

Gemini Chat

```
nm.c
#include "protocol.h"
//our file which tells this code of our own defined stuff for the network protocol (shared constants, command string 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

#include <stdbool.h>
//for bool

#include <arpa/inet.h>
// For inet_pton

#include <time.h>
#include <fcntl.h> // For fcntl
#include <time.h>

// --- User Registry (records all users ever connected) ---
typedef struct {
    char username[MAX_USERNAME_LEN];
} UserRecord;

UserRecord g_user_registry[500];
int g_user_registry_count = 0;

// Mutex (optional but recommended)
pthread_mutex_t registry_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```

// Globals for logging
FILE* g_log_fp = NULL;
pthread_mutex_t g_log_mutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    char username[MAX_USERNAME_LEN];
    char permission; // Will hold 'R' or 'W'
} AclEntry;

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;
    char owner[MAX_USERNAME_LEN];
    AclEntry acl_list[MAX_CLIENTS]; // Max permissions = max clients
    int acl_count;           // Counter for how many are in the list
    int word_count;
    int char_count;
    time_t created_at;
    time_t modified_at;
    time_t accessed_at;
    bool is_directory;        // NEW: Flag for folder vs file
}FileMapEntry;

#define ASCII_SIZE 256

typedef struct TrieNode {
    struct TrieNode* children[ASCII_SIZE];
    // -1 = Not end of a file
    // >= 0 = End of a file; value is the index in g_file_map
}

```

```

        int file_index;
    } TrieNode;

typedef struct CacheEntry {
    char filename[MAX_PATH_LEN];
    int file_index;
    struct CacheEntry* prev;
    struct CacheEntry* next;
} CacheEntry;

#define MAX_FILES 1000
#define MAX_CACHE_SIZE 50

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

// Access Request System
typedef struct {
    int request_id;
    char requester[MAX_USERNAME_LEN];
    char filename[MAX_PATH_LEN];
    char permission;
    char status;
    time_t requested_at;
    time_t processed_at;
} AccessRequest;

#define MAX_ACCESS_REQUESTS 500
AccessRequest g_access_requests[MAX_ACCESS_REQUESTS];
int g_access_request_count = 0;
int g_next_request_id = 1;
pthread_mutex_t access_request_mutex = PTHREAD_MUTEX_INITIALIZER;

TrieNode* g_file_trie_root;
CacheEntry* g_cache_head = NULL;
CacheEntry* g_cache_tail = NULL;
int g_cache_size = 0;

void* handle_client_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_write(int client_fd, char* username, char* filename);
void do_exec(int client_fd, char* username, char* filename);
void do_undo(int client_fd, char* username, char* filename);
void do_add_access(int client_fd, char* requester_username, char* filename, char* target_user, char permission_flag);
void do_delete(int client_fd, char* requester_username, char* filename);
void do_list_users(int client_fd);
void do_view(int client_fd, char* requester_username, char* flags);
void do_info(int client_fd, char* requester_username, char* filename);
void do_rem_access(int client_fd, char* requester_username, char* filename, char* target_user);
void do_request_access(int client_fd, char* requester_username, char* filename, char permission_flag);

```

```

void do_view_requests(int client_fd, char* requester_username, char* filename);
void do_approve_request(int client_fd, char* requester_username, int req_id);
void do_deny_request(int client_fd, char* requester_username, int req_id);
void do_my_requests(int client_fd, char* requester_username);
void do_create_folder(int client_fd, char* username, char* foldername);
void do_move(int client_fd, char* username, char* filename, char* dest_folder);
void do_view_folder(int client_fd, char* username, char* foldername);
TrieNode* create_trie_node();
void trie_insert(const char* filename, int file_index);
int trie_search(const char* filename);
void trie_remove(const char* filename);
void trie_update_index(const char* filename, int new_index);
void cache_move_to_front(CacheEntry* entry);
void cache_evict_last();
CacheEntry* cache_find(const char* filename);
void cache_add(const char* filename, int file_index);
void cache_remove(const char* filename);
void registry_add_user(const char* name);
int registry_has_user(const char* name);

//Creates a new, empty TrieNode and returns a pointer to the newly allocated TrieNode.
TrieNode* create_trie_node() {
    TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
    if (node) {
        node->file_index = -1; // Not the end of a path
        memset(node->children, 0, sizeof(node->children));
    }
    return node;
}

//Inserts a file's index into the global Trie. Assumes g_file_trie_root is already initialized. The filename is the full path
//file_index is the index in g_file_map to store.
void trie_insert(const char* filename, int file_index) {
    TrieNode* current = g_file_trie_root;

    for (int i = 0; filename[i] != '\0'; i++) {
        unsigned char ch = (unsigned char)filename[i];
        if (current->children[ch] == NULL) {
            current->children[ch] = create_trie_node();
        }
        current = current->children[ch];
    }
    // At the end of the string, mark it as a file
    current->file_index = file_index;
}

//It searches the global Trie for a filename. The filename is the full path to search for. It returns the file's index in g_
//otherwise -1.
int trie_search(const char* filename) {
    TrieNode* current = g_file_trie_root;
}

```

```

for (int i = 0; filename[i] != '\0'; i++) {
    unsigned char ch = (unsigned char)filename[i];
    if (current->children[ch] == NULL) {
        return -1; // Path does not exist
    }
    current = current->children[ch];
}

// At the end, return the index.
// This will be -1 if the path is just a prefix (e.g., "foo")
// but not a complete file (e.g., only "foo/bar.txt" exists).
return current->file_index;
}

// It lazy removes a file from the Trie by un-marking it (with -1)
void trie_remove(const char* filename) {
    TrieNode* current = g_file_trie_root;

    for (int i = 0; filename[i] != '\0'; i++) {
        unsigned char ch = (unsigned char)filename[i];
        if (current->children[ch] == NULL) {
            return; // File doesn't exist anyway
        }
        current = current->children[ch];
    }

    // "Remove" it by setting its index to -1
    current->file_index = -1;
}

// It updates the file index for an existing entry in the Trie (Used when a file is moved in the g_file_map array).
void trie_update_index(const char* filename, int new_index) {
    TrieNode* current = g_file_trie_root;

    for (int i = 0; filename[i] != '\0'; i++) {
        unsigned char ch = (unsigned char)filename[i];
        if (current->children[ch] == NULL) {
            return; // File doesn't exist, can't update
        }
        current = current->children[ch];
    }

    // Update the index
    current->file_index = new_index;
}

// (It helps the NM act as a client)
int connect_to_server(const char* ip, int port) {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("connect_to_server: socket");
        return -1;
}

```

```

    }

    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port);
    if (inet_pton(AF_INET, ip, &serv_addr.sin_addr) <= 0) {
        perror("connect_to_server: inet_pton");
        close(sock);
        return -1;
    }
    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("connect_to_server: connect");
        close(sock);
        return -1;
    }
    return sock;
}

void log_event(const char* message) {
    char time_str[100];
    time_t now = time(NULL);
    struct tm ltime; // A local struct for the thread-safe version

    // Use thread-safe localtime_r
    localtime_r(&now, &ltime);
    strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);

    // Lock, write, unlock
    pthread_mutex_lock(&g_log_mutex);
    if (g_log_fp) {
        fprintf(g_log_fp, "[%s] %s\n", time_str, message);
        fflush(g_log_fp); // Ensure it writes immediately
    }
    pthread_mutex_unlock(&g_log_mutex);
}

/**
 * =====
 * SECTION: LRU Cache Helper Functions
 *
 * These functions also assume the file_map_mutex is HELD by the caller.
 * =====
 */

// Moves an existing cache entry to be the head (most recent).
void cache_move_to_front(CacheEntry* entry) {
    if (entry == g_cache_head) {
        return; // Already at the front
    }

    // Unlink from its current position
    if (entry->prev) {

```

```

        entry->prev->next = entry->next;
    }
    if (entry->next) {
        entry->next->prev = entry->prev;
    }

    // Check if it was the tail
    if (entry == g_cache_tail) {
        g_cache_tail = entry->prev;
    }

    // Link at the front
    entry->next = g_cache_head;
    entry->prev = NULL;
    if (g_cache_head) {
        g_cache_head->prev = entry;
    }
    g_cache_head = entry;

    // If the list was empty, it's also the tail
    if (g_cache_tail == NULL) {
        g_cache_tail = entry;
    }
}

// Evicts the last (least recently used) entry from the cache.
void cache_evict_last() {
    if (g_cache_tail == NULL) {
        return; // Cache is empty
    }

    CacheEntry* old_tail = g_cache_tail;

    // Update the tail pointer
    g_cache_tail = old_tail->prev;

    if (g_cache_tail) {
        g_cache_tail->next = NULL; // New tail has no next
    } else {
        g_cache_head = NULL; // Cache is now empty
    }

    free(old_tail);
    g_cache_size--;
}

// Finds a file in the cache by its name.
// Returns a pointer to the CacheEntry if found, otherwise NULL.
CacheEntry* cache_find(const char* filename) {
    CacheEntry* current = g_cache_head;
    while (current) {
        if (strcmp(current->filename, filename) == 0) {

```

```

        return current; // Found
    }
    current = current->next;
}
return NULL; // Not found
}

// Adds a new file index to the front of the cache.
// Evicts the last entry if the cache is full.
void cache_add(const char* filename, int file_index) {
    // 1. Create the new entry
    CacheEntry* new_entry = (CacheEntry*)malloc(sizeof(CacheEntry));
    if (!new_entry) {
        perror("malloc cache_add");
        return; // Failed to add
    }
    strcpy(new_entry->filename, filename);
    new_entry->file_index = file_index;
    new_entry->prev = NULL;

    // 2. Link at the front
    new_entry->next = g_cache_head;
    if (g_cache_head) {
        g_cache_head->prev = new_entry;
    }
    g_cache_head = new_entry;

    // 3. If list was empty, it's also the tail
    if (g_cache_tail == NULL) {
        g_cache_tail = new_entry;
    }

    // 4. Update size and evict if full
    g_cache_size++;
    if (g_cache_size > MAX_CACHE_SIZE) {
        cache_evict_last();
    }
}

// Removes an entry from the cache by its name.
// (Used for cache invalidation, e.g., on delete).
void cache_remove(const char* filename) {
    CacheEntry* entry = cache_find(filename);
    if (entry == NULL) {
        return; // Not in cache
    }

    // Unlink it
    if (entry->prev) {
        entry->prev->next = entry->next;
    }
    if (entry->next) {

```

```

        entry->next->prev = entry->prev;
    }

    // Update head/tail pointers if necessary
    if (entry == g_cache_head) {
        g_cache_head = entry->next;
    }
    if (entry == g_cache_tail) {
        g_cache_tail = entry->prev;
    }

    free(entry);
    g_cache_size--;
}

// --- PERSISTENCE FUNCTIONS ---

// Saves the entire file map and count to disk
void save_metadata_to_disk() {
    printf("[NM-Persist] Saving metadata to disk...\n");

    // We lock the mutex to ensure no other thread
    // is modifying the list while we save.
    // pthread_mutex_lock(&file_map_mutex);

    FILE* fp = fopen("nm_metadata.dat", "w");
    if (fp == NULL) {
        perror("fopen (save_metadata_to_disk)");
        // pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 1. Write the count
    if (fwrite(&g_file_count, sizeof(int), 1, fp) != 1) {
        printf("[NM-Persist] Error writing file count.\n");
    }

    // 2. Write the entire array
    if (fwrite(g_file_map, sizeof(FileMapEntry), g_file_count, fp) != g_file_count) {
        printf("[NM-Persist] Error writing file map data.\n");
    }

    fclose(fp);
    // pthread_mutex_unlock(&file_map_mutex);
    printf("[NM-Persist] Save complete.\n");
}

// Loads the entire file map and count from disk
void load_metadata_from_disk() {
    printf("[NM-Persist] Loading metadata from disk...\n");

    // We lock here just to be safe, though this should
}

```

```

// only be called once at startup.
pthread_mutex_lock(&file_map_mutex);

FILE* fp = fopen("nm_metadata.dat", "r");
if (fp == NULL) {
    // This is not an error, it just means no file exists yet.
    printf("[NM-Persist] No metadata file found. Starting fresh.\n");
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 1. Read the count
if (fread(&g_file_count, sizeof(int), 1, fp) != 1) {
    printf("[NM-Persist] Error reading file count.\n");
    g_file_count = 0; // Reset on error
}

// 2. Read the entire array
if (fread(g_file_map, sizeof(FileMapEntry), g_file_count, fp) != g_file_count) {
    printf("[NM-Persist] Error reading file map data.\n");
    g_file_count = 0; // Reset on error
}

fclose(fp);
pthread_mutex_unlock(&file_map_mutex);
printf("[NM-Persist] Load complete. %d files loaded.\n", g_file_count);
}

void build_trie_from_map() {
    printf("[NM] Building Trie from loaded metadata...\n");
    pthread_mutex_lock(&file_map_mutex);
    for (int i = 0; i < g_file_count; i++) {
        trie_insert(g_file_map[i].path, i);
    }
    pthread_mutex_unlock(&file_map_mutex);
    printf("[NM] Trie build complete.\n");
}

void do_exec(int client_fd, char* username, char* filename) {

    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: EXEC, FILE: %s", username, filename);
    log_event(log_msg);
    printf("Client %s requesting EXEC: %s\n", username, filename);

    char resp_buf[MAX_MSG_LEN];

    // 1. Lock the central mutex
    pthread_mutex_lock(&file_map_mutex);

    int file_index = -1;
}

```

```

// 2. Check cache (fastest)
CacheEntry* entry = cache_find(filename);

if (entry) {
    file_index = entry->file_index;
    cache_move_to_front(entry);
} else {
    file_index = trie_search(filename);
    if (file_index != -1) {
        cache_add(filename, file_index);
    }
}

// 5. Handle "Not Found"
if (file_index == -1) {
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

FileMapEntry* file = &g_file_map[file_index];

// --- SAFETY CHECK ---
if (file->is_directory) {
    sprintf(resp_buf, MAX_MSG_LEN, "%s Cannot perform file operation on a folder\n", RESP_BAD_REQ);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}
// -------

file->accessed_at = time(NULL);

// 6. ACL Check
bool has_access = false;
if (strcmp(username, file->owner) == 0) has_access = true;
if (!has_access) {
    for (int i = 0; i < file->acl_count; i++) {
        if (strcmp(username, file->acl_list[i].username) == 0) {
            has_access = true;
            break;
        }
    }
}
if (!has_access) {
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

```

```

int ss_index = file->ss_index;
pthread_mutex_unlock(&file_map_mutex);

// 7. Get SS Info
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);

// 8. NM connects TO SS
int ss_sock = connect_to_server(ss_ip, ss_port);
if (ss_sock < 0) {
    printf("EXEC: NM failed to connect to SS\n");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SS_DOWN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}
printf("EXEC: NM connected to SS successfully.\n");

// 9. Request file from SS
char req_buf[MAX_MSG_LEN];
char file_buf[4096];
snprintf(req_buf, MAX_MSG_LEN, "%s %s\n", SS_GET_FILE, filename);
send(ss_sock, req_buf, strlen(req_buf), 0);

// 10. Read file content from SS
memset(file_buf, 0, sizeof(file_buf));
int bytes_read = read(ss_sock, file_buf, sizeof(file_buf) - 1);
if (bytes_read <= 0) {
    printf("EXEC: NM failed to read file from SS\n");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    close(ss_sock);
    return;
}
file_buf[bytes_read] = '\0';
close(ss_sock);
printf("EXEC: NM received file content from SS.\n");

// 11. Create temp file
char temp_filename[] = "/tmp/nm_exec_XXXXXX";
int temp_fd = mkstemp(temp_filename);
if (temp_fd < 0) {
    perror("mkstemp");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}
write(temp_fd, file_buf, bytes_read);
close(temp_fd);

```

```

// 12. Execute with popen()
char cmd_buf[MAX_PATH_LEN + 10];
snprintf(cmd_buf, sizeof(cmd_buf), "sh %s 2>&1", temp_filename);

FILE* pipe = popen(cmd_buf, "r");
if (!pipe) {
    perror("popen");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    remove(temp_filename);
    return;
}

// 13. Send 200 OK
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
send(client_fd, resp_buf, strlen(resp_buf), 0);

// 14. Pipe output
char pipe_buf[1024];
while (fgets(pipe_buf, sizeof(pipe_buf), pipe) != NULL) {
    send(client_fd, pipe_buf, strlen(pipe_buf), 0);
}

pclose(pipe);
remove(temp_filename);
printf("EXEC: Command executed and output sent to client.\n");

// Send the special "End of Exec" message
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_EXEC_DONE);
send(client_fd, resp_buf, strlen(resp_buf), 0);
return; // Return (keeps connection alive)
}

void do_create(int client_fd, char* username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: CREATE, FILE: %s", username, filename);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];
    int ss_index = -1;
    int ss_fd = -1;

    // --- Phase 1: Check for conflicts and get SS info ---
    // We lock, get the info we need, then unlock *before* network I/O.

    pthread_mutex_lock(&file_map_mutex);

    // 1. Check for conflicts using the Trie
    int file_index = trie_search(filename);
    if (file_index != -1) {
        // File already exists.
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_CONFLICT);
    }
}

```

```

        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 2. Check if any SS is available
    if (g_ss_count == 0) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SS_DOWN);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 3. Get SS info using round-robin. We must lock ss_list.
    // We lock in a consistent order (file_map -> ss_list) to prevent deadlock.
    ss_index = g_file_count % g_ss_count;

    pthread_mutex_lock(&ss_list_mutex);
    ss_fd = g_ss_list[ss_index].conn_fd;
    pthread_mutex_unlock(&ss_list_mutex);

    // 4. *** CRITICAL *** Unlock before network I/O
    pthread_mutex_unlock(&file_map_mutex);

    // --- Phase 2: Communicate with SS (No locks held) ---

    // 5. 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");
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        return;
    }

    // 6. 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");
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        return;
    }

    // --- Phase 3: Commit changes (if SS was successful) ---

    if (strcmp(ss_resp, RESP_OK, strlen(RESP_OK)) == 0) {
        // 7. *** RE-ACQUIRE LOCK *** to commit to the shared map
        pthread_mutex_lock(&file_map_mutex);

```

```

// 8. *** RE-CHECK CONFLICT ***
// (Another thread might have created the same file
// while we were unlocked and talking to the SS)
file_index = trie_search(filename);
if (file_index != -1) {
    // Another thread won the race. Tell our client "Conflict".
    // In a v2, we would tell the SS to *delete* the
    // file it just created, but for now this is safe.
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_CONFLICT);
    pthread_mutex_unlock(&file_map_mutex);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// 9. No conflict. We are clear to commit.
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; // Use the index from Phase 1
g_file_map[g_file_count].is_directory = false; // It's a file, not a folder
g_file_map[g_file_count].acl_count = 0;
g_file_map[g_file_count].word_count = 0;
g_file_map[g_file_count].char_count = 0;

time_t now = time(NULL);
g_file_map[g_file_count].created_at = now;
g_file_map[g_file_count].modified_at = now;
g_file_map[g_file_count].accessed_at = now;

trie_insert(filename, g_file_count); // Add to Trie
cache_add(filename, g_file_count); // Add to Cache

g_file_count++;
save_metadata_to_disk();

pthread_mutex_unlock(&file_map_mutex);

// 10. Send final success to client
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
send(client_fd, resp_buf, strlen(resp_buf), 0);

} else {
    // SS responded with an error
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
}
}

void do_undo(int client_fd, char* username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: UNDO, FILE: %s", username, filename);
    log_event(log_msg);
}

```

```

printf("Client %s requesting UNDO: %s\n", username, filename);

char resp_buf[MAX_MSG_LEN];

// 1. Lock the central mutex
pthread_mutex_lock(&file_map_mutex);

int file_index = -1;
// char resp_buf[MAX_MSG_LEN]; // Make sure resp_buf is declared

// 2. Check cache (fastest)
CacheEntry* entry = cache_find(filename);

if (entry) {
    // --- CACHE HIT ---
    file_index = entry->file_index;
    cache_move_to_front(entry); // Mark as recently used
} else {
    // --- CACHE MISS ---
    // 3. Check trie (fast)
    file_index = trie_search(filename);
    if (file_index != -1) {
        // 4. Add to cache if we found it
        cache_add(filename, file_index);
    }
}

// 5. Handle "Not Found"
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return; // Exit the function
}

// --- We now have a valid file_index, and the lock is still held ---
FileMapEntry* file = &g_file_map[file_index];

bool has_write_access = false;
if (strcmp(username, file->owner) == 0) {
    has_write_access = true;
}
if (!has_write_access) {
    for (int i = 0; i < file->acl_count; i++) {
        if (strcmp(username, file->acl_list[i].username) == 0) {
            if (file->acl_list[i].permission == PERM_WRITE) {
                has_write_access = true;
            }
            break;
        }
    }
}

```

```

if (!has_write_access) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 3. User has access. Get the SS info.
int ss_index = g_file_map[file_index].ss_index;
pthread_mutex_unlock(&file_map_mutex);

// 4. Get the SS's COMMAND-LINE socket
pthread_mutex_lock(&ss_list_mutex);
int ss_fd = g_ss_list[ss_index].conn_fd; // <-- This is the important part
pthread_mutex_unlock(&ss_list_mutex);

// 5. Send command to the SS
char command_buf[MAX_MSG_LEN];
sprintf(command_buf, "%s %s\n", NM_UNDO, filename);

if (send(ss_fd, command_buf, strlen(command_buf), 0) < 0) {
    perror("Failed to send NM_UNDO to SS");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// 6. 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 UNDO\n");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// 7. Check SS response and commit metadata change
if (strcmp(ss_resp, RESP_OK, strlen(RESP_OK)) == 0) {
    // The UNDO was successful, now we must update our metadata
    pthread_mutex_lock(&file_map_mutex);

    // We must re-find the file index, in case it changed
    // while we were talking to the SS.
    int current_file_index = trie_search(filename);

    if (current_file_index != -1) {
        FileMapEntry* file = &g_file_map[current_file_index];
        file->modified_at = time(NULL);
        save_metadata_to_disk(); // Save the new timestamp
    }
}

```

```

// else: file was deleted while we were unlocked. Nothing to do.

    pthread_mutex_unlock(&file_map_mutex);
}

// 8. Relay SS response (e.g., "200" or "404") to the client
send(client_fd, ss_resp, strlen(ss_resp), 0);

}

void do_read(int client_fd, char* username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: READ, FILE: %s", username, filename);
    log_event(log_msg);
    printf("Client %s requesting READ: %s\n", username, filename); // <-- Add this line back
    char resp_buf[MAX_MSG_LEN];

    // 1. Lock the central mutex
    pthread_mutex_lock(&file_map_mutex);

    int file_index = -1;
    // char resp_buf[MAX_MSG_LEN]; // Make sure resp_buf is declared

    // 2. Check cache (fastest)
    CacheEntry* entry = cache_find(filename);

    if (entry) {
        // --- CACHE HIT ---
        file_index = entry->file_index;
        cache_move_to_front(entry); // Mark as recently used
    } else {
        // --- CACHE MISS ---
        // 3. Check trie (fast)
        file_index = trie_search(filename);
        if (file_index != -1) {
            // 4. Add to cache if we found it
            cache_add(filename, file_index);
        }
    }

    // 5. Handle "Not Found"
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return; // Exit the function
    }

    // --- We now have a valid file_index, and the lock is still held ---
    FileMapEntry* file = &g_file_map[file_index];

    // --- SAFETY CHECK ---

```

```

if (file->is_directory) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Cannot perform file operation on a folder\n", RESP_BAD_REQ);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}
// ----

file->accessed_at = time(NULL);

bool has_access = false;

// Check if the requester is the owner
if (strcmp(username, file->owner) == 0) {
    has_access = true;
    printf("ACL Check: User %s is OWNER of %s. Access granted.\n", username, filename);
}

// If not owner, check the ACL list
if (!has_access) {
    for (int i = 0; i < file->acl_count; i++) {
        // Check username AND if they have 'R' or 'W' permission
        if (strcmp(username, file->acl_list[i].username) == 0) {
            has_access = true;
            printf("ACL Check: User %s found in ACL for %s. Access granted.\n", username, filename);
            break;
        }
    }
}

// 3. ENFORCE
if (!has_access) {
    printf("ACL Check: User %s has NO ACCESS to %s. Denying.\n", username, filename);
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 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);
    }

// Add this new function to nm.c

void do_write(int client_fd, char* username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: WRITE, FILE: %s", username, filename);
    log_event(log_msg);
    printf("Client %s requesting WRITE: %s\n", username, filename);
    char resp_buf[MAX_MSG_LEN];

    // 1. Lock the central mutex
    pthread_mutex_lock(&file_map_mutex);

    int file_index = -1;
    // char resp_buf[MAX_MSG_LEN]; // Make sure resp_buf is declared

    // 2. Check cache (fastest)
    CacheEntry* entry = cache_find(filename);

    if (entry) {
        // --- CACHE HIT ---
        file_index = entry->file_index;
        cache_move_to_front(entry); // Mark as recently used
    } else {
        // --- CACHE MISS ---
        // 3. Check trie (fast)
        file_index = trie_search(filename);
        if (file_index != -1) {
            // 4. Add to cache if we found it
            cache_add(filename, file_index);
        }
    }

    // 5. Handle "Not Found"
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return; // Exit the function
    }

    // --- We now have a valid file_index, and the lock is still held ---
    FileMapEntry* file = &g_file_map[file_index];

    // --- SAFETY CHECK ---
    if (file->is_directory) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Cannot perform file operation on a folder\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
    }
}

```

```

        return;
    }
    // ----

    file->accessed_at = time(NULL);

    bool has_write_access = false;

    // --- Stricter ACL Check for WRITE ---
    // 1. Check if owner
    if (strcmp(username, file->owner) == 0) {
        has_write_access = true;
        printf("ACL Check: User %s is OWNER of %s. Write access granted.\n", username, filename);
    }

    // 2. If not owner, check ACL for 'W' permission
    if (!has_write_access) {
        for (int i = 0; i < file->acl_count; i++) {
            if (strcmp(username, file->acl_list[i].username) == 0) {
                // MUST have 'W' (PERM_WRITE)
                if (file->acl_list[i].permission == PERM_WRITE) {
                    has_write_access = true;
                    printf("ACL Check: User %s found in ACL with 'W' perm for %s. Write access granted.\n", username, file);
                }
                break; // Found user, no need to search more
            }
        }
    }
    // --- End of ACL Check ---

    // 3. ENFORCE
    if (!has_write_access) {
        printf("ACL Check: User %s has NO WRITE ACCESS to %s. Denying.\n", username, filename);
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 4. Get the SS's info (User has access, proceed same as do_read)
    int ss_index = g_file_map[file_index].ss_index;
    pthread_mutex_unlock(&file_map_mutex); // Done with file map

    // 5. 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);

    // 6. Send the referral to the client
    sprintf(resp_buf, "%s %s %d\n", RESP_SS_INFO, ss_ip, ss_port);

```

```

        send(client_fd, resp_buf, strlen(resp_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 THIS BLOCK -----
        pthread_mutex_lock(&registry_mutex);
        registry_add_user(username);
        pthread_mutex_unlock(&registry_mutex);
        // ----

        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);

        char log_msg[MAX_MSG_LEN];
        snprintf(log_msg, MAX_MSG_LEN, "New connection: CLIENT, USER: %s", username);
        log_event(log_msg);
        char resp_buf[MAX_MSG_LEN];
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
        send(conn_fd, resp_buf, strlen(resp_buf), 0); // Send "OK\n"

        // 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> <n> <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);
char resp_buf[MAX_MSG_LEN];
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
send(conn_fd, resp_buf, strlen(resp_buf), 0); // Send "OK\n"
free(arg);

}

else if (strncmp(buffer, S_META_UPDATE, strlen(S_META_UPDATE)) == 0) {
    // --- It's an SS sending a metadata update ---
    printf("[NM] Received S_META_UPDATE: %s\n", buffer);
    char cmd_tmp[100], filename[MAX_PATH_LEN], wc_str[20], cc_str[20];

    if (sscanf(buffer, "%s %s %s %s", cmd_tmp, filename, wc_str, cc_str) < 4) {
        printf("[NM] Malformed S_META_UPDATE. Ignoring.\n");
    } else {
        int word_count = atoi(wc_str);
        int char_count = atoi(cc_str);

        pthread_mutex_lock(&file_map_mutex);
        for (int i = 0; i < g_file_count; i++) {
            if (strcmp(g_file_map[i].path, filename) == 0) {
                g_file_map[i].word_count = word_count;
                g_file_map[i].char_count = char_count;
                g_file_map[i].modified_at = time(NULL);
                printf("[NM] Metadata updated for %s.\n", filename);
                break;
            }
        }
        save_metadata_to_disk();
        pthread_mutex_unlock(&file_map_mutex);
    }

    // This was a temporary connection, so we close it.
    close(conn_fd);
    free(arg);
    return NULL; // This thread ends
}

