Gemini Chat

hiii , can you read and understand this

Alright, Teammate 1, let's do a proper deep dive. You're the backend specialist, so your mission this weekend is to build the entire "plumbing" of this distributed system. By Sunday night, your components (NM and SS) should be robust, concurrent, and ready to serve CREATE and READ requests.
Here is an extremely detailed, step-by-step breakdown of your tasks, the technical concepts involved, and the why behind each decision.
🚀 Task 1: Build the Concurrent Skeletons (Target: Tonight, Oct 31)
Your goal tonight is to get two executables, name_server and storage_server, that can run, listen for connections, and handle multiple connections at once without crashing.
Name Server (NM) - The "Central Coordinator"
This is your central listener. It's a pure server.
Socket Creation (socket): Create a standard IPv4 TCP socket. socket(AF_INET, SOCK_STREAM, 0).
Set Socket Option (setsockopt): This is a critical quality-of-life step for development. Use setsockopt with the SO_REUSEADDR flag.
Why? When you kill your server, the OS keeps its port in a TIME_WAIT state for 30-60 seconds. SO_REUSEADDR allows you to immediately restart your server without getting an "Address already in use" error. This will save you hours of frustration.
Binding (bind): bind() your socket to the well-known port from your protocol (NM_LISTEN_PORT, e.g., 9001) and the address INADDR_ANY.
Why INADDR_ANY? Binding to 127.0.0.1 (localhost) only allows connections from your own machine. INADDR_ANY tells the OS to listen on all available network interfaces (Wi-Fi, Ethernet, etc.), which is what a real server does.
Listening (listen): Call listen() with a backlog queue size (e.g., 20). This tells the OS how many incoming connections to queue up while your main thread is busy.
The "Main Loop" (accept): Your main() function's final step is an infinite while(1) loop. Its only job is to call accept().
Concurrency (The Most Critical Part): As soon as accept() returns a new client file descriptor (conn_fd), you must handle it concurrently. Do not put any recv() or send() logic in this main loop.
Action: Immediately call pthread_create(). Pass a new thread function (e.g., handle_connection_thread) the conn_fd as its argument.
Why? This is the "thread-per-connection" model. It isolates every client and SS. If one client is slow, it doesn't block the entire server. Your main loop is now free to immediately call accept() again, ready for the next connection.
Storage Server (SS) - The "Hybrid Worker"
This component is more complex: it's both a server (listening for clients) and a client (talking to the NM).
Dual Role - Part A: The Server (Listener):
Just like the NM, you need to socket(), setsockopt(SO_REUSEADDR), bind(), and listen().
You must do this for your Client-Facing Port (e.g., 9002, which you'll get from argv).
This listener also needs its own accept() loop in its own thread. main() should spawn a "listener thread" (e.g., run_client_listener_loop) that does this.
This listener thread will also use the "thread-per-connection" model, spawning a handle_client_data_thread for every client that connects.
Dual Role - Part B: The Client (Initiator):

This is the second thing your main() function does (after spawning the listener thread).
It must act as a client: socket() and connect() to the Name Server's well-known address (127.0.0.1) and port (NM_LISTEN_PORT).
This connection is permanent. You must save this single file descriptor in a global variable (e.g., g_nm_fd). This is your "hotline" to the NM.
If this connect() fails, the SS should exit(1) because it cannot function without its NM.
🚀 Task 2: Implement Registration (Target: Saturday Morning, Nov 1)
Your goal now is to make the components talk and introduce themselves.
Storage Server (SS) - "Reporting for Duty"
Action: Immediately after the connect() to the NM (in g_nm_fd) succeeds, you must send your registration message.
Message: Format a string based on your protocol.h. Example: sprintf(buffer, "%s %s %d\n", S_INIT, "127.0.0.1", 9002).
You're sending S_INIT, your public IP (use "127.0.0.1" for local testing), and your Client-Facing Port (9002). The NM doesn't need to know about your other ports.
Send: send(g_nm_fd, buffer, strlen(buffer), 0).
Heartbeat Loop: After sending, your main thread now enters its own while(1) recv() loop, listening only on g_nm_fd. This is where it will receive commands from the NM (like NM_CREATE).
Name Server (NM) - "The Handler Logic"
Action: This is the code inside your handle_connection_thread function.
The Handshake: The first recv() on the conn_fd is your identification. You must read it.
The if-else:
if (strncmp(buffer, C_INIT, ...)) -> It's a Client.
if (strncmp(buffer, S_INIT, ...)) -> It's a Storage Server.
If S_INIT:
Parse: sscanf(buffer, "%*s %s %d", ip, &client_port) to extract the SS's details.
Data Structures: You need a global struct StorageServer g_ss_list[MAX_SS] and a global int g_ss_count.
Atomicity: This global list is shared memory. Any time you touch it, you must use a mutex.
pthread_mutex_lock(&ss_list_mutex).
Store: Add the parsed IP, port, and—most importantly—this conn_fd to your g_ss_list. Increment g_ss_count.
pthread_mutex_unlock(&ss_list_mutex).
The Command Loop: This thread cannot exit. It now enters its own while(1) recv() loop, listening on this same conn_fd. This is where it will receive future messages from this SS (like S_META_UPDATE in Phase 3).
If C_INIT (Teammate 2's part):
Parse: sscanf(buffer, "%*s %s", username).
Data Structures: You need a struct Client g_client_list[MAX_CLIENTS] and a client_list_mutex.
Store: Lock, store the username and conn_fd, unlock.
The Command Loop: This thread also enters a while(1) recv() loop. This is where you'll get C_REQ_CREATE, C_REQ_READ, etc., from this specific client.
🚀 Task 3: Implement CREATE (Target: Saturday Afternoon, Nov 1)
Your goal is to handle the C_REQ_CREATE command by delegating it to an SS.
Name Server (NM) - "The Delegator"
Action: Inside the while(1) loop of a Client handler thread.
Receive: You get a recv() with the message C_REQ_CREATE foo.txt\n.
Parse: Extract the filename.
Data Structures: You need a global "file map." A hash map is best, but for now, a simple array is fine: struct FileMapEntry g_file_map[MAX_FILES] and g_file_map_mutex.

Atomicity (Critical): pthread_mutex_lock(&file_map_mutex). You must hold this lock while you check for existence and create the entry, to prevent a race condition where two clients try to create the same file.
Check Existence: Loop through g_file_map to see if filename already exists.
If yes: send(client_conn_fd, RESP_CONFLICT, ...) and pthread_mutex_unlock(...) and continue; (wait for next command).
Storage Policy (If New):
pthread_mutex_lock(&ss_list_mutex).
Pick an SS. A simple round-robin is perfect: int ss_index = g_file_count % g_ss_count;
Get the conn_fd for that SS: int ss_fd = g_ss_list[ss_index].conn_fd.
pthread_mutex_unlock(&ss_list_mutex).
Forward Command:
Format the command: sprintf(buffer, "%s %s\n", NM_CREATE, filename).
send(ss_fd, buffer, ...) to the chosen Storage Server.
Wait for ACK: This is a blocking recv(ss_fd, response, ...) on the SS's socket.
Commit or Abort:
If RESP_OK from SS: Success! Add the file to g_file_map (e.g., g_file_map[g_file_count] = {filename, ss_index}). Also, add the creator to your new g_acl_map as the owner. Increment g_file_count.
If RESP_SRV_ERR from SS: The SS failed. Send RESP_SRV_ERR to the client. Do not add the file to your map.
Release & Reply:
pthread_mutex_unlock(&file_map_mutex).
Send the final status (RESP_OK or RESP_SRV_ERR) to the Client.
Storage Server (SS) - "The Worker"
Action: Inside the while(1) loop of your NM-facing thread (the one listening on g_nm_fd).
Receive: You get NM_CREATE foo.txt\n.
Parse: Extract filename.
Path Sanity: Never trust a filename. Prepend your data directory: sprintf(local_path, "ss_data/ss1/%s", filename). (You must mkdir -p ss_data/ss1 on your machine manually).
File I/O: FILE* fp = fopen(local_path, "w"); The "w" flag creates a new, empty file.
Error Handling: If fp == NULL (disk full, bad permissions), send RESP_SRV_ERR back to the NM.
Success: If fp is good, fclose(fp) immediately. Send RESP_OK back to the NM.
🚀 Task 4: Implement READ (Target: Sunday, Nov 2)
Your goal is to have the NM refer a client to the correct SS, and have the SS stream the file.
Name Server (NM) - "The Matchmaker"
Action: Inside the while(1) loop of a Client handler thread.
Receive: You get C_REQ_READ foo.txt\n.
Parse: Extract filename.
Find File:
pthread_mutex_lock(&file_map_mutex).
Search g_file_map for filename.
If not found: send(client_conn_fd, RESP_NOT_FOUND, ...), pthread_mutex_unlock(...), continue;.
If found: Get the ss_index associated with that file.
pthread_mutex_unlock(&file_map_mutex).
Find SS Info:
pthread_mutex_lock(&ss_list_mutex).
Use ss_index to get the client-facing IP and port: g_ss_list[ss_index].client_ip, g_ss_list[ss_index].client_port.
pthread_mutex_unlock(&ss_list_mutex).
ACL Stub: The plan says to skip the access check for now. This is a good simplification.

Send Referral: Format the special RESP_SS_INFO message: sprintf(buffer, "%s %s %d\n", RESP_SS_INFO, ip, port).
send(client_conn_fd, buffer, ...) to the Client.
Done! The NM's job for this READ is finished.
Storage Server (SS) - "The Data Server"
Action: This logic is not in your NM thread. This is in the client listener thread spawned in Task 1.
Receive Connection: Your accept() loop (listening on your Client-Facing Port, e.g., 9002) gets a new connection. It spawns a handle_client_data_thread with the new client_fd.
One-Shot Transaction: This connection is temporary. It has one job.
Receive Command: recv(client_fd, buffer, ...) to get the one and only command: SS_GET_FILE foo.txt\n.
Parse & Path: sscanf(buffer, "%*s %s", filename). sprintf(local_path, "ss_data/ss1/%s", filename).
File I/O: FILE* fp = fopen(local_path, "r");.
Error Handling: If fp == NULL (file doesn't exist, which shouldn't happen if the NM is correct), just close(client_fd) and pthread_exit(). The client's read will fail, which is the correct behavior.
The Data Stream: This is the core.
Use a loop with fread() into a buffer (e.g., char file_buffer[4096]).
while ((bytes_read = fread(file_buffer, 1, 4096, fp)) > 0)
Inside the loop: send(client_fd, file_buffer, bytes_read, 0). You must check the return of send() for errors (e.g., if the client disconnected).
Cleanup (Critical):
fclose(fp);
close(client_fd); This close is the signal. It's how Teammate 2's client will know the file transfer is complete (their recv() will return 0).
The thread's job is done. pthread_exit().

Yes, I've read and understood this document.

It's a very detailed, step-by-step technical plan for "Teammate 1" to build the backend for a distributed file system. The goal is to create two main components:

1. **Name Server (NM):** A central coordinator that manages metadata (which SS has which file) and handles client/SS registration.
2. **Storage Server (SS):** A worker component that both reports to the NM and serves files directly to clients.

The plan is broken down into four main tasks with a clear timeline:

- **Task 1 (Tonight):** Build the basic, concurrent server skeletons for both NM and SS using a "thread-per-connection" model.
- **Task 2 (Saturday Morning):** Implement the registration protocol so the SS can connect and register itself with the NM.
- **Task 3 (Saturday Afternoon):** Implement the `CREATE` file functionality, where the NM delegates the file creation to an SS.
- **Task 4 (Sunday):** Implement the `READ` file functionality, where the NM refers the client to the correct SS, and the SS streams the file data.

The document covers key technical details like socket programming, using `SO_REUSEADDR`, concurrency with `pthread`s, and protecting shared data with mutexes.

How can I help you with this? For example, I could write the C code for a specific part (like the Task 1 skeleton), explain a concept in more detail, or help you structure the data-sharing structs.

i have made the below toy server clients to learn the core concepts related to above project
can you tell me how much of above is done / or similar to this
like it will help me figure out if i have anything else new left to learn conceptually

simpleclient.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
//headers explained in simpleserver.c
#include <netdb.h> // For gethostbyname

int main(){
int client_fd;
client_fd=socket(AF_INET, SOCK_STREAM, 0);
if(client_fd<0){
perror("client socket() failed");
exit(1);
}
printf("1. Client socket created succesfully");

int port=9000;
struct sockaddr_in server_addr;
struct hostent *server;
//struct to hold info about the server

server=gethostbyname("localhost");
if(server==NULL){
fprintf(stderr, "No such host as localhost, error");
exit(1);
}

// 2. setting up the server_addr struct (same like in the server)
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(port);

memcpy(&server_addr.sin_addr.s_addr, server->h_addr_list[0], server->h_length);
// 3. Copy the server's IP address into the struct
// We use h_addr_list[0] - the first address in the list

if(connect(client_fd,(struct sockaddr *) &server_addr, sizeof(server_addr))<0){
perror("connect() failed");
exit(1);
}
// 4. Connect to the server!
// This call blocks until the connection is made or fails.
```

```
printf("2. Connected to server yayy!!");

char buffer[1024];
char* message="Hello from client\n";
write(client_fd, message, strlen(message));
printf("Sent message: %s", message);
memset(buffer, 0, 1024);
read(client_fd, buffer, 1023);

printf("Server echoed: %s",buffer);
// close(client_fd);
return 0;

}

simpleserver.c
#include <stdio.h>
//for printf and perror

#include <stdlib.h>
//for exit

#include <string.h>
//for bzero or memset

#include <unistd.h>
//for read write close all

#include <sys/socket.h>
//for main socket functions

#include <netinet/in.h>
//for the struct sockaddr_in and htons()/htonl() macros.

#include <pthread.h>
//for threading

// This is the function that each thread will run
void *handle_client(void *arg){
// 1. Get our client file descriptor back by casting arg to (int*) then dereferencing to get the integer
file descriptor.
int client_fd = *( (int*)arg );
free(arg); // We've copied the value, so we free the memory
// free(arg) releases the heap memory you allocated in the main thread to avoid leaks.

char buffer[1024]; // A buffer to hold the client's message
int bytes_read;

// Clear the buffer
memset(buffer, 0, 1024);
```

```
// read() will block and wait for the client to send data
bytes_read = read(client_fd, buffer, 1023); // 1023 to leave room for '\0'

if (bytes_read < 0) {
perror("read() failed");
} else if (bytes_read == 0) {
printf("Client disconnected before sending a message.\n");
} else {
// We got a message!
printf("Client said: %s", buffer); // printf the message

// Now, echo it back
write(client_fd, buffer, bytes_read);
printf("Echoed message back to client.\n");
}

// Now we're done with this client
close(client_fd);
printf("Client disconnected. Waiting for next client...\n");
return NULL;
}

int main(int argc, char*argv[]){
printf("Starting server...\n");
int server_fd; //server file descriptor
server_fd=socket(AF_INET,SOCK_STREAM,0);
// AF_INET = Use IPv4 (the standard internet protocol)
// SOCK_STREAM = Use TCP (a reliable, streaming connection, not UDP)
// 0 = Use the default protocol (which is TCP for SOCK_STREAM)
// ALWAYS check for errors.
// A negative return value means the function failed.
if(server_fd<0){
// perror prints your message ("socket() failed")
// AND the specific system error (like "Permission denied").
perror("socket() function failed");
exit(1);
}
printf("1. Socket created successfully (fd=%d) \n",server_fd);

int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
perror("setsockopt(SO_REUSEADDR) failed");
exit(1);
}
//When your server exits (or crashes), the operating system keeps port 9000 in a "TIME_WAIT" state for about 30-60 seconds. It's "reserving" the port just in case any last-second data packets arrive.
//The Solution: SO_REUSEADDR, We need to tell the OS, "Hey, I'm the new server, and I give you permission to reuse that address right now.", We do this with a function called setsockopt().

struct sockaddr_in server_addr;
```

```
//Different kinds of networks (IPv4, IPv6, local domain sockets, etc.) each have their own struct to
represent an address.
//AF_INET and struct sockaddr_in for IPv4, AF_INET6 and struct sockaddr_in6 for IPv6, etc.



int port=9000;
//we can also take this from command-line using if (argc > 1) port = atoi(argv[1]);

// Clear the entire struct to make sure it's clean
// We want all fields clean before assigning values.
// Prevents accidental garbage values from stack memory.
memset(&server_addr, 0, sizeof(server_addr));

server_addr.sin_family = AF_INET;
// Sets the address family to IPv4 (must match what we used in the socket else bind() will fail with
EINVAL (invalid argument))

server_addr.sin_addr.s_addr = INADDR_ANY;
// Sets the IP address to INADDR_ANY which means "listen on any available IP address" This is
what we want for a server.
//INADDR_ANY is a macro meaning “all network interfaces”.In binary, it’s 0.0.0.0.

server_addr.sin_port = htons(port);
// Sets the port number, converting from host byte order to network byte order
// htons() = "Host to Network Short"
// All TCP/IP headers use network byte order (big-endian).
// If you skip this conversion, other computers would interpret the bytes backwards — e.g.,
// 9000 (0x2328) might become 0x2823 = 10275, i.e., completely wrong port.

