NFS Server Initialization

Initialize the Naming Server (NM): The first step is to initialize the Naming Server, which serves as the central coordination point in the NFS. It is responsible for managing the directory structure and maintaining essential information about file locations. The IP address and port of the Naming Server can be assumed to be known publicly so that it can be provided to Clients and Storage servers while registering. Note that since the IP address is dynamic, you have to inform each storage server and client about it when they attempt to register.
Do not hardcode IP address to 127.0.0.1 and port since we will test on multiple laptops. (2 for SS, 1 for NM, 1 for client. You have to open multiple SS on each laptop)

Initialize Storage Server 1 (SS_1): Each Storage Server (SS) is responsible for storing files and interacting with both the Naming Server and clients. Initialization of SS_1 involves several sub-steps:
Upon initialization, SS_1 sends vital details about its existence to the Naming Server. This information includes:
IP address: To facilitate communication and location tracking.
Port for NM Connection: A dedicated port for direct communication with the Naming Server.
Port for Client Connection: A separate port for clients to interact with SS_1.
List of Accessible Paths: A comprehensive list of file and folder paths that are accessible on SS_1. The Storage server, while registering should be able to decide which path has to be made accessible to the NFS. You are free to give this as a command-line input or any other way.
Initialize SS_2 to SS_n: Following the same procedure as SS_1, additional Storage Servers (SS_2 to SS_n) are initialized, each providing their details to the Naming Server.
write this in C. give me two codes one for server and one for storage servers

Memory updated

Here's a basic implementation for initializing a Naming Server (NM) and a Storage Server (SS) in C. The Naming Server listens for registration requests from Storage Servers, while each Storage Server registers itself and shares its details.

The example includes the essential components for this setup and can be expanded with additional functionality.

Naming Server Code (NamingServer.c)

This code sets up a Naming Server that listens for connections from Storage Servers, receives their details, and stores the information.

```c
// NamingServer.c
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 9000
#define BUFFER_SIZE 1024

typedef struct {
    char ip[INET_ADDRSTRLEN];
    int portNM;
    int portClient;
    char paths[BUFFER_SIZE];
} StorageServerInfo;

void handle_registration(int client_socket) {
    StorageServerInfo ss_info;
    if (recv(client_socket, &ss_info, sizeof(ss_info), 0) < 0) {
        perror("Failed to receive storage server info");
        return;
    }

    printf("Received registration from Storage Server:\n");
    printf("IP Address: %s\n", ss_info.ip);
    printf("NM Port: %d\n", ss_info.portNM);
    printf("Client Port: %d\n", ss_info.portClient);
    printf("Accessible Paths: %s\n", ss_info.paths);
}

int main() {
    int server_socket, client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size = sizeof(client_addr);

    // Create socket
    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Configure server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind and listen
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }
    if (listen(server_socket, 5) < 0) {
        perror("Listen failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }
    printf("Naming Server is running on port %d...\n", PORT);

    // Accept connections
    while (1) {
        client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &addr_size);
        if (client_socket < 0) {
            perror("Accept failed");
            continue;
        }
        printf("Connection established with Storage Server\n");
        handle_registration(client_socket);
        close(client_socket);
    }

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

Storage Server Code (StorageServer.c)

This code sets up a Storage Server that connects to the Naming Server and registers itself with its details.

```c
// StorageServer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define NM_PORT 9000
#define BUFFER_SIZE 1024

typedef struct {
    char ip[INET_ADDRSTRLEN];
    int portNM;
    int portClient;
    char paths[BUFFER_SIZE];
} StorageServerInfo;

void register_with_nm(const char* nm_ip, int nm_port, StorageServerInfo* ss_info) {
    int sock;
    struct sockaddr_in nm_addr;

    // Create socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set up NM address
    nm_addr.sin_family = AF_INET;
    nm_addr.sin_port = htons(nm_port);
    if (inet_pton(AF_INET, nm_ip, &nm_addr.sin_addr) <= 0) {
        perror("Invalid NM IP address");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Connect to Naming Server
    if (connect(sock, (struct sockaddr*)&nm_addr, sizeof(nm_addr)) < 0) {
        perror("Connection to NM failed");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Send Storage Server info
    if (send(sock, ss_info, sizeof(*ss_info), 0) < 0) {
        perror("Failed to send Storage Server info");
    }
    printf("Registered with Naming Server at %s:%d\n", nm_ip, nm_port);
    close(sock);
}

int main(int argc, char* argv[]) {
    if (argc < 5) {
        fprintf(stderr, "Usage: %s <NM_IP> <Port for NM> <Port for Client> <Accessible Pat
hs>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Initialize Storage Server info
    StorageServerInfo ss_info;
    strncpy(ss_info.ip, argv[1], INET_ADDRSTRLEN);
    ss_info.portNM = atoi(argv[2]);
    ss_info.portClient = atoi(argv[3]);
    strncpy(ss_info.paths, argv[4], BUFFER_SIZE);
    // Register with Naming Server
    register_with_nm("127.0.0.1", NM_PORT, &ss_info);