```

```

// --- END OF NEW BLOCK ---


}

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 do_add_access(int client_fd, char* requester_username, char* filename, char* target_user, char permission_f
char log_msg[MAX_MSG_LEN];
snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: ADDACCESS, FILE: %s, TARGET: %s, PERM: %c", requester_
target_user, permission_flag);
log_event(log_msg);

char resp_buf[MAX_MSG_LEN];

// 1. Lock the central mutex
pthread_mutex_lock(&file_map_mutex);

int file_index = -1;
// char resp_buf[MAX_MSG_LEN]; // Make sure resp_buf is declared

// 2. Check cache (fastest)
CacheEntry* entry = cache_find(filename);

if (entry) {
    // --- CACHE HIT ---
    file_index = entry->file_index;
    cache_move_to_front(entry); // Mark as recently used
} else {
    // --- CACHE MISS ---
    // 3. Check trie (fast)
    file_index = trie_search(filename);
    if (file_index != -1) {
        // 4. Add to cache if we found it
        cache_add(filename, file_index);
    }
}

// 5. Handle "Not Found"
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return; // Exit the function
}

```

```

// --- We now have a valid file_index, and the lock is still held ---
FileMapEntry* file = &g_file_map[file_index];

if (strcmp(requester_username, file->owner) != 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 3. Add or update the permission
bool user_found = false;
for (int i = 0; i < file->acl_count; i++) {
    if (strcmp(file->acl_list[i].username, target_user) == 0) {
        // User already in list, just update permission
        file->acl_list[i].permission = permission_flag;
        user_found = true;
        break;
    }
}

if (!user_found) {
    // User not in list, add new entry (if space is available)
    if (file->acl_count < MAX_CLIENTS) {
        strcpy(file->acl_list[file->acl_count].username, target_user);
        file->acl_list[file->acl_count].permission = permission_flag;
        file->acl_count++;
    } else {
        // Handle case where ACL list is full (optional)
    }
}

// 4. Send success
printf("Client %s granted %c access for %s to %s\n", requester_username, permission_flag, filename, target_user);
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
send(client_fd, resp_buf, strlen(resp_buf), 0);
save_metadata_to_disk();
pthread_mutex_unlock(&file_map_mutex);
}

void do_rem_access(int client_fd, char* requester_username, char* filename, char* target_user) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: REMACCESS, FILE: %s, TARGET: %s", requester_username, filename, target_user);
    log_event(log_msg);
    printf("Client %s requesting REMACCESS for %s from %s\n", requester_username, target_user, filename);

    char resp_buf[MAX_MSG_LEN];

    // 1. Lock the central mutex
    pthread_mutex_lock(&file_map_mutex);
}

```

```

int file_index = -1;
// char resp_buf[MAX_MSG_LEN]; // Make sure resp_buf is declared

// 2. Check cache (fastest)
CacheEntry* entry = cache_find(filename);

if (entry) {
    // --- CACHE HIT ---
    file_index = entry->file_index;
    cache_move_to_front(entry); // Mark as recently used
} else {
    // --- CACHE MISS ---
    // 3. Check trie (fast)
    file_index = trie_search(filename);
    if (file_index != -1) {
        // 4. Add to cache if we found it
        cache_add(filename, file_index);
    }
}

// 5. Handle "Not Found"
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return; // Exit the function
}

// --- We now have a valid file_index, and the lock is still held ---
FileMapEntry* file = &g_file_map[file_index];

if (strcmp(requester_username, file->owner) != 0) {
    printf("ACL Check: User %s is NOT OWNER of %s. RemAccess denied.\n", requester_username, filename);
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 3. Find the user in the ACL list
int user_index = -1;
for (int i = 0; i < file->acl_count; i++) {
    if (strcmp(file->acl_list[i].username, target_user) == 0) {
        user_index = i;
        break;
    }
}

// 4. If found, remove them using "swap-with-last"
if (user_index != -1) {
    printf("Removing %s from ACL for %s\n", target_user, filename);
    // Copy the last element over the one we're deleting
}

```

```

        file->acl_list[user_index] = file->acl_list[file->acl_count - 1];
        file->acl_count--; // Decrease the count
    } else {
        // User wasn't in the list anyway, but that's not an error.
        printf("User %s was not in ACL for %s. No action taken.\n", target_user, filename);
    }

    // 5. Send success
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    save_metadata_to_disk();
    pthread_mutex_unlock(&file_map_mutex);
}

/*
 * Handle: C_REQ_ACC <filename> <-R|-W>
 * User requests READ or WRITE access to a file they don't own
 */
void do_request_access(int client_fd, char* requester_username, char* filename, char permission_flag) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: REQUEST_ACCESS, FILE: %s, PERM: %c",
             requester_username, filename, permission_flag);
    log_event(log_msg);
    printf("Client %s requesting access to %s (perm: %c)\n", requester_username, filename, permission_flag);

    char resp_buf[MAX_MSG_LEN];

    // 1. Check if file exists
    pthread_mutex_lock(&file_map_mutex);

    int file_index = trie_search(filename);
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s File not found\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    FileMapEntry* file = &g_file_map[file_index];

    // 2. Check if requester is already the owner
    if (strcmp(requester_username, file->owner) == 0) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s You are already the owner\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 3. Check if requester already has access
    for (int i = 0; i < file->acl_count; i++) {
        if (strcmp(requester_username, file->acl_list[i].username) == 0) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s You already have access to this file\n", RESP_CONFLICT);

```

```

        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }
}

pthread_mutex_unlock(&file_map_mutex);

// 4. Check if there's already a pending request
pthread_mutex_lock(&access_request_mutex);

for (int i = 0; i < g_access_request_count; i++) {
    if (strcmp(g_access_requests[i].requester, requester_username) == 0 &&
        strcmp(g_access_requests[i].filename, filename) == 0 &&
        g_access_requests[i].status == REQ_STATUS_PENDING) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s You already have a pending request for this file\n", RESP_CONFLICT);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&access_request_mutex);
        return;
    }
}

// 5. Create new access request
if (g_access_request_count >= MAX_ACCESS_REQUESTS) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Request queue is full\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&access_request_mutex);
    return;
}

AccessRequest* new_req = &g_access_requests[g_access_request_count];
new_req->request_id = g_next_request_id++;
strcpy(new_req->requester, requester_username);
strcpy(new_req->filename, filename);
new_req->permission = permission_flag;
new_req->status = REQ_STATUS_PENDING;
new_req->requested_at = time(NULL);
new_req->processed_at = 0;

g_access_request_count++;

pthread_mutex_unlock(&access_request_mutex);

// 6. Send success
snprintf(resp_buf, MAX_MSG_LEN, "%s Request submitted (ID: %d)\n", RESP_OK, new_req->request_id);
send(client_fd, resp_buf, strlen(resp_buf), 0);
printf("Access request created: ID=%d, User=%s, File=%s\n",
       new_req->request_id, requester_username, filename);
}

/*
 * Handle: C_VIEW_REQ <filename>

```

```

* File owner views pending requests for their file
*/
void do_view_requests(int client_fd, char* requester_username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: VIEW_REQUESTS, FILE: %s", requester_username, filename)
    log_event(log_msg);

    char resp_buf[4096];
    char payload[4096] = "";

    // 1. Check if file exists and if requester is owner
    pthread_mutex_lock(&file_map_mutex);

    int file_index = trie_search(filename);
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s File not found\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    FileMapEntry* file = &g_file_map[file_index];

    if (strcmp(requester_username, file->owner) != 0) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Only the file owner can view requests\n", RESP_FORBIDDEN);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    pthread_mutex_unlock(&file_map_mutex);

    // 2. Build list of pending requests for this file
    snprintf(payload, sizeof(payload), "%s\n==== Pending Access Requests for %s ===\n", RESP_OK, filename);

    pthread_mutex_lock(&access_request_mutex);

    int found_count = 0;
    for (int i = 0; i < g_access_request_count; i++) {
        if (strcmp(g_access_requests[i].filename, filename) == 0 &&
            g_access_requests[i].status == REQ_STATUS_PENDING) {

            char time_str[100];
            struct tm ltime;
            localtime_r(&g_access_requests[i].requested_at, &ltime);
            strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);

            snprintf(payload + strlen(payload), sizeof(payload) - strlen(payload),
                    "ID: %d | User: %s | Permission: %c | Requested: %s\n",
                    g_access_requests[i].request_id,
                    g_access_requests[i].requester,
                    g_access_requests[i].permission,

```

```

        time_str);
    found_count++;
}
}

pthread_mutex_unlock(&access_request_mutex);

if (found_count == 0) {
    snprintf(payload + strlen(payload), sizeof(payload) - strlen(payload),
              "No pending requests.\n");
}

send(client_fd, payload, strlen(payload), 0);
}

/*
 * Handle: C_APPROVE <request_id>
 * File owner approves a pending access request
 */
void do_approve_request(int client_fd, char* requester_username, int request_id) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: APPROVE, REQ_ID: %d", requester_username, request_id);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];

    pthread_mutex_lock(&access_request_mutex);

    // 1. Find the request
    AccessRequest* req = NULL;
    for (int i = 0; i < g_access_request_count; i++) {
        if (g_access_requests[i].request_id == request_id) {
            req = &g_access_requests[i];
            break;
        }
    }

    if (req == NULL) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Request not found\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&access_request_mutex);
        return;
    }

    // 2. Check if already processed
    if (req->status != REQ_STATUS_PENDING) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Request already processed\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&access_request_mutex);
        return;
    }
}

```

```

// Store request details before unlocking
char req_filename[MAX_PATH_LEN];
char req_requester[MAX_USERNAME_LEN];
char req_permission = req->permission;
strcpy(req_filename, req->filename);
strcpy(req_requester, req->requester);

pthread_mutex_unlock(&access_request_mutex);

// 3. Verify requester is the file owner
pthread_mutex_lock(&file_map_mutex);

int file_index = trie_search(req_filename);
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s File no longer exists\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

FileMapEntry* file = &g_file_map[file_index];

if (strcmp(requester_username, file->owner) != 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Only the file owner can approve requests\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 4. Add user to ACL
bool user_found = false;
for (int i = 0; i < file->acl_count; i++) {
    if (strcmp(file->acl_list[i].username, req_requester) == 0) {
        file->acl_list[i].permission = req_permission;
        user_found = true;
        break;
    }
}

if (!user_found) {
    if (file->acl_count < MAX_CLIENTS) {
        strcpy(file->acl_list[file->acl_count].username, req_requester);
        file->acl_list[file->acl_count].permission = req_permission;
        file->acl_count++;
    } else {
        snprintf(resp_buf, MAX_MSG_LEN, "%s ACL is full\n", RESP_SRV_ERR);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }
}

```

```

    save_metadata_to_disk();
    pthread_mutex_unlock(&file_map_mutex);

    // 5. Mark request as approved
    pthread_mutex_lock(&access_request_mutex);
    req->status = REQ_STATUS_APPROVED;
    req->processed_at = time(NULL);
    pthread_mutex_unlock(&access_request_mutex);

    // 6. Send success
    snprintf(resp_buf, MAX_MSG_LEN, "%s Access granted to %s\n", RESP_OK, req_requester);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    printf("Request %d approved by %s\n", request_id, requester_username);
}

/*
 * Handle: C_DENY <request_id>
 * File owner denies a pending access request
 */
void do_deny_request(int client_fd, char* requester_username, int request_id) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: DENY, REQ_ID: %d", requester_username, request_id);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];

    pthread_mutex_lock(&access_request_mutex);

    // 1. Find the request
    AccessRequest* req = NULL;
    for (int i = 0; i < g_access_request_count; i++) {
        if (g_access_requests[i].request_id == request_id) {
            req = &g_access_requests[i];
            break;
        }
    }

    if (req == NULL) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Request not found\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&access_request_mutex);
        return;
    }

    // 2. Check if already processed
    if (req->status != REQ_STATUS_PENDING) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Request already processed\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&access_request_mutex);
        return;
    }
}

```

```

// Store filename for ownership check
char req_filename[MAX_PATH_LEN];
strcpy(req_filename, req->filename);

pthread_mutex_unlock(&access_request_mutex);

// 3. Verify requester is the file owner
pthread_mutex_lock(&file_map_mutex);

int file_index = trie_search(req_filename);
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s File no longer exists\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

FileMapEntry* file = &g_file_map[file_index];

if (strcmp(requester_username, file->owner) != 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Only the file owner can deny requests\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

pthread_mutex_unlock(&file_map_mutex);

// 4. Mark request as denied
pthread_mutex_lock(&access_request_mutex);
req->status = REQ_STATUS_DENIED;
req->processed_at = time(NULL);
pthread_mutex_unlock(&access_request_mutex);

// 5. Send success
snprintf(resp_buf, MAX_MSG_LEN, "%s Request denied\n", RESP_OK);
send(client_fd, resp_buf, strlen(resp_buf), 0);
printf("Request %d denied by %s\n", request_id, requester_username);
}

/*
 * Handle: C_MY_REQ
 * User views their own access requests (pending, approved, denied)
 */
void do_my_requests(int client_fd, char* requester_username) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: MY_REQUESTS", requester_username);
    log_event(log_msg);

    char payload[4096] = "";
    snprintf(payload, sizeof(payload), "%s\n==== Your Access Requests ====\n", RESP_OK);
}

```

```

pthread_mutex_lock(&access_request_mutex);

int found_count = 0;
for (int i = 0; i < g_access_request_count; i++) {
    if (strcmp(g_access_requests[i].requester, requester_username) == 0) {
        char time_str[100];
        struct tm ltime;
        localtime_r(&g_access_requests[i].requested_at, &ltime);
        strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);

        char status_str[20];
        if (g_access_requests[i].status == REQ_STATUS_PENDING) {
            strcpy(status_str, "PENDING");
        } else if (g_access_requests[i].status == REQ_STATUS_APPROVED) {
            strcpy(status_str, "APPROVED");
        } else {
            strcpy(status_str, "DENIED");
        }

        snprintf(payload + strlen(payload), sizeof(payload) - strlen(payload),
                  "ID: %d | File: %s | Perm: %c | Status: %s | Date: %s\n",
                  g_access_requests[i].request_id,
                  g_access_requests[i].filename,
                  g_access_requests[i].permission,
                  status_str,
                  time_str);
        found_count++;
    }
}

pthread_mutex_unlock(&access_request_mutex);

if (found_count == 0) {
    snprintf(payload + strlen(payload), sizeof(payload) - strlen(payload),
              "No requests found.\n");
}

send(client_fd, payload, strlen(payload), 0);
}

void do_create_folder(int client_fd, char* username, char* foldername) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: CREATEFOLDER, NAME: %s", username, foldername);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];

    pthread_mutex_lock(&file_map_mutex);

    // 1. Check existence
    if (trie_search(foldername) != -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Folder/File already exists\n", RESP_CONFLICT);
    }
}

```

```

        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 2. Create Metadata Entry
    strcpy(g_file_map[g_file_count].path, foldername);
    strcpy(g_file_map[g_file_count].owner, username);
    g_file_map[g_file_count].ss_index = -1; // No SS for folders
    g_file_map[g_file_count].is_directory = true; // <--- MARK AS FOLDER
    g_file_map[g_file_count].acl_count = 0;

    time_t now = time(NULL);
    g_file_map[g_file_count].created_at = now;
    g_file_map[g_file_count].modified_at = now;

    trie_insert(foldername, g_file_count);
    g_file_count++;
    save_metadata_to_disk();

    pthread_mutex_unlock(&file_map_mutex);

    snprintf(resp_buf, MAX_MSG_LEN, "%s Folder created\n", RESP_OK);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
}

void do_move(int client_fd, char* username, char* filename, char* dest_folder) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: MOVE, FILE: %s, DEST: %s", username, filename, dest_folder);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];

    // Lock GLOBAL mutex to prevent race conditions during the complex move
    pthread_mutex_lock(&file_map_mutex);

    // 1. Find Source
    int file_index = trie_search(filename);
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Source file not found\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // 2. Check permissions
    if (strcmp(g_file_map[file_index].owner, username) != 0) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s Permission denied\n", RESP_FORBIDDEN);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }
}

```

```

// 3. Reject Folder Move (For Correctness/Simplicity)
if (g_file_map[file_index].is_directory) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Moving folders is not supported\n", RESP_BAD_REQ);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 4. Construct New Path
char new_path[MAX_PATH_LEN];

// Handle "." as destination (move to root)
if (strcmp(dest_folder, ".") == 0) {
    // If dest is ".", strip directory prefix from filename.
    // E.g. "a/b.txt" -> "b.txt"
    char *base_name = strrchr(filename, '/');
    if (base_name) {
        strcpy(new_path, base_name + 1); // +1 skips '/'
    } else {
        // Already at root
        strcpy(new_path, filename);
    }
} else {
    // Move into folder: "dest/filename_base"
    char *base_name = strrchr(filename, '/');
    if (base_name) {
        snprintf(new_path, MAX_PATH_LEN, "%s/%s", dest_folder, base_name + 1);
    } else {
        snprintf(new_path, MAX_PATH_LEN, "%s/%s", dest_folder, filename);
    }
}

// 5. Check Collision
if (trie_search(new_path) != -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Destination already exists\n", RESP_CONFLICT);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 6. SYNCHRONOUS SS UPDATE
// We talk to the SS while holding the lock.
// This blocks other metadata ops, ensuring safety.
int ss_index = g_file_map[file_index].ss_index;

pthread_mutex_lock(&ss_list_mutex);
int ss_fd = g_ss_list[ss_index].conn_fd;
pthread_mutex_unlock(&ss_list_mutex);

char cmd_buf[MAX_MSG_LEN];
snprintf(cmd_buf, MAX_MSG_LEN, "%s %s %s\n", NM_RENAME, filename, new_path);

```

```

// Send
if (send(ss_fd, cmd_buf, strlen(cmd_buf), 0) < 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// Wait for ACK (This prevents the race condition)
char ss_resp[MAX_MSG_LEN];
memset(ss_resp, 0, MAX_MSG_LEN);
if (read(ss_fd, ss_resp, MAX_MSG_LEN - 1) <= 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// Check ACK
if (strncmp(ss_resp, RESP_OK, 3) != 0) {
    // SS failed (e.g., disk error). Do NOT update metadata.
    send(client_fd, ss_resp, strlen(ss_resp), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 7. SS Succeeded. Update Metadata safely.
trie_remove(filename);
cache_remove(filename);

strcpy(g_file_map[file_index].path, new_path);
trie_insert(new_path, file_index);
save_metadata_to_disk();

pthread_mutex_unlock(&file_map_mutex);

snprintf(resp_buf, MAX_MSG_LEN, "%s File moved\n", RESP_OK);
send(client_fd, resp_buf, strlen(resp_buf), 0);
}

void do_view_folder(int client_fd, char* username, char* foldername) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: VIEWFOLDER, FOLDER: %s", username, foldername);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];
    char payload[4096] = "";

    snprintf(payload, sizeof(payload), "%s\n==== Contents of %s ====\n", RESP_OK, foldername);

    pthread_mutex_lock(&file_map_mutex);
}

```

```

// Validation: Check if folder exists and IS a folder
int folder_idx = trie_search(foldername);
if (folder_idx != -1 && !g_file_map[folder_idx].is_directory) {
    // It's a file, not a folder.
    snprintf(resp_buf, MAX_MSG_LEN, "%s Not a folder\n", RESP_BAD_REQ);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

int len = strlen(foldername);
int count = 0;

for(int i=0; i<g_file_count; i++) {
    // Check if path starts with "foldername/"
    if (strncmp(g_file_map[i].path, foldername, len) == 0) {
        // Ensure it matches exactly as a directory (next char is '/')
        if (g_file_map[i].path[len] == '/') {
            // It is a child.
            char* relative_name = g_file_map[i].path + len + 1;

            // Simple check to only show direct children (no '/' in relative part)
            // If relative_name has a '/', it's a grandchild.
            // For a simple listing, we can just list everything or filter.
            // Let's just list everything recursive for visibility.

            char type_char = g_file_map[i].is_directory ? 'D' : 'F';
            snprintf(payload + strlen(payload), sizeof(payload) - strlen(payload),
                    "[%c] %s\n", type_char, relative_name);
            count++;
        }
    }
}
pthread_mutex_unlock(&file_map_mutex);

if (count == 0) strcat(payload, "(Empty)\n");
send(client_fd, payload, strlen(payload), 0);
}

void do_delete(int client_fd, char* requester_username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: DELETE, FILE: %s", requester_username, filename);
    log_event(log_msg);
    printf("Client %s requesting DELETE: %s\n", requester_username, filename);

    char resp_buf[MAX_MSG_LEN]; // Declared once

    // --- 1. Find file, check ACL, get SS index ---
    pthread_mutex_lock(&file_map_mutex);
}

```

```

int file_index = -1;

// 2. Check cache (fastest)
CacheEntry* entry = cache_find(filename);

if (entry) {
    file_index = entry->file_index;
    cache_move_to_front(entry);
} else {
    file_index = trie_search(filename);
    if (file_index != -1) {
        cache_add(filename, file_index);
    }
}

// 5. Handle "Not Found"
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

FileMapEntry* file = &g_file_map[file_index];

// 6. ACL Check
if (strcmp(requester_username, file->owner) != 0) {
    printf("ACL Check: User %s is NOT OWNER of %. Delete denied.\n", requester_username, filename);
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 7. Get the SS index *before* unlocking
int ss_index = file->ss_index;

// 8. *** UNLOCK THE MUTEX *** before network I/O
pthread_mutex_unlock(&file_map_mutex);

// --- 2. Communicate with SS (No lock held) ---

// 9. Get SS info
pthread_mutex_lock(&ss_list_mutex);
int ss_fd = g_ss_list[ss_index].conn_fd;
pthread_mutex_unlock(&ss_list_mutex);

// 10. Send command to SS
char command_buf[MAX_MSG_LEN];
sprintf(command_buf, "%s %s\n", NM_DELETE, filename);
if (send(ss_fd, command_buf, strlen(command_buf), 0) < 0) {
    perror("Failed to send NM_DELETE to SS");
}

```

```

snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
send(client_fd, resp_buf, strlen(resp_buf), 0);
return; // No lock to unlock
}

// 11. 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 DELETE\n");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return; // No lock to unlock
}

// --- 3. Commit changes (Re-acquire lock) ---

// 12. If SS says OK, delete from our map
if (strncmp(ss_resp, RESP_OK, strlen(RESP_OK)) == 0) {

    // *** RE-ACQUIRE THE LOCK ***
    pthread_mutex_lock(&file_map_mutex);

    printf("SS confirmed deletion. Removing from map.\n");

    // We must re-find the file index, in case it changed
    // while we were talking to the SS.
    int current_file_index = trie_search(filename);
    if (current_file_index == -1) {
        // This is weird, but could happen.
        printf("File was already deleted by another thread.\n");
        pthread_mutex_unlock(&file_map_mutex);
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK); // Just say OK
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        return;
    }

    // Get info about the *last* file before we overwrite
    int last_index = g_file_count - 1;
    char last_filename[MAX_PATH_LEN];
    strcpy(last_filename, g_file_map[last_index].path);

    // 1. Invalidate deleted file from Trie and Cache
    trie_remove(filename);
    cache_remove(filename);

    // 2. Perform the "swap-with-last"
    if (current_file_index != last_index) {
        g_file_map[current_file_index] = g_file_map[last_index];

        // 3. Update the *moved* file's index in the Trie
        trie_update_index(last_filename, current_file_index);
    }
}

```

```

        // 4. Invalidate the *moved* file's stale entry from Cache
        cache_remove(last_filename);
    }

    g_file_count--;

    save_metadata_to_disk();

    // 5. Unlock
    pthread_mutex_unlock(&file_map_mutex);

    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
    send(client_fd, resp_buf, strlen(resp_buf), 0);

} else {
    // SS reported an error
    printf("SS reported an error during deletion.\n");
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
}

}

void do_view(int client_fd, char* requester_username, char* flags) {
    char log_msg[MAX_MSG_LEN];
    sprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: VIEW, FLAGS: %s", requester_username, flags);
    log_event(log_msg);
    printf("Client %s requesting VIEW with flags: %s\n", requester_username, flags);

    // Parse the flags
    bool flag_a = (strstr(flags, "a") != NULL);
    bool flag_l = (strstr(flags, "l") != NULL);

    char payload[4096] = ""; // Big buffer
    sprintf(payload, sizeof(payload), "%s\n", RESP_OK);

    pthread_mutex_lock(&file_map_mutex);
    for (int i = 0; i < g_file_count; i++) {
        FileMapEntry* file = &g_file_map[i];
        bool has_access = false;

        // 1. Check for '-a' flag (admin view)
        if (flag_a) {
            has_access = true;
        } else {
            // 2. Run the same ACL check as do_read
            if (strcmp(requester_username, file->owner) == 0) {
                has_access = true;
            }
            if (!has_access) {
                for (int j = 0; j < file->acl_count; j++) {
                    if (strcmp(requester_username, file->acl_list[j].username) == 0) {
                        has_access = true;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        }

    }

}

// 3. If they have access, add this file to the payload
if (has_access) {
    char line_buf[512];
    if (flag_l) {
        // Stub metadata as 0,0 for now, as planned
        char time_str[100];
        struct tm ltime;
        localtime_r(&file->accessed_at, &ltime);
        strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);

        snprintf(line_buf, sizeof(line_buf), "| %-20s | %-5d | %-5d | %-20s | %-10s |\n", file->path, file->word_count, file->owner);
    } else {
        snprintf(line_buf, sizeof(line_buf), "%s\n", file->path);
    }
    // Safely append to the payload
    strncat(payload, line_buf, sizeof(payload) - strlen(payload) - 1);
}
}

pthread_mutex_unlock(&file_map_mutex);

send(client_fd, payload, strlen(payload), 0);
}

int registry_has_user(const char* name) {
    for (int i = 0; i < g_user_registry_count; i++) {
        if (strcmp(g_user_registry[i].username, name) == 0)
            return 1;
    }
    return 0;
}

void registry_add_user(const char* name) {
    // Don't add duplicates
    if (registry_has_user(name))
        return;

    strcpy(g_user_registry[g_user_registry_count].username, name);
    g_user_registry_count++;
}

void do_list_users(int client_fd) {
    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);

char log_msg[MAX_MSG_LEN];
snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: LIST", username);
log_event(log_msg);
printf("Client %s requested user list.\n", username);

// We'll build a big string payload.
// 4096 is a safe, large size.
char payload[4096] = "";

// Start the response with 200 OK
snprintf(payload, sizeof(payload), "%s\n", RESP_OK);

pthread_mutex_lock(&registry_mutex);
for (int i = 0; i < g_user_registry_count; i++) {
    snprintf(payload + strlen(payload), sizeof(payload) - strlen(payload),
              "%s\n", g_user_registry[i].username);
}
pthread_mutex_unlock(&registry_mutex);

// Send the whole list in one go
send(client_fd, payload, strlen(payload), 0);
}

void do_info(int client_fd, char* requester_username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: INFO, FILE: %s", requester_username, filename);
    log_event(log_msg);
    printf("Client %s requesting INFO for: %s\n", requester_username, filename);

    char payload[4096] = ""; // Big buffer
    char resp_buf[MAX_MSG_LEN];

    // 1. Lock the central mutex
    pthread_mutex_lock(&file_map_mutex);

    int file_index = -1;
    // char resp_buf[MAX_MSG_LEN]; // Make sure resp_buf is declared

    // 2. Check cache (fastest)
    CacheEntry* entry = cache_find(filename);

    if (entry) {
        // --- CACHE HIT ---
        file_index = entry->file_index;
        cache_move_to_front(entry); // Mark as recently used
    }
}

```

```

} else {
    // --- CACHE MISS ---
    // 3. Check trie (fast)
    file_index = trie_search(filename);
    if (file_index != -1) {
        // 4. Add to cache if we found it
        cache_add(filename, file_index);
    }
}

// 5. Handle "Not Found"
if (file_index == -1) {
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return; // Exit the function
}

// --- We now have a valid file_index, and the lock is still held ---
FileMapEntry* file = &g_file_map[file_index];

bool has_access = false;

if (strcmp(requester_username, file->owner) == 0) {
    has_access = true;
}
if (!has_access) {
    for (int j = 0; j < file->acl_count; j++) {
        if (strcmp(requester_username, file->acl_list[j].username) == 0) {
            has_access = true;
            break;
        }
    }
}
if (!has_access) {
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// 3. If access OK, build the detailed payload
sprintf(payload, sizeof(payload), "%s\n", RESP_OK); // Start with 200 OK
sprintf(payload + strlen(payload), "File: %s\n", file->path);
sprintf(payload + strlen(payload), "Owner: %s\n", file->owner);
sprintf(payload + strlen(payload), "Size: %d words, %d chars\n", file->word_count, file->char_count);

char time_str[100];
struct tm ltime;

// Format Created time
localtime_r(&file->created_at, &ltime);