//bind() is the system call that tells the operating system, "I want the socket server_fd to be the
one responsible for port 9000 (we have assigned server_addr's sin_port as port)."
if (bind(server_fd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
// We cast our 'struct sockaddr_in' (which is internet-specific)
// to the generic 'struct sockaddr' that bind() expects.
perror("bind() failed");
//checking for errors. A common error here is "Address already in use," which means another
program (or your old, crashed server) is still holding onto port 9000.
exit(1);
}
printf("2. Socket bound to port %d\n", port);

//???????/ '5' is the "backlog" — how many connections can be
waiting???????????????????????????
// in a queue while we're busy. 5 is a fine number.
if (listen(server_fd, 5) < 0) {
perror("listen() failed");
exit(1);
}
printf("3. Server is listening on port %d...\n", port);

int client_fd; // This will be the NEW file descriptor for the client
```

```
struct sockaddr_in client_addr; // This will hold the CLIENT's address info
socklen_t client_len = sizeof(client_addr);
printf("Waiting for a client to connect...\n");
while (1) {
// now accept() blocks the program and waits for a connection
client_fd = accept(server_fd, (struct sockaddr *) &client_addr, &client_len);
if (client_fd < 0) {
perror("accept() failed");
continue; // Go back to the start of the loop and wait again
}

// printf("4. Client connected successfully! Waiting for a message...\n");
printf("4. Client connected! Handing off to a new thread...\n");

pthread_t tid;

int *new_sock=malloc(sizeof(int));
*new_sock=client_fd;
// We can't just pass &client_fd to the thread. Because the main loop will immediately loop back, accept a new client, and change the value of client_fd. The first thread would suddenly have its file descriptor changed! By malloc-ing new memory, we give each thread its own private copy of the file descriptor.
// Why not pass &client_fd? If you passed the address of the stack variable client_fd from the main thread, that memory could change when the main loop accepts the next connection; threads would race and get wrong FDs. Allocating per-thread memory avoids that race.

if(pthread_create(&tid, NULL, handle_client, (void*)new_sock)!=0){
perror("pthread_create() failed");
}
// Create a new thread:
// 1. &tid: Store the thread's ID here
// 2. NULL: Use default thread attributes
// 3. handle_client: The function the new thread should run
// 4. new_sock: The argument to pass to that function
// We need to pass the client_fd to the thread, but pthread_create only accepts a (void*) so we cast.

}
close(server_fd);
return 0;
}
```

You've built an excellent foundation! Your two toy programs cover almost all the core socket programming concepts from **Task 1** of the project.

Here’s a breakdown of what you've already done and what the new conceptual leaps are.

✅ What You Have Done (Conceptually)

Your `simpleserver.c` and `simpleclient.c` are a near-perfect implementation of the fundamental building blocks:

- **Task 1: The Server Skeleton (NM & SS-Listener):**
  - You have correctly implemented `socket()`, `setsockopt(SO_REUSEADDR)`, `bind()`, and `listen()`. Your comments on *why* you're doing each (especially `SO_REUSEADDR` and `htons`) are spot-on.
  - Your `main()` function's `while(1)` loop with `accept()` is exactly what the spec describes.
- **Task 1: The Concurrency Model (Critical!):**
  - You have **100% implemented the "thread-per-connection" model.**
  - Your use of `pthread_create()` immediately after `accept()` is the core requirement.
  - Your use of `malloc` to pass the `client_fd` to the thread is the correct, safe way to avoid the race condition you described in your comments. This is a common hurdle, and you've nailed it.
- **Task 1 & 2: The Client Skeleton (Client & SS-Initiator):**
  - Your `simpleclient.c` correctly implements `socket()`, `gethostbyname()` (for `localhost`), and `connect()`. This is exactly what the SS will do in **Task 1, Part B** to connect to the NM.
- **The Handler Thread:**
  - Your `handle_client` function shows you understand the life cycle of a connection: `read` the request, `write` the response, and `close` the connection.

🚀 What's New / Left to Learn (Conceptually)

The *new* parts aren't about sockets (you've got that); they're about **state management** and **distributed logic**. Your toy server is an "echo" server—it's stateless. The project requires a *stateful* system.

Here are the new concepts you'll be adding:

1. The "Hybrid" Component (The SS):

- **Your Code:** You have one server and one client.
- **The Project:** The Storage Server (SS) is **both at the same time**. Its `main()` function will first spawn a *listener thread* (your `simpleserver.c` logic) and *then* it will act as a client (your `simpleclient.c` logic) to `connect()` to the NM and keep that connection open.

2. Persistent Connections & Command Loops:

- **Your Code:** Your `handle_client` thread does one `read`, one `write`, and then exits.
- **The Project:** The threads in the NM (handling Clients and SSs) and the SS's main thread (handling the NM connection) must **not** exit. After the first message (like `S_INIT`), they must enter a `while(1) { recv(...) }` loop to listen for *future* commands (`NM_CREATE`, `C_REQ_READ`, etc.) on that same, persistent socket.

3. Global State & Mutexes (The Biggest New Concept):

- **Your Code:** Your `handle_client` thread is totally isolated. It shares no data with any other thread. It needs no mutexes.
- **The Project:** The NM *must* have global, shared data (like `g_ss_list[]` and `g_file_map[]`).
- **The New Skill:** When one thread adds a new SS to `g_ss_list` (Task 2) or another thread tries to *read* that list to find an SS (Task 3), you have a **race condition**. You *must* protect this shared data with a mutex.
  - `pthread_mutex_lock(&ss_list_mutex);`
  - ... (access the global list) ...
  - `pthread_mutex_unlock(&ss_list_mutex);`

- This is the "Atomicity" a-ha moment in the spec.

4. Protocol Parsing & Logic:

- **Your Code:** You just echo the buffer: `write(client_fd, buffer, ...)`
- **The Project:** You have to *understand* the buffer. You'll use `strncmp` to see if the message is `C_INIT` or `S_INIT`. You'll use `sscanf` to parse the details from the string. Your server becomes a "router" that makes decisions instead of an "echo."

5. Multi-Component Logic (Delegation & Referral):

- **Your Code:** The logic is simple: Client <-> Server.
- **The Project:** The logic is distributed:
  - **Delegation (CREATE):** Client -> NM, then NM -> SS. The NM thread *blocks* and waits for the SS to reply before it replies to the Client.
  - **Referral (READ):** Client -> NM, NM -> Client (with SS info). Then the Client -> SS directly. This is a new communication pattern.

6. Real File I/O & Streaming:

- **Your Code:** You're just sending a buffer of text.
- **The Project:** You'll use `fopen("w")` to create empty files (Task 3) and `fopen("r")` with a `while(fread(...)) { send(...) }` loop to stream large files (Task 4).

💡 Answering Your `????` in the Code

> `//???????/ '5' is the "backlog" – how many connections can be waiting???????????????????????????`

You are exactly right. `listen(server_fd, 5)` tells the OS: "If I'm busy in my `accept()` loop handling a new connection, you can hold up to 5 other new connections in a 'pending' queue. If a 6th connection arrives while the queue is full, just reject it."

Since your `accept()` loop is so fast (it just calls `pthread_create` and loops), the backlog is rarely hit, but it's important to have. `5` or `20` is a fine number for this.

In summary: You have the perfect foundation. You've mastered the complex parts of socket setup and threading. Your next steps are all about building on that foundation with state management (mutexes) and application-level logic (parsing and routing commands).

Would you like me to help you refactor your `simpleserver.c` to look more like the **Task 1 Name Server (NM) skeleton**? We could add the empty `handle_connection_thread` function and the basic `if-else` for parsing.

i have made this, how much of the above timeline / plan am i done with can you explain
#include "protocol.h"
//our file which tells this code of our own defined stuff for the network protocol (shared constants, command strings, and response codes)

#include <stdio.h>
//for printf and perror

#include <stdlib.h>
//for exit

```
#include <string.h>
//for bzero or memset

#include <unistd.h>
//for read write close all

#include <sys/socket.h>
//for main socket functions

#include <netinet/in.h>
//for the struct sockaddr_in and htons()/htonl() macros.

#include <pthread.h>
//for threading

typedef struct{
int conn_fd; //persistent file descriptor to this storage server
char ip[INET_ADDRSTRLEN]; //client facing ip
int client_port; //client facing ip
//maybe more will be added later
}StorageServer;

# define MAX_SS 10 // WHAT IS MAX NO OF STORAGE SERVERS WE ARE ALLOWING, CAN BE
CHANGED LATER
StorageServer g_ss_list[MAX_SS];
int g_ss_count=0;
pthread_mutex_t ss_list_mutex=PTHREAD_MUTEX_INITIALIZER;

typedef struct{
int conn_fd;
char username[MAX_USERNAME_LEN];
}Client;

# define MAX_CLIENTS 50 // WHAT IS MAX NO OF CLIENTS WE ARE ALLOWING, CAN BE
CHANGED LATER
Client g_client_list[MAX_CLIENTS];
int g_client_count=0;
pthread_mutex_t client_list_mutex=PTHREAD_MUTEX_INITIALIZER;

// 3. File Map (Which file is on which SS?)
typedef struct {
char PATH[MAX_PATH_LEN];
int ss_index;
//more info will be added here later
char owner[MAX_USERNAME_LEN];
}FileMapEntry;

#define MAX_FILES 1000
FileMapEntry g_file_map[MAX_FILES];
int g_file_count = 0;
```

```
pthread_mutex_t file_map_mutex = PTHREAD_MUTEX_INITIALIZER;

void* handle_client_commands(void* arg);
void* handle_ss_commands(void* arg);
void do_create(int client_fd, char* username, char* filename);
void do_read(int client_fd, char* username, char* filename);

void do_create(int client_fd, char* username, char* filename) {
printf("Client %s requesting CREATE: %s\n", username, filename);
// 1. Lock file map to check existence
pthread_mutex_lock(&file_map_mutex);
for (int i = 0; i < g_file_count; i++) {
if (strcmp(g_file_map[i].path, filename) == 0) {
send(client_fd, RESP_CONFLICT, strlen(RESP_CONFLICT), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}
}
// 2. File doesn't exist. Find an SS to send it to.
pthread_mutex_lock(&ss_list_mutex);
if (g_ss_count == 0) {
send(client_fd, RESP_SS_DOWN, strlen(RESP_SS_DOWN), 0);
pthread_mutex_unlock(&ss_list_mutex);
pthread_mutex_unlock(&file_map_mutex);
return;
}
// 3. Round-robin policy
int ss_index = g_file_count % g_ss_count;
int ss_fd = g_ss_list[ss_index].conn_fd;
pthread_mutex_unlock(&ss_list_mutex);

// 4. Send command to the SS
char command_buf[MAX_MSG_LEN];
sprintf(command_buf, "%s %s\n", NM_CREATE, filename);
if (send(ss_fd, command_buf, strlen(command_buf), 0) < 0) {
perror("Failed to send to SS");
send(client_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}
// 5. Wait for ACK from SS
char ss_resp[MAX_MSG_LEN];
memset(ss_resp, 0, MAX_MSG_LEN);
if (read(ss_fd, ss_resp, MAX_MSG_LEN - 1) <= 0) {
printf("SS failed to respond to CREATE\n");
send(client_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}

// 6. If SS says OK, commit to our map
```

```
if (strncmp(ss_resp, RESP_OK, strlen(RESP_OK)) == 0) {
strcpy(g_file_map[g_file_count].path, filename);
strcpy(g_file_map[g_file_count].owner, username);
g_file_map[g_file_count].ss_index = ss_index;
g_file_count++;
send(client_fd, RESP_OK, strlen(RESP_OK), 0); // Success!
} else {
send(client_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR), 0);
}
pthread_mutex_unlock(&file_map_mutex);
}

void do_read(int client_fd, char* username, char* filename) {
printf("Client %s requesting READ: %s\n", username, filename);
// 1. Lock map and find the file
pthread_mutex_lock(&file_map_mutex);
int file_index = -1;
for (int i = 0; i < g_file_count; i++) {
if (strcmp(g_file_map[i].path, filename) == 0) {
file_index = i;
break;
}
}
if (file_index == -1) {
send(client_fd, RESP_NOT_FOUND, strlen(RESP_NOT_FOUND), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}
// 2. (ACL Stub: We'll add this in Phase 2)
// For now, just let them read

// 3. Get the SS's info
int ss_index = g_file_map[file_index].ss_index;
pthread_mutex_unlock(&file_map_mutex); // Done with file map

// 4. Get SS client-facing IP/port
pthread_mutex_lock(&ss_list_mutex);
char ss_ip[INET_ADDRSTRLEN];
int ss_port = g_ss_list[ss_index].client_port;
strcpy(ss_ip, g_ss_list[ss_index].ip);
pthread_mutex_unlock(&ss_list_mutex);

// 5. Send the referral to the client
char response_buf[MAX_MSG_LEN];
sprintf(response_buf, "%s %s %d\n", RESP_SS_INFO, ss_ip, ss_port);
send(client_fd, response_buf, strlen(response_buf), 0);
}

// This is the function that each thread will run
void *handle_connection(void *arg){
int conn_fd = *((int*)arg);
```

```
free(arg);
char buffer[MAX_MSG_LEN];
memset(buffer, 0, MAX_MSG_LEN);

// 1. Read the HELLO message (the handshake)
if (read(conn_fd, buffer, MAX_MSG_LEN - 1) <= 0) {
printf("Handshake failed. Closing connection.\n");
close(conn_fd);
return NULL;
}

printf("Handshake received: %s", buffer);