    return 0;
}
```

Usage

1. Compile both programs:

   bash

   ```
   gcc NamingServer.c -o NamingServer
   gcc StorageServer.c -o StorageServer
   ```

2. Start the Naming Server on a designated machine:

   bash

   ```
   ./NamingServer
   ```

3. Start a Storage Server instance on a different machine, passing IP, port, and accessible paths:

   bash

   ```
   ./StorageServer <NM_IP> <Port_for_NM> <Port_for_Client> <Accessible_Paths>
   ```

This setup provides a simple communication between the Naming Server and Storage Servers, allowing Storage Servers to register and send their accessible paths.

is this using TCP

Yes, the provided code uses TCP sockets. The `` `SOCK_STREAM` `` parameter in the `` `socket()` `` function indicates TCP communication. Here's a breakdown:

- In both `` `NamingServer.c` `` and `` `StorageServer.c` ``, the socket is created with `` `socket(AF_INET, SOCK_STREAM, 0)` ``, which specifies TCP as the transport protocol.
- The Naming Server (`` `NamingServer.c` ``) listens for incoming connections, which is a typical setup for TCP servers, as it maintains a persistent connection to allow reliable data transfer.
- The Storage Server (`` `StorageServer.c` ``) connects to the Naming Server using `` `connect()` ``, also specific to TCP, allowing it to establish a reliable connection to the Naming Server and send its registration data.

This code is set up for TCP, ensuring reliable, ordered communication between the Naming and Storage Servers.

in naming server, do not close the connection use threads to handle multiple clients at same time. every 5 sec keep sending like message to storage server to simulate data transfer

To modify the Naming Server to handle multiple Storage Servers simultaneously and maintain persistent connections, we can use threads for each incoming connection. Additionally, we'll set up a "heartbeat" mechanism to send a "like" message every 5 seconds to each connected Storage Server, simulating ongoing data transfer.

Here's the updated code:

Updated Naming Server Code (NamingServer.c)

This version uses POSIX threads to handle each Storage Server connection independently, keeping the connection open. Each thread will continuously send a "like" message to its connected Storage Server every 5 seconds.

c

```c
// NamingServer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 9000
#define BUFFER_SIZE 1024

typedef struct {
    int socket;
    struct sockaddr_in addr;
} ClientInfo;

typedef struct {
    char ip[INET_ADDRSTRLEN];
    int portNM;
    int portClient;
    char paths[BUFFER_SIZE];
} StorageServerInfo;

void* handle_storage_server(void* arg) {
    ClientInfo* client_info = (ClientInfo*)arg;
    int client_socket = client_info->socket;
    StorageServerInfo ss_info;

    // Receive initial registration data
    if (recv(client_socket, &ss_info, sizeof(ss_info), 0) <= 0) {
        perror("Failed to receive storage server info");
        close(client_socket);
        free(client_info);
        return NULL;
    }
    printf("Storage Server registered:\n");
    printf("IP: %s\n", ss_info.ip);
    printf("NM Port: %d, Client Port: %d\n", ss_info.portNM, ss_info.portClient);
    printf("Accessible Paths: %s\n\n", ss_info.paths);

    // Send "like" message every 5 seconds
    while (1) {
        const char* like_message = "like";
        if (send(client_socket, like_message, strlen(like_message), 0) < 0) {
            perror("Failed to send 'like' message");
            break;
        }
        printf("Sent 'like' to Storage Server at %s:%d\n", ss_info.ip, ss_info.portNM);
        sleep(5);
    }

    close(client_socket);
    free(client_info);
    return NULL;
}

int main() {
    int server_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size = sizeof(client_addr);

    // Create the server socket
    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Configure server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind and listen
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }
    if (listen(server_socket, 5) < 0) {
        perror("Listen failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }
    printf("Naming Server is running on port %d...\n", PORT);

    // Accept connections
    while (1) {
        int client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &addr_si
ze);
        if (client_socket < 0) {
            perror("Accept failed");
            continue;
        }

        // Allocate and initialize client info
        ClientInfo* client_info = malloc(sizeof(ClientInfo));
        client_info->socket = client_socket;
        client_info->addr = client_addr;