```

```

strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);
sprintf(payload + strlen(payload), "Created: %s\n", time_str);

// Format Modified time
localtime_r(&file->modified_at, &ltime);
strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);
sprintf(payload + strlen(payload), "Last Modified: %s\n", time_str);

// Format Accessed time
localtime_r(&file->accessed_at, &ltime);
strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);
sprintf(payload + strlen(payload), "Last Accessed: %s\n", time_str);

sprintf(payload + strlen(payload), "---Access List---\n");
sprintf(payload + strlen(payload), " %s (Owner)\n", file->owner);

for (int j = 0; j < file->acl_count; j++) {
    sprintf(payload + strlen(payload), " %s (%c)\n", file->acl_list[j].username, file->acl_list[j].permission);
}

pthread_mutex_unlock(&file_map_mutex);
send(client_fd, payload, strlen(payload), 0);
}

/*
* =====
* CHECKPOINT HANDLER FUNCTIONS
* =====
*/
void do_checkpoint(int client_fd, char* username, char* filename, char* tag) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: CHECKPOINT, FILE: %s, TAG: %s",
             username, filename, tag);
    log_event(log_msg);
    printf("Client %s requesting CHECKPOINT for: %s (tag: %s)\n", username, filename, tag);

    char resp_buf[MAX_MSG_LEN];

    // 1. Find file and check ownership
    pthread_mutex_lock(&file_map_mutex);

    int file_index = trie_search(filename);
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    FileMapEntry* file = &g_file_map[file_index];
}

```

```

// --- SAFETY CHECK ---
if (file->is_directory) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Cannot perform file operation on a folder\n", RESP_BAD_REQ);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

// -------

// Only owner can create checkpoints
if (strcmp(username, file->owner) != 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

int ss_index = file->ss_index;
pthread_mutex_unlock(&file_map_mutex);

// 2. Forward command to SS
pthread_mutex_lock(&ss_list_mutex);
int ss_fd = g_ss_list[ss_index].conn_fd;
pthread_mutex_unlock(&ss_list_mutex);

char command_buf[MAX_MSG_LEN];
snprintf(command_buf, MAX_MSG_LEN, "%s %s %s\n", NM_CHECKPOINT, filename, tag);

if (send(ss_fd, command_buf, strlen(command_buf), 0) < 0) {
    perror("Failed to send NM_CHECKPOINT to SS");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// 3. Wait for SS response
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 CHECKPOINT\n");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// 4. Relay response to client
send(client_fd, ss_resp, strlen(ss_resp), 0);
}

void do_viewcheckpoint(int client_fd, char* username, char* filename, char* tag) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: VIEWCHECKPOINT, FILE: %s, TAG: %s",

```

```

        username, filename, tag);
log_event(log_msg);

char resp_buf[MAX_MSG_LEN];

// --- 1. ACL Check (Same as do_read) ---
pthread_mutex_lock(&file_map_mutex);

int file_index = trie_search(filename);
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

FileMapEntry* file = &g_file_map[file_index];
bool has_access = false;
if (strcmp(username, file->owner) == 0) has_access = true;
if (!has_access) {
    for (int i = 0; i < file->acl_count; i++) {
        if (strcmp(username, file->acl_list[i].username) == 0) {
            has_access = true;
            break;
        }
    }
}

if (!has_access) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

int ss_index = file->ss_index;
pthread_mutex_unlock(&file_map_mutex);

// --- 2. Get SS Client-Facing Info (like do_exec) ---
pthread_mutex_lock(&ss_list_mutex);
char ss_ip[INET_ADDRSTRLEN];
int ss_port = g_ss_list[ss_index].client_port; // <- The client port!
strcpy(ss_ip, g_ss_list[ss_index].ip);
pthread_mutex_unlock(&ss_list_mutex);

// --- 3. NM connects TO SS on a NEW socket ---
int ss_sock = connect_to_server(ss_ip, ss_port);
if (ss_sock < 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SS_DOWN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

```

```

// --- 4. Request Checkpoint from SS (using new command) ---
char req_buf[MAX_MSG_LEN];
snprintf(req_buf, MAX_MSG_LEN, "%s %s %s\n", SS_VIEWCHECKPOINT, filename, tag);
send(ss_sock, req_buf, strlen(req_buf), 0);

// --- 5. Read SS response (status code first) ---
char ss_resp[MAX_MSG_LEN];
int n = read(ss_sock, ss_resp, sizeof(ss_resp) - 1);
if (n <= 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    close(ss_sock);
    return;
}
ss_resp[n] = '\0';

// --- 6. Check SS response and relay to client ---
if (strncmp(ss_resp, RESP_OK, 3) != 0) {
    // It's an error (e.g., 404), just forward it
    send(client_fd, ss_resp, strlen(ss_resp), 0);
} else {
    // It's 200 OK. Send 200 to our client
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
    send(client_fd, resp_buf, strlen(resp_buf), 0);

    // Check if data was already in the first read
    char* payload = strchr(ss_resp, '\n');
    if (payload && strlen(payload + 1) > 0) {
        send(client_fd, payload + 1, strlen(payload + 1), 0);
    }

    // Loop to read and relay the rest of the content
    char content_buffer[4096];
    while ((n = read(ss_sock, content_buffer, sizeof(content_buffer))) > 0) {
        send(client_fd, content_buffer, n, 0);
    }

    // Send the "done" marker to our client
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_EXEC_DONE);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
}

close(ss_sock); // Close the temporary socket
}

void do_revert(int client_fd, char* username, char* filename, char* tag) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: REVERT, FILE: %s, TAG: %s",
            username, filename, tag);
    log_event(log_msg);
}

```

```

char resp_buf[MAX_MSG_LEN];

pthread_mutex_lock(&file_map_mutex);

int file_index = trie_search(filename);
if (file_index == -1) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

FileMapEntry* file = &g_file_map[file_index];

// --- SAFETY CHECK ---
if (file->is_directory) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s Cannot perform file operation on a folder\n", RESP_BAD_REQ);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}
// -----

// Check write access (same as WRITE)
bool has_write_access = false;
if (strcmp(username, file->owner) == 0) {
    has_write_access = true;
}
if (!has_write_access) {
    for (int i = 0; i < file->acl_count; i++) {
        if (strcmp(username, file->acl_list[i].username) == 0) {
            if (file->acl_list[i].permission == PERM_WRITE) {
                has_write_access = true;
            }
            break;
        }
    }
}

if (!has_write_access) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

int ss_index = file->ss_index;
pthread_mutex_unlock(&file_map_mutex);

// Forward to SS
pthread_mutex_lock(&ss_list_mutex);
int ss_fd = g_ss_list[ss_index].conn_fd;

```

```

pthread_mutex_unlock(&ss_list_mutex);

char command_buf[MAX_MSG_LEN];
snprintf(command_buf, MAX_MSG_LEN, "%s %s %s\n", NM_REVERT, filename, tag);
send(ss_fd, command_buf, strlen(command_buf), 0);

char ss_resp[MAX_MSG_LEN];
memset(ss_resp, 0, MAX_MSG_LEN);
if (read(ss_fd, ss_resp, MAX_MSG_LEN - 1) <= 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// Update metadata on success
if (strncmp(ss_resp, RESP_OK, 3) == 0) {
    pthread_mutex_lock(&file_map_mutex);
    int current_index = trie_search(filename);
    if (current_index != -1) {
        g_file_map[current_index].modified_at = time(NULL);
        save_metadata_to_disk();
    }
    pthread_mutex_unlock(&file_map_mutex);
}

send(client_fd, ss_resp, strlen(ss_resp), 0);
}

void do_listcheckpoints(int client_fd, char* username, char* filename) {
    char log_msg[MAX_MSG_LEN];
    snprintf(log_msg, MAX_MSG_LEN, "USER: %s, REQ: LISTCHECKPOINTS, FILE: %s",
             username, filename);
    log_event(log_msg);

    char resp_buf[MAX_MSG_LEN];

    // --- 1. ACL Check (Same as do_read) ---
    pthread_mutex_lock(&file_map_mutex);

    int file_index = trie_search(filename);
    if (file_index == -1) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    FileMapEntry* file = &g_file_map[file_index];
    bool has_access = false;
    if (strcmp(username, file->owner) == 0) has_access = true;
    if (!has_access) {
        for (int i = 0; i < file->acl_count; i++) {

```

```

        if (strcmp(username, file->acl_list[i].username) == 0) {
            has_access = true;
            break;
        }
    }
}

if (!has_access) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_FORBIDDEN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    pthread_mutex_unlock(&file_map_mutex);
    return;
}

int ss_index = file->ss_index;
pthread_mutex_unlock(&file_map_mutex);

// --- 2. Get SS Client-Facing Info (like do_exec) ---
pthread_mutex_lock(&ss_list_mutex);
char ss_ip[INET_ADDRSTRLEN];
int ss_port = g_ss_list[ss_index].client_port; // <- The client port!
strcpy(ss_ip, g_ss_list[ss_index].ip);
pthread_mutex_unlock(&ss_list_mutex);

// --- 3. NM connects TO SS on a NEW socket ---
int ss_sock = connect_to_server(ss_ip, ss_port);
if (ss_sock < 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SS_DOWN);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    return;
}

// --- 4. Request List from SS (using new command) ---
char req_buf[MAX_MSG_LEN];
snprintf(req_buf, MAX_MSG_LEN, "%s %s\n", SS_LISTCHECKPOINTS, filename);
send(ss_sock, req_buf, strlen(req_buf), 0);

// --- 5. Read SS response (status code first) ---
char ss_resp[MAX_MSG_LEN];
int n = read(ss_sock, ss_resp, sizeof(ss_resp) - 1);
if (n <= 0) {
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
    close(ss_sock);
    return;
}
ss_resp[n] = '\0';

// --- 6. Check SS response and relay to client ---
if (strncmp(ss_resp, RESP_OK, 3) != 0) {
    // It's an error (e.g., 500), just forward it
    send(client_fd, ss_resp, strlen(ss_resp), 0);
}

```

```

} else {
    // It's 200 OK. Send 200 to our client
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
    send(client_fd, resp_buf, strlen(resp_buf), 0);

    // Check if data was already in the first read
    char* payload = strchr(ss_resp, '\n');
    if (payload && strlen(payload + 1) > 0) {
        send(client_fd, payload + 1, strlen(payload + 1), 0);
    }

    // Loop to read and relay the rest of the list
    char content_buffer[4096];
    while ((n = read(ss_sock, content_buffer, sizeof(content_buffer))) > 0) {
        send(client_fd, content_buffer, n, 0);
    }

    // Send the "done" marker to our client
    sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_EXEC_DONE);
    send(client_fd, resp_buf, strlen(resp_buf), 0);
}

close(ss_sock); // Close the temporary socket
}

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 arg2[MAX_USERNAME_LEN]; // For target_user
    char arg3[10]; // For flags (" -R" or " -W")
    char resp_buf[MAX_MSG_LEN];

    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);
        memset(arg2, 0, MAX_USERNAME_LEN);
        memset(arg3, 0, 10);

```

```

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

if (bytes_read <= 0) {
    printf("Client %s disconnected.\n", username);
    pthread_mutex_lock(&client_list_mutex);
    int client_index = -1;
    for (int i = 0; i < g_client_count; i++) {
        if (g_client_list[i].conn_fd == client_fd) {
            client_index = i;
            break;
        }
    }

    if (client_index != -1) {
        // "Swap-with-last" delete
        g_client_list[client_index] = g_client_list[g_client_count - 1];
        g_client_count--;
    }
    pthread_mutex_unlock(&client_list_mutex);
    break; // Exit loop
}

int items_scanned = sscanf(buffer, "%s %s %s %s", cmd, arg1, arg2, arg3);
if (items_scanned <= 0) {
    continue; // Ignore empty lines (like just hitting Enter)
}

if (strcmp(cmd, C_REQ_CREATE, strlen(C_REQ_CREATE)) == 0) {
    do_create(client_fd, username, arg1);
}
else if (strcmp(cmd, C_REQ_ADD_ACC, strlen(C_REQ_ADD_ACC)) == 0) {
    if (items_scanned < 4) { // Needs 4 args: CMD, file, user, flag
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    } else {
        do_add_access(client_fd, username, arg1, arg2, arg3[1]); // arg3[1] is 'R' or 'W'
    }
}
else if (strcmp(cmd, C_REQ_REM_ACC, strlen(C_REQ_REM_ACC)) == 0) {
    if (items_scanned < 3) { // Needs 3 args: CMD, file, user
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    } else {
        do_rem_access(client_fd, username, arg1, arg2); // arg1=file, arg2=user
    }
}
else if (strcmp(cmd, C_REQ_VIEWCHECKPOINT, strlen(C_REQ_VIEWCHECKPOINT)) == 0) {
    if (items_scanned < 3) {
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    } else {
        do_viewcheckpoint(client_fd, username, arg1, arg2);
    }
}

```

```

        }
    }

    else if (strncmp(cmd, C_REQ_VIEW_REQUESTS, strlen(C_REQ_VIEW_REQUESTS)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_view_requests(client_fd, username, arg1);
        }
    }

    else if (strncmp(cmd, C_REQ_READ, strlen(C_REQ_READ)) == 0) {
        do_read(client_fd, username, arg1);
    }

    else if (strncmp(cmd, C_REQ_WRITE, strlen(C_REQ_WRITE)) == 0) {
        if (items_scanned < 2) { // Needs CMD and filename
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_write(client_fd, username, arg1); // arg1 is filename
        }
    }

    else if (strncmp(cmd, C_REQ_UNDO, strlen(C_REQ_UNDO)) == 0) {
        if (items_scanned < 2) { // Needs CMD and filename
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_undo(client_fd, username, arg1); // arg1 is filename
        }
    }

    else if (strncmp(cmd, C_REQ_STREAM, strlen(C_REQ_STREAM)) == 0) {
        if (items_scanned < 2) { // Needs CMD and filename
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            // C_STREAM uses the exact same logic as C_READ
            do_read(client_fd, username, arg1);
        }
    }

    else if (strncmp(cmd, C_REQ_DELETE, strlen(C_REQ_DELETE)) == 0) {
        if (items_scanned < 2) { // Needs CMD and filename
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_delete(client_fd, username, arg1); // arg1 is filename
        }
    }

    else if (strncmp(cmd, C_REQ_LISTCHECKPOINTS, strlen(C_REQ_LISTCHECKPOINTS)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_listcheckpoints(client_fd, username, arg1);
        }
    }
}

```

```

        }
    }

    else if (strncmp(cmd, C_REQ_LIST, strlen(C_REQ_LIST)) == 0) {
        do_list_users(client_fd);
    }

    else if (strncmp(cmd, C_REQ_VIEW, strlen(C_REQ_VIEW)) == 0) {
        do_view(client_fd, username, arg1); // arg1 holds the flags
    }

    else if (strncmp(cmd, C_REQ_INFO, strlen(C_REQ_INFO)) == 0) {
        do_info(client_fd, username, arg1); // arg1 is filename
    }

    else if (strncmp(cmd, C_REQ_EXEC, strlen(C_REQ_EXEC)) == 0) {
        if (items_scanned < 2) { // Needs CMD and filename
            sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_exec(client_fd, username, arg1);
        }
    }

    else if (strncmp(cmd, C_REQ_CHECKPOINT, strlen(C_REQ_CHECKPOINT)) == 0) {
        if (items_scanned < 3) {
            sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_checkpoint(client_fd, username, arg1, arg2);
        }
    }

}

else if (strncmp(cmd, C_REQ_REVERT, strlen(C_REQ_REVERT)) == 0) {
    if (items_scanned < 3) {
        sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    } else {
        do_revert(client_fd, username, arg1, arg2);
    }
}

else if (strncmp(cmd, C_REQ_REQUEST_ACC, strlen(C_REQ_REQUEST_ACC)) == 0) {
    if (items_scanned < 3) {
        sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    } else {
        do_request_access(client_fd, username, arg1, arg2[1]);
    }
}

else if (strncmp(cmd, C_REQ_APPROVE_ACC, strlen(C_REQ_APPROVE_ACC)) == 0) {
    if (items_scanned < 2) {
        sprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    } else {
        int req_id = atoi(arg1);
        do_approve_request(client_fd, username, req_id);
    }
}

```

```

    }

    else if (strncmp(cmd, C_REQ_DENY_ACC, strlen(C_REQ_DENY_ACC)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            int req_id = atoi(arg1);
            do_deny_request(client_fd, username, req_id);
        }
    }

    else if (strncmp(cmd, C_REQ_MY_REQUESTS, strlen(C_REQ_MY_REQUESTS)) == 0) {
        do_my_requests(client_fd, username);
    }

    else if (strncmp(cmd, C_REQ_CREATEFOLDER, strlen(C_REQ_CREATEFOLDER)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_create_folder(client_fd, username, arg1);
        }
    }

    else if (strncmp(cmd, C_REQ_MOVE, strlen(C_REQ_MOVE)) == 0) {
        if (items_scanned < 3) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_move(client_fd, username, arg1, arg2);
        }
    }

    else if (strncmp(cmd, C_REQ_VIEWFOLDER, strlen(C_REQ_VIEWFOLDER)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_view_folder(client_fd, username, arg1);
        }
    }

    else {
        // Now this is a *real* unknown command
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
        send(client_fd, resp_buf, strlen(resp_buf), 0);
    }
}

close(client_fd);
free(arg);
return NULL;
}

int main(int argc, char*argv[]){
    printf("Starting server..\n");
    load_metadata_from_disk();
}

```

```

g_file_trie_root = create_trie_node();
build_trie_from_map();

g_log_fp = fopen(NM_LOG_FILE, "a"); // "a" for append
if (g_log_fp == NULL) {
    perror("FATAL: Failed to open log file");
    exit(1);
}
// Set the "close-on-exec" flag for the log file
int log_fd = fileno(g_log_fp);
fcntl(log_fd, F_SETFD, FD_CLOEXEC);

log_event("--- Name Server Started ---");

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 3
// "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 re
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 a
//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

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
//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
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 you
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
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 imp
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 value of client_fd. The first thread would suddenly have its file descriptor changed! By malloc-ing new memory, we 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 is when the main loop accepts the next connection; threads would race and get wrong FDs. Allocating per-thread memory solves this.

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
#include <stdbool.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>

FILE* g_log_fp = NULL;
pthread_mutex_t g_log_mutex = PTHREAD_MUTEX_INITIALIZER;
char g_ss_root_path[64];

#define NM_IP "127.0.0.1" // IP for the Name Server (NM)

// --- Global Lock Manager ---
typedef struct {

```

```

char path[MAX_PATH_LEN];
bool is_locked;
int sentence_num;
// We could also store the user who locked it
} FileLock;

#define MAX_LOCKS 50
FileLock g_lock_list[MAX_LOCKS];
int g_lock_count = 0;
pthread_mutex_t g_lock_list_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t g_commit_mutex = PTHREAD_MUTEX_INITIALIZER;
// --- End of Lock Manager ---

// --- Checkpoint Manager ---
typedef struct {
    char tag[MAX_CHECKPOINT_TAG];
    char filename[MAX_PATH_LEN];
    time_t created_at;
    char checkpoint_path[MAX_PATH_LEN]; // Path to the checkpoint file
} Checkpoint;

#define MAX_CHECKPOINTS 200 // Total checkpoints across all files
Checkpoint g_checkpoint_list[MAX_CHECKPOINTS];
int g_checkpoint_count = 0;
pthread_mutex_t g_checkpoint_mutex = PTHREAD_MUTEX_INITIALIZER;
// --- End of Checkpoint Manager ---

// global for our connection to the NM
int g_nm_fd;

// Struct for passing args to client threads
typedef struct {
    int client_fd;
    int ss_port;
} ClientThreadArgs;

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

// Forward declarations for checkpoint functions
char* read_checkpoint(const char* filename, const char* tag);
char* list_checkpoints(const char* filename);

// Helper function to read an entire file into a new, allocated string.
// Returns NULL on failure. Caller MUST free() the result.
char* read_file_to_memory(const char* filepath) {
    FILE *fp = fopen(filepath, "r");
    if (fp == NULL) {
        perror("fopen (read_file_to_memory)");
        return NULL;
    }
}

```

```

fseek(fp, 0, SEEK_END);
long file_size = ftell(fp);
fseek(fp, 0, SEEK_SET);

char *buffer = (char*)malloc(file_size + 1);
if (buffer == NULL) {
    perror("malloc");
    fclose(fp);
    return NULL;
}

fread(buffer, 1, file_size, fp);
buffer[file_size] = '\0';
fclose(fp);
return buffer;
}

// Validates a filename to prevent path traversal attacks.
// Returns true if the filename is safe, false otherwise.
bool is_valid_filename(const char* filename) {
    // 1. Check for absolute paths
    if (filename[0] == '/') {
        printf("[SS-Security] Rejecting absolute path: %s\n", filename);
        return false;
    }

    // 2. Check for ".." (directory traversal)
    if (strstr(filename, "..") != NULL) {
        printf("[SS-Security] Rejecting path traversal: %s\n", filename);
        return false;
    }

    // 3. Check for empty filename
    if (strlen(filename) == 0) {
        return false;
    }
}

return true;
}

// Counts words (space-separated) and chars
void count_metadata(char* content, int* word_count, int* char_count) {
    *char_count = strlen(content);
    *word_count = 0;

    char* temp_content = strdup(content); // Make a copy
    char* word = strtok(temp_content, " \n\t");
    while (word != NULL) {
        (*word_count)++;
        word = strtok(NULL, " \n\t");
    }
    free(temp_content);
}

```

```
}

// Helper function to check if string ends with sentence delimiter
bool ends_with_delimiter(const char* str) {
    if (str == NULL || strlen(str) == 0) return false;
    char last = str[strlen(str) - 1];
    return (last == '.' || last == '!' || last == '?');
}

/*
 * Counts the number of sentences in a file's content.
 * An empty file has 0 sentences.
 * A file with content but no delimiter has 1 sentence.
 * Each delimiter (. ! ?) marks the end of a sentence.
 */

int count_sentences(char* content) {
    if (content == NULL || *content == '\0') {
        return 0; // Empty content has 0 sentences
    }

    int sentence_count = 0;
    char* ptr = content;

    // Skip leading whitespace
    while (*ptr == ' ' || *ptr == '\n' || *ptr == '\t') {
        ptr++;
    }

    if (*ptr == '\0') {
        return 0; // Only whitespace
    }

    // We have at least one sentence (even without delimiter)
    sentence_count = 1;

    // Count additional sentences by counting delimiters
    while (*ptr != '\0') {
        if (*ptr == '.' || *ptr == '!' || *ptr == '?') {
            // Skip consecutive delimiters and whitespace
            ptr++;
            while (*ptr == ' ' || *ptr == '\n' || *ptr == '\t' ||
                   *ptr == '.' || *ptr == '!' || *ptr == '?') {
                ptr++;
            }
        }

        // If there's more content after the delimiter, it's a new sentence
        if (*ptr != '\0') {
            sentence_count++;
        }
    }
}
```

```

    }

    return sentence_count;
}

void log_event(const char* message) {
    char time_str[100];
    time_t now = time(NULL);
    struct tm ltime; // A local struct for the thread-safe version

    // Use thread-safe localtime_r
    localtime_r(&now, &ltime);
    strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);

    // Lock, write, unlock
    pthread_mutex_lock(&g_log_mutex);
    if (g_log_fp) {
        fprintf(g_log_fp, "[%s] %s\n", time_str, message);
        fflush(g_log_fp); // Ensure it writes immediately
    }
    pthread_mutex_unlock(&g_log_mutex);
}

// Connects to NM, sends one message, and disconnects.
void send_async_update_to_nm(char* message) {
    int temp_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (temp_sock < 0) {
        perror("send_async: socket");
        return;
    }

    struct sockaddr_in nm_addr;
    struct hostent *nm_server = gethostbyname(NM_IP);
    if (nm_server == NULL) {
        fprintf(stderr, "ERROR, no such host as %s\n", NM_IP);
        close(temp_sock);
        return;
    }

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

    if (connect(temp_sock, (struct sockaddr *) &nm_addr, sizeof(nm_addr)) < 0) {
        perror("send_async: connect");
        close(temp_sock);
        return;
    }

    // Send the message and immediately close
    send(temp_sock, message, strlen(message), 0);
}

```

```

        close(temp_sock);
    }

/*
 * Helper function to find the Nth sentence in a file's content.
 * A sentence is defined by the delimiters ".!?"
 *
 * Parameters:
 *   content - The full string buffer of the file.
 *   sent_num - The sentence number to find (0-indexed).
 *   start_ptr - A (pointer to a char*) to store the start of the sentence.
 *   end_ptr - A (pointer to a char*) to store the end (the delimiter) of the sentence.
 *
 * Returns:
 *   true (1) if the sentence was found.
 *   false (0) if the sentence number is out of bounds.
 */
bool find_sentence_pointers(char* content, int sent_num, char** start_ptr, char** end_ptr) {
    char* current_start = content;
    int current_sent = 0;

    // Loop until we find the sentence or run out of content
    while (*current_start != '\0') {
        // 1. Skip any leading whitespace (like spaces or newlines)
        while (*current_start == ' ' || *current_start == '\n' || *current_start == '\t') {
            current_start++;
        }

        // If we hit the end of the string while skipping, we're done.
        if (*current_start == '\0') {
            return false;
        }

        // 2. Find the end of this sentence
        char* current_end = strpbrk(current_start, ".!?");

        // 3. Check if we found a delimiter
        if (current_end == NULL) {
            // No delimiter found, but there's content. This is the last sentence.
            // Check if this is the sentence we were looking for.
            if (current_sent == sent_num) {
                *start_ptr = current_start;
                // Set the end to the end of the string
                *end_ptr = current_start + strlen(current_start) - 1;
                return true;
            } else {
                // This was the last sentence, but not the one we wanted.
                return false;
            }
        }
    }

    // 4. We found a delimiter. Is this the sentence we want?
}

```

```

        if (current_sent == sent_num) {
            *start_ptr = current_start;
            *end_ptr = current_end;
            return true; // Success!
        }

        // 5. It wasn't. Move to the next sentence.
        current_sent++;
        current_start = current_end + 1; // Move past the delimiter
    }

    // We reached the end of the string without finding the sentence
    return false;
}

/*
 * Helper function to find the Nth word in a specific sentence.
 * A word is defined by the delimiter " " (space).
 *
 * Parameters:
 * sentence_start - Pointer to the start of the sentence (from find_sentence_pointers).
 * sentence_end - Pointer to the end of the sentence (the delimiter .!?)
 * word_idx - The word number to find (0-indexed).
 * start_ptr - A (pointer to a char*) to store the start of the word.
 * end_ptr - A (pointer to a char*) to store the end (the space) of the word.
 *
 * Returns:
 * true (1) if the word was found.
 * false (0) if the word index is out of bounds.
 */
bool find_word_pointers(char* sentence_start, char* sentence_end, int word_idx, char** start_ptr, char** end_ptr)
{
    char* current_start = sentence_start;
    int current_word = 0;

    while (current_start <= sentence_end) {
        // 1. Skip leading spaces (if any)
        while (*current_start == ' ' && current_start <= sentence_end) {
            current_start++;
        }

        // If we hit the end of the sentence while skipping, we're done.
        if (current_start > sentence_end) {
            return false;
        }

        // 2. Find the end of this word (the next space)
        char* current_end = strchr(current_start, ' ');

        // 3. Check if we found a space *within* the sentence
        if (current_end == NULL || current_end > sentence_end) {
            // No more spaces. This is the last word.
            if (current_word == word_idx) {

```

```

        *start_ptr = current_start;
        // The end of this word is the end of the sentence
        *end_ptr = sentence_end;
        return true;
    } else {
        return false;
    }
}

// 4. We found a space. Is this the word we want?
if (current_word == word_idx) {
    *start_ptr = current_start;
    *end_ptr = current_end; // The space is the end
    return true;
}

// 5. It wasn't. Move to the next word.
current_word++;
current_start = current_end + 1; // Move past the space
}

return false;
}

// Helper function to ensure directory exists (recursive mkdir)
void ensure_directory_exists(const char* filepath) {
    char path_copy[MAX_PATH_LEN];
    strcpy(path_copy, filepath);

    // Remove the filename to get just the dir
    char* last_slash = strrchr(path_copy, '/');
    if (last_slash) {
        *last_slash = '\0'; // Truncate to dir path

        char cmd[MAX_MSG_LEN];
        // Use mkdir -p. It's the safest, simplest way in C without writing 50 lines of code.
        snprintf(cmd, sizeof(cmd), "mkdir -p \"%s/%s\"", g_ss_root_path, path_copy);
        system(cmd);
    }
}

//thread function for a CLIENT connection
void *handle_client_request(void *arg) {
    ClientThreadArgs* args = (ClientThreadArgs*)arg;
    int client_fd = args->client_fd;
    int my_port = args->ss_port;
    free(arg);

    char buffer[MAX_MSG_LEN];
    char cmd[MAX_MSG_LEN];
    char filename[MAX_PATH_LEN - sizeof(g_ss_root_path) - 1];
    char log_msg[MAX_MSG_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);

        if (!is_valid_filename(filename)) {
            send(client_fd, RESP_BAD_REQ "\n", strlen(RESP_BAD_REQ "\n"), 0);
            close(client_fd);
            return NULL;
        }

        // 2. Check which command it is
        if (strcmp(cmd, SS_GET_FILE, strlen(SS_GET_FILE)) == 0) {

            snprintf(log_msg, MAX_MSG_LEN, "REQ: SS_GET_FILE, FILE: %s", filename);
            log_event(log_msg);
            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];
            snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, 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 if (strcmp(cmd, SS_LOCK, strlen(SS_LOCK)) == 0) {
            char local_path[MAX_PATH_LEN]; // Need to define local_path here
            snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
        }
    }
}