// 2. Decide WHO it is
if (strncmp(buffer, C_INIT, strlen(C_INIT)) == 0) {
// --- It's a CLIENT ---
char username[MAX_USERNAME_LEN];
sscanf(buffer, "%*s %s", username); // Parse the username
// Add to global client list (MUST be thread-safe)
pthread_mutex_lock(&client_list_mutex);
g_client_list[g_client_count].conn_fd = conn_fd;
strcpy(g_client_list[g_client_count].username, username);
g_client_count++;
pthread_mutex_unlock(&client_list_mutex);
printf("Registered new client: %s\n", username);
send(conn_fd, RESP_OK, strlen(RESP_OK), 0); // Send "OK"
// Pass to the client command loop
handle_client_commands((void*)&conn_fd);
} else if (strncmp(buffer, S_INIT, strlen(S_INIT)) == 0) {
// --- It's a STORAGE SERVER ---
char ip[INET_ADDRSTRLEN];
int client_port; // The port for *clients* to connect to
// S_INIT <ip> <nm_port> <client_port>
// We only care about the client-facing IP and port
sscanf(buffer, "%*s %s %*d %d", ip, &client_port);
// Add to global SS list (MUST be thread-safe)
pthread_mutex_lock(&ss_list_mutex);
g_ss_list[g_ss_count].conn_fd = conn_fd; // This is the NM-SS command socket
strcpy(g_ss_list[g_ss_count].ip, ip);
g_ss_list[g_ss_count].client_port = client_port;
g_ss_count++;
pthread_mutex_unlock(&ss_list_mutex);
printf("Registered new Storage Server at %s:%d\n", ip, client_port);
send(conn_fd, RESP_OK, strlen(RESP_OK), 0); // Send "OK"

// Pass to the SS command loop
handle_ss_commands((void*)&conn_fd);

} else {
printf("Unknown handshake. Closing connection.\n");
close(conn_fd);
```

```
}
// The thread exits when the helper function (handle_client/ss_commands) returns
printf("Connection handler thread exiting.\n");
return NULL;
}

void* handle_client_commands(void* arg) {
int client_fd = *((int*)arg);
char buffer[MAX_MSG_LEN];
char cmd[MAX_MSG_LEN];
char arg1[MAX_MSG_LEN];

// Get the username for this FD (This is a simple, non-performant lookup)
char username[MAX_USERNAME_LEN] = "unknown";
pthread_mutex_lock(&client_list_mutex);
for(int i=0; i<g_client_count; i++) {
if(g_client_list[i].conn_fd == client_fd) {
strcpy(username, g_client_list[i].username);
break;
}
}
pthread_mutex_unlock(&client_list_mutex);

// This is the new "echo" loop
while (1) {
memset(buffer, 0, MAX_MSG_LEN);
int bytes_read = read(client_fd, buffer, MAX_MSG_LEN - 1);

if (bytes_read <= 0) {
printf("Client %s disconnected.\n", username);
break; // Exit loop, thread will end
}
// Parse the command
sscanf(buffer, "%s %s", cmd, arg1);
if (strncmp(cmd, C_REQ_CREATE, strlen(C_REQ_CREATE)) == 0) {
do_create(client_fd, username, arg1);
} else if (strncmp(cmd, C_REQ_READ, strlen(C_REQ_READ)) == 0) {
do_read(client_fd, username, arg1);
} else {
// Unknown command
send(client_fd, RESP_BAD_REQ, strlen(RESP_BAD_REQ), 0);
}
}
// --- Cleanup ---
// Remove client from list
// (We'll add this later, for now just close)
close(client_fd);
return NULL;
}

// This function runs in a loop for a STORAGE SERVER
```

```
void* handle_ss_commands(void* arg) {
int ss_fd = *((int*)arg);
char buffer[MAX_MSG_LEN];

while (1) {
memset(buffer, 0, MAX_MSG_LEN);
int bytes_read = read(ss_fd, buffer, MAX_MSG_LEN - 1);

if (bytes_read <= 0) {
printf("Storage Server disconnected.\n");
// CRITICAL: We need to mark this SS as DEAD
// (We'll add this logic later)
break; // Exit loop
}
// Right now, the SS only sends S_META_UPDATE (Phase 3)
// So for Phase 1, this loop just sits and waits.
printf("Received message from SS: %s", buffer);
}
close(ss_fd);
return NULL;
}

int main(int argc, char*argv[]){
printf("Starting server...\n");
int server_fd; //server file descriptor
server_fd=socket(AF_INET,SOCK_STREAM,0);
// AF_INET = Use IPv4 (the standard internet protocol)
// SOCK_STREAM = Use TCP (a reliable, streaming connection, not UDP)
// 0 = Use the default protocol (which is TCP for SOCK_STREAM)
// ALWAYS check for errors.
// A negative return value means the function failed.
if(server_fd<0){
// perror prints your message ("socket() failed")
// AND the specific system error (like "Permission denied").
perror("socket() function failed");
exit(1);
}
printf("1. Socket created successfully (fd=%d) \n",server_fd);

int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
perror("setsockopt(SO_REUSEADDR) failed");
exit(1);
}
//When your server exits (or crashes), the operating system keeps port 9000 in a "TIME_WAIT" state for about 30-60 seconds. It's "reserving" the port just in case any last-second data packets arrive.
//The Solution: SO_REUSEADDR, We need to tell the OS, "Hey, I'm the new server, and I give you permission to reuse that address right now.", We do this with a function called setsockopt().

struct sockaddr_in server_addr;
```

```
//Different kinds of networks (IPv4, IPv6, local domain sockets, etc.) each have their own struct to represent an address.
//AF_INET and struct sockaddr_in for IPv4, AF_INET6 and struct sockaddr_in6 for IPv6, etc.


int port=9000;
//we can also take this from command-line using if (argc > 1) port = atoi(argv[1]);

// Clear the entire struct to make sure it's clean
// We want all fields clean before assigning values.
// Prevents accidental garbage values from stack memory.
memset(&server_addr, 0, sizeof(server_addr));

server_addr.sin_family = AF_INET;
// Sets the address family to IPv4 (must match what we used in the socket else bind() will fail with EINVAL (invalid argument))

server_addr.sin_addr.s_addr = INADDR_ANY;
// Sets the IP address to INADDR_ANY which means "listen on any available IP address" This is what we want for a server.
//INADDR_ANY is a macro meaning “all network interfaces”.In binary, it’s 0.0.0.0.

server_addr.sin_port = htons(port);
// Sets the port number, converting from host byte order to network byte order
// htons() = "Host to Network Short"
// All TCP/IP headers use network byte order (big-endian).
// If you skip this conversion, other computers would interpret the bytes backwards — e.g.,
// 9000 (0x2328) might become 0x2823 = 10275, i.e., completely wrong port.

//bind() is the system call that tells the operating system, "I want the socket server_fd to be the one responsible for port 9000 (we have assigned server_addr's sin_port as port)."
if (bind(server_fd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
// We cast our 'struct sockaddr_in' (which is internet-specific)
// to the generic 'struct sockaddr' that bind() expects.
perror("bind() failed");
//checking for errors. A common error here is "Address already in use," which means another program (or your old, crashed server) is still holding onto port 9000.
exit(1);
}
printf("2. Socket bound to port %d\n", port);

// listen(server_fd, 5) tells the OS: "If I'm busy in my accept() loop handling a new connection, you can hold up to 5 other new connections in a 'pending' queue. If a 6th connection arrives while the queue is full, just reject it."
// Since your accept() loop is so fast (it just calls pthread_create and loops), the backlog is rarely hit, but it's important to have. 5 or 20 is a fine number for this.
if (listen(server_fd, 5) < 0) {
perror("listen() failed");
exit(1);
}
printf("3. Server is listening on port %d...\n", port);
```

```
int client_fd; // This will be the NEW file descriptor for the client
struct sockaddr_in client_addr; // This will hold the CLIENT's address info
socklen_t client_len = sizeof(client_addr);
printf("Waiting for a client to connect...\n");
while (1) {
// now accept() blocks the program and waits for a connection
client_fd = accept(server_fd, (struct sockaddr *) &client_addr, &client_len);
if (client_fd < 0) {
perror("accept() failed");
continue; // Go back to the start of the loop and wait again
}

// printf("4. Client connected successfully! Waiting for a message...\n");
printf("4. Client connected! Handing off to a new thread...\n");

pthread_t tid;

int *new_sock=malloc(sizeof(int));
*new_sock=client_fd;
// We can't just pass &client_fd to the thread. Because the main loop will immediately loop back, accept a new client, and change the value of client_fd. The first thread would suddenly have its file descriptor changed! By malloc-ing new memory, we give each thread its own private copy of the file descriptor.
// Why not pass &client_fd? If you passed the address of the stack variable client_fd from the main thread, that memory could change when the main loop accepts the next connection; threads would race and get wrong FDs. Allocating per-thread memory avoids that race.

if(pthread_create(&tid, NULL, handle_connection, (void*)new_sock)!=0){
perror("pthread_create() failed");
}
// Create a new thread:
// 1. &tid: Store the thread's ID here
// 2. NULL: Use default thread attributes
// 3. handle_client: The function the new thread should run
// 4. new_sock: The argument to pass to that function
// We need to pass the client_fd to the thread, but pthread_create only accepts a (void*) so we cast.

}
close(server_fd);
return 0;
}
```

This is a fantastic implementation. Based on the plan, you have **fully completed the Name Server (NM) side** of all four tasks.

Your single `name_server.c` file correctly implements all the logic for the "Central Coordinator."

Here's a detailed breakdown of your progress against the timeline:

✅ What You Have Completed (The Name Server)

You have successfully built the entire `name_server` executable.

- **Task 1: Build the Concurrent Skeletons (NM Part)**
  - **100% Done.** Your `main()` function correctly implements `socket`, `setsockopt(SO_REUSEADDR)`, `bind`, and `listen`.
  - **100% Done.** Your `main()`'s `while(1)` loop correctly calls `accept` and immediately uses `pthread_create` to hand off the connection, which is the "thread-per-connection" model.
- **Task 2: Implement Registration (NM Part)**
  - **100% Done.** Your `handle_connection` function is the perfect "handshake" logic.
  - It correctly does the first `read` and uses `strncmp` to differentiate `C_INIT` from `S_INIT`.
  - It correctly parses the SS details (`sscanf(buffer, "%*s %s %*d %d", ip, &client_port)`).
  - It correctly locks `ss_list_mutex`, adds the SS to the `g_ss_list`, and unlocks.
  - It does the same for clients with `client_list_mutex` and `g_client_list`.
  - It correctly calls `handle_client_commands` or `handle_ss_commands` to enter the persistent command loop.
- **Task 3: Implement CREATE (NM Part)**
  - **100% Done.** Your `do_create` function is a perfect implementation of the "Delegator" logic.
  - It locks the `file_map_mutex` to check for conflicts.
  - It locks the `ss_list_mutex` to pick an SS (using the round-robin policy `g_file_count % g_ss_count`).
  - It `send`s the `NM_CREATE` command to the chosen `ss_fd`.
  - It performs the critical blocking `read` to wait for the SS's response.
  - It correctly "commits" the file to `g_file_map` only if the SS returns `RESP_OK`.
- **Task 4: Implement READ (NM Part)**
  - **100% Done.** Your `do_read` function is the "Matchmaker."
  - It locks/unlocks the `file_map_mutex` to find the file and its `ss_index`.
  - It handles the `RESP_NOT_FOUND` case.
  - It locks/unlocks the `ss_list_mutex` to get the `client_ip` and `client_port` from the `ss_index`.
  - It sends the `RESP_SS_INFO` referral to the client, finishing its job for this request.

🚀 What's Next (The Storage Server)

Your code file is the `name_server` executable. The **entire second executable, the** `storage_server`, is the next piece to build. You have not started the code for this component yet, but your NM is fully ready for it.

Here is what's left, which matches the other half of the plan:

- **Task 1 & 2: SS Skeleton & Registration**
  - Create `storage_server.c`.
  - Its `main()` function must be a "Hybrid":
    1. Spawn a **listener thread** (like your NM's `main`) to listen for *clients*.
    2. Act as a **client** itself: `socket()`, `connect()` to the NM (at `127.0.0.1:9000`).

3. `send()` its `S_INIT` registration message.
4. Enter a `while(1) { read(...) }` loop, listening on the NM socket for commands.

- **Task 3: SS `CREATE` Worker**
  - Inside that `while(1)` loop (listening to the NM), add an `if` statement.
  - When it receives `NM_CREATE foo.txt`, it must:
    1. Create the local path (e.g., `ss_data/ss1/foo.txt`).
    2. `FILE* fp = fopen(local_path, "w");`
    3. Check if `fp` is `NULL`.
    4. `fclose(fp);`
    5. `send()` either `RESP_OK` or `RESP_SRV_ERR` back to the NM.
- **Task 4: SS `READ` Data Server**
  - This is the code for the **listener thread** you spawned in the SS `main`.
  - This thread will `accept()` a connection from a client.
  - It will `read` one command from the client (which will be `SS_GET_FILE foo.txt`).
  - It will `fopen(local_path, "r")`.
  - It will enter the **data streaming loop**:

```c
while ((bytes_read = fread(file_buffer, 1, 4096, fp)) > 0) {
    send(client_fd, file_buffer, bytes_read, 0);
}
```

  - `fclose(fp)` and `close(client_fd)`.

In short, you've perfectly built one of the two required executables. All the logic for the *coordinator* (NM) is done. Now you just need to build the *worker* (SS) that the NM coordinates.

Would you like help starting the skeleton for the `storage_server.c`? We could outline its `main` function and its two main jobs (the listener thread and the NM connection).

i have done a lot, have pasted all my files for your info, can you check all of below and tell me what all parts of phase 1 are done and what all is left / also mention right / wrong if needed
nm.c
#include "protocol.h"
//our file which tells this code of our own defined stuff for the network protocol (shared constants, command strings, and response codes)

#include <stdio.h>
//for printf and perror

#include <stdlib.h>
//for exit

#include <string.h>
//for bzero or memset

#include <unistd.h>

```
//for read write close all

#include <sys/socket.h>
//for main socket functions

#include <netinet/in.h>
//for the struct sockaddr_in and htons()/htonl() macros.

