures that data is redundantly stored for fault tolerance.
- **Failure Detection** : The NM should be equipped to detect SS failures. This ensures that the system can respond promptly to any disruptions in SS availability.
- **SS Recovery** : When an SS comes back online (reconnects to the NM), the duplicated stores should be matched back to the original SS. This ensures that the SS is synchronized with the current state of the system and can resume its role in data storage and retrieval seamlessly.

- **[5] The Unique Factor**: What sets your implementation apart from others? Welp, this is where you showcase your innovation and creativity.

## Examples

> Note: There is no specification on the exact format of the commands. The commands mentioned in the examples are indicative. You may choose to implement them in any format you like, as long as the functionality remains the same.

### Example 1: View File

```
Client: VIEW  # Lists files accessible to the user
--> wowee.txt
--> nuh_uh.txt

Client: VIEW -a  # Lists all files on the system
--> wowee.txt
--> nuh_uh.txt
--> grades.txt

Client: VIEW -l  # Lists files accessible to the user with details
---------------------------------------------------------
|  Filename   | Words | Chars | Last Access Time | Owner |
|-------------|-------|-------|------------------|-------|
| wowee.txt   |   69  |  420  | 2025-10-10 14:32 | user1 |
| nuh_uh.txt  |   37  |  123  | 2025-10-10 14:32 | user1 |
---------------------------------------------------------
```

```
Client: VIEW -al
-------------------------------------------------------
|  Filename  | Words | Chars | Last Access Time | Owner |
|------------|-------|-------|------------------|-------|
| wowee.txt  |   69  |  420  | 2025-10-10 14:32 | user1 |
| nuh_uh.txt |   37  |  123  | 2025-10-10 14:32 | user1 |
| grades.txt |   51  |  273  | 2025-10-10 14:32 | kaevi |
-------------------------------------------------------
```

### Example 2: Read File

```
Client: READ wowee.txt  # Displays the content of the file
OSN assignments are so fun!

I love doing them.          Wish we had more of them.
```

### Example 3: Create File

```
Client: CREATE mouse.txt  # Creates an empty file named mouse.txt
File Created Successfully!

Client: VIEW
--> wowee.txt
--> nuh_uh.txt
--> mouse.txt
```

> **Note:** NS dynamically adds the new file to the list of available files and updates backup SSs. If the file already exists, NS responds with an appropriate error.

### Example 4: Write to a File

```
Client: WRITE mouse.txt 0 # Adding to the start of file
Client: 1 Im just a mouse.
Client: ETIRW
Write Successful!

Client: READ mouse.txt
Im just a mouse.
```

```
Client: WRITE mouse.txt 1  # In essence, appending to the file
Client: 1 I dont like PNS
Client: ETIRW
Write Successful!

Client: READ mouse.txt
Im just a mouse. I dont like PNS

Client WRITE mouse.txt 2  # Caveat, note the lack of delimiter after last senten
ERROR: Sentence index out of range.  # Similarly for word indexes (negative or >

Client: WRITE mouse.txt 1  # Inserting into the second sentence
Client: 3 T-T
Client: ETIRW
Write Successful!

Client: READ mouse.txt
Im just a mouse. I dont like T-T PNS

Client: WRITE mouse.txt 0  # Inserting multiple times into a sentence
Client: 4 deeply mistaken hollow lil gei-fwen # New sentence : Im just a deeply
Client: 6 pocket-sized  # New sentence : Im just a deeply mistaken hollow pocket
Client: ETIRW
Write Successful!

Client: READ mouse.txt
Im just a deeply mistaken hollow pocket-sized lil gei-fwen mouse. I dont like T-

Client: WRITE mouse.txt 1  # Inserting a sentence delimiter
Client: 5 and AAD. aaaah # New sentences : [I dont like T-T PNS and AAD.]* [aaaa
Client: 0 But,  # New sentence : [But, I dont like T-T PNS and AAD.]* [aaaah].
Client: ETIRW
Write Successful!

Client: READ mouse.txt
Im just a deeply mistaken hollow pocket-sized lil gei-fwen mouse. But, I dont li
```

The multiples writes within a single WRITE call, are all considered a single operation. Note this while implementing features like UNDO

Proper error handling should cover:

- Attempting to write without access
- Attempting to write a locked sentence
- Invalid indices
- Updates are applied in order received, so later updates operate on the already modified sentence.

### Example 5: Undo Change

```
Client: READ nuh uh.txt
rm -rf /  # Oops!

Client: WRITE nuh_uh.txt 0
Client: 0 sudo
Client: ETIRW
Write Successful!

Client: READ nuh_uh.txt
sudo rm -rf /

Client: UNDO nuh_uh.txt
Undo Successful!

Client: READ nuh_uh.txt
rm -rf /
```

**Note:**

- Only one undo operation for a file needs to be supported. Multiple undos are beyond the scope of this project.
- Undo operates at the Storage Server level and only reverts the most recent change.
- If the current user is user1 and the most recent modification to the file was made by user2, then an UNDO command issued by user1 should revert user2's last change.

### Example 6: Get Additional Information

```
Client: INFO nuh_uh.txt
--> File: feedback.txt
```

```
--> Owner: user1
--> Created: 2025-10-10 14:21
--> Last Modified: 2025-10-10 14:32
--> Size: 52 bytes
--> Access: user1 (RW)
--> Last Accessed: 2025-10-10 14:32 by user1
```

### Example 7: Delete a File

```
Client: VIEW
--> wowee.txt
--> nuh_uh.txt
--> mouse.txt

Client: DELETE mouse.txt
File 'mouse.txt' deleted successfully!

Client: VIEW
--> wowee.txt
--> nuh_uh.txt
```

### Example 8: Stream File

Same as READ, but the client receives the file content word-by-word (from the storage server) with a delay of 0.1 seconds between each word.