```

```

int sentence_num;
sscanf(buffer, "%*s %*s %d", &sentence_num);
snprintf(log_msg, MAX_MSG_LEN, "REQ: SS_LOCK, FILE: %s, SENTENCE: %d", filename, sentence_num);
log_event(log_msg);
printf("[SS-Client-Write] Lock requested for %s sen %d\n", filename, sentence_num);

// 1. CHECK IF ALREADY LOCKED
pthread_mutex_lock(&g_lock_list_mutex);
bool already_locked = false;
for (int i = 0; i < g_lock_count; i++) {
    if (strcmp(g_lock_list[i].path, filename) == 0 &&
        g_lock_list[i].sentence_num == sentence_num &&
        g_lock_list[i].is_locked) {
        already_locked = true;
        break;
    }
}

if (already_locked) {
    pthread_mutex_unlock(&g_lock_list_mutex);
    printf("[SS-Client-Write] Lock FAILED for %s sen %d (already locked)\n", filename, sentence_num);
    send(client_fd, RESP_LOCKED_ERR "\n", strlen(RESP_LOCKED_ERR "\n"), 0);
} else {
    // 2. OPTIMISTICALLY ACQUIRE LOCK (we'll rollback if validation fails)
    strcpy(g_lock_list[g_lock_count].path, filename);
    g_lock_list[g_lock_count].is_locked = true;
    g_lock_list[g_lock_count].sentence_num = sentence_num;
    g_lock_count++;
    pthread_mutex_unlock(&g_lock_list_mutex);

    time_t original_mod_time = 0; // Will store the file's "version"

    // 3. READ FILE TO MEMORY
    char* file_content_buffer = read_file_to_memory(local_path);
    if (file_content_buffer == NULL) {
        if (errno == ENOENT) { // File doesn't exist (new file or deleted)
            printf("[SS-Client-Write] File %s is new (empty).\n", filename);
            file_content_buffer = strdup(""); // Allocate empty string
            original_mod_time = 0; // Set timestamp to 0 for a new file
        } else {
            // Read error (not just ENOENT)
            printf("[SS-Client-Write] Error: File not found or unreadable, releasing lock: %s\n", filename);
        }
    }

    // Release the lock we just acquired
    pthread_mutex_lock(&g_lock_list_mutex);
    for (int i = 0; i < g_lock_count; i++) {
        if (strcmp(g_lock_list[i].path, filename) == 0 &&
            g_lock_list[i].sentence_num == sentence_num &&
            g_lock_list[i].is_locked) {
            g_lock_list[i] = g_lock_list[g_lock_count - 1];
            g_lock_count--;
            break;
        }
    }
}

```

```

        }

    }

    pthread_mutex_unlock(&g_lock_list_mutex);

    send(client_fd, RESP_NOT_FOUND "\n", strlen(RESP_NOT_FOUND "\n"), 0);
    file_content_buffer = NULL; // Ensure it stays NULL
}

} else {
    // File exists, get its timestamp
    struct stat file_stat;
    if (stat(local_path, &file_stat) == 0) {
        original_mod_time = file_stat.st_mtime;
    }
}

// Only proceed if the buffer is valid ---
if (file_content_buffer != NULL)
{
    // 4. VALIDATE SENTENCE INDEX
    int total_sentences = count_sentences(file_content_buffer);

    printf("[SS-Client-Write] File %s has %d sentences. Request for sentence %d.\n",
           filename, total_sentences, sentence_num);

    // Valid indices: 0 to total_sentences-1 (existing) OR total_sentences (append)
    if (sentence_num < 0 || sentence_num > total_sentences) {
        printf("[SS-Client-Write] Lock FAILED for %. Invalid sentence index %d (file has %d sentences)\n",
               filename, sentence_num, total_sentences);

        // Send error to client
        char error_msg[256];
        snprintf(error_msg, sizeof(error_msg),
                 "%s Sentence index %d out of range (file has %d sentences)\n",
                 RESP_NOT_FOUND, sentence_num, total_sentences);
        send(client_fd, error_msg, strlen(error_msg), 0);

        // ROLLBACK: Remove the lock we just added
        pthread_mutex_lock(&g_lock_list_mutex);
        for (int i = 0; i < g_lock_count; i++) {
            if (strcmp(g_lock_list[i].path, filename) == 0 &&
                g_lock_list[i].sentence_num == sentence_num) {
                // Remove this lock by replacing it with the last one
                g_lock_list[i] = g_lock_list[g_lock_count - 1];
                g_lock_count--;
                break;
            }
        }
        pthread_mutex_unlock(&g_lock_list_mutex);

        free(file_content_buffer);
        // Don't enter transaction loop, just continue to next command
    } else {
}
}

```

```

// 5. VALIDATION PASSED - Send LOCKED response
printf("[SS-Client-Write] Lock ACQUIRED for %s sen %d\n", filename, sentence_num);
send(client_fd, RESP_LOCKED "\n", strlen(RESP_LOCKED "\n"), 0);
while (1) { // Transaction loop
    memset(buffer, 0, MAX_MSG_LEN);
    int bytes_read = read(client_fd, buffer, MAX_MSG_LEN - 1);
    if (bytes_read <= 0) {
        printf("[SS-Client-Write] Client disconnected. Aborting write.\n");
        pthread_mutex_lock(&g_lock_list_mutex);

        for (int i = 0; i < g_lock_count; i++) {
            if (strcmp(g_lock_list[i].path, filename) == 0 &&
                g_lock_list[i].is_locked) {
                g_lock_list[i] = g_lock_list[g_lock_count - 1];
                g_lock_count--;
                printf("[SS-Client-Write] Lock RELEASED for sen %d due to disconnect.\n", sentence_num);
                break;
            }
        }
        pthread_mutex_unlock(&g_lock_list_mutex);
        free(file_content_buffer); // Free the memory we were editing
        break; // Client disconnected, abort
    }
}

sscanf(buffer, "%s", cmd);
if (strncmp(cmd, SS_UPDATE, strlen(SS_UPDATE)) == 0) {
    int word_idx;
    char new_content[MAX_MSG_LEN];
    char cmd_tmp[100];

    if (sscanf(buffer, "%s %d %[^\n]", cmd_tmp, &word_idx, new_content) < 3) {
        printf("[SS-Client-Write] Error: Malformed SS_UPDATE (expected 3+ parts, got <3).\n");
        continue;
    }

    char *sent_start, *sent_end, *word_start, *word_end;

    // --- FIX: Special Case for writing to an empty file ---
    if (sentence_num == 0 && strlen(file_content_buffer) == 0 && word_idx == 0) {
        printf("[SS-Client-Write] Special Case: Writing to empty file.\n");

        char* new_buffer = (char*)malloc(strlen(new_content) + 1);
        if (new_buffer == NULL) {
            perror("malloc");
            continue; // Skip update
        }
        strcpy(new_buffer, new_content);

        free(file_content_buffer);
        file_content_buffer = new_buffer;
    }
}

```

```

printf("[SS-Client-Write] Update successful for sen 0, word 0.\n");

continue; // Go to next update
}

// --- Normal Case: Find the sentence ---
if (!find_sentence_pointers(file_content_buffer, sentence_num, &sent_start, &sent_end)) {
    int sentence_count = 0;
    char* temp_start = file_content_buffer;
    while (*temp_start != '\0') {
        // Skip whitespace
        while (*temp_start == ' ' || *temp_start == '\n' || *temp_start == '\t') temp_start++;
        if (*temp_start == '\0') break; // Reached end

        char* temp_end = strpbrk(temp_start, ".!?");
        if (temp_end == NULL) {
            sentence_count++; // This is the last sentence
            break;
        }
        sentence_count++;
        temp_start = temp_end + 1;
    }
}

// If user wants to write to the *next* new sentence (and is writing word 0)
if (sentence_num == sentence_count && word_idx == 0) {
    printf("[SS-Client-Write] Appending new sentence %d.\n", sentence_num);

    int old_len = strlen(file_content_buffer);
    int new_content_len = strlen(new_content);
    // Need space for: existing content + space/period + new content + null
    // If old content exists and doesn't end with a delimiter, add one
    bool needs_delimiter = false;
    if (old_len > 0) {
        char last_char = file_content_buffer[old_len - 1];
        if (last_char != '.' && last_char != '!' && last_char != '?') {
            needs_delimiter = true;
        }
    }
}

int extra_chars = (old_len > 0 ? 1 : 0) + (needs_delimiter ? 1 : 0); // space and/or delimiter
char* new_buffer = (char*)malloc(old_len + extra_chars + new_content_len + 1);

if (new_buffer == NULL) {
    perror("malloc");
    continue;
}

// Copy old content, add delimiter if needed, add space, and the new content
if (old_len > 0) {
    strcpy(new_buffer, file_content_buffer);
    if (needs_delimiter) {

```

```

        new_buffer[old_len] = ':';
        new_buffer[old_len + 1] = ' ';
        strcpy(new_buffer + old_len + 2, new_content);
    } else {
        new_buffer[old_len] = ' ';
        strcpy(new_buffer + old_len + 1, new_content);
    }
} else {
    strcpy(new_buffer, new_content);
}

free(file_content_buffer);
file_content_buffer = new_buffer;

printf("[SS-Client-Write] New sentence append successful.\n");

continue; // Go to next update
} else {
    // This is a real error
    printf("[SS-Client-Write] Error: Sentence %d not found (File has %d sentences).\n", sentence_num,
    continue;
}
}

// --- Normal Case: Find the word ---
if (!find_word_pointers(sent_start, sent_end, word_idx, &word_start, &word_end)) {

    // --- FIX: Special Case for appending to a sentence ---
    // (word_idx is the *next* word index)
    int word_count = 0;
    char* temp_start = sent_start;
    while(temp_start <= sent_end) {
        while (*temp_start == ' ' && temp_start <= sent_end) temp_start++;
        if (temp_start > sent_end) break;
        char* temp_end = strchr(temp_start, ' ');
        if (temp_end == NULL || temp_end > sent_end) {
            word_count++; // Last word
            break;
        }
        word_count++;
        temp_start = temp_end + 1;
    }

    if (word_idx == word_count) {
        printf("[SS-Client-Write] Appending to sentence %d (word %d).\n", sentence_num, word_idx);

        int old_len = strlen(file_content_buffer);
        int new_content_len = strlen(new_content) + 1; // +1 for the space
        char* new_buffer = (char*)malloc(old_len + new_content_len + 1); // +1 for null

        if (new_buffer == NULL) {
            perror("malloc");

```

```

        continue;
    }

    // --- NEW SIMPLER LOGIC ---

    // Part 1: Copy up to (and INCLUDING) the last char of the sentence
    // (This is the char at sent_end)
    int part1_len = (sent_end - file_content_buffer) + 1;
    memcpy(new_buffer, file_content_buffer, part1_len);

    // Part 2: Add a space and the new word
    new_buffer[part1_len] = ' ';
    memcpy(new_buffer + part1_len + 1, new_content, new_content_len - 1);

    // Part 3: Copy the rest of the file (from AFTER sent_end)
    int part3_start_offset = part1_len;
    int part3_len = old_len - part3_start_offset;
    memcpy(new_buffer + part1_len + new_content_len, file_content_buffer + part3_start_offset, part3_len);

    // Null-terminate the new string
    new_buffer[old_len + new_content_len] = '\0';

    // --- END NEW LOGIC ---

    free(file_content_buffer);
    file_content_buffer = new_buffer;

    printf("[SS-Client-Write] Append successful.\n");

    // Recompute sentence pointers after buffer reallocation
    if (!find_sentence_pointers(file_content_buffer, sentence_num, &sent_start, &sent_end)) {
        printf("[SS-Client-Write] Warning: Could not recompute sentence pointers after append.\n");
    }

    continue; // Done
} else {
    printf("[SS-Client-Write] Error: Word %d not found (count %d) in sentence %d.\n", word_idx, word_count, sentence_num);
    continue; // Ignore this update
}
}

// --- Normal Case: Inserting a word mid-sentence ---
printf("[SS-Client-Write] Inserting at sen %d, word %d.\n", sentence_num, word_idx);

int old_len = strlen(file_content_buffer);
// We are inserting the new content AND a space
int new_content_len = strlen(new_content) + 1; // +1 for the space
int new_len = old_len + new_content_len;

char* new_buffer = (char*)malloc(new_len + 1);
if (new_buffer == NULL) {

```

```

        perror("malloc");
        continue;
    }

    // Part 1: Copy everything *before* the insertion point (word_start)
    int part1_len = word_start - file_content_buffer;
    memcpy(new_buffer, file_content_buffer, part1_len);

    // Part 2: Copy the *new word* and add a space
    memcpy(new_buffer + part1_len, new_content, new_content_len - 1);
    new_buffer[part1_len + new_content_len - 1] = ' '; // Add the space

    // Part 3: Copy everything *from* the insertion point to the end
    int part3_start_offset = part1_len; // This is the start of the old word
    int part3_len = old_len - part3_start_offset;
    memcpy(new_buffer + part1_len + new_content_len, file_content_buffer + part3_start_offset, part3_len);

    new_buffer[new_len] = '\0'; // Null-terminate

    free(file_content_buffer);
    file_content_buffer = new_buffer;

    printf("[SS-Client-Write] Update successful for sen %d, word %d.\n", sentence_num, word_idx);

    // Recompute sentence pointers after buffer reallocation
    if (!find_sentence_pointers(file_content_buffer, sentence_num, &sent_start, &sent_end)) {
        printf("[SS-Client-Write] Warning: Could not recompute sentence pointers after update.\n");
    }

}

else if (strncmp(cmd, SS_COMMIT, strlen(SS_COMMIT)) == 0) {
    snprintf(log_msg, MAX_MSG_LEN, "REQ: SS_COMMIT, FILE: %s", filename);
    log_event(log_msg); // We'll fix the "UNKNOWN" user later

    // --- CONFLICT CHECK & MERGE LOGIC ---
    pthread_mutex_lock(&g_commit_mutex); // Lock the global commit mutex

    struct stat file_stat;
    time_t current_mod_time = 0;
    char bak_path[MAX_PATH_LEN + 5]; // Define bak_path here
    snprintf(bak_path, MAX_PATH_LEN + 5, "%s.bak", local_path);

    if (stat(local_path, &file_stat) == 0) {
        current_mod_time = file_stat.st_mtime;
    } else if (errno != ENOENT) {
        perror("stat (commit)"); // Real stat error
    }
    // Note: if errno == ENOENT, current_mod_time remains 0. This is fine.

    if (current_mod_time == original_mod_time) {
        // --- PATH 1: NO CONFLICT (We are the first to commit) ---
        printf("[SS-Client-Write] COMMIT received for %s (No conflict).\n", filename);
    }
}

```

```

rename(local_path, bak_path); // Create backup

FILE *fp = fopen(local_path, "w");
if (fp) {
    fwrite(file_content_buffer, 1, strlen(file_content_buffer), fp);
    fclose(fp);
    send(client_fd, RESP_OK "\n", strlen(RESP_OK "\n"), 0);

    // Send metadata update (using my_port from Task 2)
    int wc = 0, cc = 0;
    count_metadata(file_content_buffer, &wc, &cc);
    char update_msg[MAX_MSG_LEN];
    snprintf(update_msg, MAX_MSG_LEN, "%s %d %s %d %d\n", S_META_UPDATE, my_port, filename,
    send_async_update_to_nm(update_msg);

} else {
    perror("fopen (commit)");
    send(client_fd, RESP_SRV_ERR "\n", strlen(RESP_SRV_ERR "\n"), 0);
}

} else {
    // --- PATH 2: CONFLICT (We are the second to commit) ---
    printf("[SS-Client-Write] COMMIT for %s (Conflict detected, attempting merge).\n", filename);

    char* my_changes_buffer = file_content_buffer; // Already in memory
    char* current_file_buffer = read_file_to_memory(local_path); // User A's changes
    char* original_file_buffer = read_file_to_memory(bak_path); // Common ancestor

    if (current_file_buffer == NULL) {
        send(client_fd, RESP_SRV_ERR "\n", strlen(RESP_SRV_ERR "\n"), 0);
        if(original_file_buffer) free(original_file_buffer);
        pthread_mutex_unlock(&g_commit_mutex);
        break; // Break the while loop
    }

    if (original_file_buffer == NULL) {
        printf("[SS-Client-Write] MERGE FAILED for %s (No common ancestor).\n", filename);
        send(client_fd, RESP_CONFLICT "\n", strlen(RESP_CONFLICT "\n"), 0);
        free(current_file_buffer);
        pthread_mutex_unlock(&g_commit_mutex);
        break; // Break the while loop
    }

    // 2. Check for a *true* conflict (did they edit our sentence?)
    char *original_text_start, *original_text_end;
    char *their_text_start, *their_text_end;
    char *my_change_start, *my_change_end;

    bool original_found = find_sentence_pointers(original_file_buffer, sentence_num, &original_text_start);
    bool their_found = find_sentence_pointers(current_file_buffer, sentence_num, &their_text_start, &their_text_end);
    bool my_found = find_sentence_pointers(my_changes_buffer, sentence_num, &my_change_start, &my_change_end);
}

```

```

bool true_conflict = false;

if (original_found != their_found) {
    true_conflict = true; // One person deleted the sentence, the other didn't.
} else if (original_found && their_found) {
    // Both found it, let's compare.
    int original_len = (original_text_end - original_text_start) + 1;
    int their_len = (their_text_end - their_text_start) + 1;

    if (original_len != their_len || strncmp(original_text_start, their_text_start, original_len) != 0) {
        true_conflict = true; // The sentence we edited has been changed by someone else.
    }
}
else if (!original_found && their_found) {
    true_conflict = true; // They added the sentence we were trying to add.
}

if (true_conflict) {
    // 3. Unmergeable Conflict
    printf("[SS-Client-Write] MERGE FAILED for %s (True conflict on sen %d).\n", filename, sentence_r
    send(client_fd, RESP_CONFLICT "\n", strlen(RESP_CONFLICT "\n"), 0);
} else {
    // 4. Perform Merge
    printf("[SS-Client-Write] MERGE success for %s (Applying changes to sen %d).\n", filename, sente

char* new_merged_buffer;

if (!my_found) {
    // This should not happen (we edited nothing?), but safe to just keep their buffer.
    new_merged_buffer = strdup(current_file_buffer);
} else if (!original_found && !their_found) {
    // Clean append: We added a new sentence, they didn't.
    int my_change_len = (my_change_end - my_change_start) + 1;
    int current_len = strlen(current_file_buffer);

    // Check if current file needs a delimiter before our new sentence
    bool needs_delimiter = false;
    if (current_len > 0) {
        char last_char = current_file_buffer[current_len - 1];
        if (last_char != '.' && last_char != '!' && last_char != '?') {
            needs_delimiter = true;
        }
    }
}

int extra_chars = (current_len > 0 ? 1 : 0) + (needs_delimiter ? 1 : 0);
int new_buffer_size = current_len + extra_chars + my_change_len + 1;
new_merged_buffer = (char*)malloc(new_buffer_size);

if (current_len > 0) {
    strcpy(new_merged_buffer, current_file_buffer);
    if (needs_delimiter) {
        new_merged_buffer[current_len] = '.';
    }
}

```

```

        new_merged_buffer[current_len + 1] = ' ';
        memcpy(new_merged_buffer + current_len + 2, my_change_start, my_change_len);
        new_merged_buffer[current_len + 2 + my_change_len] = '\0';
    } else {
        new_merged_buffer[current_len] = ' ';
        memcpy(new_merged_buffer + current_len + 1, my_change_start, my_change_len);
        new_merged_buffer[current_len + 1 + my_change_len] = '\0';
    }
} else {
    memcpy(new_merged_buffer, my_change_start, my_change_len);
    new_merged_buffer[my_change_len] = '\0';
}

} else {
    // Clean replace: We edited a sentence, they didn't.
    int my_change_len = (my_change_end - my_change_start) + 1;
    int their_change_len = (their_text_end - their_text_start) + 1;

    int new_buffer_size = strlen(current_file_buffer) - their_change_len + my_change_len + 1;
    new_merged_buffer = (char*)malloc(new_buffer_size);

    int part1_len = (their_text_start - current_file_buffer);
    memcpy(new_merged_buffer, current_file_buffer, part1_len);

    memcpy(new_merged_buffer + part1_len, my_change_start, my_change_len);

    int part3_start_offset = (their_text_end - current_file_buffer) + 1;
    int part3_len = strlen(current_file_buffer) - part3_start_offset;
    memcpy(new_merged_buffer + part1_len + my_change_len, current_file_buffer + part3_start_offset, part3_len);

    new_merged_buffer[new_buffer_size - 1] = '\0';
}

// 5. Commit Merged File
rename(local_path, bak_path); // Save User A's version as the new backup
FILE *fp = fopen(local_path, "w");
if (fp) {
    fwrite(new_merged_buffer, 1, strlen(new_merged_buffer), fp);
    fclose(fp);
    send(client_fd, RESP_OK "\n", strlen(RESP_OK "\n"), 0);

    // Send metadata update (using my_port from Step 2)
    int wc = 0, cc = 0;
    count_metadata(new_merged_buffer, &wc, &cc);
    char update_msg[MAX_MSG_LEN];
    snprintf(update_msg, MAX_MSG_LEN, "%s %d %s %d %d\n", S_META_UPDATE, my_port, filename);
    send_async_update_to_nm(update_msg);
}

} else {
    send(client_fd, RESP_SRV_ERR "\n", strlen(RESP_SRV_ERR "\n"), 0);
}
free(new_merged_buffer);

```

```

    }

    // Cleanup
    free(current_file_buffer);
    free(original_file_buffer);
}

pthread_mutex_unlock(&g_commit_mutex); // Unlock the global commit mutex
free(file_content_buffer); // Free our in-memory changes
break; // Exit transaction loop
}
}

} // end while(1)

// 7. RELEASE LOCK (after loop)
pthread_mutex_lock(&g_lock_list_mutex);
for (int i = 0; i < g_lock_count; i++) {
    if (strcmp(g_lock_list[i].path, filename) == 0 &&
        g_lock_list[i].sentence_num == sentence_num &&
        g_lock_list[i].is_locked) {
        // "Swap with last" to delete from list
        g_lock_list[i] = g_lock_list[g_lock_count - 1];
        g_lock_count--;
        printf("[SS-Client-Write] Lock RELEASED for %s, sen %d\n", filename, sentence_num);
        break;
    }
}
pthread_mutex_unlock(&g_lock_list_mutex);
} // end if (file_content_buffer != NULL)
} // end else (lock not already locked)
} // end SS_LOCK handler
else if (strncmp(cmd, SS_GET_STREAM, strlen(SS_GET_STREAM)) == 0) {
    printf("[SS-Client] Received STREAM request for: %s\n", filename);
    snprintf(log_msg, MAX_MSG_LEN, "REQ: SS_GET_STREAM, FILE: %s", filename);
    log_event(log_msg);

    char local_path[MAX_PATH_LEN];
    snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);

    FILE *fp = fopen(local_path, "r");
    if (fp == NULL) {
        perror("[SS-Client] fopen failed for STREAM");
    } else {
        char word_buffer[256];
        // Read one WORD at a time (fscanf handles whitespace)
        while (fscanf(fp, "%255s", word_buffer) == 1) {
            // Send the word
            if (send(client_fd, word_buffer, strlen(word_buffer), 0) == -1) {
                break; // Client disconnected
            }
            // Send a space after the word
            if (send(client_fd, " ", 1, 0) == -1) {

```

```

        break; // Client disconnected
    }

    // The 0.1 second delay
    usleep(STREAM_DELAY_US); // From protocol.h
}
fclose(fp);
printf("[SS-Client] Stream complete for: %s\n", filename);
}

else if (strncmp(cmd, SS_VIEWCHECKPOINT, strlen(SS_VIEWCHECKPOINT)) == 0) {
    char tag[MAX_CHECKPOINT_TAG];
    sscanf(buffer, "%*s %s %s", filename, tag);

    sprintf(log_msg, MAX_MSG_LEN, "REQ: SS_VIEWCHECKPOINT, FILE: %s, TAG: %s", filename, tag);
    log_event(log_msg);

    char* content = read_checkpoint(filename, tag);
    if (content == NULL) {
        // Checkpoint or file not found
        send(client_fd, RESP_NOT_FOUND "\n", strlen(RESP_NOT_FOUND "\n"), 0);
    } else {
        // Send 200 OK, then the content
        send(client_fd, RESP_OK "\n", strlen(RESP_OK "\n"), 0);
        send(client_fd, content, strlen(content), 0);
        free(content);
    }
}

else if (strncmp(cmd, SS_LISTCHECKPOINTS, strlen(SS_LISTCHECKPOINTS)) == 0) {
    sscanf(buffer, "%*s %s", filename);

    sprintf(log_msg, MAX_MSG_LEN, "REQ: SS_LISTCHECKPOINTS, FILE: %s", filename);
    log_event(log_msg);

    char* list = list_checkpoints(filename);
    if (list == NULL) {
        // This indicates a server error (e.g., malloc failed)
        send(client_fd, RESP_SRV_ERR "\n", strlen(RESP_SRV_ERR "\n"), 0);
    } else {
        // Send 200 OK, then the list
        send(client_fd, RESP_OK "\n", strlen(RESP_OK "\n"), 0);
        send(client_fd, list, strlen(list), 0);
        free(list);
    }
}

else {
    // Unknown command
    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);
    log_event("Client listener thread started.");

    // 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;
ClientThreadArgs* args = malloc(sizeof(ClientThreadArgs));
args->client_fd = client_fd;
args->ss_port = port; // 'port' is the argument this function received

if (pthread_create(&client_tid, NULL, handle_client_request, (void *)args) != 0) {
    perror("SS failed to create client handler thread");
    free(args);
    close(client_fd);
}
}

// This part is never reached
close(listener_fd);
return NULL;
}

/*
 * =====
 * CHECKPOINT HELPER FUNCTIONS
 * =====
 */

/*
 * Creates a checkpoint by copying the current file to a checkpoint file
 * Checkpoint files are stored as: <filename>.checkpoint.<tag>
 * Returns true on success, false on failure
 */
bool create_checkpoint(const char* filename, const char* tag) {
    char source_path[MAX_PATH_LEN];
    char checkpoint_path[MAX_PATH_LEN];

    snprintf(source_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
    snprintf(checkpoint_path, MAX_PATH_LEN, "%s/%s.checkpoint.%s",
            g_ss_root_path, filename, tag);

    // Check if source file exists
    FILE* source = fopen(source_path, "r");
    if (source == NULL) {
        perror("create_checkpoint: source file not found");
        return false;
    }
}

```

```

// Create checkpoint file
FILE* checkpoint = fopen(checkpoint_path, "w");
if (checkpoint == NULL) {
    perror("create_checkpoint: failed to create checkpoint");
    fclose(source);
    return false;
}

// Copy content
char buffer[4096];
size_t bytes;
while ((bytes = fread(buffer, 1, sizeof(buffer), source)) > 0) {
    if (fwrite(buffer, 1, bytes, checkpoint) != bytes) {
        perror("create_checkpoint: write failed");
        fclose(source);
        fclose(checkpoint);
        remove(checkpoint_path);
        return false;
    }
}

fclose(source);
fclose(checkpoint);

// Add to checkpoint list
pthread_mutex_lock(&g_checkpoint_mutex);

// Check if checkpoint with this tag already exists for this file
for (int i = 0; i < g_checkpoint_count; i++) {
    if (strcmp(g_checkpoint_list[i].filename, filename) == 0 &&
        strcmp(g_checkpoint_list[i].tag, tag) == 0) {
        // Update existing checkpoint
        g_checkpoint_list[i].created_at = time(NULL);
        strcpy(g_checkpoint_list[i].checkpoint_path, checkpoint_path);
        pthread_mutex_unlock(&g_checkpoint_mutex);
        return true;
    }
}

// Add new checkpoint
if (g_checkpoint_count < MAX_CHECKPOINTS) {
    strcpy(g_checkpoint_list[g_checkpoint_count].tag, tag);
    strcpy(g_checkpoint_list[g_checkpoint_count].filename, filename);
    strcpy(g_checkpoint_list[g_checkpoint_count].checkpoint_path, checkpoint_path);
    g_checkpoint_list[g_checkpoint_count].created_at = time(NULL);
    g_checkpoint_count++;
} else {
    pthread_mutex_unlock(&g_checkpoint_mutex);
    remove(checkpoint_path);
    return false;
}