#include <pthread.h>
//for threading

typedef struct{
int conn_fd; //persistent file descriptor to this storage server
char ip[INET_ADDRSTRLEN]; //client facing ip
int client_port; //client facing ip
//maybe more will be added later
}StorageServer;

# define MAX_SS 10 // WHAT IS MAX NO OF STORAGE SERVERS WE ARE ALLOWING, CAN BE
CHANGED LATER
StorageServer g_ss_list[MAX_SS];
int g_ss_count=0;
pthread_mutex_t ss_list_mutex=PTHREAD_MUTEX_INITIALIZER;

typedef struct{
int conn_fd;
char username[MAX_USERNAME_LEN];
}Client;

# define MAX_CLIENTS 50 // WHAT IS MAX NO OF CLIENTS WE ARE ALLOWING, CAN BE
CHANGED LATER
Client g_client_list[MAX_CLIENTS];
int g_client_count=0;
pthread_mutex_t client_list_mutex=PTHREAD_MUTEX_INITIALIZER;

// 3. File Map (Which file is on which SS?)
typedef struct {
char path[MAX_PATH_LEN];
int ss_index;
//more info will be added here later
char owner[MAX_USERNAME_LEN];
}FileMapEntry;

#define MAX_FILES 1000
FileMapEntry g_file_map[MAX_FILES];
int g_file_count = 0;
pthread_mutex_t file_map_mutex = PTHREAD_MUTEX_INITIALIZER;

void* handle_client_commands(void* arg);
void* handle_ss_commands(void* arg);
void do_create(int client_fd, char* username, char* filename);
```

```c
void do_read(int client_fd, char* username, char* filename);

void do_create(int client_fd, char* username, char* filename) {
printf("Client %s requesting CREATE: %s\n", username, filename);
// 1. Lock file map to check existence
pthread_mutex_lock(&file_map_mutex);
for (int i = 0; i < g_file_count; i++) {
if (strcmp(g_file_map[i].path, filename) == 0) {
send(client_fd, RESP_CONFLICT, strlen(RESP_CONFLICT), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}
}
// 2. File doesn't exist. Find an SS to send it to.
pthread_mutex_lock(&ss_list_mutex);
if (g_ss_count == 0) {
send(client_fd, RESP_SS_DOWN, strlen(RESP_SS_DOWN), 0);
pthread_mutex_unlock(&ss_list_mutex);
pthread_mutex_unlock(&file_map_mutex);
return;
}
// 3. Round-robin policy
int ss_index = g_file_count % g_ss_count;
int ss_fd = g_ss_list[ss_index].conn_fd;
pthread_mutex_unlock(&ss_list_mutex);

// 4. Send command to the SS
char command_buf[MAX_MSG_LEN];
sprintf(command_buf, "%s %s\n", NM_CREATE, filename);
if (send(ss_fd, command_buf, strlen(command_buf), 0) < 0) {
perror("Failed to send to SS");
send(client_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}
// 5. Wait for ACK from SS
char ss_resp[MAX_MSG_LEN];
memset(ss_resp, 0, MAX_MSG_LEN);
if (read(ss_fd, ss_resp, MAX_MSG_LEN - 1) <= 0) {
printf("SS failed to respond to CREATE\n");
send(client_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}

// 6. If SS says OK, commit to our map
if (strncmp(ss_resp, RESP_OK, strlen(RESP_OK)) == 0) {
strcpy(g_file_map[g_file_count].path, filename);
strcpy(g_file_map[g_file_count].owner, username);
g_file_map[g_file_count].ss_index = ss_index;
g_file_count++;
```

```
send(client_fd, RESP_OK, strlen(RESP_OK), 0); // Success!
} else {
send(client_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR), 0);
}
pthread_mutex_unlock(&file_map_mutex);
}

void do_read(int client_fd, char* username, char* filename) {
printf("Client %s requesting READ: %s\n", username, filename);
// 1. Lock map and find the file
pthread_mutex_lock(&file_map_mutex);
int file_index = -1;
for (int i = 0; i < g_file_count; i++) {
if (strcmp(g_file_map[i].path, filename) == 0) {
file_index = i;
break;
}
}
if (file_index == -1) {
send(client_fd, RESP_NOT_FOUND, strlen(RESP_NOT_FOUND), 0);
pthread_mutex_unlock(&file_map_mutex);
return;
}
// 2. (ACL Stub: We'll add this in Phase 2)
// For now, just let them read

// 3. Get the SS's info
int ss_index = g_file_map[file_index].ss_index;
pthread_mutex_unlock(&file_map_mutex); // Done with file map

// 4. Get SS client-facing IP/port
pthread_mutex_lock(&ss_list_mutex);
char ss_ip[INET_ADDRSTRLEN];
int ss_port = g_ss_list[ss_index].client_port;
strcpy(ss_ip, g_ss_list[ss_index].ip);
pthread_mutex_unlock(&ss_list_mutex);

// 5. Send the referral to the client
char response_buf[MAX_MSG_LEN];
sprintf(response_buf, "%s %s %d\n", RESP_SS_INFO, ss_ip, ss_port);
send(client_fd, response_buf, strlen(response_buf), 0);
}

// This is the function that each thread will run
void *handle_connection(void *arg){
int conn_fd = *((int*)arg);
char buffer[MAX_MSG_LEN];
memset(buffer, 0, MAX_MSG_LEN);

// 1. Read the HELLO message (the handshake)
if (read(conn_fd, buffer, MAX_MSG_LEN - 1) <= 0) {
```

```
printf("Handshake failed. Closing connection.\n");
close(conn_fd);
return NULL;
}

printf("Handshake received: %s\n", buffer);

// 2. Decide WHO it is
if (strncmp(buffer, C_INIT, strlen(C_INIT)) == 0) {
// --- It's a CLIENT ---
char username[MAX_USERNAME_LEN];
sscanf(buffer, "%*s %s", username); // Parse the username
// Add to global client list (MUST be thread-safe)
pthread_mutex_lock(&client_list_mutex);
g_client_list[g_client_count].conn_fd = conn_fd;
strcpy(g_client_list[g_client_count].username, username);
g_client_count++;
pthread_mutex_unlock(&client_list_mutex);
printf("Registered new client: %s\n", username);
send(conn_fd, RESP_OK, strlen(RESP_OK), 0); // Send "OK"
// Pass to the client command loop
handle_client_commands(arg);
} else if (strncmp(buffer, S_INIT, strlen(S_INIT)) == 0) {
// --- It's a STORAGE SERVER ---
char ip[INET_ADDRSTRLEN];
int client_port; // The port for *clients* to connect to
// S_INIT <ip> <nm_port> <client_port>
// We only care about the client-facing IP and port
sscanf(buffer, "%*s %s %*d %d", ip, &client_port);
// Add to global SS list (MUST be thread-safe)
pthread_mutex_lock(&ss_list_mutex);
g_ss_list[g_ss_count].conn_fd = conn_fd; // This is the NM-SS command socket
strcpy(g_ss_list[g_ss_count].ip, ip);
g_ss_list[g_ss_count].client_port = client_port;
g_ss_count++;
pthread_mutex_unlock(&ss_list_mutex);
printf("Registered new Storage Server at %s:%d\n", ip, client_port);
send(conn_fd, RESP_OK, strlen(RESP_OK), 0); // Send "OK"

// Pass to the SS command loop
handle_ss_commands(arg);

} else {
printf("Unknown handshake. Closing connection.\n");
close(conn_fd);
free(arg);
}
// The thread exits when the helper function (handle_client/ss_commands) returns
printf("Connection handler thread exiting.\n");
return NULL;
}
```

```
void* handle_client_commands(void* arg) {
int client_fd = *((int*)arg); // Get the FD
char buffer[MAX_MSG_LEN];
char cmd[MAX_MSG_LEN];
char arg1[MAX_PATH_LEN]; // Use MAX_PATH_LEN for filenames

char username[MAX_USERNAME_LEN] = "unknown";
pthread_mutex_lock(&client_list_mutex);
for(int i=0; i<g_client_count; i++) {
if(g_client_list[i].conn_fd == client_fd) {
strcpy(username, g_client_list[i].username);
break;
}
}
pthread_mutex_unlock(&client_list_mutex);


while (1) {
memset(buffer, 0, MAX_MSG_LEN);
memset(cmd, 0, MAX_MSG_LEN);
memset(arg1, 0, MAX_PATH_LEN);

int bytes_read = read(client_fd, buffer, MAX_MSG_LEN - 1);

if (bytes_read <= 0) {
printf("Client %s disconnected.\n", username);
break; // Exit loop
}
int items_scanned = sscanf(buffer, "%s %s", cmd, arg1);
if (items_scanned <= 0) {
continue; // Ignore empty lines (like just hitting Enter)
}
if (strncmp(cmd, C_REQ_CREATE, strlen(C_REQ_CREATE)) == 0) {
do_create(client_fd, username, arg1);
} else if (strncmp(cmd, C_REQ_READ, strlen(C_REQ_READ)) == 0) {
do_read(client_fd, username, arg1);
} else {
// Now this is a *real* unknown command
send(client_fd, RESP_BAD_REQ, strlen(RESP_BAD_REQ), 0);
}
}
close(client_fd);
free(arg);
return NULL;
}

// This function runs in a loop for a STORAGE SERVER
void* handle_ss_commands(void* arg) {
int ss_fd = *((int*)arg);
char buffer[MAX_MSG_LEN];
```

```
while (1) {
memset(buffer, 0, MAX_MSG_LEN);
int bytes_read = read(ss_fd, buffer, MAX_MSG_LEN - 1);

if (bytes_read <= 0) {
printf("Storage Server disconnected.\n");
// CRITICAL: We need to mark this SS as DEAD
// (We'll add this logic later)
break; // Exit loop
}
// Right now, the SS only sends S_META_UPDATE (Phase 3)
// So for Phase 1, this loop just sits and waits.
printf("Received message from SS: %s\n", buffer);
}
close(ss_fd);
free(arg);
return NULL;
}

int main(int argc, char*argv[]){
printf("Starting server...\n");
int server_fd; //server file descriptor
server_fd=socket(AF_INET,SOCK_STREAM,0);
// AF_INET = Use IPv4 (the standard internet protocol)
// SOCK_STREAM = Use TCP (a reliable, streaming connection, not UDP)
// 0 = Use the default protocol (which is TCP for SOCK_STREAM)
// ALWAYS check for errors.
// A negative return value means the function failed.
if(server_fd<0){
// perror prints your message ("socket() failed")
// AND the specific system error (like "Permission denied").
perror("socket() function failed");
exit(1);
}
printf("1. Socket created successfully (fd=%d) \n",server_fd);

int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
perror("setsockopt(SO_REUSEADDR) failed");
exit(1);
}
//When your server exits (or crashes), the operating system keeps port 9000 in a "TIME_WAIT" state for about 30-60 seconds. It's "reserving" the port just in case any last-second data packets arrive.
//The Solution: SO_REUSEADDR, We need to tell the OS, "Hey, I'm the new server, and I give you permission to reuse that address right now.", We do this with a function called setsockopt().

struct sockaddr_in server_addr;
//Different kinds of networks (IPv4, IPv6, local domain sockets, etc.) each have their own struct to represent an address.
```

```
//AF_INET and struct sockaddr_in for IPv4, AF_INET6 and struct sockaddr_in6 for IPv6, etc.


int port=NM_LISTEN_PORT;
//we can also take this from command-line using if (argc > 1) port = atoi(argv[1]);

// Clear the entire struct to make sure it's clean
// We want all fields clean before assigning values.
// Prevents accidental garbage values from stack memory.
memset(&server_addr, 0, sizeof(server_addr));

server_addr.sin_family = AF_INET;
// Sets the address family to IPv4 (must match what we used in the socket else bind() will fail with
EINVAL (invalid argument))

server_addr.sin_addr.s_addr = INADDR_ANY;
// Sets the IP address to INADDR_ANY which means "listen on any available IP address" This is
what we want for a server.
//INADDR_ANY is a macro meaning “all network interfaces”.In binary, it’s 0.0.0.0.

server_addr.sin_port = htons(port);
// Sets the port number, converting from host byte order to network byte order
// htons() = "Host to Network Short"
// All TCP/IP headers use network byte order (big-endian).
// If you skip this conversion, other computers would interpret the bytes backwards — e.g.,
// 9000 (0x2328) might become 0x2823 = 10275, i.e., completely wrong port.

//bind() is the system call that tells the operating system, "I want the socket server_fd to be the
one responsible for port 9000 (we have assigned server_addr's sin_port as port)."
if (bind(server_fd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
// We cast our 'struct sockaddr_in' (which is internet-specific)
// to the generic 'struct sockaddr' that bind() expects.
perror("bind() failed");
//checking for errors. A common error here is "Address already in use," which means another
program (or your old, crashed server) is still holding onto port 9000.
exit(1);
}
printf("2. Socket bound to port %d\n", port);

// listen(server_fd, 5) tells the OS: "If I'm busy in my accept() loop handling a new connection, you
can hold up to 5 other new connections in a 'pending' queue. If a 6th connection arrives while the
queue is full, just reject it."
// Since your accept() loop is so fast (it just calls pthread_create and loops), the backlog is rarely
hit, but it's important to have. 5 or 20 is a fine number for this.
if (listen(server_fd, 5) < 0) {
perror("listen() failed");
exit(1);
}
printf("3. Server is listening on port %d...\n", port);

int client_fd; // This will be the NEW file descriptor for the client
```

```
struct sockaddr_in client_addr; // This will hold the CLIENT's address info
socklen_t client_len = sizeof(client_addr);
printf("Waiting for a client to connect...\n");
while (1) {
// now accept() blocks the program and waits for a connection
client_fd = accept(server_fd, (struct sockaddr *) &client_addr, &client_len);
if (client_fd < 0) {
perror("accept() failed");
continue; // Go back to the start of the loop and wait again
}

// printf("4. Client connected successfully! Waiting for a message...\n");
printf("4. Client connected! Handing off to a new thread...\n");

pthread_t tid;

int *new_sock=malloc(sizeof(int));
*new_sock=client_fd;
// We can't just pass &client_fd to the thread. Because the main loop will immediately loop back,
accept a new client, and change the value of client_fd. The first thread would suddenly have its
file descriptor changed! By malloc-ing new memory, we give each thread its own private copy of
the file descriptor.
// Why not pass &client_fd? If you passed the address of the stack variable client_fd from the main
thread, that memory could change when the main loop accepts the next connection; threads
would race and get wrong FDs. Allocating per-thread memory avoids that race.

if(pthread_create(&tid, NULL, handle_connection, (void*)new_sock)!=0){
perror("pthread_create() failed");
}
// Create a new thread:
// 1. &tid: Store the thread's ID here
// 2. NULL: Use default thread attributes
// 3. handle_client: The function the new thread should run
// 4. new_sock: The argument to pass to that function
// We need to pass the client_fd to the thread, but pthread_create only accepts a (void*) so we
cast.

}
close(server_fd);
return 0;
}

ss.c
#include "protocol.h" // Your protocol
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h> // For gethostbyname
```

```
#include <pthread.h> // For threads

// We'll need a global for our connection to the NM
int g_nm_fd;

// --- We will create this listener function in the next session ---
void* run_listener_thread(void* arg);

// --- This is the thread function for a CLIENT connection ---
void *handle_client_request(void *arg) {
int client_fd = *((int*)arg);
free(arg);
char buffer[MAX_MSG_LEN];
char cmd[MAX_MSG_LEN];
char filename[MAX_PATH_LEN];
memset(buffer, 0, MAX_MSG_LEN);

// 1. Read the one and only command from the client (e.g., SS_GET_FILE)
if (read(client_fd, buffer, MAX_MSG_LEN - 1) <= 0) {
printf("[SS-Client] Client disconnected before sending command.\n");
close(client_fd);
return NULL;
}
sscanf(buffer, "%s %s", cmd, filename);
// 2. Check which command it is
if (strncmp(cmd, SS_GET_FILE, strlen(SS_GET_FILE)) == 0) {
printf("[SS-Client] Received request for file: %s\n", filename);

// --- This is your file streaming logic from simpleserver ---
// TODO: This path is hardcoded. You'll make this dynamic later.
char local_path[MAX_PATH_LEN];
sprintf(local_path, "ss_data/ss1/%s", filename);
FILE *fp = fopen(local_path, "r");
if (fp == NULL) {
perror("fopen failed");
// We don't send an error, we just close the connection.
// The client's read() will fail.
} else {
char file_buffer[4096];
size_t bytes_read;
while ((bytes_read = fread(file_buffer, 1, 4096, fp)) > 0) {
if (send(client_fd, file_buffer, bytes_read, 0) == -1) {
printf("[SS-Client] Client disconnected during file transfer.\n");
break;
}
}
fclose(fp);
printf("[SS-Client] File transfer complete for: %s\n", filename);
}
} else {
// Unknown command
```

```c
printf("[SS-Client] Unknown command: %s\n", cmd);
}
close(client_fd);
return NULL;
}