### Example 9: List Users

```
Client: LIST
--> user1
--> user2
--> kaevi
```

### Example 10: Access Control

```
Client: INFO nuh_uh.txt
--> File: feedback.txt
--> Owner: user1
--> Created: 2025-10-10 14:21
--> Last Modified: 2025-10-10 14:32
--> Size: 52 bytes
--> Access: user1 (RW)
--> Last Accessed: 2025-10-10 14:32 by user1

Client: ADDACCESS -R nuh_uh.txt user2
Access granted successfully!

Client: INFO nuh_uh.txt
--> File: feedback.txt
--> Owner: user1
--> Created: 2025-10-10 14:21
--> Last Modified: 2025-10-10 14:32
--> Size: 52 bytes
--> Access: user1 (RW), user2 (R)
--> Last Accessed: 2025-10-10 14:32 by user1

Client: ADDACCESS -W nuh_uh.txt user2
Access granted successfully!

--> File: feedback.txt
--> Owner: user1
--> Created: 2025-10-10 14:21
--> Last Modified: 2025-10-10 14:32
--> Size: 52 bytes
--> Access: user1 (RW), user2 (RW)
--> Last Accessed: 2025-10-10 14:32 by user1

Client: REMACCESS nuh_uh.txt user2
Access removed successfully!

Client: INFO nuh_uh.txt
--> File: feedback.txt
--> Owner: user1
--> Created: 2025-10-10 14:21
--> Last Modified: 2025-10-10 14:32
```

```
--> Size: 52 bytes
--> Access: user1 (RW)
--> Last Accessed: 2025-10-10 14:32 by user1
```

### Example 11: Execute File

Suppose the content of `LMAA0.txt` is as follows:

```
echo "Running diagnostics..."
ls
echo "Done!"
```

```
Client: EXEC LMAA0.txt
Running siagnostics...
--> LMAA0.txt
--> RotT.txt
--> AUG.txt
Done!
```

## Grading

The whole implementation grading can be broadly divided into 4 parts:

1. User Functionalities - 150 marks
2. System Requirements - 40 marks
3. Specifications - 10 marks
4. Bonus Functionalities - 50 marks (Optional)

So, the total marks add up to 250. The further breakdown of marks for each functionality is mentioned in the respective sections above.

## TA Section

### How do I start?

- Define the calls. For example, define the format of the functionality of the read call for the client as `READ path` and then divide the work amongst yourselves with each team member implementing either the client, the naming server or the storage server code.

- Identify things that can be decoupled. For example, the specification where an SS can join the NM at any given moment of execution doesn't depend on neither the client or the SS (Assuming you have figured out how SSs attach themselves at the beginning of the execution). One team member can implement this while the other thinks of ways to implement caching (You probably won't get merge conflicts as you'll be changing different parts of the naming server code)

> Remember Figuring things out is just as important as coding the implementation in this project. Not everyone needs to be programming at once. One could design the redundancy attribute and start coding it up later.

### Some pointers:

- Use TCP sockets
- You may use any *POSIX C library* (opengroup link)
- Use wireshark for debugging TCP communications by inspecting your packets when required.
- You can use netcat to create client/server stubs so that you can start testing your components even if some of it hasn't been coded up.
- As always, decompose the problem and write modular code
- Cite your resources if you take any ideas or code
- Make necessary assumptions

### Resources

- A few slides on the topic
- CMU slides on Distrbuted File Systems
- Rutgers University's resources
- Least Recently Used (LRU) Caching
- Handling multiple clients
- Handling multiple clients without multithreading (This section will be updated on specific requests in the doubts document)
- The Apache Hadoop DFS

### Tips from the TAs *:)*

- Reference Code to get started
- Hints in the doc are merely for providing a direction to one of the solutions. You may choose to ignore them and come up with your own solutions.
- Communicate with your teammates if you cannot fulfill your deliverables on time. That is the entire point of team-work.

- Last, but not the least, START EARLY. This is a big project and will take time and coordination to complete.

### Why are the requirements so trash?

- The requirements might feel very unideal for the usecase at times. However, we would like to highlight that this is a learning exercise (through a usecase) and not a production system. The requirements are designed to make you think about the various aspects of distributed systems and file systems in particular.
- We would also like to ensure that all the requirements and implementables were throughly discussed in TA meets before reaching this document and finally you. To give a few examples (with hopes of igniting that system thinking spark and no thought of flex, okay maybe a bit)
  - Using period as delimiter might seem like a bad idea, especially when it is in the middle of a word.
    - You might think that use of newline character would be better. However, this would mean that every time a user wants to write a new sentence, they would have to add a newline character at the end. NOT USER_FRIENDLY DESIGN
    - Another approach could be using fixed character lengths for sentences. But, this would require the system to pad sentences with spaces to reach the fixed length (or even worse, splitting words mid-way). NOT SPACE_EFFICIENT DESIGN
- The point is, there is no perfect solution. Every design decision has its pros and cons. The requirements are designed to make you think about these trade-offs and come up with a solution that balances them effectively.
  - Anyways in the industry (and DASS next sem), you will be working with, probably, worse requirements. So, this is a good practice run.

**ALL THE BEST! BUT MOST IMPORTANTLY, HAVE FUN!**