```

```

pthread_mutex_unlock(&g_checkpoint_mutex);
return true;
}

/*
 * Reads and returns the content of a checkpoint
 * Caller must free() the returned string
 */
char* read_checkpoint(const char* filename, const char* tag) {
    pthread_mutex_lock(&g_checkpoint_mutex);

    char checkpoint_path[MAX_PATH_LEN] = "";
    bool found = false;

    for (int i = 0; i < g_checkpoint_count; i++) {
        if (strcmp(g_checkpoint_list[i].filename, filename) == 0 &&
            strcmp(g_checkpoint_list[i].tag, tag) == 0) {
            strcpy(checkpoint_path, g_checkpoint_list[i].checkpoint_path);
            found = true;
            break;
        }
    }

    pthread_mutex_unlock(&g_checkpoint_mutex);

    if (!found) {
        return NULL;
    }

    return read_file_to_memory(checkpoint_path);
}

/*
 * Reverts a file to a checkpoint by copying the checkpoint file
 * back to the original file location
 */
bool revert_to_checkpoint(const char* filename, const char* tag) {
    char current_path[MAX_PATH_LEN];
    char checkpoint_path[MAX_PATH_LEN];

    snprintf(current_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);

    pthread_mutex_lock(&g_checkpoint_mutex);

    bool found = false;
    for (int i = 0; i < g_checkpoint_count; i++) {
        if (strcmp(g_checkpoint_list[i].filename, filename) == 0 &&
            strcmp(g_checkpoint_list[i].tag, tag) == 0) {
            strcpy(checkpoint_path, g_checkpoint_list[i].checkpoint_path);
            found = true;
            break;
        }
    }
}

```

```
}

pthread_mutex_unlock(&g_checkpoint_mutex);

if (!found) {
    return false;
}

// Check if file is locked
pthread_mutex_lock(&g_lock_list_mutex);
bool is_locked = false;
for (int i = 0; i < g_lock_count; i++) {
    if (strcmp(g_lock_list[i].path, filename) == 0 && g_lock_list[i].is_locked) {
        is_locked = true;
        break;
    }
}
pthread_mutex_unlock(&g_lock_list_mutex);

if (is_locked) {
    return false; // Can't revert while file is locked
}

// Copy checkpoint to current file
FILE* checkpoint = fopen(checkpoint_path, "r");
if (checkpoint == NULL) {
    return false;
}

FILE* current = fopen(current_path, "w");
if (current == NULL) {
    fclose(checkpoint);
    return false;
}

char buffer[4096];
size_t bytes;
while ((bytes = fread(buffer, 1, sizeof(buffer), checkpoint)) > 0) {
    fwrite(buffer, 1, bytes, current);
}

fclose(checkpoint);
fclose(current);

return true;
}

/*
 * Lists all checkpoints for a given file
 * Returns a dynamically allocated string containing the list
 * Caller must free() the returned string
 */
```

```

char* list_checkpoints(const char* filename) {
    pthread_mutex_lock(&g_checkpoint_mutex);

    // Allocate a large buffer for the list
    char* list = (char*)malloc(4096);
    if (list == NULL) {
        pthread_mutex_unlock(&g_checkpoint_mutex);
        return NULL;
    }

    strcpy(list, "");
    int count = 0;

    for (int i = 0; i < g_checkpoint_count; i++) {
        if (strcmp(g_checkpoint_list[i].filename, filename) == 0) {
            char time_str[100];
            struct tm ltime;
            localtime_r(&g_checkpoint_list[i].created_at, &ltime);
            strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", &ltime);

            char line[256];
            snprintf(line, sizeof(line), "%s | %s\n",
                     g_checkpoint_list[i].tag, time_str);
            strcat(list, line);
            count++;
        }
    }

    pthread_mutex_unlock(&g_checkpoint_mutex);

    if (count == 0) {
        strcpy(list, "No checkpoints found.\n");
    }

    return list;
}

int main(int argc, char* argv[]) {

    g_log_fp = fopen(SS_LOG_FILE, "a"); // "a" for append
    if (g_log_fp == NULL) {
        perror("FATAL: Failed to open log file");
        exit(1);
    }
    log_event("---- Storage Server Started ----");

    // 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]);

snprintf(g_ss_root_path, sizeof(g_ss_root_path), "ss_data/%d", client_port_for_ss);

// 2. Create parent directory "ss_data"
// We check for -1 AND if the error is *not* "File Exists"
if (mkdir("ss_data", 0777) == -1 && errno != EEXIST) {
    perror("FATAL: Failed to create 'ss_data' directory");
    exit(1);
}

// 3. Create the specific server directory (e.g., "ss_data/9002")
if (mkdir(g_ss_root_path, 0777) == -1 && errno != EEXIST) {
    perror("FATAL: Failed to create server-specific directory");
    exit(1);
}

printf("[SS] Storage root path set to: %s\n", g_ss_root_path);

// --- 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");
log_event("Registered with Name Server");

// --- 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 ---
char nm_buffer[MAX_MSG_LEN];
char cmd[MAX_MSG_LEN];
char filename[MAX_PATH_LEN - sizeof(g_ss_root_path) - 1];
char local_path[MAX_PATH_LEN];
char resp_buf[MAX_MSG_LEN]; // Declare all buffers outside the loop

while(1) {
    // Clear all buffers for this new command
    char log_msg[MAX_MSG_LEN];
    memset(nm_buffer, 0, MAX_MSG_LEN);

```

```

        memset(cmd, 0, MAX_MSG_LEN);
        memset(filename, 0, sizeof(filename));
        memset(local_path, 0, MAX_PATH_LEN);
        memset(resp_buf, 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);

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

        if (!is_valid_filename(filename)) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            write(g_nm_fd, resp_buf, strlen(resp_buf));
            continue; // Skip to next command
        }

        if (strncmp(cmd, NM_CREATE, strlen(NM_CREATE)) == 0) {
            snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_CREATE, FILE: %s", filename); // <-- ADD
            log_event(log_msg);
            printf("[SS-NM Loop] Received NM_CREATE for: %s\n", filename);

            // --- ADD THIS LINE ---
            ensure_directory_exists(filename);
            // -----
            snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);

            FILE *fp = fopen(local_path, "w");
            if (fp == NULL) {
                perror("fopen failed");
                snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
            } else {
                fclose(fp);
                snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);

                // printf("[SS-NM Loop] SIMULATING SLOW CREATE...\n");
                // sleep(5); // Simulate a 5-second disk write

                printf("[SS-NM Loop] Successfully created file: %s\n", filename);
            }
        }
    else if (strncmp(cmd, NM_DELETE, strlen(NM_DELETE)) == 0) {
        snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_DELETE, FILE: %s", filename);
        log_event(log_msg);
        printf("[SS-NM Loop] Received NM_DELETE for: %s\n", filename);
        snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
    }
}

```

```

pthread_mutex_lock(&g_lock_list_mutex);
bool is_file_locked = false;
for (int i = 0; i < g_lock_count; i++) {
    if (strcmp(g_lock_list[i].path, filename) == 0 && g_lock_list[i].is_locked) {
        is_file_locked = true;
        break;
    }
}
pthread_mutex_unlock(&g_lock_list_mutex);

if (is_file_locked) {
    printf("[SS-NM Loop] DELETE failed for %s (file is locked by a user).\n", filename);
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_LOCKED_ERR);
} else if (remove(local_path) == 0) {
    printf("[SS-NM Loop] Successfully deleted file: %s\n", filename);
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
} else {
    perror("[SS-NM Loop] remove() failed");
    snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
}
}

else if (strncmp(cmd, NM_UNDO, strlen(NM_UNDO)) == 0) {
    snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_UNDO, FILE: %s", filename);
    log_event(log_msg);
    printf("[SS-NM Loop] Received NM_UNDO for: %s\n", filename);
    snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
    char bak_path[MAX_PATH_LEN + 5];
    snprintf(bak_path, MAX_PATH_LEN+5, "%s.bak", local_path);

    pthread_mutex_lock(&g_lock_list_mutex);
    bool is_file_locked = false;
    for (int i = 0; i < g_lock_count; i++) {
        // If *any* sentence is locked for this file, block the undo.
        if (strcmp(g_lock_list[i].path, filename) == 0 && g_lock_list[i].is_locked) {
            is_file_locked = true;
            break;
        }
    }
    pthread_mutex_unlock(&g_lock_list_mutex);

    if (is_file_locked) {
        printf("[SS-NM Loop] UNDO failed for %s (file is locked by a user).\n", filename);
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_LOCKED_ERR);
    } else {

        // This is the core logic: swap the .bak file back to the original
        if (rename(bak_path, local_path) == 0) {
            printf("[SS-NM Loop] Successfully reverted file: %s\n", filename);
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
        } else {
            perror("[SS-NM Loop] rename(undo) failed");
        }
    }
}

```

```

// This happens if there's no .bak file
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
}
}
}

else if (strncmp(cmd, NM_CHECKPOINT, strlen(NM_CHECKPOINT)) == 0) {
char tag[MAX_CHECKPOINT_TAG];
sscanf(nm_buffer, "%*s %s %s", filename, tag);

snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_CHECKPOINT, FILE: %s, TAG: %s", filename, tag);
log_event(log_msg);
printf("[SS-NM Loop] Received NM_CHECKPOINT for: %s (tag: %s)\n", filename, tag);

if (!is_valid_filename(filename)) {
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
write(g_nm_fd, resp_buf, strlen(resp_buf));
continue;
}

snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);

// Check if file exists
if (access(local_path, F_OK) != 0) {
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_NOT_FOUND);
} else if (create_checkpoint(filename, tag)) {
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
printf("[SS-NM Loop] Checkpoint created: %s (tag: %s)\n", filename, tag);
} else {
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
}

write(g_nm_fd, resp_buf, strlen(resp_buf));
continue;
}

else if (strncmp(cmd, NM_REVERT, strlen(NM_REVERT)) == 0) {
char tag[MAX_CHECKPOINT_TAG];
sscanf(nm_buffer, "%*s %s %s", filename, tag);

snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_REVERT, FILE: %s, TAG: %s", filename, tag);
log_event(log_msg);
printf("[SS-NM Loop] Received NM_REVERT for: %s (tag: %s)\n", filename, tag);

if (!is_valid_filename(filename)) {
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
write(g_nm_fd, resp_buf, strlen(resp_buf));
continue;
}

if (revert_to_checkpoint(filename, tag)) {
snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
printf("[SS-NM Loop] File reverted: %s (tag: %s)\n", filename, tag);
} else {

```

```

        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_LOCKED_ERR);
    }

    write(g_nm_fd, resp_buf, strlen(resp_buf));
    continue;
}

else if (strncmp(cmd, NM_RENAME, strlen(NM_RENAME)) == 0) {
    char new_path[MAX_PATH_LEN];
    sscanf(nm_buffer, "%*s %s %s", filename, new_path);

    snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_RENAME, OLD: %s, NEW: %s", filename, new_path);
    log_event(log_msg);

    char old_full_path[MAX_PATH_LEN];
    char new_full_path[MAX_PATH_LEN];
    snprintf(old_full_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
    snprintf(new_full_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, new_path);

    // Ensure destination folder exists before moving
    ensure_directory_exists(new_path);

    if (rename(old_full_path, new_full_path) == 0) {
        printf("Renamed %s to %s\n", filename, new_path);
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK);
    } else {
        perror("rename failed");
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR);
    }

    write(g_nm_fd, resp_buf, strlen(resp_buf)); // Send ACK
    continue;
}

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

// Send the single, formatted response
write(g_nm_fd, resp_buf, strlen(resp_buf));
}

close(g_nm_fd);
return 0;
}

client.c
#include "protocol.h"
#include <stdio.h> // For printf, fgets
#include <stdlib.h> // For exit, EXIT_FAILURE
#include <string.h> // For strlen, strtok, memset
#include <unistd.h> // For close
#include <arpa/inet.h> // For sockaddr_in, inet_pton

```

```

#include <sys/socket.h> // For socket, connect
#include <stdbool.h>

#define NM_IP "127.0.0.1" // IP for the Name Server (NM)

void print_error(const char *code)
{
if (!code)
return;
if (!strncmp(code, "400", 3))
printf("Error: Bad request. Check your syntax.\n");
else if (!strncmp(code, "403", 3))
printf("Error: Access denied.\n");
else if (!strncmp(code, "404", 3))
printf("Error: File not found.\n");
else if (!strncmp(code, "409", 3))
printf("Error: Conflict. File was updated by another user. Your changes were not saved.\n");
else if (!strncmp(code, "503", 3))
printf("Error: Storage server unavailable.\n");
else if (!strncmp(code, "504", 3))
printf("Error: Sentence locked by another user.\n");
else
printf("Unexpected response: %s\n", code);
}

int main()
{
int sockfd;
struct sockaddr_in nm_addr;
char username[100];
char init_msg[120];

// Prompt the user to enter their username
printf("Enter username: ");
fgets(username, sizeof(username), stdin);
// Remove trailing newline character from username, if present
size_t len = strlen(username);
if (len > 0 && username[len - 1] == '\n')
{
username[len - 1] = '\0';
}

// Create a TCP socket for IPv4 communication
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
perror("Failed to create socket");
exit(EXIT_FAILURE);
}

// Set up the server address structure for the Name Server
memset(&nm_addr, 0, sizeof(nm_addr));

```

```

nm_addr.sin_family = AF_INET;
nm_addr.sin_port = htons(NM_LISTEN_PORT);
if (inet_pton(AF_INET, NM_IP, &nm_addr.sin_addr) <= 0)
{
    perror("Invalid address for Name Server");
    exit(EXIT_FAILURE);
}

// Connect to the Name Server at NM_IP:NM_LISTEN_PORT
if (connect(sockfd, (struct sockaddr *)&nm_addr, sizeof(nm_addr)) < 0)
{
    perror("Connection to Name Server failed");
    exit(EXIT_FAILURE);
}

// Format and send the INIT_CLIENT <username> message to register this client
snprintf(init_msg, sizeof(init_msg), "%s %s\n", C_INIT, username);
if (send(sockfd, init_msg, strlen(init_msg), 0) < 0)
{
    perror("Failed to send INIT_CLIENT message");
    close(sockfd);
    exit(EXIT_FAILURE);
}
printf("Registered with Name Server as '%s'.\n", username);

char response_buf[MAX_MSG_LEN];
memset(response_buf, 0, MAX_MSG_LEN);
int n = recv(sockfd, response_buf, MAX_MSG_LEN - 1, 0);
if (n <= 0)
{
    perror("Failed to receive registration ACK from NM");
    close(sockfd);
    exit(EXIT_FAILURE);
}
response_buf[n] = '\0';
if (strncmp(response_buf, RESP_OK, 3) != 0)
{
    printf("Error: Name Server rejected registration: %s", response_buf);
    close(sockfd);
    exit(EXIT_FAILURE);
}

// Command input loop for user commands (CREATE, READ, EXIT)
char cmdline[256]; // Buffer for user input
char request[256]; // Buffer for request sent to NM
char response[256]; // Buffer for response from NM

while (1)
{
    printf("> ");
    if (fgets(cmdline, sizeof(cmdline), stdin) == NULL)
    {

```

```

printf("Input error or EOF. Exiting.\n");
break;
}
size_t clen = strlen(cmdline);
if (clen > 0 && cmdline[clen - 1] == '\n')
{
cmdline[clen - 1] = '\0';
}
char *cmd = strtok(cmdline, " ");
if (!cmd)
continue;

// ----- CREATE command (already implemented) -----
if (strcmp(cmd, "CREATE") == 0)
{
char *filename = strtok(NULL, " ");
if (!filename)
{
printf("Error: No filename given.\n");
continue;
}
snprintf(request, sizeof(request), "%s %s\n", C_REQ_CREATE, filename);
if (send(sockfd, request, strlen(request), 0) < 0)
{
perror("Failed to send REQ_CREATE");
continue;
}
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n < 0)
{
perror("Failed to receive NM response");
continue;
}
response[n] = '\0';
if (strncmp(response, "200", 3) == 0)
{
printf("File created successfully.\n");
}
else if (strncmp(response, "409", 3) == 0)
{
printf("Error: File already exists.\n");
}
else if (strncmp(response, "400", 3) == 0)
{
printf("Error: Bad request.\n");
}
else
{
printf("Unexpected response: %s\n", response);
}
}

// ----- READ command -----

```

```

else if (strcmp(cmd, "READ") == 0) {
    char *filename = strtok(NULL, " ");
    if (!filename) {
        printf("Error: No filename given.\n");
        continue;
    }
    // Step 1: Request file location from NM
    snprintf(request, sizeof(request), "%s %s\n", C_REQ_READ, filename);
    if (send(sockfd, request, strlen(request), 0) < 0) {
        perror("Failed to send REQ_READ");
        continue;
    }
    int n = recv(sockfd, response, sizeof(response)-1, 0);
    if (n < 0) {
        perror("Failed to receive NM response");
        continue;
    }
    response[n] = '\0';

    char ip[100], port_str[20];
    // Step 2: Handle NM's response
    // --- THIS IS THE NEW, CLEANER LOGIC ---
    if (strncmp(response, RESP_SS_INFO, 3) == 0) {
        // Success! Parse the SS info
        if (sscanf(response, "%*s %s %s", ip, port_str) != 2) {
            printf("Error: Invalid NM response: %s\n", response);
            continue;
        }
        int ss_port = atoi(port_str);

        // Step 3: Connect to Storage Server
        int ss_sock = socket(AF_INET, SOCK_STREAM, 0);
        if (ss_sock < 0) {
            perror("Failed to create socket to Storage Server");
            continue;
        }
        struct sockaddr_in ss_addr;
        memset(&ss_addr, 0, sizeof(ss_addr));
        ss_addr.sin_family = AF_INET;
        ss_addr.sin_port = htons(ss_port);
        if (inet_pton(AF_INET, ip, &ss_addr.sin_addr) <= 0) {
            perror("Invalid Storage Server address");
            close(ss_sock);
            continue;
        }
        if (connect(ss_sock, (struct sockaddr*)&ss_addr, sizeof(ss_addr)) < 0) {
            perror("Connection to Storage Server failed");
            close(ss_sock);
            continue;
        }
        // Step 4: Send GET_FILE <filename> to SS
    }
}

```

```

snprintf(request, sizeof(request), "%s %s\n", SS_GET_FILE, filename);
if (send(ss_sock, request, strlen(request), 0) < 0) {
    perror("Failed to send GET_FILE");
    close(ss_sock);
    continue;
}

// Step 5: Receive and print file content from SS
printf("===== File: %s =====\n", filename);
// --- THIS FIXES THE 'xx' BUG ---
memset(response, 0, sizeof(response));
while (1) {
    int bytes_recv = recv(ss_sock, response, sizeof(response)-1, 0);
    if (bytes_recv <= 0) break; // 0 = closed, <0 = error
    response[bytes_recv] = '\0';
    printf("%s", response); // Print as received
}
printf("\n===== End of file =====\n");
close(ss_sock);
} else {
// --- THIS FIXES THE '403' BUG ---
print_error(response);
}

// ----- ADDACCESS -----
else if (strcmp(cmd, "ADDACCESS") == 0)
{
char *filename = strtok(NULL, " ");
char *user = strtok(NULL, " ");
char *perm = strtok(NULL, " ");
if (!filename || !user || !perm)
{
printf("Usage: ADDACCESS <filename> <username> <-R|-W>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s %s %s\n",
C_REQ_ADD_ACC, filename, user, perm);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[0] = '\0';
if (!strncmp(response, "200", 3))
printf("Access granted successfully.\n");
else if (!strncmp(response, "403", 3))
printf("Error: You are not the owner.\n");
else
printf("Unexpected response: %s\n", response);
}

```

```

// ----- REMACCESS -----
else if (strcmp(cmd, "REMACCESS") == 0)
{
    char *filename = strtok(NULL, " ");
    char *user = strtok(NULL, " ");
    if (!filename || !user)
    {
        printf("Usage: REMACCESS <filename> <username>\n");
        continue;
    }
    snprintf(request, sizeof(request), "%s %s %s\n",
C_REQ_Rem_ACC, filename, user);
    send(sockfd, request, strlen(request), 0);
    int n = recv(sockfd, response, sizeof(response) - 1, 0);
    if (n <= 0)
    {
        perror("recv");
        continue;
    }
    response[n] = '\0';
    if (!strncmp(response, "200", 3))
        printf("Access removed successfully.\n");
    else if (!strncmp(response, "403", 3))
        printf("Error: You are not the owner.\n");
    else
        printf("Unexpected response: %s\n", response);
    }

// ----- DELETE -----
else if (strcmp(cmd, "DELETE") == 0)
{
    char *filename = strtok(NULL, " ");
    if (!filename)
    {
        printf("Usage: DELETE <filename>\n");
        continue;
    }
    snprintf(request, sizeof(request), "%s %s\n", C_REQ_DELETE, filename);
    send(sockfd, request, strlen(request), 0);
    int n = recv(sockfd, response, sizeof(response) - 1, 0);
    if (n <= 0)
    {
        perror("recv");
        continue;
    }
    response[n] = '\0';
    if (!strncmp(response, "200", 3))
        printf("File deleted successfully.\n");
    else if (!strncmp(response, "403", 3))
        printf("Error: You are not the owner.\n");
    else if (!strncmp(response, "404", 3))

```

```
printf("Error: File not found.\n");
else
printf("Unexpected response: %s\n", response);
}

// ----- LIST -----
else if (strcmp(cmd, "LIST") == 0)
{
snprintf(request, sizeof(request), "%s\n", C_REQ_LIST);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
char *payload = strchr(response, '\n'); // find first newline
if (payload)
printf("%s\n", payload + 1); // print after the "200\n"
}

// ----- VIEW -----
else if (strcmp(cmd, "VIEW") == 0)
{
char *flags = strtok(NULL, " ");
if (!flags)
flags = "-";
snprintf(request, sizeof(request), "%s %s\n", C_REQ_VIEW, flags);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
printf("==== Files Visible to You ===\n%s\n", response + 4);
}

// ----- INFO -----
else if (strcmp(cmd, "INFO") == 0)
{
char *filename = strtok(NULL, " ");
if (!filename)
{
printf("Usage: INFO <filename>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s\n", C_REQ_INFO, filename);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
}
```

```

if (n <= 0)
{
    perror("recv");
    continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
printf("==== File Info ====\n%s\n", response + 4);
else if (!strncmp(response, "403", 3))
printf("Access denied.\n");
else if (!strncmp(response, "404", 3))
printf("File not found.\n");
else
printf("Unexpected response: %s\n", response);
}

// ----- WRITE -----
else if (strcmp(cmd, "WRITE") == 0)
{
char *filename = strtok(NULL, " ");
char *sentence_str = strtok(NULL, " ");
if (!filename || !sentence_str)
{
printf("Usage: WRITE <filename> <sentence_num>\n");
continue;
}
int sentence_num = atoi(sentence_str);

// Step 1: Ask NM for SS info
snprintf(request, sizeof(request), "%s %s\n", C_REQ_WRITE, filename);
if (send(sockfd, request, strlen(request), 0) < 0)
{
    perror("send");
    continue;
}

int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
    perror("recv");
    continue;
}
response[n] = '\0';

if (strncmp(response, RESP_SS_INFO, 3) != 0)
{
    // Check if it's our detailed 404 error
    if (strncmp(response, RESP_NOT_FOUND, 3) == 0) {
        // Print the WHOLE message from the server (e.g. "404 Sentence index 5...")
        // We skip the first 4 chars ("404 ")
        printf("Error: %s", response + 4);
    } else {
}
}
}

```

```

// Otherwise, use the generic printer
print_error(response);
}

continue;
}

// Parse SS IP + port
char ip[64], port_str[16];
if (sscanf(response, "%*s %s %s", ip, port_str) != 2)
{
printf("Error: Invalid SS info.\n");
continue;
}
int ss_port = atoi(port_str);

// Step 2: Connect to SS
int ss_sock = socket(AF_INET, SOCK_STREAM, 0);
if (ss_sock < 0)
{
perror("socket");
continue;
}

struct sockaddr_in ss_addr;
memset(&ss_addr, 0, sizeof(ss_addr));
ss_addr.sin_family = AF_INET;
ss_addr.sin_port = htons(ss_port);
inet_pton(AF_INET, ip, &ss_addr.sin_addr);

if (connect(ss_sock, (struct sockaddr *)&ss_addr, sizeof(ss_addr)) < 0)
{
perror("connect to SS");
close(ss_sock);
continue;
}

// Step 3: Request lock
snprintf(request, sizeof(request), "%s %s %d\n", SS_LOCK, filename, sentence_num);
send(ss_sock, request, strlen(request), 0);

n = recv(ss_sock, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
close(ss_sock);
continue;
}
response[n] = '\0';

if (!strcmp(response, RESP_LOCKED_ERR, 3))
{
printf("Error: Sentence is locked by another user.\n");
}

```

```

close(ss_sock);
continue;
}
else if (!strcmp(response, RESP_LOCKED, 3))
{
printf("Sentence locked. Enter updates (e.g., '3 new_word') or 'ETIRW' to finish:\n");
}
else
{
if (strcmp(response, RESP_NOT_FOUND, 3) == 0) {
// Print the WHOLE message from the server
// We skip the first 4 chars ("404 ") and the trailing newline
response[strcspn(response, "\n")] = 0; // Remove trailing newline
printf("Error: %s\n", response + 4);
} else {
// Otherwise, use the generic printer
print_error(response);
}
close(ss_sock);
continue;
}

// Step 4: Interactive update loop
char line[256];
while (1)
{
printf("update> ");
if (!fgets(line, sizeof(line), stdin))
break;
if (strcmp(line, "ETIRW\n") == 0)
{
snprintf(request, sizeof(request), "%s\n", SS_COMMIT);
send(ss_sock, request, strlen(request), 0);
break;
}
else
{
int word_idx;
char new_word[128];
if (sscanf(line, "%d %[^\n]", &word_idx, new_word) != 2) {
printf("Usage: <word_index> <content>\n");
continue;
}
snprintf(request, sizeof(request), "%s %d %s\n", SS_UPDATE, word_idx, new_word);
send(ss_sock, request, strlen(request), 0);
}
}

// Step 5: Wait for commit response
n = recv(ss_sock, response, sizeof(response) - 1, 0);
if (n <= 0)
{

```

```

    perror("recv");
    close(ss_sock);
    continue;
}
response[n] = '\0';

if (!strncmp(response, RESP_OK, 3))
printf("Write successful.\n");
else
print_error(response);

close(ss_sock);
}

// ----- UNDO -----
else if (strcmp(cmd, "UNDO") == 0)
{
char *filename = strtok(NULL, " ");
if (!filename)
{
printf("Usage: UNDO <filename>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s\n", C_REQ_UNDO, filename);
send(sockfd, request, strlen(request), 0);

int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';

if (!strncmp(response, RESP_OK, 3))
printf("Undo successful.\n");
else
print_error(response);
}

// ----- STREAM -----
else if (strcmp(cmd, "STREAM") == 0)
{
char *filename = strtok(NULL, " ");
if (!filename)
{
printf("Usage: STREAM <filename>\n");
continue;
}

snprintf(request, sizeof(request), "%s %s\n", C_REQ_STREAM, filename);
send(sockfd, request, strlen(request), 0);
}

```

```
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
    perror("recv");
    continue;
}
response[n] = '\0';

if (strncmp(response, RESP_SS_INFO, 3) != 0)
{
    print_error(response);
    continue;
}

char ip[64], port_str[16];
sscanf(response, "%*s %s %s", ip, port_str);
int ss_port = atoi(port_str);

int ss_sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in ss_addr;
memset(&ss_addr, 0, sizeof(ss_addr));
ss_addr.sin_family = AF_INET;
ss_addr.sin_port = htons(ss_port);
inet_pton(AF_INET, ip, &ss_addr.sin_addr);

if (connect(ss_sock, (struct sockaddr *)&ss_addr, sizeof(ss_addr)) < 0)
{
    perror("connect SS");
    close(ss_sock);
    continue;
}

snprintf(request, sizeof(request), "%s %s\n", SS_GET_STREAM, filename);
send(ss_sock, request, strlen(request), 0);

printf("Streaming %s...\n", filename);
while ((n = recv(ss_sock, response, sizeof(response) - 1, 0)) > 0)
{
    response[n] = '\0';
    printf("%s", response);
    fflush(stdout);
}
printf("\n--- End of stream ---\n");
close(ss_sock);
}

// ----- EXEC -----
else if (strcmp(cmd, "EXEC") == 0)
{
    char *filename = strtok(NULL, " ");
    if (!filename)
```

```

{
printf("Usage: EXEC <filename>\n");
continue;
}

snprintf(request, sizeof(request), "%s %s\n", C_REQ_EXEC, filename);
send(sockfd, request, strlen(request), 0);

// Do the FIRST read
n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';

// Check for 200 OK
if (strncmp(response, RESP_OK, 3) != 0)
{
print_error(response); // Handle errors like 404, 403
}
else
{
printf("Executing script remotely... Success:\n");

// --- START NEW ROBUST LOGIC ---
char* payload_start = response + 4; // Skip "200\n"
char* end_marker = NULL;
bool done = false;

// Check if the FIRST buffer already contains the end marker
end_marker = strstr(payload_start, RESP_EXEC_DONE);
if (end_marker) {
*end_marker = '\0'; // Truncate the buffer before the marker
done = true;
}
// Print whatever payload was in the first buffer
if (strlen(payload_start) > 0) {
printf("%s", payload_start);
fflush(stdout);
}

// If we didn't find the end marker, loop for the rest
if (!done) {
while ((n = recv(sockfd, response, sizeof(response) - 1, 0)) > 0)
{
response[n] = '\0';
// Check if this buffer has the end marker
end_marker = strstr(response, RESP_EXEC_DONE);
if (end_marker) {
*end_marker = '\0'; // Truncate
}
}
}
}