// --- This is the SERVER part of the SS ---
// It runs in its own thread and just accepts clients
void* run_listener_thread(void* arg) {
int port = *((int*)arg);
free(arg);

int listener_fd;
struct sockaddr_in ss_server_addr;
// 1. Create the listener socket
listener_fd = socket(AF_INET, SOCK_STREAM, 0);
if (listener_fd < 0) {
perror("SS listener socket() failed");
pthread_exit(NULL);
}
// 2. Set SO_REUSEADDR (so you can restart it)
int opt = 1;
if (setsockopt(listener_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
perror("SS setsockopt() failed");
pthread_exit(NULL);
}
// 3. Setup the address struct
memset(&ss_server_addr, 0, sizeof(ss_server_addr));
ss_server_addr.sin_family = AF_INET;
ss_server_addr.sin_addr.s_addr = INADDR_ANY;
ss_server_addr.sin_port = htons(port);
// 4. Bind
if (bind(listener_fd, (struct sockaddr *) &ss_server_addr, sizeof(ss_server_addr)) < 0) {
perror("SS bind() failed");
pthread_exit(NULL);
}
// 5. Listen
if (listen(listener_fd, 10) < 0) {
perror("SS listen() failed");
pthread_exit(NULL);
}
printf("[SS-Listener] SS is now listening for clients on port %d\n", port);
// 6. The Accept Loop (stolen from simpleserver's main)
struct sockaddr_in client_addr;
socklen_t client_len = sizeof(client_addr);
int client_fd;
while(1) {
client_fd = accept(listener_fd, (struct sockaddr *) &client_addr, &client_len);
if (client_fd < 0) {
perror("SS accept() failed");
continue; // Keep listening
```

```
}
printf("[SS-Listener] New client connection accepted.\n");
// --- Spawn a new thread to handle this client's request ---
pthread_t client_tid;
int *new_sock = malloc(sizeof(int));
*new_sock = client_fd;
if (pthread_create(&client_tid, NULL, handle_client_request, (void *)new_sock) != 0) {
perror("SS failed to create client handler thread");
free(new_sock);
close(client_fd);
}
}
// This part is never reached
close(listener_fd);
return NULL;
}

int main(int argc, char* argv[]) {
// --- Argument Check ---
// We need the user to tell us what port to listen on
// e.g., ./bin/storage_server 9002
if (argc < 2) {
fprintf(stderr, "Usage: %s <client-facing-port>\n", argv[0]);
exit(1);
}
int client_port_for_ss = atoi(argv[1]);
// --- CLIENT PART: Connect to Name Server ---
printf("SS starting... connecting to Name Server...\n");
struct sockaddr_in nm_addr;
struct hostent *nm_server;

g_nm_fd = socket(AF_INET, SOCK_STREAM, 0);
if (g_nm_fd < 0) {
perror("SS socket() to NM failed");
exit(1);
}

nm_server = gethostbyname("localhost");
if (nm_server == NULL) {
fprintf(stderr, "ERROR, no such host as localhost\n");
exit(1);
}

memset(&nm_addr, 0, sizeof(nm_addr));
nm_addr.sin_family = AF_INET;
nm_addr.sin_port = htons(NM_LISTEN_PORT); // From protocol.h
memcpy(&nm_addr.sin_addr.s_addr, nm_server->h_addr_list[0], nm_server->h_length);

if (connect(g_nm_fd, (struct sockaddr *) &nm_addr, sizeof(nm_addr)) < 0) {
perror("SS connect() to NM failed");
exit(1);
```

```
}

printf("1. Connected to Name Server successfully.\n");
// --- REGISTRATION PART ---
char reg_buffer[MAX_MSG_LEN];
memset(reg_buffer, 0, MAX_MSG_LEN);
// S_INIT <ip> <nm_port> <client_port>
// We'll hardcode "127.0.0.1" for the IP
// We can send 0 for nm_port, our NM doesn't use it
sprintf(reg_buffer, "%s %s %d %d\n", S_INIT, "127.0.0.1", 0, client_port_for_ss);
printf("Sending registration: %s", reg_buffer);
write(g_nm_fd, reg_buffer, strlen(reg_buffer));
// Wait for the "OK" from the NM
memset(reg_buffer, 0, MAX_MSG_LEN);
if (read(g_nm_fd, reg_buffer, MAX_MSG_LEN - 1) <= 0) {
perror("NM closed connection during registration");
exit(1);
}
if (strncmp(reg_buffer, RESP_OK, strlen(RESP_OK)) != 0) {
printf("Name Server rejected registration: %s\n", reg_buffer);
exit(1);
}
printf("2. Registered with Name Server successfully.\n");
// --- SERVER PART ---
printf("3. Spawning client-listener thread...\n");
pthread_t listener_tid;
// We must pass the port number to the new thread
// We must use malloc to avoid a race condition
int *port_arg = malloc(sizeof(int));
*port_arg = client_port_for_ss;

if (pthread_create(&listener_tid, NULL, run_listener_thread, (void *)port_arg) != 0) {
perror("Failed to create listener thread");
exit(1);
}

printf("SS initialization complete. Main thread is now waiting for NM commands.\n");

// --- NM COMMAND LOOP ---
// The main thread now just waits for commands from the NM
char nm_buffer[MAX_MSG_LEN];
while(1) {
memset(nm_buffer, 0, MAX_MSG_LEN);
int bytes_read = read(g_nm_fd, nm_buffer, MAX_MSG_LEN - 1);
if (bytes_read <= 0) {
printf("Name Server disconnected. Exiting.\n");
exit(1); // If NM dies, SS should die
}
printf("Received command from NM: %s", nm_buffer);
char cmd[MAX_MSG_LEN];
char filename[MAX_PATH_LEN];
```

```c
sscanf(nm_buffer, "%s %s", cmd, filename);

if (strncmp(cmd, NM_CREATE, strlen(NM_CREATE)) == 0) {
printf("[SS-NM Loop] Received NM_CREATE for: %s\n", filename);

// --- This is your "CREATE" logic ---
// TODO: This path is hardcoded.
char local_path[MAX_PATH_LEN];
sprintf(local_path, "ss_data/ss1/%s", filename);

// Create the empty file
FILE *fp = fopen(local_path, "w");
if (fp == NULL) {
perror("fopen failed");
write(g_nm_fd, RESP_SRV_ERR, strlen(RESP_SRV_ERR));
} else {
fclose(fp);
write(g_nm_fd, RESP_OK, strlen(RESP_OK));
printf("[SS-NM Loop] Successfully created file: %s\n", filename);
}

} else {
printf("[SS-NM Loop] Unknown NM command: %s\n", cmd);
}
}

close(g_nm_fd);
return 0;
}
```

Makefile

```makefile
# --- Variables ---
# CC is our C Compiler
CC = gcc

# CFLAGS are the "compiler flags" (arguments)
# -g: Adds debug symbols (for GDB)
# -Wall: Turns on "all" warnings (very helpful)
# -Iinclude: Tells gcc to look for headers (like protocol.h) in the 'include' folder
# -pthread: Links the POSIX threads library
CFLAGS = -g -Wall -Iinclude -pthread

# LDFLAGS are the "linker flags"
# We need -pthread here too for the final linking stage
LDFLAGS = -pthread

# --- Targets ---

# 'all' is the default target. If you just type 'make', it will try to build these.
all: bin/name_server bin/storage_server bin/client
```

```
# Target for the Name Server
bin/name_server: name_server/nm.c
@mkdir -p bin # Create the 'bin' directory if it doesn't exist
$(CC) $(CFLAGS) -o bin/name_server name_server/nm.c $(LDFLAGS)
@echo "Compiled Name Server!"

# Placeholder for the Storage Server
bin/storage_server: storage_server/ss.c
@mkdir -p bin
$(CC) $(CFLAGS) -o bin/storage_server storage_server/ss.c $(LDFLAGS)
@echo "Compiled Storage Server!"

# Placeholder for the Client
bin/client:
@echo "Client target not yet implemented."

# 'clean' is a special target to clean up your project
clean:
rm -f bin/name_server bin/storage_server bin/client
@echo "Cleanup complete."

protocol.h
#ifndef PROTOCOL_H
#define PROTOCOL_H

/*
*
============================================================================
* protocol.h
*
* This header file defines the complete network protocol for the
* Distributed File System project.
*
* It must be included by:
* 1. client
* 2. name_server
* 3. storage_server
*
* It contains all shared constants, command strings, and response codes.
*
============================================================================
*/


/*
*
============================================================================
* SECTION 1: CORE NETWORK & BUFFER CONSTANTS
*
============================================================================
*/
```

```
/* The single "well-known" public port the Name Server listens on. */
#define NM_LISTEN_PORT 9001

/* Standard buffer size for sending/receiving command strings. */
#define MAX_MSG_LEN 1024

/* Standard max length for file paths, usernames, etc. */
#define MAX_PATH_LEN 256
#define MAX_USERNAME_LEN 64


/*
*
================================================================================
* SECTION 2: COMMANDS (CLIENT -> NAME SERVER)
*
================================================================================
*/

/* [Phase 1] Initial registration: "C_INIT <username>" */
#define C_INIT "C_INIT"

/* [Phase 1] Create file: "C_CREATE <filename>" */
#define C_REQ_CREATE "C_CREATE"

/* [Phase 1 & 3] Requests for file ops (NM replies with SS info) */
#define C_REQ_READ "C_READ" /* "C_READ <filename>" */
#define C_REQ_WRITE "C_WRITE" /* "C_WRITE <filename>" */
#define C_REQ_STREAM "C_STREAM" /* "C_STREAM <filename>" */

/* [Phase 2] Get file/user metadata (NM handles directly) */
#define C_REQ_VIEW "C_VIEW" /* "C_VIEW <flags>" (e.g., "C_VIEW -al") */
#define C_REQ_INFO "C_INFO" /* "C_INFO <filename>" */
#define C_REQ_LIST "C_LIST" /* "C_LIST" (Lists all registered users) */

/* [Phase 2] Delete file */
#define C_REQ_DELETE "C_DELETE" /* "C_DELETE <filename>" */

/* [Phase 2] Access control */
#define C_REQ_ADD_ACC "C_ADD_ACC" /* "C_ADD_ACC <filename> <username> <perm_flag>" */
#define C_REQ_REM_ACC "C_REM_ACC" /* "C_REM_ACC <filename> <username>" */

/* [Phase 3] Undo command */
#define C_REQ_UNDO "C_UNDO" /* "C_UNDO <filename>" */

/* [Phase 3] Execute command */
#define C_REQ_EXEC "C_EXEC" /* "C_EXEC <filename>" */


/*
```

```
*
================================================================================
* SECTION 3: COMMANDS (STORAGE SERVER -> NAME SERVER)
*
================================================================================
*/

/* [Phase 1] Initial registration */
/* "S_INIT <ip_addr> <nm_facing_port> <client_facing_port>" */
#define S_INIT "S_INIT"

/* [Phase 3] After a successful write, SS tells NM new metadata */
/* "S_META_UPDATE <filename> <word_count> <char_count>" */
#define S_META_UPDATE "S_META_UPDATE"


/*
*
================================================================================
* SECTION 4: COMMANDS (NAME SERVER -> STORAGE SERVER)
*
================================================================================
*/

/* [Phase 1] Tell SS to create a new, empty file */
#define NM_CREATE "NM_CREATE" /* "NM_CREATE <filename>" */

/* [Phase 2] Tell SS to delete a file */
#define NM_DELETE "NM_DELETE" /* "NM_DELETE <filename>" */

/* [Phase 3] Tell SS to revert the last change */
#define NM_UNDO "NM_UNDO" /* "NM_UNDO <filename>" */

/* [Phase 3] NM needs a file's content (for EXEC) */
#define NM_GET_FILE "NM_GET_FILE" /* "NM_GET_FILE <filename>" */


/*
*
================================================================================
* SECTION 5: COMMANDS (CLIENT -> STORAGE SERVER)
* (Used on the direct C-SS connection)
*
================================================================================
*/

/* [Phase 1] Request file contents for READ */
#define SS_GET_FILE "SS_GET_FILE" /* "SS_GET_FILE <filename>" */

/* [Phase 3] Request file contents for STREAM */
#define SS_GET_STREAM "SS_GET_STREAM" /* "SS_GET_STREAM <filename>" */
```

```
/* [Phase 3] The multi-step WRITE protocol */
/* 1. Lock: "SS_LOCK <filename> <sentence_number>" */
#define SS_LOCK "SS_LOCK"
/* 2. Update: "SS_UPDATE <word_index> <content>" */
#define SS_UPDATE "SS_UPDATE"
/* 3. Commit: "SS_COMMIT" (the ETIRW command) */
#define SS_COMMIT "SS_COMMIT"


/*
*
==========================================================================
* SECTION 6: UNIVERSAL RESPONSE PREFIXES (STATUS CODES)
*
==========================================================================
*/

/* --- Success (2xx) --- */

/* "200" (Generic success) */
#define RESP_OK "200"
/* "201" (SS to Client on successful lock) */
#define RESP_LOCKED "201"
/* "202 <ip> <port>" (NM to Client with SS info) */
#define RESP_SS_INFO "202"

/* --- Client Errors (4xx) --- */

/* "400" (Malformed command, bad args) */
#define RESP_BAD_REQ "400"
/* "403" (No permission, not owner, etc.) */
#define RESP_FORBIDDEN "403"
/* "404" (File, user, or sentence not found) */
#define RESP_NOT_FOUND "404"
/* "409" (File already exists on CREATE) */
#define RESP_CONFLICT "409"

/* --- Server Errors (5xx) --- */

/* "500" (Generic server-side crash) */
#define RESP_SRV_ERR "500"
/* "503" (NM can't reach SS for a task) */
#define RESP_SS_DOWN "503"
/* "504" (WRITE failed, sentence locked by another user) */
#define RESP_LOCKED_ERR "504"


/*
*
==========================================================================
```

```
* SECTION 7: PROTOCOL ARGUMENT CONSTANTS
*
==================================================================================
*/

/* [Phase 2] For ADDACCESS command */
#define PERM_READ 'R'
#define PERM_WRITE 'W'

/* [Phase 2] For VIEW command */
#define FLAG_ALL 'a'
#define FLAG_LONG 'l'

/* [Phase 3] For STREAM command */
/* Delay in microseconds (0.1 seconds = 100,000 us) */
#define STREAM_DELAY_US 100000