        // Create a thread to handle the Storage Server
        pthread_t thread_id;
        if (pthread_create(&thread_id, NULL, handle_storage_server, client_info) != 0) {
            perror("Failed to create thread");
            close(client_socket);
            free(client_info);
        }
        pthread_detach(thread_id);  // Detach the thread to clean up automatically
    }
    close(server_socket);
    return 0;
}
```

Explanation

1. **Multithreading**: Each new Storage Server connection spawns a thread using `pthread_create`. Each thread runs the `handle_storage_server` function independently, allowing the Naming Server to handle multiple connections at the same time.
2. **Persistent Connection**: The `client_socket` remains open, maintaining a persistent connection with each Storage Server.
3. **Heartbeat/Simulated Data Transfer**: Every 5 seconds, the Naming Server sends a "like" message to the connected Storage Server. This simulates ongoing data transfer.

With this setup, the Naming Server can handle multiple Storage Servers in parallel, and each connection receives periodic messages. The server is now ready for real-time communication in a distributed file system.

modify client also

To modify the Storage Server (client) to maintain a persistent connection with the Naming Server and handle periodic "like" messages, we need to implement a loop that continuously listens for incoming messages from the Naming Server.

Here’s the updated Storage Server code that:

- Establishes a persistent connection to the Naming Server.
- Listens for the periodic "like" messages from the Naming Server.

Updated Storage Server Code (StorageServer.c)

```c
// StorageServer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define NM_PORT 9000
#define BUFFER_SIZE 1024

typedef struct {
    char ip[INET_ADDRSTRLEN];
    int portNM;
    int portClient;
    char paths[BUFFER_SIZE];
} StorageServerInfo;
```

```
// Function to connect and register with the Naming Server
void connect_to_nm(const char* nm_ip, int nm_port, StorageServerInfo* ss_info) {
    int sock;
    struct sockaddr_in nm_addr;

    // Create socket
```

use only port number and ip address in arg no need for more

Memory updated

Here's the simplified Storage Server code, using only the IP address and port number as arguments. This version establishes a persistent connection to the Naming Server, listens for the "like" messages, and prints them when received.

Updated Storage Server Code (StorageServer.c)

```c
// StorageServer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 1024

// Function to connect to the Naming Server and listen for messages
void connect_to_nm(const char* nm_ip, int nm_port) {
    int sock;
    struct sockaddr_in nm_addr;
    char buffer[BUFFER_SIZE];

    // Create the socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set up Naming Server address
    nm_addr.sin_family = AF_INET;
    nm_addr.sin_port = htons(nm_port);
    if (inet_pton(AF_INET, nm_ip, &nm_addr.sin_addr) <= 0) {
        perror("Invalid IP address");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Connect to Naming Server
    if (connect(sock, (struct sockaddr*)&nm_addr, sizeof(nm_addr)) < 0) {
        perror("Connection to Naming Server failed");
        close(sock);
        exit(EXIT_FAILURE);
    }
    printf("Connected to Naming Server at %s:%d\n", nm_ip, nm_port);

    // Listen for "like" messages
    while (1) {
        memset(buffer, 0, BUFFER_SIZE);
        int bytes_received = recv(sock, buffer, BUFFER_SIZE, 0);
        if (bytes_received <= 0) {
            printf("Connection closed by Naming Server\n");
            break;
        }
        printf("Received from Naming Server: %s\n", buffer);
    }

    close(sock);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <NM_IP> <NM_Port>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Parse command-line arguments
    const char* nm_ip = argv[1];
    int nm_port = atoi(argv[2]);

    // Connect to the Naming Server and listen for messages
    connect_to_nm(nm_ip, nm_port);

    return 0;
}
```

Explanation

1. **Arguments**: Only the IP address and port number of the Naming Server are taken as command-line arguments.
2. **Connection**: The Storage Server connects to the Naming Server and maintains a persistent connection.
3. **Message Listening**: In a loop, the Storage Server listens for incoming "like" messages from the Naming Server. When a message is received, it's printed to the console.
4. **Disconnection**: If the Naming Server closes the connection, the Storage Server detects it and exits the loop.

This modified client setup aligns with your preference for a simpler interface while handling the required persistent connection and message handling.

```
ClientInfo* client_info = malloc(sizeof(ClientInfo));
        client_info->socket = client_socket;
        client_info->addr = client_addr;
        printf("");
```

write the debug stamtnt for this only give me the snupppet

Here's a debug statement snippet that prints the `client_info` structure details:

```c


  ClientInfo* client_info = malloc(sizeof(ClientInfo));
  client_info->socket = client_socket;
  client_info->addr = client_addr;

  printf("Debug: Client connected - Socket: %d, IP: %s, Port: %d\n",
         client_info->socket,
         inet_ntoa(client_info->addr.sin_addr),
         ntohs(client_info->addr.sin_port));
```

This debug statement will output:

- The socket descriptor.
- The IP address of the client.
- The port number of the client.

Make sure to include `<arpa

ChatGPT can make mistakes. Check important info.