```

```

done = true;
}

// Print the (possibly truncated) buffer
if (strlen(response) > 0) {
printf("%s", response);
fflush(stdout);
}

if (done) {
break; // Exit the loop
}
}

printf("\n--- End of execution output ---\n");
// --- END NEW ROBUST LOGIC ---
}

// ----- CHECKPOINT -----
else if (strcmp(cmd, "CHECKPOINT") == 0)
{
char *filename = strtok(NULL, " ");
char *tag = strtok(NULL, " ");
if (!filename || !tag)
{
printf("Usage: CHECKPOINT <filename> <tag>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s %s\n", C_REQ_CHECKPOINT, filename, tag);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
printf("Checkpoint '%s' created successfully for '%s'.\n", tag, filename);
else if (!strncmp(response, "403", 3))
printf("Error: Only the file owner can create checkpoints.\n");
else if (!strncmp(response, "404", 3))
printf("Error: File not found.\n");
else
print_error(response);
}

// ----- VIEWCHECKPOINT -----
else if (strcmp(cmd, "VIEWCHECKPOINT") == 0)
{
char *filename = strtok(NULL, " ");
char *tag = strtok(NULL, " ");

```

```
if (!filename || !tag)
{
    printf("Usage: VIEWCHECKPOINT <filename> <tag>\n");
    continue;
}
snprintf(request, sizeof(request), "%s %s %s\n", C_REQ_VIEWCHECKPOINT, filename, tag);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
    perror("recv");
    continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
{
    printf("===== Checkpoint '%s' for %s =====\n", tag, filename);
    // Use robust marker-based detection (like EXEC command)
    char* payload_start = response + 4; // Skip "200\n"
    char* end_marker = NULL;
    bool done = false;

    end_marker = strstr(payload_start, RESP_EXEC_DONE);
    if (end_marker) {
        *end_marker = '\0';
        done = true;
    }
    if (strlen(payload_start) > 0) {
        printf("%s", payload_start);
        fflush(stdout);
    }

    if (!done) {
        while ((n = recv(sockfd, response, sizeof(response) - 1, 0)) > 0)
        {
            response[n] = '\0';
            end_marker = strstr(response, RESP_EXEC_DONE);
            if (end_marker) {
                *end_marker = '\0';
                done = true;
            }
            if (strlen(response) > 0) {
                printf("%s", response);
                fflush(stdout);
            }
            if (done) {
                break;
            }
        }
    }
    printf("\n===== End of checkpoint =====\n");
}
```

```

else if (!strcmp(response, "403", 3))
printf("Error: Access denied.\n");
else if (!strcmp(response, "404", 3))
printf("Error: Checkpoint not found.\n");
else
print_error(response);
}

// ----- REVERT -----
else if (strcmp(cmd, "REVERT") == 0)
{
char *filename = strtok(NULL, " ");
char *tag = strtok(NULL, " ");
if (!filename || !tag)
{
printf("Usage: REVERT <filename> <tag>\n");
continue;
}
// Confirmation prompt
printf("Are you sure you want to revert '%s' to checkpoint '%s'? (yes/no): ",
filename, tag);
char confirm[10];
if (fgets(confirm, sizeof(confirm), stdin) == NULL) {
printf("Input error.\n");
continue;
}
// Remove newline
size_t clen = strlen(confirm);
if (clen > 0 && confirm[clen - 1] == '\n') {
confirm[clen - 1] = '\0';
}
if (strcmp(confirm, "yes") != 0) {
printf("Revert cancelled.\n");
continue;
}
snprintf(request, sizeof(request), "%s %s %s\n", C_REQ_REVERT, filename, tag);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
if (!strcmp(response, "200", 3))
printf("File '%s' successfully reverted to checkpoint '%s'.\n", filename, tag);
else if (!strcmp(response, "403", 3))
printf("Error: You don't have write access to this file.\n");
else if (!strcmp(response, "404", 3))
printf("Error: Checkpoint not found.\n");
else if (!strcmp(response, "504", 3))
printf("Error: Cannot revert while file is locked by another user.\n");
}

```



```
printf("%s", response);
fflush(stdout);
}
if (done) {
break;
}
}
printf("=====\n");
}
else if (!strncmp(response, "403", 3))
printf("Error: Access denied.\n");
else if (!strncmp(response, "404", 3))
printf("Error: File not found.\n");
else
print_error(response);
}

// ----- REQACCESS (Request access to a file) -----
else if (strcmp(cmd, "REQACCESS") == 0)
{
char *filename = strtok(NULL, " ");
char *perm = strtok(NULL, " ");
if (!filename || !perm)
{
printf("Usage: REQACCESS <filename> <-R|-W>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s %s\n",
C_REQ_REQUEST_ACC, filename, perm);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
{
printf("%s", response + 4); // Skip "200 "
}
else
{
printf("%s", response + 4); // Error message
}
}

// ----- VIEWREQUESTS (View pending requests for your file) -----
else if (strcmp(cmd, "VIEWREQUESTS") == 0)
{
char *filename = strtok(NULL, " ");
if (!filename)
{
```

```

printf("Usage: VIEWREQUESTS <filename>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s\n",
C_REQ_VIEW_REQUESTS, filename);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
{
printf("%s", response + 4); // Skip "200\n"
}
else
{
print_error(response);
}
}

// ----- APPROVE (Approve an access request) -----
else if (strcmp(cmd, "APPROVE") == 0)
{
char *req_id = strtok(NULL, " ");
if (!req_id)
{
printf("Usage: APPROVE <request_id>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s\n",
C_REQ_APPROVE_ACC, req_id);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
{
printf("%s", response + 4); // Skip "200 "
}
else
{
printf("%s", response + 4); // Error message
}
}

// ----- DENY (Deny an access request) -----
else if (strcmp(cmd, "DENY") == 0)
{
}
}

```

```

{
char *req_id = strtok(NULL, " ");
if (!req_id)
{
printf("Usage: DENY <request_id>\n");
continue;
}
snprintf(request, sizeof(request), "%s %s\n",
C_REQ_DENY_ACC, req_id);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
if (!strncmp(response, "200", 3))
{
printf("%s", response + 4); // Skip "200 "
}
else
{
printf("%s", response + 4); // Error message
}
}

// ----- MYREQUESTS (View your own access requests) -----
else if (strcmp(cmd, "MYREQUESTS") == 0)
{
snprintf(request, sizeof(request), "%s\n", C_REQ_MY_REQUESTS);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response) - 1, 0);
if (n <= 0)
{
perror("recv");
continue;
}
response[n] = '\0';
printf("%s", response + 4); // Skip "200\n"
}

// ----- Folder commands -----
else if (strcmp(cmd, "CREATEFOLDER") == 0) {
char *folder = strtok(NULL, " ");
if (!folder) { printf("Usage: CREATEFOLDER <name>\n"); continue; }
snprintf(request, sizeof(request), "%s %s\n", C_REQ_CREATEFOLDER, folder);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response)-1, 0);
response[0] = 0;
if(strncmp(response, "200", 3) == 0) printf("%s", response+4);
else print_error(response);
}

else if (strcmp(cmd, "MOVE") == 0) {

```

```

char *file = strtok(NULL, " ");
char *folder = strtok(NULL, " ");
if (!file || !folder) { printf("Usage: MOVE <file> <folder>\n"); continue; }
snprintf(request, sizeof(request), "%s %s %s\n", C_REQ_MOVE, file, folder);
send(sockfd, request, strlen(request), 0);
int n = recv(sockfd, response, sizeof(response)-1, 0);
response[n] = 0;
if(strncmp(response, "200", 3) == 0) printf("%s", response+4);
else print_error(response);
}

else if (strcmp(cmd, "VIEWFOLDER") == 0) {
char *folder = strtok(NULL, " ");
if (!folder) { printf("Usage: VIEWFOLDER <name>\n"); continue; }
snprintf(request, sizeof(request), "%s %s\n", C_REQ_VIEWFOLDER, folder);
send(sockfd, request, strlen(request), 0);
// This response might be multi-line, but isn't streaming.
// Single recv is fine for MVP.
int n = recv(sockfd, response, sizeof(response)-1, 0);
response[n] = 0;
if(strncmp(response, "200", 3) == 0) printf("%s", response+4);
else print_error(response);
}

// ----- Exit command -----
else if (strcmp(cmd, "QUIT") == 0 || strcmp(cmd, "EXIT") == 0)
{
printf("Exiting client...\n");
break;
}

// ----- Other commands -----
else
{
printf("Unsupported command (for now). Try CREATE <filename>, READ <filename>, or EXIT.\n");
}
}

// Clean up socket and exit
close(sockfd);
return 0;
}

```

◆ I found the exact issue! It is the **"Prefix Match" bug** again, exactly like the one we fixed for `VIEW` and `VIEWREQUESTS`.

⌚ The Bug: Command Hijacking

In your `nm.c`, inside `handle_client_commands`, you check for `C_CREATE` before `C_CREATEFOLDER`.

- Client sends: `"C_CREATEFOLDER documents"`
- Name Server checks: `strncmp("C_CREATEFOLDER", "C_CREATE", 8)`
- Result: **Match!** (Because `"C_CREATE"` is the first part of `"C_CREATEFOLDER"`)

So, the Name Server incorrectly runs `do_create()` instead of `do_create_folder()`. This tells the Storage Server to create "documents". Later, when you try to move a file into it, the SS tries to make a directory named "documents", fails because it exists, and crashes the operation.

You also have the same bug with `VIEW` hijacking `VIEWFOLDER`.

🔧 The Fix: Reorder `nm.c`

You must check for the **longer** commands first.

In `nm.c`, find the `handle_client_commands` function (bottom of the file). Rearrange the `if/else if` blocks so `CREATEFC` `CREATE`, and `VIEWFOLDER` comes **before** `VIEW`.

Here is the corrected code block for `nm.c`:

```
C

// In nm.c -> handle_client_commands

while (1) {
    // ... (reading code) ...

    int items_scanned = sscanf(buffer, "%s %s %s %s", cmd, arg1, arg2, arg3);
    if (items_scanned <= 0) {
        continue;
    }

    // --- FIX START: Move CREATEFOLDER before CREATE ---
    if (strncmp(cmd, C_REQ_CREATEFOLDER, strlen(C_REQ_CREATEFOLDER)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_create_folder(client_fd, username, arg1);
        }
    }
    else if (strncmp(cmd, C_REQ_CREATE, strlen(C_REQ_CREATE)) == 0) {
        do_create(client_fd, username, arg1);
    }
    // --- FIX END ---

    else if (strncmp(cmd, C_REQ_ADD_ACC, strlen(C_REQ_ADD_ACC)) == 0) {
        // ...
    }
    // ... (keep REM_ACC, VIEWCHECKPOINT, VIEW_REQUESTS, READ, WRITE, UNDO, STREAM, DELETE, LISTCHECK)
    // ... (Somewhere down the list you have VIEW) ...

    // --- FIX START: Move VIEWFOLDER before VIEW ---
    else if (strncmp(cmd, C_REQ_VIEWFOLDER, strlen(C_REQ_VIEWFOLDER)) == 0) {
        if (items_scanned < 2) {
            snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_BAD_REQ);
            send(client_fd, resp_buf, strlen(resp_buf), 0);
        } else {
            do_view_folder(client_fd, username, arg1);
        }
    }
    else if (strncmp(cmd, C_REQ_VIEW, strlen(C_REQ_VIEW)) == 0) {
        do_view(client_fd, username, arg1);
    }
    // --- FIX END ---

    // ... (keep INFO, EXEC, CHECKPOINT, REVERT, REQUEST_ACC, etc.) ...
}
```

🧹 Cleanup (Critical!)

Because the bug ran once, your system is currently in a broken state. You **must** clean it up before testing again.

1. **Stop all servers** (`Ctrl+C`).

2. Delete the "documents" file on the Storage Server:

```
Bash
```

```
rm ss_data/9002/documents
```

(If you don't do this, the fix won't matter because the file is already blocking the directory creation).

3. Delete the Metadata file:

```
Bash
```

```
rm nm_metadata.dat
```

4. Recompile:

```
Bash
```

```
make
```



Now start your servers and client again.

```
> CREATEFOLDER documents
```

(This should now ONLY create metadata on NM. The SS logs should be silent).

```
> CREATE notes.txt  
> MOVE notes.txt documents
```

(The SS should now say "rename successful" because it will successfully run `ensure_directory_exists` and create the re

ss

C

nm

C

client

C

check my implementation of the project as given below and give me a detailed list of what all i have implemented as per requirement

Docs++

Simply put - You need to implement a simplified, shared document system (similar in spirit to Google Docs) from scratch. It should support basic operations like creating, reading, updating and deleting documents. It should also support 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 refer to them 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 by entering 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 repository 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. You have been given a brief, which outlined their vision for the system, summarised below. Your task is to design and implement this system from the provided specifications, within the next month. The goal is a Christmas launch, so timely delivery is crucial (with some deadline extensions). The outcome will impact both the company's future and your career (with the added bonus of a bonus - 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 must 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 will go 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 format. A file consists of multiple sentences, and each sentence is made up of words. A sentence is defined as a sequence of words separated by a period (.), exclamation mark (!), or question mark (?). Words within a sentence are separated by spaces. This means that if a user wants to access a file, the file must be handled by the system, the user should be able to access the file as a whole. For completeness and in interest of readability, 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 files (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 by others until the operation is complete. This allows multiple users to view or edit the file simultaneously, but prevents them from editing 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 user. The "-l" flag should list files along with details like word count, character count, last access, owner, etc. Note, a combination of "-l" and "-a" flags should list all files with details.
VIEW # Lists all files user has access to
VIEW -a # Lists all files user has access to
-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 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 an existing file.`WRITE <filename> <sentence_number>` # Locks the sentence for other users (if no file write access, should return error)
`<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. The content may contain characters like period (.), exclamation mark (!), or question mark (?). The system should handle these sentence delimiters and create separate sentences accordingly. (Please refer to examples given below, for reference.) Hint: For resolving concurrent read-write issues, you may write to a temporary swap file initially, and move the content once all updates are complete. You may also consider using locks, semaphores, some algorithmic approach, etc.

[15] Undo Change: Users can revert the last changes made to a file.`UNDO <filename>` # Reverts the last change made

Note: The undo-ees are file specific, and not user specific. So, if user A makes a change, and user B wants to undo 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 file size, access rights, timestamps, and other metadata, providing users with comprehensive insights into the file.

[10] Delete a File: Owners should be able to remove files from the system when they are no longer needed, contributing to management. All data like user access should be accordingly updated to reflect this change.`DELETE <filename>` # Deletes the file

[15] Stream Content: The client establishes direct connection with the Storage Server and fetches & displays the content of the file word by word. This simulates a streaming effect, allowing users to experience the content with a delay of 0.1 seconds between each word. This simulates a streaming effect, allowing users to experience the content in a more natural 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
`REDACT <username>` # Removes all access

[15] Executable File: Users (with read access) can “execute” the file. Execute, here, means executing the file content. The output of the command should be displayed to the user.`EXEC <filename>` # Executes the file content as shell command

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. The data must remain 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 and 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. The NM should display (print in terminal) relevant messages indicating the status and outcome of each operation. This ensures traceability and aids in debugging and system monitoring. Each entry should include relevant information such as timestamp, file path, 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 errors, including interactions between clients, Name Server (NM), and Storage Servers (SS). Define a comprehensive set of error codes for various scenarios such as unauthorized access, file not found, resource contention (e.g., file locked for writing) and system errors. These codes should be universal throughout the system.

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

Note: An approach faster than O(N) time complexity is expected here. Efficient data structures like Tries, Hashmaps, or Bloom filters 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 system. 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 the Storage servers while registering.

Storage Server (SS): Each Storage Server is responsible for physically storing the files and interacting with the Naming Server. During initialization, the SS sends vital details about its existence to the Naming Server. This information includes: IP address, port for client 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 their IP 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 client information provided by Storage Servers (SS) upon connection. This information is maintained by NM, to later direct data requests to the 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 acknowledgement 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 ability to dynamically add their entries to the NM at any point during execution. This flexibility ensures that the system can adapt to scaling requirements seamlessly. The initialisation process at the storage server side follows the same protocol as specified in Specification 1.

Commands Issued by NM: The Name Server can issue specific commands to the Storage Servers, such as creating 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 system 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. The system should ensure that users can only perform actions on files they have permissions for, based on their user profile. The profile 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 (or servers, if 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 communication between the client and the appropriate Storage Server. The operations can be broadly categorized into:

- Writing, Streaming :** The NM identifies the correct Storage Server and returns the precise IP address and client port 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 of 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 gathers 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 S processes the request and performs the specified action, such as creating / deleting the file. After successful execution, it sends an acknowledgment (ACK) to the NM to confirm task completion. The NM, in turn, conveys this information back to the user, 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 executes the commands contained within the file and captures the output. This output is then relayed back to the 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 this hierarchy when performing file operations. Some associated commands that are expected to be implemented:

```
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. The user 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. There is no need of a push-notification mechanism, a simple storing of requests and an owner-side feature to view requests is sufficient.

[15] Fault Tolerance: To ensure the robustness and reliability of the system, the following fault tolerance and data redundancy 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 data from one of the replicated stores. Every write command should be duplicated asynchronously across all replicated stores, 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 proactively to 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 store. This ensures that the SS is synchronized with the current state of the system and can resume its role in data storage seamlessly.

[5] The Unique Factor: What sets your implementation apart from others? Well, this is where you showcase your ingenuity.

Examples

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

Example 1: View File

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 time.

Example 3: Create File

```
Client: CREATE mouse.txt # Creates an empty file named mouse.txtFile Created Successfully!Client: VIEW--> wowee  
mouse.txt
```

Note: NS dynamically adds the new file to the list of available files and updates backup SSSs. If the file already exists, it will result in an appropriate error.

Example 4: Write to a File

```
Client: WRITE mouse.txt 0 # Adding to the start of fileClient: I'm just a mouse.Client: ETIRWWWrite Successful!Client:  
just a mouse.Client: WRITE mouse.txt 1 # In essence, appending to the fileClient: I don't like PNSClient: ETIRWWWrite  
READ mouse.txtI'm just a mouse. I don't like PNSClient: WRITE mouse.txt 2 # Caveat, note the lack of delimiter after  
are only 2 sentences.ERROR: Sentence index out of range. # Similarly for word indexes (negative or > number of words  
in errors)Client: WRITE mouse.txt 1 # Inserting into the second sentenceClient: 3 T-TClient: ETIRWWWrite Successful!  
mouse.txtI'm just a mouse. I don't like T-T PNSClient: WRITE mouse.txt 0 # Inserting multiple times into a sentenceClient:  
hollow lil gei-fwen # New sentence : I'm just a deeply mistaken hollow lil gei-fwen pocket-sized mouse.Client: 6 pocket  
sentence : I'm just a deeply mistaken hollow pocket-sized lil gei-fwen mouse.Client: ETIRWWWrite Successful!Client:  
a deeply mistaken hollow pocket-sized lil gei-fwen mouse. I don't like T-T PNSClient: WRITE mouse.txt 1 # Inserting  
delimiterClient: 5 and AAD. aaaah # New sentences : [I don't like T-T PNS and AAD.]* [aaaah]. Currently active status  
index at index 1Client: 0 But, # New sentence : [But, I don't like T-T PNS and AAD.]* [aaaah].Client: ETIRWWWrite Successful!  
mouse.txtI'm just a deeply mistaken hollow pocket-sized lil gei-fwen mouse. But, I don't 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 it.  
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.txtrm -rf / # Oops!Client: WRITE nuh_uh.txt 0Client: 0 sudoClient: ETIRWWWrite Successful!Client:  
nuh_uh.txtsudo rm -rf /Client: UNDO nuh_uh.txtUndo Successful!Client: READ nuh_uh.txtrm -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 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  
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.txtClient: DELETE mouse.txtFile 'mouse.txt' deleted successfully  
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 per 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  
bytes--> Access: user1 (RW)--> Last Accessed: 2025-10-10 14:32 by user1Client: ADDACCESS -R nuh_uh.txt user2
```

```
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 user1Client: ADDAccess 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 user1Client: REMAccess removed successfully!Client: INFO nuh_uh.txt--> File: feedback.txt--> Owner: user1--> Created: 2025-10-10 14:32--> 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.txtRunning siagnostics...--> LMAAO.txt--> RotT.txt--> AUG.txtDone!
```

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.

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 amongst yourselves with each team member implementing either the client, the naming server or the storage server. Identify things that can be decoupled. For example, the specification where an SS can join the NM at any given moment doesn't depend on neither the client or the SS (Assuming you have figured out how SSs attach themselves at the time of execution). One team member can implement this while the other thinks of ways to implement caching (You probably 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 do it 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 them are not implemented yet.

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 Distributed 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 discussion)

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.

Communicate with your teammates if you cannot fulfill your deliverables on time. That is the entire point of team-based projects.

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 usecase (through a usecase) and not a production system. The requirements are designed to make you think about the various distributed systems and file systems in particular.

We would also like to ensure that all the requirements and implementables were thoroughly discussed in TA meets k document and finally you. To give a few examples (with hopes of igniting that system thinking spark and no thought bit) Using period as delimiter might seem like a bad idea, especially when it is in the middle of a word. You might think a character would be better. However, this would mean that every time a user wants to write a new sentence, they would need 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 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 highlight these trade-offs and come up with a solution that balances them effectively. Anyways in the industry (and D) you will be working with, probably, worse requirements. So, this is a good practice run.

ALL THE BEST! BUT MOST IMPORTANTLY, HAVE FUN!

Copyright © 2024 Karthik Vaidhyanathan. Distributed by an MIT license.

- ◆ Based on the code provided (`client.c`, `nm.c`, `ss.c`, `protocol.h`) and the project requirements (`Docs++`), here is a quick summary of your implementation status.
 - ⦿ You have done an **excellent job** covering the vast majority of the requirements, including complex features like concurrency and bonuses.

✓ Fully Implemented Functionalities

1. User Functionalities (150/150)

- **View Files:** Implemented via `do_view` (handles `-a`, `-l` flags).
- **Read File:** Implemented via `C_REQ_READ` -> `SS_GET_FILE`.
- **Create File:** Implemented via `do_create`.
- **Write to File (Concurrent):** This is the hardest part, and you have a robust implementation:
 - **Locking:** `SS_LOCK` locks specific sentences using `g_lock_list` mutexes.
 - **Updating:** `SS_UPDATE` handles word insertion/appending logic in memory.
 - **Committing:** `SS_COMMIT` checks for timestamp conflicts (3-way merge logic) and writes to disk.
- **Undo:** Implemented via `NM_UNDO` swapping the `.bak` file.
- **Info:** Implemented via `do_info` (returns size, owner, timestamps).
- **Delete:** Implemented via `do_delete`.
- **Stream:** Implemented via `SS_GET_STREAM` with `usleep(STREAM_DELAY_US)`.
- **List Users:** Implemented via `do_list_users`.
- **Access Control:** Implemented `ADDACCESS` and `REMACCESS`.
- **Execute:** Implemented via `do_exec` (fetches file to temp, runs `popen`).

2. System Requirements (40/40)

- **Data Persistence:** `nm.c` saves `nm_metadata.dat` and `favorites.dat`. `ss.c` saves files to disk.
- **Access Control:** All `do_` functions in `nm.c` check permissions against `acl_list`.
- **Logging:** Implemented `log_event` in both servers.
- **Error Handling:** `print_error` in client translates codes (404, 403, etc.).
- **Efficient Search:** You implemented a **Trie** (`trie_search`) for path lookups and an **LRU Cache** (`cache_find`) for hot $O(L)$ and $O(1)$ respectively).

3. Bonus Functionalities Implemented

- **[10] Hierarchical Folders:** Implemented.
 - `CREATEFOLDER` : Adds folder metadata to Trie.
 - `MOVE` : Handles renaming/moving files and updating paths.
 - `VIEWFOLDER` : Lists directory contents.
 - Note: You handled the normalization of paths (/) well in `client.c`.
- **[15] Checkpoints:** Implemented.
 - `CHECKPOINT`, `VIEWCHECKPOINT`, `REVERT`, `LISTCHECKPOINTS` logic exists in SS and NM.
- **[5] Requesting Access:** Implemented.
 - Queue system for requests (`g_access_requests`) with `APPROVE` and `DENY` commands.
- **[5] Unique Factor (Favorites & Trending):** Implemented.
 - You added a "Favorites" system (`C_FAVORITE`, `C_TRENDING`) which counts likes and shows trending files. This is a

⚠ Partially Implemented / Missing

The only significant gap is in the **Fault Tolerance** bonus section.

1. [15] Fault Tolerance (Replication & Recovery)

- **What you have:**
 - **NM Side:** The `nm.c` has the logic to assign a `replica_ss_index` in `do_create`. It has a background thread `rep` a queue `queue_replication_task` to send `SS_REPLICATE` commands asynchronously. This is great architecture.
- **What is missing:**
 - **SS Side (Processing):** Your `ss.c` main loop (lines 1435-1650) handles `NM_CREATE`, `NM_DELETE`, `NM_MOVE`, etc., I handlers for `SS_REPLICATE` and `SS_REPLICATE_WRITE`. Even though the NM sends these commands, the SS will Unknown NM command } block and do nothing.
 - **Failure Detection:** The NM doesn't seem to actively heartbeat the SS. It detects failure only when a `connect` fails request.
 - **Recovery:** There is no logic for an SS to say "I just restarted, please give me the data I missed" (`SS_Recovery`).

📝 Detailed Recommendations to Finalize

To get full marks on the Fault Tolerance bonus, you need to patch `ss.c`.

1. Update `ss.c` Main Loop In `ss.c`, inside the `while(1)` loop where you handle NM commands, add:

```
C

else if (strncmp(cmd, SS_REPLICATE, strlen(SS_REPLICATE)) == 0) {
    // 1. Parse filename
    sscanf(nm_buffer, "%*s %s", filename);
    // 2. Create file (similar to NM_CREATE logic)
    snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
```

```

FILE *fp = fopen(local_path, "w");
if (fp) fclose(fp);
// 3. Log event
}
else if (strncmp(cmd, SS_REPLICATE_WRITE, strlen(SS_REPLICATE_WRITE)) == 0) {
    // 1. Parse filename and content
    // Note: Content might contain spaces, use proper scanning
    char content[MAX_MSG_LEN];
    sscanf(nm_buffer, "%*s %s %[^\n]", filename, content);