#endif // PROTOCOL_H
```

protocol.md
OSN Course Project: Final Network Protocol Specification

1. Overview

This document defines the complete network protocol for the **fault-tolerant, hierarchical** Distributed File System. It details all communication flows between the three components:
* **Client (`client`)**
* **Name Server (`name_server` or NM)**
* **Storage Server (`storage_server` or SS)**

This is the definitive "contract" for the project. All messages and constants are defined in `protocol.h`.

2. General Principles

* **Transport:** All communication uses **TCP**.
* **Format:** All messages are **text-based (ASCII)** and **newline-terminated (`\n`)**. Servers must read from sockets in a loop until a `\n` is found.
* **Structure:** `COMMAND_PREFIX <arg1> <arg2> ...\n`. All prefixes are in `protocol.h`.
* **Paths:** All file and folder paths are **absolute** (e.g., `/`, `/docs/file.txt`). The root is `/`.
* **Responses:** All server responses start with a 3-digit Status Code (from `protocol.h`).
* `CODE <optional_payload>\n`
* `200 OK` (Success)
* `202 127.0.0.1 9002` (Success, with SS info)
* `404 File not found` (Error)
* **Connections:**
* **NM-Client & NM-SS:** A single, persistent TCP connection is maintained. A `read()` returning 0 signals a disconnect/crash, which the NM must handle.
* **Client-SS:** A new, temporary TCP connection is made for each data operation (`READ`, `WRITE`, `STREAM`, etc.).

3. Component Initialization

3.1. Client Initialization

1. Client starts, prompts user for `username`.
2. Client connects to the NM at `NM_LISTEN_PORT`.
3. **Client -> NM:** `C_INIT <username>\n`
4. NM registers the client and keeps the connection.

3.2. Storage Server Initialization & Recovery

This flow handles both a *new* SS and a *recovering* (restarting) SS.

1. SS starts. It's given its IP, NM-facing port, and Client-facing port.
2. SS listens on its two ports.
3. SS connects to the NM at `NM_LISTEN_PORT`.
4. **SS -> NM:** `S_INIT <ip> <nm_facing_port> <client_facing_port>\n`
5. **NM Logic:**
* **If New SS:** NM adds it to the pool of available servers.
* **If Recovering SS:** NM marks it as "RECOVERING". It finds all files this SS *should* have (as Primary or Replica). For each file, it finds a "good" copy on another live SS.
* **NM -> SS (Recovering):** `NM_PULL_SYNC <path> <source_ss_ip> <source_ss_port>\n`
* (This tells the recovering SS to connect to the "good" SS and pull the latest file version. This must be done for all files and checkpoints.)
* After all sync commands are *sent*, the NM marks the SS as "ALIVE".

4. Response Code Summary (Reference)

Code	`protocol.h` Name	Meaning
200	`RESP_OK`	**OK:** Generic success.
201	`RESP_LOCKED`	**OK (Locked):** SS to Client. Sentence lock acquired.
202	`RESP_SS_INFO`	**OK (Info):** NM to Client. Payload is the IP/Port of the SS to contact.
400	`RESP_BAD_REQ`	**Bad Request:** Malformed command, wrong args.
403	`RESP_FORBIDDEN`	**Forbidden:** No permission (not owner, no R/W access, etc.).
404	`RESP_NOT_FOUND`	**Not Found:** File, folder, or sentence index does not exist.
405	`RESP_NOT_EMPTY`	**Not Empty:** Cannot delete a folder that is not empty.
409	`RESP_CONFLICT`	**Conflict:** Item already exists at this path (on `CREATE`).
500	`RESP_SRV_ERR`	**Internal Server Error:** A generic error on the NM or SS.
5.0.3	`RESP_SS_DOWN`	**SS Unavailable:** All replicas for this file are currently down.
504	`RESP_LOCKED_ERR`	**Sentence Locked:** Cannot lock; already locked by another user.

5. Data Payload Formats (Reference)

* **`VIEW` Payload:** A single string, items separated by `;`, fields by `,`.
* `D,/docs,userA;F,/file.txt,userB,69,420;`

* Format: `Type(D/F),Path,Owner[,WordCount,CharCount]`
* **`LISTREQS` Payload:**
* `reqID1,userA,R;reqID2,userC,W;`
* Format: `RequestID,Username,PermissionSought`
* **`LISTCHECKPOINTS` Payload:**
* `tag1,2025-10-10_14:32;tag2,2025-10-11_09:01;`
* Format: `Tag,Timestamp`

---

6. Core & Bonus Command Flows

6.1. File & Folder Operations

`CREATE <path>` (e.g., `/docs/file.txt`)
1. **Client -> NM:** `C_REQ_CREATE /docs/file.txt\n`
2. NM checks if parent (`/docs`) exists and if user has 'W' access in it.
3. NM checks if `/docs/file.txt` already exists (`409`).
4. **Replication:** NM selects a **Primary SS** (SS1) and a **Replica SS** (SS2).
5. **NM -> SS1 (Primary):** `NM_CREATE /docs/file.txt\n`
6. `SS1 -> NM: 200\n`
7. NM updates metadata: `/docs/file.txt -> {primary: SS1, replica: SS2, ...}`
8. **NM -> Client:** `200\n`
9. **Asynchronously:** NM sends the create command to the replica.
* **NM -> SS2 (Replica):** `NM_CREATE /docs/file.txt\n` (NM does not wait for this ACK).

`CREATEFOLDER <path>`
1. **Client -> NM:** `C_CREATE_FOLDER /new_folder\n`
2. NM checks parent (`/`) for 'W' access.
3. NM checks if `/new_folder` exists (`409`).
4. NM updates its *internal directory tree* metadata. (No SS is contacted).
5. **NM -> Client:** `200\n`

`READ <path>`
1. **Client -> NM:** `C_REQ_READ /docs/file.txt\n`
2. NM checks for file existence (`404`) and 'R'/'W' access (`403`).
3. **Fault Tolerance:**
* NM finds file metadata: `{primary: SS1, replica: SS2}`.
* NM checks liveness of SS1.
* **If SS1 is ALIVE:** `NM -> Client: 202 <SS1_ip> <SS1_port>\n`
* **If SS1 is DEAD:** NM checks liveness of SS2.
* **If SS2 is ALIVE:** `NM -> Client: 202 <SS2_ip> <SS2_port>\n`
* **If SS1 & SS2 are DEAD:** `NM -> Client: 503\n`
4. Client opens a **new connection** to the provided SS.
5. **Client -> SS:** `SS_GET_FILE /docs/file.txt\n`
6. **SS -> Client:** `[Raw file bytes streamed]`
7. SS closes connection.

`WRITE <path> <sentence_num>`
1. **Client -> NM:** `C_REQ_WRITE /docs/file.txt\n`
2. NM checks for existence (`404`) and 'W' access (`403`).

3. **Fault Tolerance:** NM finds Primary SS (SS1).
* **If SS1 (Primary) is ALIVE:** `NM -> Client: 202 <SS1_ip> <SS1_port>\n`
* **If SS1 is DEAD:** NM promotes Replica (SS2) to be the new Primary.
* `NM -> Client: 202 <SS2_ip> <SS2_port>\n`
4. Client opens a **new connection** to the provided SS (now the Primary).
5. **Client -> SS (Primary):** `SS_LOCK /docs/file.txt 3\n`
6. `SS -> Client: 201\n` (Or `504` if locked).
7. Client sends updates:
* **Client -> SS (Primary):** `SS_UPDATE 2 hello\n`
8. Client commits:
* **Client -> SS (Primary):** `SS_COMMIT\n`
9. SS (Primary) saves to its disk, releases lock.
10. **SS (Primary) -> Client:** `200\n` (Client is now done).
11. **Replication (Async):**
* **SS (Primary) -> NM:** `S_META_UPDATE /docs/file.txt 150 780\n`
* NM receives this update (e.g., new size/timestamp).
* NM commands the Replica (SS2) to pull the update from the Primary (SS1).
* **NM -> SS2 (Replica):** `NM_PULL_SYNC /docs/file.txt <SS1_ip> <SS1_port>\n`

`DELETE <path>` (File)
1. **Client -> NM:** `C_REQ_DELETE /docs/file.txt\n`
2. NM checks if user is owner (`403`).
3. NM finds all SSs for this file (SS1, SS2).
4. **NM -> SS1:** `NM_DELETE /docs/file.txt\n`
5. **NM -> SS2:** `NM_DELETE /docs/file.txt\n` (NM sends to all replicas).
6. After primary ACKs, NM removes file from its metadata.
7. **NM -> Client:** `200\n`

`DELETE <path>` (Folder)
1. **Client -> NM:** `C_REQ_DELETE /docs\n`
2. NM checks if user is owner (`403`).
3. NM checks its metadata to see if folder `/docs` is empty. If not: `NM -> Client: 405\n`
4. If empty, NM removes folder from its internal directory tree.
5. **NM -> Client:** `200\n`

`MOVE <source_path> <dest_path>`
1. **Client -> NM:** `C_MOVE /file.txt /docs/\n` (or `C_MOVE /file.txt /docs/new.txt`)
2. NM checks ownership of `file.txt` and 'W' access in `/docs`.
3. NM checks if destination path already exists (`409`).
4. This is a **pure metadata operation** on the NM. No SSs are contacted.
5. NM updates its internal map: `file.txt` metadata is now associated with the path `/docs/file.txt`.
6. **NM -> Client:** `200\n`

6.2. Metadata & Access Control

`VIEW <path>`
1. **Client -> NM:** `C_VIEW -al /docs\n` (Path can be `/` or any folder).
2. NM checks for 'R' access on `/docs`.
3. NM looks at its directory tree for all items inside `/docs`.
4. NM filters list based on user (unless `-a` is present).
5. NM constructs the payload string (see Section 5).

6. **NM -> Client:** `200 D,/docs/sub,userA;F,/docs/file.txt,userB,69,420;\n`

`INFO <path>`
1. **Client -> NM:** `C_INFO /docs/file.txt\n`
2. NM checks 'R' access.
3. NM gathers all metadata (Owner, Timestamps, Size, Full ACL) from its internal state.
4. NM formats this into a human-readable string (payload format is flexible).
5. **NM -> Client:** `200 Owner: userB\nSize: 420 bytes\n...\n`

`LIST` (Users)
1. **Client -> NM:** `C_REQ_LIST\n`
2. NM iterates its list of all *ever-registered* usernames.
3. **NM -> Client:** `200 userA;userB;kaevi;\n` (Payload is flexible, e.g., newline-separated).

`ADDACCESS` / `REMACCESS`
1. **Client -> NM:** `C_ADD_ACC /file.txt userC W\n` (or `C_REM_ACC /file.txt userC`)
2. NM checks if requestor is the *owner* of `/file.txt` (`403`).
3. NM updates its internal ACL for `/file.txt`.
4. **NM -> Client:** `200\n`

`REQACCESS <path> <R/W>` (Bonus)
1. **ClientA -> NM:** `C_REQ_ACCESS /file.txt R\n`
2. NM checks if ClientA *already* has access. If yes, `200\n`.
3. If no, NM adds to a "Pending Requests" list: `reqID1 -> {file, userA, R, PENDING}`.
4. **NM -> ClientA:** `200 Request submitted.\n`

`LISTREQS <path>` (Bonus)
1. **ClientB (Owner) -> NM:** `C_LIST_REQS /file.txt\n`
2. NM checks if ClientB is owner (`403`).
3. NM finds all pending requests for `/file.txt`.
4. NM formats payload (see Section 5).
5. **NM -> ClientB:** `200 reqID1,userA,R;reqID2,userD,W;\n`

`APPROVEREJ <req_id> <A/D>` (Bonus)
1. **ClientB (Owner) -> NM:** `C_APPROVE_REQ reqID1 A\n`
2. NM finds `reqID1` and verifies ClientB is the owner of the associated file.
3. **If 'A' (Approve):** NM performs the `ADDACCESS` logic internally.
4. **If 'D' (Deny):** NM does nothing to the ACL.
5. NM *deletes* the pending request (`reqID1`).
6. **NM -> ClientB:** `200\n`

6.3. Advanced File Operations

`STREAM <path>`
* Flow is **identical to `READ`**.
* The *only* difference is the Client-SS command:
* **Client -> SS:** `SS_GET_STREAM /docs/file.txt\n`
* The SS, upon seeing `SS_GET_STREAM`, will `send()` one word at a time, followed by `usleep(STREAM_DELAY_US)`.

`UNDO <path>`

1. **Client -> NM:** `C_REQ_UNDO /file.txt\n`
2. NM checks 'W' access (`403`).
3. NM finds the file's Primary SS (SS1).
4. **NM -> SS1 (Primary):** `NM_UNDO /file.txt\n`
5. `SS1 -> NM: 200\n` (SS1 swaps `file.txt.bak` with `file.txt`).
6. **SS1 -> NM:** `S_META_UPDATE /file.txt 100 500\n` (Tells NM the new size).
7. **NM -> Client:** `200\n`
8. **Asynchronously:** NM tells the replica (SS2) to sync this undo.
* **NM -> SS2 (Replica):** `NM_PULL_SYNC /file.txt <SS1_ip> <SS1_port>\n`

`EXEC <path>`
1. **Client -> NM:** `C_REQ_EXEC /script.sh\n`
2. NM checks for 'R' access (`403`).
3. NM finds a live SS (Primary or Replica) for the file (e.g., SS1).
4. **NM -> SS1:** `NM_GET_FILE /script.sh\n` (NM acts as a client to the SS).
5. **SS1 -> NM:** `[Raw file bytes streamed]`
6. NM saves content to a temp file (e.g., `/tmp/exec_userA.sh`).
7. NM `popen()`s the script.
8. NM sends `200\n` to Client to signal success (and that output is coming).
9. **NM -> Client:** `[stdout from popen is streamed]`
10. NM `pclose()`s, deletes temp file, and the response is complete.

`CHECKPOINT <path> <tag>` (Bonus)
1. **Client -> NM:** `C_REQ_CHECKPOINT /file.txt my_tag\n`
2. NM checks 'W' access (`403`).
3. NM finds Primary SS (SS1) and Replica SS (SS2).
4. **NM -> SS1 (Primary):** `NM_CHECKPOINT /file.txt my_tag\n`
5. `SS1 -> NM: 200\n` (SS1 copies `file.txt` to `file.txt.chk.my_tag`).
6. **NM -> Client:** `200\n`
7. **Asynchronously:** NM relays the command to the replica.
* **NM -> SS2 (Replica):** `NM_CHECKPOINT /file.txt my_tag\n`

`REVERT <path> <tag>` (Bonus)
1. **Client -> NM:** `C_REQ_REVERT /file.txt my_tag\n`
2. NM checks 'W' access (`403`).
3. NM finds Primary SS (SS1).
4. **NM -> SS1 (Primary):** `NM_REVERT /file.txt my_tag\n`
5. `SS1 -> NM: 200\n` (SS1 creates `.bak`, then copies `.chk.my_tag` over `file.txt`).
6. **SS1 -> NM:** `S_META_UPDATE /file.txt 80 350\n` (Tells NM the new size).
7. **NM -> Client:** `200\n`
8. **Asynchronously:** NM tells replica (SS2) to pull the change.
* **NM -> SS2 (Replica):** `NM_PULL_SYNC /file.txt <SS1_ip> <SS1_port>\n`

`LISTCHECKPOINTS <path>` (Bonus)
1. **Client -> NM:** `C_REQ_LIST_CHKPTS /file.txt\n`
2. NM checks 'R' access (`403`).
3. NM finds *any* live replica (e.g., SS1).
4. **NM -> Client:** `202 <SS1_ip> <SS1_port>\n`
5. Client opens new connection to SS1.
6. **Client -> SS1:** `SS_LIST_CHKPTS /file.txt\n`
7. SS1 scans its disk for `file.txt.chk.*`, formats payload (see Section 5).