    // 2. Write content to file
    snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
    FILE *fp = fopen(local_path, "w"); // Overwrite mode for sync
    if (fp) {
        fprintf(fp, "%s", content);
        fclose(fp);
    }
}

```

2. Cross-System Connectivity Currently, you have `#define NM_IP "127.0.0.1"` in `client.c` and `ss.c`.

- **To run on different machines:** You must change this to the actual LAN IP of the machine running the Name Server (e.g., 192.168.1.10).
- **SS Registration:** In `ss.c`, you send `S_INIT 127.0.0.1 ...`. The SS needs to send its *actual* IP to the NM so the NM can connect to the correct remote machine.

Summary Checklist

Feature	Status	Notes
Core User Ops	✓ Done	View, Read, Write, Create, Delete, etc.
Concurrency	✓ Done	Sentence locking and 3-way timestamp merge.
System Req	✓ Done	Persistence, Logging, Error Codes.
Search	✓ Done	Trie + LRU Cache.
Folders	✓ Done	Create, Move, View folders.
Checkpoints	✓ Done	Versioning implemented.
Access Request	✓ Done	Request/Approve flow implemented.
Unique Factor	✓ Done	Favorites/Trending system.
Replication	⚠ Partial	NM sends commands, SS ignores them. Fix <code>ss.c</code> .
Recovery	✗ Missing	SS sync on restart not implemented.



Overall: You have a very strong project (approx. 230/250 marks as is). Fixing the `ss.c` replication handler will secure the R

can you check the project specifications in detail and give me a list of everything not yet implemented

CS3.301

Course Project Answers

Try HackMD Logo HackMD

"The goal is a Christmas launch, so timely delivery is critical for this MVP (no deadline extensions)." ... This is written in the section of the Course project doc. Does this mean we cannot use late days?

[AG] - No, you can use late days; in accordance with the usual course policies, as shared before.

Say one client is editing a particular file while someone else is streaming the same file. How shall it output?

[AG] - As long as the WRITE is not completed (the client has sent ETIRW), the file content remains the original content. It should display the original file content.

[10] Get Additional Information: Users can access a wealth of supplementary information about specific files. This includes file size, access rights, timestamps, and other metadata, providing users with comprehensive insights into the file's properties.

What does timestamps mean in this case?

[AG] - Time of file creation, last edited time and anything more you would like to add.

In example 4: write to a file: Client: WRITE mouse.txt 2 # Inserting a sentence delimiter Client: 5 and AAD. aaaah # I like T-T PNS and AAD.]* [aaaah]. Currently active status remains with the index at index 2 Client: 0 But, # New sentence Client: T-T PNS and AAD.]* [aaaah]. Client: ETIRW Write Successful!, shouldn't the sentence number be 1 and not 2 in the response? (because up until the previous line, we havent added a delimiter to sentence 1) and the active status also remains at index 2 right?

[AG] - Ah yes, true. It should be 1 only (mb)

In Example 10: Access Control: Client: ADDACCESS -W nuh_uh.txt user3 Access granted successfully! -> File: feed user1 -> Created: 2025-10-10 14:21 -> Last Modified: 2025-10-10 14:32 -> Size: 52 bytes -> Access: user1 (RW), user3 (R) Accessed: 2025-10-10 14:32 by user1, isn't "File:" supposed to display the file's name (which is nuh_uh.txt)? and in the message, user3 added (assuming there exists user3 coz the message says "Access granted successfully")?

[AG] - Yes yes user3 is a typo, it was meant to be user2 only. Will fix, thnx for pointing out!

"After each sentence write update, the index must word_index must update for the next sentence." Could you please explain what this means?

[AG] - Fixed the wording, pls go through that. Also, an example issue (with possible solution) was shared in the 18th slide, could you please go through that for more clarity.

Just to confirm, -W flag would provide BOTH read and write access to the user right?

[AG] - Yes

Referring to the word-index in example 4. Should we take it as 0-indexed or 1-indexed? And is the index referring to the positions between the words? For example, say a sentence has "A B C D E" will "1 Z" result in "A Z B C D E" and "A B Z C D E"? There are 4 cases of inserting a word using word index in example 4, sentence 0 word 4 and word 6, and sentence 0, and the policy for inserting words doesn't match in all the cases.

[AG] - The example assumes 0-index, but feel free to choose whatever suits you. While there shouldn't be any discrepancy in the output, it's good to know that examples (I'll reconfirm and fix, if any), but the indexing and index-values can be chosen by you in whatever system you're using. The underlying structure for WRITE calls is preserved.

Under Bonus Functionalities, in Hierarchical Folder Structure, is the structure created by a user expected to be persistent? If a user logs back in, he should start off with the folder structure he left with?

[AG] - Not necessarily, you can have them starting from the root folder.

Under Bonus Functionalities, do checkpoints need to be persistent? Also, are checkpoints file specific or user specific?

[AG] - Yes, that's the whole point of checkpoints: To be able to revert to them from anytime in the future. They are persistent.

It is mentioned that Reading, Writing, Streaming : The NM identifies the correct Storage Server and returns the pre-allocated port for that SS to the client. Subsequently, the client directly communicates with the designated SS. How exactly does the NM identify the correct SS for the client (in case there are multiple SS hosting the same file)?

[AG] - Any of the SS containing that file would be the "correct SS". So, NM can return the IP and port of any of the servers, and the client can choose the one with the lower load if you can employ some mechanism to judge that.

Lets consider File1 is in SS1 and File2 is in SS2. Say direct connection between SS1 and client is established. Now if File2 will a new connection establish between SS2 and client or will it just say file not exists?

[AG] Client-SS relations are per request. So, after the client's File1 request is finished the Client-SS1 connection is next request to File2 from SS2 (or even File1 from SS1), a fresh connection between the two need to be established.

What exactly does it mean by sending a predefined STOP packet? Does it mean that the user has to send a STOP after read, write or stream task is completed, it automatically terminates the connection?

[AG] - A STOP packet signifies the end of the communication. It must be explicitly sent for the receiver to realise the communication.

Do we have to define each data blocks size and the number of data blocks in the storage? Also, is there any restriction on capacity of each storage server?

[AG] - Depending on your implementation, you might have to. There is no specific restriction, as in your system storage capacity of the server. But, you can expect the evaluation requirements to not go beyond a few MBs of data.

In example 4,

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 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.

Why has there been a delimiter added at the end of aaaah?

[AG] - For a user, it is just a full stop, they can add it anywhere. The system interprets it as a delimiter. So, no reason to chose to end his statement with a full-stop.

Should we include LLM Generations with the files?

[AG] - Yes, just clearly demarkate it, and add appropriate links in a seperate README file.

image

this was shown in the tut online explanation, just wanted to make it clear

does it work like, lets say it was

[S1->hello->folks

S2-> hows->life

so on ..]

I want to insert the following after hello

S->hi

S'->!

S''->!

so will the final be

[

S1-> hello hi

!

! folks

]
is this correct?

[AG] - Yes (it's just a demo implementation, you may choose to implement different data structure, etc)

does backup start automatically when we connect one of the storage server, or does that also require explicit connection to storage server 1, does the backup for it start automatically, or I need to start its backup also myself?

[AG] - As soon as a new 'empty' server comes up online, it can start working as a backup for some other server. This acts as a backup or not, depends on your implementation. But you should ensure that if resources are available, all copy for backup.

For READ/WRITE/STREAM operations: After NM gives SS details to client, should the client open a new TCP socket to two listening ports: one for NM, one for clients?

[AG] - Yes, while maintaining its connection to NM. Yes

How exactly is a user trying to write affected when there are $>=1$ readers or streamers? For example, if the user is writing while the file is being read/streamed. Should this request be queued after the readsstreams? Or should the user be interrupted by interrupting the readers/streamers?

[AG] - (Answered previously) Unless the WRITE is completed, all accesses to the file return the original data (before it is given to read/stream).

Till what extent are we expected to handle packet loss? Can we assume no "ACK"s will be lost or are we supposed to as well? Please clarify the extent of error handling for the networking side of things.

[AG] - The user should never exit abruptly. If there is a loss of packet, there should be mechanisms for retransmission. Handle worst-case scenarios of whole network chain broken, and appropriate and graceful error handling. Some errors are expected.

The 5 marks of bonus will be granted based on how we implement the project specifications or will they also be granted for new/extra features we devise?

[AG] - Assuming this doubt is for the "Unique Factor" part. New / extra feature.

Undo Change: Users can revert the last changes made to a file. Does this mean that if I do WRITE <some_sentences> and after that when I undo it removes ALL the sentences that were written by WRITE? Also if I do undo again, should it remove sentences or undo whatever change was made before that or do nothing?

[AG] - It undoes all the sentences within that single WRITE command. Undo again, is not redo. It will undo whatever was done before that.

Client: WRITE mouse.txt 1 # Inserting a sentence delimiter Client: 5 and AAD. aaaah # New sentences : [I dont like T-aaaah]. Currently active status remains with the index at index 1 Client: 0 But, # New sentence : [But, I dont like T-aaaah]. Here, if I try the command : Client: 9 abcd Should I get an error because I'm referencing next sentence or not?

[AG] - 9 would give error. But 8 would append to the first sentence. You cannot jump sentences by giving a large index.

The SS sends a list of files on it when it registers. Can we assume that 4096 bytes is enough for a very long list?

[AG] - No, we'll prefer a more robust way, like a stream of packets you send until the whole list is sent with an acknowledgement to confirm the whole list is transmitted.

Should we Load ALL files at server startup (slower startup, all files always ready) or do Lazy loading - load file on demand once loaded (faster startup) what to do?

[AG] - No, you should never load all data files. Bad design when you have humongous data stores. You should always request ONLY. And ensure you keep clearing memory (cache) after some time. The whole point of cache is to be selective.

Are the shell commands in the file to be EXEC'd singular commands? Or can they include pipes, background & etc?

[AG] - Won't be singular commands, can include file read/writes, some computation, etc. (Ideally, you should not run as is, very insecure). But, you can assume that these files would clear up all the data / files they create, not mess up the system. It would not hog resources.

If access is attempted to be given to a user that is not registered (does not show up in list), should we give an error?

[AG] - Yes, an unregistered user should ideally not even get access to the interface.

Is cJSON header allowed to be used?

[AG] - You may use any POSIX library.

If a library is not posix, is it allowed to be used?

[AG] - No, only POSIX libraries are allowed.

Following up on 27., they can be commands other than bash also? Please clarify 'shell commands'. They need not be bash commands. What else can they be? Can they be READ, WRITE, STREAM, CREATEFOLDER etc.?

[AG] - Only bash commands would be there

For the EXEC operation, do we have to separately interpret the file and then send for execution? For example, let's say we have a file hi.txt which contains 'echo hiii'. If we interpret each line as a command, then the operations lead to the file hi.txt have echo hiii. Is. Is the expected output:

hiii\n<list of files in directory>

or hiii.\n<list of files in directory>

or hiii. ls

[AG] - Treat the whole file as a bash script.

For EXEC operation, are we supposed to send request from client to ns, then fetch the file from ss and execute the file? Or the flow has to be different from the normal READ and WRITE operations right? And should the whole file be treated as one sentence or should we have to parse it sentence by sentence?

[AG] - Yes, the flow would be different from READ / WRITE. Yes, treat it as a bash script, dont parse it sentence-by-sentence.

Is the undo command supposed to store only one level of history, or is the code supposed to support multiple simultaneous undo operations? Or is it supposed to be chained together?

[AG] - Just one-level works, we only need a proof of concept

When we are implementing WRITE, do we add a newline after adding each sentence? According to example 2, yes. According to example 4, no. So what do we do?

[AG] - The content of the file would have '\n' to signify new line. Both examples follow that, just example 4 never ends the sentence.

I have 2 clients alice and bob, and then quit the client alice in the alice server, and then run LIST from the bob server. What would be the output?

->alice

->bob

since alice was technically registered before? or does the output change to:

->bob

[AG] - Registered users include users not currently online. You must store all users that have logged in till date.

In the document it is mentioned "At any point, there would be a single instance of the Name Server running, to which all User Clients and Storage Servers can connect. The User Clients and Storage Servers can disconnect and reconnect at any time. The system should handle these events gracefully." My question - Does NM need to be able to disconnect using ctrl+c or ctrl+z fine, as the exiting of this server is not mentioned in the project document?

[AG] - No, you can assume that once NM starts, it stays on for the whole duration. NM going down is not in the scope of this project.

If the owner is attempting to delete a file, but the file is being written to by another user, are we expected to:

a) Prevent the owner from deleting with an error message?

b) Give an error message to the editing user?

[AG] - Prevent deletion as the sentence is locked.

In the 23rd answer you have mentioned that multiple undoes are supported for one file while the project doc explicitly says that only one level of undo is supported. Is this correct? If so, why not(under example 5)?

[AG] - The minimum requirement is one-level undo. People might choose to expand features or develop their unique own.

For performing UNDO, does a user need to have write permission for that file?

[AG] - Yes

What is expected to be done in caching? Does it need to be implemented on the ns or ss? As on the ns, it would just store the storage server id for the corresponding file, whose access is already O(1) or O(log n) average case.

[AG] - It won't be O(1), might tend to be but not O(1). But the cached mappings should be O(1) always.

can we use glibc?

[AG] - You may use any (and only) POSIX libraries.

do access lookups also need to be fast?

[AG] - Yes, they need to be sub-linear time complexity. And cached ones need to be constant time.

If a user is writing to a file and has a sentence locked, but the client disconnects midway, should other users be allowed to write to that sentence? Do we leave the sentence as locked until the previously disconnected user writes that sentence again?

[AG] - If the client disconnects midway a WRITE (ie, before sending the ERITW), no changes of that write should propagate. The file should be as if the write never happened. And all locks are relieved for other users to access the file without any issues.

For running the EXEC command, Example 11 in the document shows commands being separated by newline characters. Should the file be in our document as well? Or separated based on sentence delimiters, as that is how sentences are handled in files?

For example:

"ls

echo "Hello"

or "ls. echo "Hello".

[AG] - The file contents, as is, should be treated as a normal bash script for running. The delimiters don't play any role here.

Can we assume that WRITE to a file will always happen via terminal ONLY and not directly to the file? If only through terminal, then we can add newline characters to the file?

[AG] - Yes, it can only happen to terminal. Something like '\n' should work, depends on how you want to implement it. It's up to the developer to decide.

For blocking operations like EXEC on the Name Server and WRITE on the Storage Server, is it acceptable to let them block, or are we required to implement a thread pool to handle them asynchronously?

[AG] - Thread pool would be preferred

only file-lookups need to be efficient?

[AG] - Can you elaborate on what you mean by 'only'...

Client: READ mouse.txt

I'm just a mouse. I don't 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 : I'm just a deeply mistaken hollow lil gei-fwen pocket

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

The words for sentence(0) are : (0) Im (1) just (2) a (3) mouse. I dont like T-T PNS

shouldn't the second line of input be "Client: 3 deeply mistaken hollow lil gei-fwen" instead of client: 4 ?

[AG] - Yes, some oopsie-woopsie from the TAs side while handling 0-based / 1-based indexing.

in q43, we are expected to implement caching for access lookups as well???

[AG] - Yes, file access (I assume you mean the SS mappings, from this), need to have some cache implementation.

lets say a client 1 is writing to a file at sentence index 1 so logically other client 2 must give sentence is locked for w
should it give the same for index 2 or it should be able to write at any other index except for 1. also can client 2 rea
is writing.

[AG] - Explained in detail in the tutorial video. Refer to that

Suppose client 2 is in a WRITE session to a file whose owner is client 1. While client 2 is writing, client 1 removes the
Should that write session succeed or not?

[AG] - You can make a justifiable assumption, and work with that.

lets say:

WRITE file.txt 0

1 hello

2 ww

ETIRW

WRITE file.txt 0

1 finally !

ETIRW

WRITE file.txt 1

1 demn

ETIRW

what should the output of this be?

is it- finally hello ww! demn

or is it- finally! demn hello ww

pls confirm asap

For STREAM, is it fine, if there is no gap between the words printed?

Last changed by

CS3.301

Operating Systems and Networks

0

721

Read more

Course Project Questions

"The goal is a Christmas launch, so timely delivery is critical for this MVP (no deadline extensions)." ... This is written section of the Course project doc. Does this mean we cannot use late days?

Nov 19, 2025

Mini Project 2 Questions

I wasn't allowed to Sync fork due to some merge conflicts that I couldn't make sense of, is it fine if i delete the fold xv6-riscv MIT repo instead?

Oct 16, 2025

Mini Project 2 Answers

I wasn't allowed to Sync fork due to some merge conflicts that I couldn't make sense of, is it fine if i delete the fold xv6-riscv MIT repo instead?

Oct 13, 2025

OSN Doubts Document – Main Page (2025)

Welcome to the OSN course!

Sep 26, 2025

Read more from CS3.301

Published on HackMD

25

:

11

1x

Docs++

Simply put - You need to implement a simplified, shared document system (similar in spirit to Google Docs) from scratch. Implement 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 them 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 using 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. They have approached you as part of their top engineering team to build the first version of their solution and present it to potential investors. They have outlined their vision for the system, summarised below. Your task is to design and implement this system from the ground up.

following the provided specifications, within the next month. The goal is a Christmas launch, so timely delivery is crucial (with extensions). The outcome will impact both the company's future and your career (with the added bonus of a raise, though 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 can connect. The User Clients and Storage Servers can disconnect and reconnect at any time, and the system must 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 will stop working, and the system 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 format, consisting of multiple sentences, and each sentence is made up of words. A sentence is defined as a sequence of words separated by spaces. A period (.), exclamation mark (!), or question mark (?). Words within a sentence are separated by spaces. This means that, even if a file is handled by the system, the user should be able to access the file as a whole. For completeness and in interest of readability, a sentence 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 files (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 by others until the operation is complete. This allows multiple users to view or edit the file simultaneously, but prevents them from editing 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 user. The "-l" flag should list files along with details like word count, character count, last access, owner, etc. Note, a combination of "-l" and "-a" flags 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 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 existing files.

WRITE <filename> <sentence_number> # Locks the sentence for other users (if no file write access, should return error)

<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. The content may contain characters like period (.), exclamation mark (!), or question mark (?). The system should ignore these sentence delimiters and create separate sentences accordingly. (Please refer to examples given below, for reference.) Yes, every period (or question / exclamation mark) is a sentence delimiter, even if it is in the middle of a word like "ackchually!"

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

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

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

Note: The undo-ees are file specific, and not user specific. So, if user A makes a change, and user B wants to undo 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 file size, access rights, timestamps, and other metadata, providing users with comprehensive insights into the file.

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 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 of the file 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 more natural 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, and 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. 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 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 on the permissions set by the file owner.

[5] Logging: Implement a logging mechanism where the NM and SS records every request, acknowledgment and r the NM should display (print in terminal) relevant messages indicating the status and outcome of each operation. ensures traceability and aids in debugging and system monitoring. Each entry should include relevant information 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 including interactions between clients, Name Server (NM), and Storage Servers (SS). Define a comprehensive set of scenarios such as unauthorized access, file not found, resource contention (e.g., file locked for writing) and system codes should be universal throughout the system.

[15] Efficient Search: The Name Server should implement efficient search algorithms to quickly locate files based on metadata, minimizing latency in file access operations. Furthermore, caching should be implemented for recent searches 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 and 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 Storage servers while registering.

Storage Server (SS): Each Storage Server is responsible for physically storing the files and interacting with the Naming Server. During initialization, the SS sends vital details about its existence to the Naming Server. This information includes: IP address, port for client 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 their IP address 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 data provided by Storage Servers (SS) upon connection. This information is maintained by NM, to later be used by clients.

- ◆ This is a detailed analysis of your implementation against the official specifications and common distributed systems requirements.
- ◎ You have built an extremely robust system, covering all **Core Functionalities** and **System Requirements** completely. The remaining points have been concentrated in the advanced **Fault Tolerance** section, which is common for projects of this scope.

✓ Section A: Fully Implemented Requirements (150/150 + 40/40 + 30/50 Bonus)

You have successfully implemented all core functionalities, demonstrating solid command over concurrency and data structures.

1. User Functionalities [150 Marks]

Functionality	Status	Implementation Details
VIEW	✓ Done	<code>do_view</code> handles listing files, including ACL checks and optional detailed output (-l , -a).
READ / STREAM	✓ Done	Implemented via the referral system (<code>RESP_SS_INFO</code>). STREAM correctly uses <code>SS_GET_STREAM</code> with streaming from <code>ss.c</code> .
CREATE / DELETE	✓ Done	Implemented with atomic commitment and Trie / Cache updates.
WRITE (Concurrency)	✓ Done	Implemented with <code>SS_LOCK</code> (sentence-level locking), <code>SS_UPDATE</code> (in-memory modification), <code>SS_CO</code> write with optimistic concurrency check/3-way merge logic. Locks are released upon disconnect.
UNDO	✓ Done	Implemented as a single-level revert using the <code>.bak</code> file created during WRITE commits.
INFO	✓ Done	<code>do_info</code> retrieves and displays full metadata (timestamps, size, ACLs).
LIST	✓ Done	Users are correctly registered and listed via <code>g_user_registry</code> .
ACCESS (ACLs)	✓ Done	<code>ADDACCESS</code> / <code>REMACCESS</code> fully implemented, updating <code>acl_list</code> in <code>FileMapEntry</code> .
EXEC	✓ Done	Implemented via <code>do_exec</code> . NM fetches content from SS, runs it via <code>popen()</code> , and relays the output via <code>RESP_EXEC_DONE</code> marker.

2. System Requirements [40 Marks]

Requirement	Status	Implementation Details
Data Persistence	✓ Done	Metadata (<code>nm_metadata.dat</code>) and files (on SS disk) are persistently stored.
Access Control	✓ Done	Enforced across all NM functions (<code>do_read</code> , <code>do_write</code> , etc.).
Logging	✓ Done	Implemented <code>log_event</code> in both NM and SS for requests/responses.
Error Handling	✓ Done	Comprehensive use of response codes (2xx, 4xx, 5xx) and corresponding client-side display log.
Efficient Search & Caching	✓ Done	Trie (<code>trie_search</code>) provides sub-linear file lookup. LRU Cache (<code>cache_find</code>) provides cache for recent files.

🚀 Section B: Bonus Functionalities (50 Marks)

Bonus Feature	Status	Implementation Details
[10] Hierarchical Folder Structure	✓ Done	Implemented CREATEFOLDER (metadata-only entry with <code>is_folder=true</code>), VIEWFOLDER (<code>g_file_map</code>), and MOVE (synchronous SS rename operation). Includes safety checks in code to reject folder operations.
[15] Checkpoints	✓ Done	Implemented CHECKPOINT , REVERT , VIEWCHECKPOINT , and LISTCHECKPOINTS . Uses client (EXEC) pattern for streaming access.
[5] Requesting Access	✓ Done	Implemented REQACCESS , VIEWREQUESTS , APPROVE , and DENY using the <code>g_access</code> system.

Bonus Feature	Status	Implementation Details
[5] Unique Factor	✓ Done	Implemented a Favorites/Trending system (<code>do_favorite</code> , <code>do_trending</code>) that tracks user activity.
[15] Fault Tolerance: Replication	⚠ Partial (NM-Side Only)	NM has logic to assign a <code>replica_ss_index</code> and a background thread (<code>replication_worker</code>) to handle <code>SS_REPLICATE</code> tasks asynchronously.

✖ Section C: Critical Missing Components (Fault Tolerance)

The core missing work lies in closing the loop on Fault Tolerance:

Missing Component	Location / Logic Gap	Consequence
1. SS Replication Handler	<code>ss.c</code> (Main while(1) Loop)	The SS currently ignores the <code>SS_REPLICATE</code> or <code>SS_REPLICATE_WRITE</code> commands sent by the replication worker. Data is only written to the primary SS, meaning replication is not functional .
2. SS Failure Detection	<code>nm.c</code>	No mechanism (like a periodic heartbeat ping) to actively check if a registered SS is down or failing (<code>connect_error</code>) to notice.
3. SS Recovery Logic	<code>nm.c</code> & <code>ss.c</code>	No mechanism to match the duplicated stores when a failed SS reconnects. The SS registers itself again and it needs to pull data from its replicas to sync its state (as required by the spec).

Final Assessment

Your implementation is exceptionally strong on functional correctness, concurrency, and architecture. To achieve the remainder of the project requirements you must implement the SS handlers for replication and the NM logic for failure detection/recovery.

okay can you give me a detailed list of commands and expected outputs so i can confirm that everything other than the basic functionality is working

- ◆ This is a detailed set of commands designed to fully verify all implemented User Functionalities, System Requirements (Concurrency, Consistency), and Bonus Features (Folders , Checkpoints , Access Requests , Favorites).

► We will use two client terminals and ensure the Name Server remains consistent throughout concurrent operations.

Setup: Terminals and Cleanup

1. **Stop all servers** (Ctrl+C).
2. **Clean up state:** Delete old file data and metadata.

```
Bash

rm nm_metadata.dat favorites.dat
rm -rf ss_data/9002/* ````
```

3. Start Servers:

- **Terminal 1 (NM):** `./bin/name_server`
- **Terminal 2 (SS):** `./bin/storage_server 9002`

4. Start Clients:

- **Terminal 3 (Alice - Owner):** `./bin/client -> alice`

- Terminal 4 (Bob - Collaborator): `./bin/client` -> `bob`

Test Suite 1: Folders, Files, and Metadata

This verifies your new directory structure and basic file operations.

Terminal	Command	Expected Output	Verification Point
Alice	CREATEFOLDER /docs	Folder '/docs' created successfully.	CREATE (Folder)
Alice	CREATE /notes.txt	File created successfully.	CREATE (File)
Alice	WRITE /notes.txt 0	Sentence locked...	WRITE (Lock Acquired)
Alice	update> 0 First draft.	(No output)	
Alice	update> ETIRW	Write successful.	WRITE (Lock Relieved)
Alice	MOVE /notes.txt /docs	File moved	MOVE (File renamed)
Alice	READ /docs/notes.txt	===== File: /docs/notes.txt =====\nFirst draft.\n===== End of file =====	READ (File contents)
Alice	VIEWFOLDER /docs	== Contents of /docs ==\n[F] notes.txt\n	VIEWF (File inside)
Alice	CREATEFOLDER /docs/archive	Folder '/docs/archive' created successfully.	Nested Folders
Alice	VIEWFOLDER /docs	== Contents of /docs ==\n[F] notes.txt\n[D] archive\n	VIEWF (File folder)
Alice	MOVE /docs/notes.txt .	File moved	MOVE (File moved)
Alice	VIEWFOLDER /docs	== Contents of /docs ==\n(Empty)\n	Folder (Empty)
Alice	INFO notes.txt	Shows File: notes.txt , Owner: alice , Size: 2 words, 11 chars .	INFO (File info)

Test Suite 2: Concurrency & Locks

This verifies your core **WRITE** and **UNDO** logic, especially sentence locking.

Terminal	Command	Expected Output	Verification Point
Alice	WRITE notes.txt 0	Sentence locked. Enter updates...	Lock Acquired
Bob	WRITE notes.txt 0	Error: Sentence is locked by another user.	Sentence Lock (Corruption)
Bob	WRITE notes.txt 1	Error: Sentence index 1 out of range...	Index validation
Bob	READ notes.txt	First draft.	READ Allowed (Read !)
Alice	update> 1 but it needs more.	(No output)	
Alice	update> ETIRW	Write successful.	Lock Relieved (Metadata)
Alice	UNDO notes.txt	Undo successful.	UNDO (File reverts to original state)

Test Suite 3: Access Control & Requests

This verifies the `ACL` system and the new `Access Request` bonus feature.

Terminal	Command	Expected Output	Verification
Alice	CREATE secret.txt	File created successfully.	
Bob	READ secret.txt	Error: Access denied.	Default Denial
Bob	REQACCESS secret.txt -R	Request submitted (ID: 1)	REQACCESS
Bob	VIEWREQUESTS secret.txt	Error: Only the file owner can view requests	Owner-Only
Alice	VIEWREQUESTS secret.txt	Shows ID:1	User: bob
Alice	APPROVE 1	Access granted to bob	APPROVE
Bob	READ secret.txt	===== File: secret.txt =====\n\n===== End of file =====	Access Granted (works)
Alice	REMACCESS secret.txt bob	Access removed successfully!	REMACCESS
Bob	REQACCESS secret.txt -W	Request submitted (ID: 2)	
Alice	DENY 2	Request denied	DENY

📝 Test Suite 4: Checkpoints & Creative Feature

This verifies the complexity of the "EXEC" pattern and the two creative bonus features.

➡ Part 4a: Checkpoints (Verifies the streaming EXEC Pattern)

Pre-Requisite: Use `test.txt` from previous tests.

Terminal	Command	Expected Output	Verification
Alice	CHECKPOINT test.txt clean	Checkpoint 'clean' created successfully...	
Alice	WRITE test.txt 0	update> 0 THIS IS GARBAGE.	
Alice	update> ETIRW	Write successful.	
Alice	LISTCHECKPOINTS test.txt	Shows clean	Listing
Alice	VIEWCHECKPOINT test.txt clean	===== Checkpoint 'clean' for test.txt =====\nFirst draft.\n===== End of checkpoint =====	VIEWCHECKPOINT (not handled)
Alice	REVERT test.txt clean	File 'test.txt' successfully reverted...	REVERT

➡ Part 4b: Favorites System

Terminal	Command	Expected Output	Verification Point
Alice	FAVORITE /docs/test.txt	✗ File not found.	File Check (File is now at root as /test.txt)
Alice	FAVORITE notes.txt	★ Added 'notes.txt' to your favorites!	Favorite
Alice	FAVORITE notes.txt	⚠ File already in favorites!	Conflict Check
Bob	FAVORITE notes.txt	★ Added 'notes.txt' to your favorites!	Favorite Count
Alice	VIEWFAV	Shows notes.txt (1file)	VIEWFAV (User list)

Terminal	Command	Expected Output	Verification Point
Alice	TRENDING	Shows notes.txt (2 likes) at position #1	TRENDING (Global aggregation)

Conclusion

If your system passes Test Cases 1 through 4, you have successfully verified all requirements other than **Fault Tolerance (R Recovery)**. The stability shown in the concurrency and checkpoint tests proves the complex architectural changes were su

```
whats the issue
terminal 1
(base) jkb@jkb-HP-ProBook-450-15-6-inch-G10-Notebook-PC:~/Documents/sem3/osn/finalproject/pversion/course
./bin/name_server
Starting server...
[NM-Persist] Loading metadata from disk...
[NM-Persist] No metadata file found. Starting fresh.
[NM] Building Trie from loaded metadata...
[NM] Trie build complete.
[REPLICATION] Worker thread started for async fault tolerance
1. Socket created successfully (fd=4)
2. Socket bound to port 9001
3. Server is listening on port 9001...
Waiting for a client to connect...
4. Client connected! Handing off to a new thread...
Handshake received: S_INIT 127.0.0.1 9001 9101

Registered new Storage Server at 127.0.0.1:9101
Connection handler thread exiting.
4. Client connected! Handing off to a new thread...
Handshake received: S_INIT 127.0.0.1 9001 9101

Registered new Storage Server at 127.0.0.1:9101
Connection handler thread exiting.
4. Client connected! Handing off to a new thread...
Handshake received: S_INIT 127.0.0.1 9001 9101

Registered new Storage Server at 127.0.0.1:9101
Connection handler thread exiting.
4. Client connected! Handing off to a new thread...
Handshake received: C_INIT alice

4. Client connected! Handing off to a new thread...
Handshake received: C_INIT bob

[NM-Persist] Saving metadata to disk...
[NM-Persist] Save complete.
[NM] Folder created: /docs (idx=0)
SS failed to respond to CREATE
```

```
terminal 2
(base) jkb@jkb-HP-ProBook-450-15-6-inch-G10-Notebook-PC:~/Documents/sem3/osn/finalproject/pversion/cour
./bin/storage_server 127.0.0.1 9001 9101
[SS] Storage root path set to: ss_data/9101
SS starting... connecting to Name Server...
1. Connected to Name Server successfully.
Sending registration: S_INIT 127.0.0.1 9001 9101
2. Registered with Name Server successfully.
3. Spawning client-listener thread...
SS initialization complete. Main thread is now waiting for NM commands.
[SS-Listener] SS is now listening for clients on port 9101
^C
(base) jkb@jkb-HP-ProBook-450-15-6-inch-G10-Notebook-PC:~/Documents/sem3/osn/finalproject/pversion/cour
./bin/storage_server 9002
Usage: ./bin/storage_server <nm-ip> <nm-port> <client-port>
Example: ./bin/storage_server 127.0.0.1 9001 9101
(base) jkb@jkb-HP-ProBook-450-15-6-inch-G10-Notebook-PC:~/Documents/sem3/osn/finalproject/pversion/cour
./bin/storage_server 127.0.0.1 9001 9101
[SS] Storage root path set to: ss_data/9101
SS starting... connecting to Name Server...
1. Connected to Name Server successfully.
Sending registration: S_INIT 127.0.0.1 9001 9101
2. Registered with Name Server successfully.
3. Spawning client-listener thread...
SS initialization complete. Main thread is now waiting for NM commands.
[SS-Listener] SS is now listening for clients on port 9101
^C
(base) jkb@jkb-HP-ProBook-450-15-6-inch-G10-Notebook-PC:~/Documents/sem3/osn/finalproject/pversion/cour
./bin/storage_server 127.0.0.1 9001 9101
[SS] Storage root path set to: ss_data/9101
SS starting... connecting to Name Server...
1. Connected to Name Server successfully.
Sending registration: S_INIT 127.0.0.1 9001 9101
2. Registered with Name Server successfully.
3. Spawning client-listener thread...
SS initialization complete. Main thread is now waiting for NM commands.
[SS-Listener] SS is now listening for clients on port 9101
Name Server disconnected. Exiting.
```

```
terminal 3
(base) jkb@jkb-HP-ProBook-450-15-6-inch-G10-Notebook-PC:~/Documents/sem3/osn/finalproject/pversion/cour
./bin/client
Enter username: alice
Registered with Name Server as 'alice'.
> CREATEFOLDER /docs
Folder '/docs/' created successfully.
> CREATE /notes.txt
Unexpected response: 500

> CREATE notes.txt
Unexpected response:
```

>

- ◆ The primary issue is a **logic flaw in the Storage Server's registration and handling of disconnected clients/servers**, causing the Name Server to believe multiple storage servers are connected when only one is. This leads to the Name Server sending commands to a dead socket, resulting in a persistent "SS failed to respond" error.

Here is the detailed analysis:

1. The Core Bug: Zombie SS Registration

Your Name Server log shows that it registered the **same Storage Server three times** with the exact same details:

```
Handshake received: S_INIT 127.0.0.1 9001 9101
Registered new Storage Server at 127.0.0.1:9101
```

This suggests you restarted the Storage Server multiple times without the Name Server realizing the old connection was dead.

- **Problem:** When the **Storage Server** dies (you hit Ctrl+C three times in Terminal 2), the Name Server does not detect the loss of connection. The Name Server's `g_ss_list` contains three references to the same SS with the same client port (`9101`).
- **Consequence:** When Alice sends `CREATE /notes.txt`, the Name Server picks a round-robin index (say, index 2) and sends the `NM_CREATE` command to a File Descriptor (`ss_fd`) that is now **closed and invalid**. The `send()` succeeds momentarily, but the `read()` (line 469 in `nm.c`) fails, printing `SS failed to respond to CREATE`.

2. Secondary Bug: `CREATEFOLDER` and `CREATE` Conflicts

The SS log indicates a mix-up in command processing due to the **NM not checking the full response**.

- When you run `CREATEFOLDER /docs`, the SS *correctly* receives the command (via the now-dead socket), but it's quick.
- The subsequent `CREATE /notes.txt` fails with `Unexpected response: 500`. This happens because the Name Server receives the response from the Storage Server (because the connection is invalid).

3. The Immediate Fix (Required for Operation)

You must fix the way the Name Server manages the `g_ss_list` array.

Fix 1: SS Registration Cleanup (In `nm.c`)

When a Storage Server connects, the NM must check if that SS is already in the list by its **Client IP and Port**. If it is, the old entry is overwritten/updated, not appended.

Add a check function to `nm.c` and use it inside `handle_connection`:

```
C

// nm.c
// Add this helper function:
int find_ss_by_ip_port(const char* ip, int port) {
    for (int i = 0; i < g_ss_count; i++) {
        if (g_ss_list[i].client_port == port && strcmp(g_ss_list[i].ip, ip) == 0) {
            return i;
        }
    }
    return -1;
}

// Then, modify handle_connection (around line 1030 in your full nm.c file):
else if (strncmp(buffer, S_INIT, strlen(S_INIT)) == 0) {
    // --- It's a STORAGE SERVER ---
    char ip[INET_ADDRSTRLEN];
    int nm_port, client_port; // Added nm_port for completeness
```

```

// S_INIT <ip> <nmp_port> <client_port>
sscanf(buffer, "%*s %s %d %d", ip, &nmp_port, &client_port);

pthread_mutex_lock(&ss_list_mutex);

int existing_index = find_ss_by_ip_port(ip, client_port);

if (existing_index != -1) {
    // SS reconnecting. Overwrite old FD, keep same index.
    g_ss_list[existing_index].conn_fd = conn_fd;
    // The index and count remain the same.
    printf("Re-registered Storage Server at %s:%d\n", ip, client_port);
} else if (g_ss_count < MAX_SS) {
    // New SS. Append.
    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++;
    printf("Registered new Storage Server at %s:%d\n", ip, client_port);
} else {
    // Handle full SS list error (not critical for this bug)
}

pthread_mutex_unlock(&ss_list_mutex);
// ... (rest of the S_INIT block, sending RESP_OK and exiting thread)
}

```

By adding the re-registration logic, you ensure that even if the SS restarts, the NM's list is clean, and commands are sent to File Descriptor.

4. Code for Review (Final Implementation Check)

Since you provided all your code, I must check the logic for the prefix bug and the folder structure one last time.

Bug 1: CREATEFOLDER Prefix Match (The root cause of the NM_CREATE documents log)

- Your `nm.c`'s `handle_client_commands` has this faulty order (the original code you uploaded):

```

C

if (strncmp(cmd, C_REQ_CREATE, strlen(C_REQ_CREATE)) == 0) {
    do_create(client_fd, username, arg1);
}
// ... later ...
else if (strncmp(cmd, C_CREATE_FOLDER, strlen(C_CREATE_FOLDER)) == 0) {
    do_createfolder(client_fd, username, arg1);
}

```

- Since `C_CREATE` is a prefix of `C_CREATE_FOLDER`, the server ran `do_create` which sent `NM_CREATE documents` to the **blocking file on disk**.

Action: You must re-order this in your live code (this was the fix in the previous turn, but the logs indicate it might not have correctly).

```

C

// Correct Order for handling C_CREATE / C_CREATE_FOLDER
if (strcmp(cmd, C_CREATE_FOLDER) == 0) { // <-- Use strcmp for exact match
    // ... logic for CREATEFOLDER ...
}
else if (strcmp(cmd, C_REQ_CREATE) == 0) {
    // ... logic for CREATE ...
}

```

Correction: The `strcmp` check is safer than `strncmp` if the command length is not specified explicitly. However, the command fix is simply reordering.

Final Verdict: Apply the **Fix 1 (SS Registration Cleanup)** and ensure the **command order** in `nm.c`'s `handle_client_commands` reordered: `CREATEFOLDER` first, then `CREATE`. Then, **clean the SS data directory**.

can you go through the code and tell me in detail what change to make

- ◆ I've performed a detailed audit of your code and found three primary areas needing correction for the **Hierarchical Folder** without the "Prefix Hijack" error and to ensure full system consistency.
- The fix involves **reordering the command checks** in `nm.c` and updating the `MOVE` logic to be fully synchronous.

Step 1: Fix the Command Prefix Hijack (in `nm.c`)

The issue is that shorter command checks (like `C_REQ_CREATE`) match longer commands (like `C_CREATE_FOLDER`) because they are checked before `CREATE` and `VIEW`. We must move the specialized folder commands to the top of the `if/else if` chain.

→ Action: Reorder `handle_client_commands`

In `nm.c`, find the `handle_client_commands` function. Move the `CREATEFOLDER`, `MOVE`, and `VIEWFOLDER` blocks to the top so they are checked before `CREATE` and `VIEW`.

```
c

// nm.c, inside handle_client_commands (around line 1675 in your full code)

// --- CORRECTED ORDER ---
while (1) {
    // ... (Reading and parsing the command) ...

    // 1. FOLDER OPERATIONS (Must be checked first)
    if (strcmp(cmd, C_CREATE_FOLDER) == 0) {
        if (items_scanned < 2) { /* ... bad req ... */ } else {
            do_createfolder(client_fd, username, arg1);
        }
    }
    else if (strcmp(cmd, C_MOVE) == 0) {
        if (items_scanned < 3) { /* ... bad req ... */ } else {
            do_move(client_fd, username, arg1, arg2);
        }
    }
    else if (strcmp(cmd, C_VIEWFOLDER) == 0) {
        if (items_scanned < 2) { /* ... bad req ... */ } else {
            do_viewfolder(client_fd, username, arg1);
        }
    }
    // 2. FILE CREATE (This can now safely be next)
    else if (strcmp(cmd, C_REQ_CREATE) == 0) {
        do_create(client_fd, username, arg1);
    }
    // 3. REMAINING COMMANDS (Keep original order for the rest)
    else if (strcmp(cmd, C_REQ_ADD_ACC) == 0) {
        // ...
    }
    // ... (All other commands follow, including VIEW and VIEWERQUESTS, which should also be reordered)
}
```

Step 2: Implement Synchronous MOVE Logic (in nm.c and ss.c)

The current implementation of `do_move` in `nm.c` updates metadata, but the SS receives the `NM_MOVE` command in its slot causing a delay and possible failure. We need to implement a full request-acknowledge cycle.

→ Action 1: Modify `do_move` to Wait for SS

In `nm.c`, inside `do_move()`, you need to block the `file_map_mutex` until the SS confirms the rename succeeded.

C

```
// nm.c, inside do_move() (around line 680)

// ... after Step 5 (check collision) ...

// 6. SYNCHRONOUS SS UPDATE (Critical for consistency)
if (!g_file_map[file_index].is_directory) {
    int ss_index = g_file_map[file_index].primary_ss_index; // <-- Use primary_ss_index

    pthread_mutex_lock(&ss_list_mutex);
    int ss_fd = g_ss_list[ss_index].conn_fd;
    pthread_mutex_unlock(&ss_list_mutex);

    char cmd_buf[MAX_MSG_LEN];
    char clean_old[MAX_PATH_LEN], clean_new[MAX_PATH_LEN];

    // Use clean paths for the SS command
    nm_to_ss_path(filename, clean_old);
    nm_to_ss_path(new_path, clean_new);

    sprintf(cmd_buf, MAX_MSG_LEN, "%s %s %s\n", NM_MOVE, clean_old, clean_new);

    // Send command
    if (send(ss_fd, cmd_buf, strlen(cmd_buf), 0) < 0) {
        // ... Error handling for send failure ...
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // Wait for ACK (BLOCKS metadata update until successful SS operation)
    char ss_resp[MAX_MSG_LEN];
    memset(ss_resp, 0, MAX_MSG_LEN);
    if (read(ss_fd, ss_resp, MAX_MSG_LEN - 1) <= 0) {
        // ... Error handling for read failure ...
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }

    // Check ACK
    if (strncmp(ss_resp, RESP_OK, 3) != 0) {
        // SS failed (e.g., disk error/404 from SS). Forward error and ABORT.
        send(client_fd, ss_resp, strlen(ss_resp), 0);
        pthread_mutex_unlock(&file_map_mutex);
        return;
    }
}

// 7. SS Succeeded. Update Metadata safely.
// ... (rest of the metadata update logic) ...
```

→ Action 2: Add NM_MOVE Handler to ss.c

In `ss.c`, inside the main `while(1)` loop where you handle NM commands, you need a handler for `NM_MOVE` that performs sends a clear ACK.

```

C

// ss.c, inside main() while(1) loop

else if (strncmp(cmd, NM_MOVE, strlen(NM_MOVE)) == 0) {
    char new_path[MAX_PATH_LEN];
    sscanf(nm_buffer, "%*s %s %s", filename, new_path);

    snprintf(log_msg, MAX_MSG_LEN, "REQ: NM_MOVE, OLD: %s, NEW: %s", filename, new_path);
    log_event(log_msg);

    char old_full_path[MAX_PATH_LEN];
    char new_full_path[MAX_PATH_LEN];
    snprintf(old_full_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);
    snprintf(new_full_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, new_path);

    // Ensure destination folder exists before moving (handles nesting)
    ensure_directory_exists(new_path);

    // Move the file
    if (rename(old_full_path, new_full_path) == 0) {
        printf("Renamed %s to %s\n", filename, new_path);
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_OK); // ACK Success
    } else {
        perror("rename failed");
        snprintf(resp_buf, MAX_MSG_LEN, "%s\n", RESP_SRV_ERR); // ACK Failure
    }

    write(g_nm_fd, resp_buf, strlen(resp_buf)); // Send ACK back to NM
    continue;
}

```

Step 3: SS File Creation Safety (in `ss.c`)

The fix for the `MOVE` command relies on `CREATE` ensuring subdirectories exist. We need to ensure that the `NM_CREATE` command uses the `ensure_directory_exists` helper function.

→ Action: Check `NM_CREATE`

In `ss.c`, inside the main `while(1)` loop, find the `NM_CREATE` handler and add the directory helper call:

```

C

// ss.c, inside main() while(1) loop

if (strncmp(cmd, NM_CREATE, strlen(NM_CREATE)) == 0) {
    // ... logs ...

    // --- ADD THIS LINE ---
    ensure_directory_exists(filename); // Ensure path (e.g., 'docs/notes.txt') exists on disk
    // -----

    snprintf(local_path, MAX_PATH_LEN, "%s/%s", g_ss_root_path, filename);

    FILE *fp = fopen(local_path, "w");
    // ... rest of the logic

```

we used to have an a
that isn't working
what happened

- Based on your provided code, the `ADDACCESS` command is **fully implemented in your Name Server** (`nm.c`), and the client can send the request.
- However, the command might not be working due to an issue with **how it's being executed or where it's located in your code**. If you have been moving code blocks around, the `ADDACCESS` function may have been accidentally placed in a section of the code that was not correctly compiled.

Here is a quick check of where the logic exists and what might be preventing it from running:

1. Concurrency and Command Definition Check

The logic for `ADDACCESS` is well-defined in your code:

Component	Code Status	Verification
Protocol (<code>protocol.h</code>)	✓ Defined	Defines <code>C_REQ_ADD_ACC</code> ("C_ADD_ACC").
Client (<code>client.c</code>)	✓ Defined	Handles the user input, parses the filename, user, and permission flags (<code>-R</code> or <code>-W</code>).
Name Server (<code>nm.c</code>)	✓ Defined	The function <code>do_add_access</code> exists and contains the logic to lock the <code>file_map_mutex</code> , find the owner permission, and update the <code>acl_list</code> .

2. Where the Error Is Likely Occurring

Since the code exists, the problem is almost certainly due to the **command order/parsing** in `nm.c`'s `handle_client_commands`. The `do_add_access` function is now defined but not correctly registered in the main loop.

→ Action 1: Check `handle_client_commands` Order

You need to verify that the `if/else if` chain in `nm.c`'s `handle_client_commands` is correctly checking for `C_REQ_ADD_ACC`:

- If you are using `strcmp(cmd, C_REQ_ADD_ACC)` (exact match), this is correct.
- If you are using `strncpy`, it might be getting incorrectly matched by a different command.

→ Action 2: Check Client Parsing

Your client code uses `strcmp(cmd, "ADDACCESS")`. This should match the user input.

However, in `nm.c`, the check must match the protocol command string:

```
C

// Check this line in nm.c handle_client_commands:

else if (strcmp(cmd, C_REQ_ADD_ACC) == 0) { // <-- USE strcmp
    if (items_scanned < 4) { /* ... bad req ... */ } else {
        do_add_access(client_fd, username, arg1, arg2, arg3[1]);
    }
}
```

Ensure you are using `strcmp` and not `strncpy` for `C_REQ_ADD_ACC` in the command handler.

→ Action 3: Check Function Definition Location

Verify that the full `do_add_access` function body is physically present in `nm.c` and is not commented out or outside of the

3. Recommended Fix

If the command isn't working, re-integrate the commands using the safer `strcmp` method.

In `nm.c`, confirm that the code block for `C_REQ_ADD_ACC` and `C_REQ_Rem_ACC` uses `strcmp` for the primary check (since they have no prefix overlap with others).

```
C

// Find this section in your nm.c handle_client_commands loop:

// ----- ADD ACCESS -----
else if (strcmp(cmd, C_REQ_ADD_ACC) == 0) {
    if (items_scanned < 4) { /* ... handle error ... */ } else {
        do_add_access(client_fd, username, arg1, arg2, arg3[1]);
    }
}

// ----- REMOVE ACCESS -----
else if (strcmp(cmd, C_REQ_Rem_ACC) == 0) {
    if (items_scanned < 3) { /* ... handle error ... */ } else {
        do_rem_access(client_fd, username, arg1, arg2);
    }
}
```

give me commands i can run to test

That's an important test! The `ADDACCESS` command is crucial for collaboration. Since your system uses usernames for access, two separate users to confirm the permissions are correctly granted and then enforced.

Here is a step-by-step test plan using two clients, Alice (the **Owner**) and Bob (the **Collaborator**).

Setup: Terminals and Initial State

1. Ensure your Name Server (`nm.c`) and Storage Server (`ss.c`) are running.
2. Open **two client terminals**:
 - **Terminal A (Owner):** Log in as `alice`
 - **Terminal B (Collaborator):** Log in as `bob`

Test Case 1: Granting and Enforcing Read Access

This test verifies that Bob can be granted read permission and that he can then successfully execute a read-only operation previously denied.

Terminal	Command	Expected Output	Verification
Alice	CREATE file1.txt	File created successfully.	Creates file.
Bob	READ file1.txt	Error: Access denied.	Initial Denial (Bob has no access).
Alice	ADDACCESS -R file1.txt bob	Access granted successfully.	Grants Read permission.
Bob	READ file1.txt	===== File: file1.txt =====\n\n===== End of file =====	Permission works.
Bob	WRITE file1.txt 0	Error: Access denied.	Write Denied.

Terminal	Command	Expected Output	Verification Point
Alice	INFO file1.txt	Shows Access List section includes bob (R) .	INFO Check

📝 Test Case 2: Granting Write Access and Revoking

This test verifies the upgrade to write access and confirms that `REMACCESS` completely revokes all privileges.

Terminal	Command	Expected Output	Verification Point
Alice	ADDACCESS -W file1.txt bob	Access granted successfully.	Grants Write Access
Bob	WRITE file1.txt 0	Sentence locked. Enter updates...	Write Works (Lock is acquired)
Bob	update> 0 Bob's first edit.	(No output)	
Bob	update> ETIRW	Write successful.	Commit Works
Alice	INFO file1.txt	Shows Access List section includes bob (W) .	INFO Check
Alice	REMACCESS file1.txt bob	Access removed successfully!	REMACCESS
Bob	READ file1.txt	Error: Access denied.	Access Revoked (All access removed)

```
alproject/pversion/course-project-codeshark$ ./bin/client
Enter username: alice
Registered with Name Server as 'alice'.
> VIEWFOLDER /docs
Contents of '/docs/':
/docs/notes.txt,FILE;/docs/archive,FOLDER;
> INFO notes.txt
File not found.
> INFO /docs/notes.txt
==== File Info ====
File: /docs/notes.txt
Owner: alice
Size: 0 words, 0 chars
Created: 2025-11-19 15:02:50
Last Modified: 2025-11-19 15:03:03
Last Accessed: 2025-11-19 15:03:06
---Access List---
alice (Owner)

> WRITE notes.txt 0
Error: >
> CREATE abc.txt
File created successfully.
> WRITE abc.txt 0
Sentence locked. Enter updates (e.g., '3 new_word') or 'ETIRW'
update> ETIRW
Write successful.
> ADDACCCESS
```

```
Unsupported command (for now). Try CREATE <filename>, RE  
> ADDACCESS  
Unsupported command (for now). Try CREATE <filename>, RE  
> ADDACCESS -R file1.txt bob  
Unexpected response: 404  
  
> ADDACCESS file1.txt bob -R  
Unexpected response: 404  
  
> ADDACCESS file1.txt -R bob  
Unexpected response: 404  
  
>
```

- ◆ It looks like you've encountered two primary issues: **Path Consistency** (using the wrong filename format) and **Command P** wrong command structure).
- ⌚ Here is a detailed breakdown of the errors and the exact step-by-step fixes required in your current code setup.

1. 📁 Path Consistency Errors (The Primary Issue)

The system is now treating the file name, **including its path**, as its unique identifier in the Name Server's Trie.

🔴 Analysis of Errors:

- Error 1:

```
> VIEWFOLDER /docs  
Contents of '/docs':  
/docs/notes.txt,FILE;/docs/archive,FOLDER;  
> INFO notes.txt  
File not found.
```

The file `notes.txt` exists, but the Name Server (NM) searches for the exact key `"notes.txt"` in the Trie. The file is a the key `/docs/notes.txt`. The NM is unaware that `notes.txt` exists at all without the full path.

- Error 2:

```
> WRITE notes.txt 0  
Error: >
```

This fails because the `WRITE` command relies on `do_write` in `nm.c` finding the file index via the path provided. Since `"notes.txt"`, the NM returns "File not found" (404), which the client translates to a general error.

🛠 Fix 1: Enforcement of Absolute Paths

You must ensure that clients refer to files using their **full, absolute path** (e.g., starting with `/`).

1. **Client (`client.c`):**

- **Action:** When the user types `READ notes.txt`, the client should prepend the current directory or simply ensure t `/`.

2. **Name Server (`nm.c`):**

- **Action:** All file operations (`do_read`, `do_write`, `do_info`, etc.) must use a helper function to ensure the `filename` normalized to start with a `/` for consistency, as your `CREATEFOLDER` uses absolute paths.

2. ❌ Command Structure Errors (ADDACCESS)

The `ADDACCESS` command is failing because you are using the wrong syntax for parsing arguments on the client side.

🔴 Analysis of Errors:

- **Error:**

```
> ADDACCESS -R file1.txt bob
Unexpected response: 404
```

The client logs show the command as `ADDACCESS -R file1.txt bob`.

- **Root Cause:** The `ADDACCESS` command takes the filename, followed by the target user, and then the flag (or relies on early). Your **client-side command parser** is splitting the string based on spaces, and the order is getting mixed up relative expected by the `C_REQ_ADD_ACC` protocol string in the Name Server.

The protocol expects: `C_ADD_ACC <filename> <username> <perm_flag>`

The client is sending: `C_ADD_ACC -R file1.txt bob` (wrong order).

🛠 Fix 2: Update `ADDACCESS` in `client.c`

You need to match the argument parsing in the client to the way the function is designed in the Name Server.

In `client.c`, find the `ADDACCESS` handler and adjust how `filename`, `user`, and `perm` are extracted using `strtok`.

Correct Parsing Flow in `client.c`:

1. `cmd = "ADDACCESS"`
2. `arg1 = Flag (-R or -W)`
3. `arg2 = Filename`
4. `arg3 = Username`

```
c

// client.c, inside the 'else if (strcmp(cmd, "ADDACCESS") == 0)' block

// Current flawed parsing logic:
// char *flag = strtok(NULL, " ");
// char *filename = strtok(NULL, " ");
// char *user = strtok(NULL, " ");

// Corrected Parsing Logic:
char *flag = strtok(NULL, " ");
char *filename = strtok(NULL, " ");
char *user = strtok(NULL, " ");

// You need to confirm the order of arguments in your specific nm.c function call.
// Based on the protocol documentation:
// C_ADD_ACC <filename> <username> <perm_flag>

// If the NM is expecting <filename> <user> <flag>, your client should send:
// C_ADD_ACC file1.txt bob -R
// (This means the input ORDER must be file, user, flag)
```

Assuming your `nm.c` expects the input in the client example format (`ADDACCESS -R file1.txt bob`):

The fix lies in how you construct the final request string sent to the NM:

```
c
```

```

// client.c, inside the 'else if (strcmp(cmd, "ADDACCESS") == 0)' block

char *flag = strtok(NULL, " ");      // -R or -W
char *filename = strtok(NULL, " ");  // file1.txt
char *user = strtok(NULL, " ");     // bob
// ... (null checks) ...

// The NM expects: C_ADD_ACC <filename> <username> <perm_char>
// But the client input is organized around the flag first.
// The safe way to send this is:
snprintf(request, sizeof(request),
    "%s %s %c\n",
    C_REQ_ADD_ACC, filename, user, flag[1]); // flag[1] is 'R' or 'W'
// This ensures the filename and user are sent first, followed by the R/W character.

```

can you tell me what is correct format for addaccess command and also let me know if i need to fix a

- ◆ Based on your project documentation and code, here is the correct format for the `ADDACCESS` command and a detailed analysis of critical bugs you need to fix in your implementation.



1. ✓ Correct Command Format

The correct format for the `ADDACCESS` command, based on your client-side parsing logic, requires the arguments to be in the correct order:

Component	Example	Notes
Command	ADDACCESS	Literal command.
File Path	/docs/file1.txt	Must use the full path (the unique identifier).
User	bob	The username receiving the access.
Permission Flag	-R or -W	-R grants read-only; -W grants read and write.



Correct Usage Example:

```
> ADDACCESS /docs/file1.txt bob -R
```

2. ✗ Critical Code Mistakes to Fix

Your current implementation has two major issues that are causing the errors in your logs: [Order of Arguments in the Client](#) and [Directory Paths in the Client](#).

A. Fix 1: Argument Order Mismatch (Client to NM)

Your client terminal input is being parsed into a format (`<flag> <filename> <user>`) that is different from how the Name Manager's `(do_add_access)` is defined (`<filename> <user> <flag>`).

- **Your Client Logic (`client.c`):**

```

C

// Reads input: ADDACCESS -R file1.txt bob
char *flag = strtok(NULL, " ");      // Gets "-R"

```

```

char *filename = strtok(NULL, " "); // Gets "file1.txt"
char *user = strtok(NULL, " "); // Gets "bob"

// Sends: C_ADD_ACC file1.txt bob -R <-- This is correct format for NM.

```

The error lies in how you are providing the command arguments.

- **Fix: Use the Correct Order for Input.** Since your client code *is* sending the correct message structure to the Name Server (`<filename> <user> <flag>`), the simplest fix is to type the command in the order expected by the parser:

In your terminal, always type: ADDACCESS <filename> <user> <flag>

```
> ADDACCESS file1.txt bob -R // Corrected command format
```

B. Fix 2: Path Normalization for Absolute Paths (In `client.c`)

The reason your `MOVE` and `INFO` commands failed earlier is that your system enforces **absolute paths** (starting with `/`). Your command is failing with `404` because you didn't include the `/` in the path, and the Name Server looks for `"file1.txt"` and it doesn't exist.

Your `CREATEFOLDER` uses the absolute path (`/docs`), so your file references must as well.

1. **Bug:** When you type `ADDACCESS file1.txt...`, the client sends `file1.txt`. The NM looks for the non-existent file `/file1.txt`.
2. **Fix:** You need to add a small normalization step in the client to ensure every file path starts with `/` if the user is operating relative to the current directory.

In `client.c`, find the parsing logic for file operations (like `READ`, `WRITE`, `INFO`, `ADDACCESS`) and apply this simple helper function to normalize the request to the Name Server.

```

C

// client.c, inside main() (or in a helper function)

// Example adjustment for ADDACCESS:
char *filename = strtok(NULL, " ");
char *user = strtok(NULL, " ");
char *flag = strtok(NULL, " ");

// --- ADD THIS LOGIC ---
char norm_filename[MAX_PATH_LEN];
if (filename[0] != '/') {
    snprintf(norm_filename, sizeof(norm_filename), "/%s", filename);
} else {
    strcpy(norm_filename, filename);
}
// -------

// Now, send the normalized path:
snprintf(request, sizeof(request),
        "%s %s %s %c\n",
        C_REQ_ADD_ACC, norm_filename, user, flag[1]);

```

After applying these fixes, the following command will work:

```
> ADDACCESS /file1.txt bob -R
```