```
8. **SS1 -> Client:** `200 tag1,ts1;tag2,ts2;\n`

#### `VIEWCHECKPOINT <path> <tag>` (Bonus)
1. **Client -> NM:** `C_REQ_VIEW_CHKPT /file.txt my_tag\n`
2. NM checks 'R' access (`403`).
3. NM finds *any* live replica (e.g., SS1).
4. **NM -> Client:** `202 <SS1_ip> <SS1_port>\n`
5. Client opens new connection to SS1.
6. **Client -> SS1:** `SS_GET_CHKPT /file.txt my_tag\n`
7. **SS1 -> Client:** `[Raw bytes of the checkpoint file streamed]`
8. SS1 closes connection.
```

You have done an incredible amount of work, and it's excellent.

Based on all the files you've provided, you have **fully implemented 99% of Phase 1**.

Your `nm.c` and `ss.c` files correctly implement all the server-side logic for every single task in the plan. The only thing missing is the "Teammate 2" component: the `client` executable to test it all.

Here is a detailed breakdown of your status:

✅ Name Server (`nm.c`): 100% Done

Your `name_server` code is a perfect implementation of all its Phase 1 responsibilities.

- **Task 1 (NM Skeleton): Done.** Your `main()` function correctly implements the `socket` / `setsockopt` / `bind` / `listen` setup and the `accept` -loop with `pthread_create`. This is the "thread-per-connection" model, implemented perfectly.
- **Task 2 (NM Registration): Done.** Your `handle_connection` is the ideal "handshake" function. It correctly reads the first message, uses `strncmp` to identify `C_INIT` vs. `S_INIT`, parses the info, and safely (using mutexes) adds the new connections to your global lists.
- **Task 3 (NM `CREATE`): Done.** Your `do_create` function is flawless. It correctly implements the "delegation" logic:
  1. Locks `file_map_mutex` to check for conflicts.
  2. Locks `ss_list_mutex` to pick an SS (using the round-robin policy).
  3. `send` s `NM_CREATE` to the SS.
  4. Blocks with `read()` waiting for the SS's ACK.
  5. Commits to `g_file_map` only on success.
- **Task 4 (NM `READ`): Done.** Your `do_read` function is a perfect "matchmaker." It correctly locks the maps to find the file's `ss_index`, then finds the `ss_ip` and `ss_port`, and sends the `RESP_SS_INFO` referral to the client.

Note on Code Quality: Your memory management in `nm.c` is **correct and safe**. Passing `arg` to the sub-functions (`handle_client_commands` / `handle_ss_commands`) and having *them* call `free(arg)` at the end of the thread's life is exactly the right way to do it.

✅ Storage Server (`ss.c`): 100% Done

Your `storage_server` code correctly implements the complex "Hybrid" role.

- **Task 1 (SS Skeleton): Done.** You've built the hybrid perfectly. The `main` function acts as a *client* (connecting to the NM), while it also spawns `run_listener_thread` to act as a *server* (listening for clients).

- **Task 2 (SS Registration): Done.** Your `main` function correctly connects to the NM, `send`s the `S_INIT` registration message, waits for the `RESP_OK`, and then enters the main "NM Command Loop" (`while(1) { read(g_nm_fd, ... }`).
- **Task 3 (SS `CREATE`): Done.** Your NM Command Loop in `main` correctly checks for `NM_CREATE`, `fopen`s the file in `"w"` mode to create it, and `write`s `RESP_OK` or `RESP_SRV_ERR` back to the NM.
- **Task 4 (SS `READ`): Done.** Your `handle_client_request` thread (spawned by the listener) correctly implements the "one-shot" transaction. It `read`s the `SS_GET_FILE` command, `fopen`s the file in `"r"` mode, and uses the `fread` / `send` loop to stream the data. Closing the socket at the end is the correct signal for "end of file."

✅ Supporting Files: 100% Done

- `protocol.h` **: Done.** It contains all the necessary commands and response codes for Phase 1 (`C_INIT`, `S_INIT`, `C_REQ_CREATE`, `C_REQ_READ`, `NM_CREATE`, `SS_GET_FILE`, `RESP_SS_INFO`, etc.).
- `Makefile` **: Done.** This correctly builds your two executables.
  - **Minor Note:** Your `Makefile` looks for files in `name_server/nm.c` and `storage_server/ss.c`. Since you've pasted them as `nm.c` and `ss.c`, just make sure your local directories match the `Makefile` paths. This is not a code error, just a pathing note.

🚀 What's Left: The Final 1% (The Client)

You have built the entire backend. The only part of Phase 1 left is to **build the `client` executable** that will talk to your backend.

Your `Makefile` already has a target for `bin/client`, but it's a placeholder. You need to create `client.c` that does the following:

1. Connects to the NM (`localhost:9001`).
2. Sends `C_INIT <username>\n`.
3. Waits for `RESP_OK`.
4. Sends `C_REQ_CREATE test.txt\n`.
5. Waits for `RESP_OK`.
6. Sends `C_REQ_READ test.txt\n`.
7. `read`s the response. If it's `RESP_SS_INFO <ip> <port>`:
   - Parses the IP and port.
   - Opens a **new socket** and `connect`s to that SS.
   - `send`s `SS_GET_FILE test.txt\n`.
   - Enters a `while(read(...) > 0)` loop, saving the received file data to a local file (e.g., `downloaded_test.txt`).
   - `close`s the SS connection when `read` returns 0.

You've built the system perfectly. Now you just need to build the "key" to use it.

Would you like help writing the `client.c` file to test your system?

okay 1 doubt what if my name server thing is stopped and started again for example using control c
would that cause issues
like would my files and data on who created them still be there

Yes, that would cause a major issue. **All your metadata would be lost.**

Here's a breakdown of what happens:

The Problem: Volatile Memory

Your Name Server (`nm.c`) stores all its critical information in global variables:

- `g_ss_list` (which Storage Servers are connected)
- `g_client_list` (which Clients are connected)
- `g_file_map` (which file is on which SS, who the owner is, etc.)

These variables exist in the process's **RAM (memory)**. When you hit `Ctrl+C`, the operating system terminates the `name_server` process and reclaims all of its memory.

When you start the server again, it's a brand new process. All those global variables are re-initialized: `g_ss_count = 0`, `g_file_count = 0`, and all the lists are empty.

Your NM has no memory of what happened before it was restarted.

Consequences of Restarting

1. **Storage Servers Disconnect:** Your `ss.c` code is designed to exit if the NM connection breaks:

```c
if (bytes_read <= 0) {
    printf("Name Server disconnected. Exiting.\n");
    exit(1); // <-- This happens
}
```

   So, when your NM dies, all your connected SSs will also exit.

2. **Total "Amnesia":** When you restart the NM, and then restart the SSs:
   - The SSs will reconnect and re-register. The NM's `g_ss_list` will be repopulated.
   - The NM's `g_file_map` is **still empty**. It has no idea that `test.txt` (or any other file) ever existed.
3. **Real-World Issues:**
   - `READ` **Fails:** A client connects and tries `C_REQ_READ test.txt`. The NM looks in its empty `g_file_map`, finds nothing, and replies with `RESP_NOT_FOUND`.
   - `CREATE` **Overwrites:** A client tries `C_REQ_CREATE test.txt`. The NM (thinking it's a new file) tells an SS to create it. The SS `fopen(".../test.txt", "w")`, which **wipes out the old file** and replaces it with a new, empty one.

The Solution (For Later)

The solution to this problem is called **persistence**. To make your NM "fault-tolerant," it cannot store its state *only* in memory. It must also save its state to the **disk**.

A simple way to do this (which you might implement in a later phase) would be:

1. **On Startup:** The NM's `main()` function first reads a file (e.g., `nm_state.txt`) to rebuild its `g_file_map` in memory.
2. **On `CREATE`:** After `do_create` successfully adds a file, it also appends a new line to `nm_state.txt` with the file info.

For now, your Phase 1 implementation is correct. This "amnesia" problem is just the *next* big challenge in building a truly robust distributed system.

this is what im trying to implement, does it require this persistence stuff?
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.
One teammate has to create the team. They need to click on this link. This opens a page of registered teams.
When a team's member visits the link first, they need to create a new team using the team name. Nothing more, nothing less.
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.
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:
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.
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.
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:
[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

[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

[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>

[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 file write access, should return appropriate here)
<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.
[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.
[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 above-mentioned details should be there

[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>

[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 of 0.1 seconds

Note: If the storage server goes down mid-streaming, an appropriate error message should be displayed to the user.
[10] List Users: Users can view a list of all users registered in the system.
LIST # Lists all users in the system

[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 user
REMACCESS <filename> <username> # Removes all access

[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:
[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.
[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.
[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.
[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.
[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
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.
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.
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 checkpoint
REVERT <filename> <checkpoint_tag> # Reverts the file to the specified checkpoint
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 sentence. There are only 2 sentences.
ERROR: Sentence index out of range.  # Similarly for word indexes (negative or > number of words + 1, should results in errors)

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 mistaken hollow lil gei-fwen pocket-sized mouse.
Client: 6 pocket-sized  # New sentence : Im just a deeply mistaken hollow pocket-sized lil gei-fwen mouse.
Client: ETIRW
Write Successful!

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

Client: WRITE mouse.txt 1  # Inserting a sentence delimiter
Client: 5 and AAD. aaaah # New sentences : [I dont like T-T PNS and AAD.]* [aaaah]. Currently active status remains with the index at index 1
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 like T-T PNS and AAD. aaaah.

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 LMAAO.txt is as follows:
echo "Running diagnostics..."
ls
echo "Done!"

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

Grading
The whole implementation grading can be broadly divided into 4 parts:
User Functionalities - 150 marks
System Requirements - 40 marks
Specifications - 10 marks

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!

Yes, it **absolutely** requires persistence.

Your previous question was a perfect insight, and this project spec confirms it. Your current implementation (where the NM loses all data on restart) would not pass the requirements.

✅ Yes, It's a 10-Point Requirement

Look at the **"System Requirements"** section of the document you sent:

> `[40] System Requirements` `[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**.`

This means your "amnesia" problem is exactly what you need to solve.

🧠 How You Must Implement Persistence

The spec gives you two different persistence models, one for the SS and one for the NM.

1. Storage Server (SS) Persistence (You've already done this!)

- **Requirement:** The files themselves must survive an SS restart.
- **Your Code:** Your `ss.c` uses `fopen("ss_data/ss1/...", "w")` and `fopen("...", "r")`. This saves the file data to the **disk**, not RAM.
- **Result: You have already implemented this.** If your SS process stops and restarts, the files will still be in the `ss_data/ss1` folder.

2. Name Server (NM) Persistence (This is the new part)

This is the "amnesia" problem we discussed. The spec has two key rules for this:

1. **Failure is OK:** `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.`
2. **Metadata must be persistent:** `...associated metadata (like access control lists) must be stored persistently.`

These two rules tell you *how* to solve the problem. The NM doesn't need to be "live" 24/7, but when it restarts, it **must reload all its old metadata**.

You can't rely *only* on the SSs to rebuild your state, because the SSs don't know about metadata like **owners** or **Access Control Lists (ACLs)**. The SSs only know about filenames.

🚀 Your Action Plan for Persistence

Here is the correct way to implement persistence based on your spec:

1. For the Name Server (`nm.c`):

- **Create a Metadata File:** You must save your critical metadata (the `g_file_map` and any ACL data) to a file on disk. For example, `nm_metadata.db` .
- **On Startup (in `main`):**
  - *Before* you `bind` and `listen` , you must try to `fopen("nm_metadata.db", "r")` .
  - Read this file and use it to completely rebuild your `g_file_map` and ACL lists in memory.
- **On Any Change (in `do_create` , `do_delete` , `do_addaccess` , etc.):**
  - After you modify the `g_file_map` or ACLs in memory (inside your mutex lock), you must **immediately re-save the entire list** to `nm_metadata.db` .
  - This is slow, but it's the simplest and safest way to guarantee persistence. (e.g., `FILE* fp = fopen("nm_metadata.db", "w");` and loop through your `g_file_map` writing each entry).

2. For the Storage Server (`ss.c`):

- The spec has a requirement: `[SS Initialization]: ...sends vital details... This information includes: ... **a list of files on it**.`
- **On Startup (in `main`):**
  - *Before* you `connect` to the NM, you must scan your `ss_data/ss1/` directory.
  - Build a list of all filenames you have (e.g., `test.txt` , `foo.bar`).
  - You need to **modify your `S_INIT` protocol** to send this list to the NM. (e.g., `S_INIT <ip> <port> test.txt,foo.bar\n`).
- **Why?** This allows the NM to check its own loaded metadata against what the SS *actually* has. This helps handle cases where the NM's metadata might be out of sync.

So, in short: **Yes, you must save your NM's `g_file_map` to disk and reload it on startup.** This is a core requirement of the project.

Would you like me to help you sketch out the `save_metadata_to_disk()` and `load_metadata_from_disk()` functions for your `nm.c` file?

can you check below and tell me if phase 3 for teammate 1 requires teammate 2 to be finished by phase 2 stuff, basically if teammate1 finishes off code for phase 3, can they check it just using nc
3 - WorkFlow

This is a tight but achievable 12-day sprint. This detailed plan assumes you have already completed Phase 0 (defining your protocol and setting up your Git repository) before your exams
Here is the highly detailed, step-by-step breakdown of the 12-day sprint from October 30th to November 10th.
Phase 1: The Foundation (Deadline: Nov 2, 4 Days) - DONE

Goal: All three components (NM, SS, Client) can start, register, and perform a complete end-to-end CREATE and READ operation. This proves your core architecture and networking logic work.

To Be Done Together (Morning of Oct 30)

Final Protocol Review: Pull up your protocol.md and protocol.h files. Do one last review.

Make sure you both 100% agree on:

The exact format of INIT_SS, INIT_CLIENT, REQ_CREATE, REQ_READ.

The exact format of RESP_OK, RESP_ERR <CODE>, and RESP_SS_INFO <IP> <PORT>.

Your basic error codes (e.g., 404 for Not Found, 409 for "File Already Exists").

Teammate 1: Backend (NM + SS)

Task 1: Build Component Skeletons (Oct 30)

Name Server (NM):

Create nm.c.

Implement the main socket logic: socket(), bind() to a public port (e.g., 9000), listen().

Implement your connection handling loop (e.g., a while(1) with accept()).

For each new connection, spawn a new thread (or use select()) to handle it. This is essential for handling multiple SSs and Clients.

Storage Server (SS):

Create ss.c.

This is more complex: it's both a client and a server.

Server Part: socket(), bind(), listen() on two ports.

Port 1 (e.g., 9001): For commands from the NM.

Port 2 (e.g., 9002): For direct data connections from Clients.

Spawn threads (or use select()) to handle incoming connections on both ports.

Client Part: Immediately on startup, implement the client-side logic to socket() and connect() to the NM (e.g., at 127.0.0.1:9000).

Task 2: Implement Registration (Oct 31)

SS: After connecting to the NM, create and send the INIT_SS message. This message must contain its IP, its NM-facing port (9001), and its Client-facing port (9002).

NM: In your connection handler thread:

Receive the first message from a new connection.

If it's an INIT_SS message:

Create an in-memory data structure (e.g., a struct StorageServer ss_list[MAX_SS]).

Parse the IP and ports and store them in this list.

Keep this connection open for future commands.

If it's an INIT_CLIENT message (from T2):

Create a separate in-memory data structure (e.g., struct Client client_list[MAX_CLIENTS]).

Parse the username and store it, associating it with this connection's file descriptor.

Keep this connection open.

Task 3: Implement CREATE (Nov 1)

NM:

Create a new handler function for REQ_CREATE <filename>.

First, create your master file map (e.g., a hash map: file_map<filename, ss_id>).

Check if the file already exists in this map. If yes, send RESP_ERR 409 to the Client.

If no, pick an SS to store the file (e.g., simple round-robin: ss_id = file_count % ss_count).

Look up the connection info for that ss_id from your ss_list.

Send a CMD_CREATE <filename> message to that SS on its NM-facing port.

Wait for an RESP_OK from the SS.

Once received, then add the file to your file_map.

Also, create your Access Control List (ACL) map. Add the creator as the owner: acl_map<filename, user_map<username, "RW">>.

Finally, send RESP_OK back to the Client.
SS:
In your NM-facing port handler, create a function for CMD_CREATE <filename>.
Parse the filename.
Physically create the file on disk (e.g., fopen("ss_data/ss1/filename.txt", "w")). Make sure the ss_data/ss1 directory exists.
fclose() the file.
Send RESP_OK back to the NM.
Task 4: Implement READ (Nov 2)
NM:
Create a handler for REQ_READ <filename>.
Look up the file in your file_map. If not found, send RESP_ERR 404.
Stub out access control: For now, just assume anyone can read.
Get the ss_id associated with the file.
Look up that SS in your ss_list.
Find its Client-facing IP and port (e.g., 127.0.0.1:9002).
Send a RESP_SS_INFO <ip> <port> message back to the Client.
SS:
In your Client-facing port handler, create a function that expects a connection.
The first message from this client will be GET_FILE <filename>.
Parse the filename.
Physically open that file for reading (e.g., fopen("ss_data/ss1/filename.txt", "r")).
In a loop, fread() data from the file into a buffer and send() that buffer to the client.
When fread() returns 0 (end of file), fclose() the file and close() the client's socket.
Teammate 2: Client
Task 1: Build Skeleton & Registration (Oct 30-31)
Create client.c.
Implement the startup:
Prompt user: "Enter username: ".
fgets() the username.
socket() and connect() to the NM's main port (e.g., 9000).
Send the INIT_CLIENT <username> message.
Implement the main command loop:
while(1)
printf("> ");
fgets() the full command line (e.g., "CREATE foo.txt").
Implement your command parser (e.g., using strtok()). This should split the input into cmd ("CREATE") and args ("foo.txt").
Task 2: Implement CREATE (Nov 1)
Inside your while loop, if cmd is "CREATE":
Check if args exist.
Format and send() the REQ_CREATE <args[0]> message to the NM.
recv() the response from the NM.
If RESP_OK, print "File created successfully.\n".
If RESP_ERR 409, print "Error: File already exists.\n".
Task 3: Implement READ (Nov 2)
This is your most complex task in this phase.
If cmd is "READ":
Format and send() the REQ_READ <args[0]> message to the NM.
recv() the response from the NM.
If RESP_ERR 404, print "Error: File not found.\n".

If the response is RESP_SS_INFO <ip> <port>:
Parse the ip and port from the message.
Create a brand new socket.
connect() to this new IP and port (the SS's Client-facing port).
Once connected, send() the GET_FILE <args[0]> message to the SS.
Enter a while loop:
bytes_recvd = recv(ss_socket, buffer, ...)
If bytes_recvd == 0, break the loop (connection closed by server).
printf("%.*s", bytes_recvd, buffer); (This prints the buffer correctly, even if it's not null-terminated).
When the loop finishes, close(ss_socket).
Phase 2: Full CRUD & Access Control (Deadline: Nov 6, 4 Days) - DONE
Goal: Implement all remaining file management (DELETE, VIEW, INFO), user management (LIST), and the entire Access Control system.
Teammate 1: Backend (NM + SS)
Task 1: Implement Full ACL System (Nov 3-4)
NM:
Your acl_map from Phase 1 needs to be robust. It should be a hash map (or similar) mapping a filename to another map: user_map<username, permission_enum>.
Create a handler for REQ_ADDACCESS <filename> <username> <permission>.
Get the requesting user's username from your client_list.
Check if this user is the owner of the file (look up the owner you stored during CREATE).
If not owner, send RESP_ERR 403 (Forbidden).
If owner, update the acl_map with the new user and permission. Send RESP_OK.
Create a handler for REQ_REMACCESS <filename> <username>. Do the same owner check.
If valid, remove the user from the acl_map[filename]. Send RESP_OK.
Task 2: Integrate ACLs into READ & CREATE (Nov 4)
NM:
Modify your REQ_READ handler: After checking if the file exists, check the acl_map. Does acl_map[filename][requesting_user] exist and is it R or RW? If not, send RESP_ERR 403.
Modify your REQ_CREATE handler: When you check if the file exists (RESP_ERR 409), this is correct. No other ACL check is needed.
Task 3: Implement DELETE (Nov 5)
NM:
Create a handler for REQ_DELETE <filename>.
Check if the requesting user is the owner. If not, RESP_ERR 403.
If yes, get the ss_id from file_map.
Send CMD_DELETE <filename> to the SS.
Wait for RESP_OK from SS.
Once received, delete the file from your file_map and delete its entry from the acl_map.
Send RESP_OK to the Client.
SS:
Create a handler for CMD_DELETE <filename>.
Execute remove("ss_data/ss1/filename.txt").
Send RESP_OK back to the NM.
Task 4: Implement VIEW, INFO, LIST (Nov 6)
NM: These are all NM-only operations.
REQ_LIST_USERS: Iterate your client_list and build a single, newline-separated string of all usernames. Send this string back.
REQ_VIEW <flags>:
Iterate through your entire file_map.

For each file:

Check flags. If -a is not present, check the ACL. Does the requesting_user have R or RW access? If not, skip this file.

If they have access (or -a is present), add the filename to a response string.

If -l is present, you also need metadata. This is a design choice. The simplest way: your file_map entry must also store word_count, char_count, owner, etc. (stubbed as 0 for now).

Send the fully formatted (or data-packed) string back to the client.

REQ_INFO <filename>: Check for R access. If allowed, pull all metadata for that one file from your file_map and acl_map (owner, size, timestamps, full access list) and send it back.

Teammate 2: Client

Task 1: Implement Access Commands (Nov 3-4)

If cmd is "ADDACCESS": Parse args[0] (flag), args[1] (file), args[2] (user). Format and send REQ_ADDACCESS to NM. recv response, print "Access granted" or "Error: You are not the owner."

If cmd is "REMACCESS": Parse args[0] (file), args[1] (user). Send REQ_REMACCESS. recv and print status.

Task 2: Implement DELETE (Nov 5)

If cmd is "DELETE": Parse args[0] (file). Send REQ_DELETE. recv and print "File deleted" or "Error: You are not the owner."

Task 3: Implement VIEW, INFO, LIST (Nov 6)

LIST: Send REQ_LIST_USERS. recv the (potentially large) string. Print it.

VIEW:

Parse the flags (-a, -l).

Send REQ_VIEW <flags> to NM.

recv the data string from NM.

This is your main job: Write the client-side code to parse this data (e.g., file1.txt,owner1,0,0;file2.txt,owner2,0,0) and format it into the pretty ASCII tables shown in the project examples.

INFO:

Send REQ_INFO <filename>.

recv the detailed data string.

Parse and format it into the human-readable list (Owner: ..., Size: ...).

Phase 3: Advanced Features & System Requirements (Deadline: Nov 10, 4 Days)

Goal: Implement the complex, stateful features (WRITE, UNDO, STREAM, EXEC) and finalize all system requirements (Logging, Caching).

Teammate 1: Backend (NM + SS)

Task 1: Implement WRITE (Nov 7-8)

This is the hardest part. All logic is on the Storage Server.

NM: Modify REQ_READ handler to become REQ_FILE_OP. If op is WRITE, check for PERM_RW in ACL. If yes, send RESP_SS_INFO just like for READ.

SS:

Sentence Parser: Write helper functions: char* get_sentence(file_content, sent_num) and char* set_sentence(file_content, sent_num, new_sentence). This involves careful string manipulation, counting ., !, ?.

Lock Manager: Create a new global data structure on the SS (e.g., hash_map<filename, lock_info>). lock_info must store bool is_locked, int sentence_index, and string username_locking.

Modify Client-facing Handler:

It can now receive REQ_WRITE_LOCK <filename> <sentence_num>.

Handler must: Check lock manager. Is filename locked at any sentence? Is this specific sentence_num locked? (Spec says sentence is locked).

If lock_map[filename].is_locked and lock_map[filename].username != requesting_user, send RESP_ERR 504 (Locked).

If not locked, set lock_map[filename] = {true, sentence_num, username}. Send RESP_OK_LOCKED.

The SS must now hold the original file content in a temporary buffer in memory.

The SS now waits for UPDATE_SENTENCE <word_idx> <content> messages. For each one, modify the in-memory buffer, not the disk.

Finally, the SS will receive ETIRW.

On ETIRW (The "Commit"):

For UNDO: rename("file.txt", "file.txt.bak").

fopen("file.txt", "w") and fwrite() the entire modified buffer to disk.

Metadata: Calculate new word/char counts.

Release Lock: delete(lock_map[filename]).

Send RESP_OK to Client.

Crucial: Send an asynchronous UPDATE_METADATA <file> <wc> <cc> message to the NM so it can update its VIEW -l info.

NM: Create a handler for UPDATE_METADATA from SSs to update your file_map metadata.

Task 2: Implement UNDO (Nov 8)

NM: Create handler for REQ_UNDO. Check for RW permission. If ok, forward CMD_UNDO to SS.

SS: Create handler for CMD_UNDO <filename>. Check if file.txt.bak exists. If yes, rename("file.txt.bak", "file.txt"). Send RESP_OK to NM.

Task 3: Implement STREAM & EXEC (Nov 9)

SS: Create handler for GET_STREAM <filename>. This is almost identical to your GET_FILE handler, but:

You will fscanf(fp, "%s", word) to read one word at a time.

In your loop: send(word), send(" "), usleep(100000).

NM:

Create handler for REQ_EXEC <filename>.

Check for R access.

Connect as a client to the SS. Send GET_FILE <filename>. recv the entire file content into a buffer.

Save this buffer to a temporary file (e.g., /tmp/exec_user_123.sh).

Use popen("sh /tmp/exec_user_123.sh", "r") to run it.

Read the output from the popen pipe in a loop.

send() this output directly to the waiting Client.

pclose() and remove() the temp file.

Task 4: System Requirements (Nov 10)

Logging (NM + SS): Go through every single handler function you've written. Add a printf (or fprintf to a log file) with: [TIMESTAMP] [IP:PORT] [COMMAND] - STATUS.

Efficient Search (NM): Confirm your file_map and acl_map are hash maps or Tries (or a balanced BST). Not a linear array search.

Caching (NM): Implement a simple cache (e.g., another hash map) for REQ_READ/REQ_WRITE. cache<filename, ss_info>. Before looking in file_map, check the cache.

Teammate 2: Client

Task 1: Implement WRITE (Nov 7-8)

This is your hardest task. It's a state machine.

If cmd is "WRITE":

Parse filename and sentence_num.

Send REQ_FILE_OP <filename> WRITE to NM. recv RESP_SS_INFO.

Connect to the SS. Send REQ_WRITE_LOCK <filename> <sentence_num>.
recv response. If RESP_ERR 504, print "Error: Sentence is locked by another user." and break.
If RESP_OK_LOCKED, print "Sentence locked. Enter updates (e.g., '3 new_word') or 'ETIRW' to finish:"
Enter a new, special input loop:
fgets() the user's update line.
if(strcmp(line, "ETIRW\n") == 0):
send("ETIRW") to SS.
break this special loop.
Else:
Parse <word_idx> <content>.
send("UPDATE_SENTENCE <word_idx> <content>") to SS.
After the loop breaks: recv() the final RESP_OK from the SS. Print "Write successful.\n".
close(ss_socket).
Task 2: Implement UNDO (Nov 8)
If cmd is "UNDO": Send REQ_UNDO <filename> to NM. recv response. Print status.
Task 3: Implement STREAM & EXEC (Nov 9)
STREAM: Almost identical to READ.
Send REQ_FILE_OP <filename> READ to NM. Get SS info.
Connect to SS. Send GET_STREAM <filename>.
In your recv loop, printf the data as it arrives (don't buffer the whole file). The 0.1s delay will be visible to the user.
EXEC:
Send REQ_EXEC <filename> to NM.
Stay connected to the NM.
Enter a recv loop (just like READ). The NM will stream the stdout of the command to you.
printf the output as it arrives.
Task 4: System Requirements (Nov 10)
Error Handling: Go through every recv() call in your entire program. Check for RESP_ERR.
Create a helper function print_error(error_code) that uses a switch statement to print a friendly message for every code (403, 404, 409, 504, etc.).
To Be Done Together (Evening of Nov 10)
Final Integration Test: Get on a call.
Start 1 NM, 2 SSs, 3 Clients (with different usernames).
Concurrency Test: Client A locks sentence 1. Client B tries to lock sentence 1 (must fail).
Client B tries to lock sentence 2 (must succeed).
Fault Test: Client A STREAMs a large file from SS1. While it's streaming, kill -9 the SS1 process. Client A must not crash; it should show an error.
Recovery Test: Restart SS1. It must re-register with the NM. Client B must be able to immediately READ a file from SS1.
Checklist: Go through the project PDF, line by line, and check off every single feature.

Yes, absolutely. **You can (and should) test all of your Phase 3 backend work just using `nc`.**

Your (Teammate 1's) Phase 3 work is **not dependent** on Teammate 2's Phase 2 work (like making pretty `VIEW -l` tables or implementing `DELETE`).

Your work only depends on a "client" (in this case, `nc`) being able to send the correct protocol strings. `nc` is the perfect tool for this.

Here’s how you can test each Phase 3 feature:

🧪 How to Test Phase 3 with `nc`

First, make sure you have your `CREATE` and `READ` (Phase 1) working so you have files to test with.

1. Testing `UNDO` (Easy)

This is a simple, single-connection test.

1. Open `nc localhost 9000` (or your NM port).
2. Type `INIT_CLIENT teammate1\n` to register.
3. Type `REQ_UNDO somefile.txt\n` (assuming you have RW access).
4. Your NM will receive this, check your (stubbed) ACL, send `CMD_UNDO` to the SS, and the SS will rename the `.bak` file.
5. You should receive a `RESP_OK\n` in your `nc` terminal.

2. Testing `EXEC` (Easy)

This is also a single-connection test.

1. Open `nc localhost 9000`.
2. Type `INIT_CLIENT teammate1\n`.
3. Type `REQ_EXEC script.sh\n`.
4. Your NM will get the file from the SS, `popen()` it, and stream the output back to you.
5. You will see the `stdout` of the script appear directly in your `nc` terminal.

3. Testing `STREAM` (2-Terminal Test)

This is a two-connection "referral" test.

1. **Terminal 1 (NM):** `nc localhost 9000`
2. Type `INIT_CLIENT teammate1\n`.
3. Type `REQ_FILE_OP somefile.txt READ\n` (or whatever your `STREAM` op is).
4. You will get `RESP_SS_INFO 127.0.0.1 9002\n` (or your SS client port).
5. **Terminal 2 (SS):** `nc 127.0.0.1 9002`
6. Type `GET_STREAM somefile.txt\n`.
7. You will see the file's content appear word... by... word... as your `usleep` code runs.

4. Testing `WRITE` (2-Terminal Test)

This is the most complex, but most important, test. It's how you'll debug your SS lock manager.

1. **Terminal 1 (NM):** `nc localhost 9000`
2. Type `INIT_CLIENT teammate1\n`.
3. Type `REQ_FILE_OP somefile.txt WRITE\n`.
4. You will get `RESP_SS_INFO 127.0.0.1 9002\n`.
5. **Terminal 2 (SS):** `nc 127.0.0.1 9002`
6. Type `REQ_WRITE_LOCK somefile.txt 1\n`. You should get `RESP_OK_LOCKED\n` back.
7. Type `UPDATE_SENTENCE 3 hello\n`. (You won't get a response, which is correct).
8. Type `UPDATE_SENTENCE 4 world\n`. (Again, no response).
9. Type `ETIRW\n`.

10. You should now receive the final `RESP_OK\n`.

💡 The One "Blind Spot" (and its solution)

There is one part of Phase 3 that `nc` can't test directly:

- `UPDATE_METADATA`**:** When your SS receives `ETIRW`, it's supposed to *asynchronously* send an `UPDATE_METADATA` message back to the NM.

You can't see this from your `nc` client terminal.

The Solution: This is exactly why **Task 4: Logging** is part of Phase 3. You will verify this by watching your **NM's terminal output**. After you type `ETIRW\n` in your `nc` client, you should see a new log line appear in your NM's terminal, like:

`[TIMESTAMP] [SS @ 127.0.0.1:...] [UPDATE_METADATA somefile.txt 15 80] - STATUS_OK`

This is the *only* way to confirm that asynchronous, backend-to-backend message works.