P2P Detailed Report

Introduction

The Tracker-Client Server is a comprehensive C++ application designed to manage user authentication, group creation, and file sharing functionalities using socket programming. Leveraging multi-threading with POSIX threads (`pthreads`), the server efficiently handles multiple client connections concurrently, ensuring robust and scalable operations. This detailed report dives into the intricacies of each function within the server, elucidating their roles, interactions, and the overall workflow facilitated by socket programming.

Architecture Overview

The Tracker Server operates as the central coordinator for clients, managing users, groups, and shared files. The primary components include:

- User Management: Handles user creation, login, and logout operations.
- Group Management: Facilitates the creation of groups, managing membership requests, and handling group-related actions.
- File Management: Manages file uploads, listings, and sharing permissions within groups.(it handles download requests too, but I haven't implemented it)
- Concurrency Control: Employs multi-threading and mutexes to handle multiple clients simultaneously while ensuring thread safety.
- Socket Communication: Utilizes TCP sockets for reliable client-server communication.

Detailed Function Descriptions

1. Utility Functions

a. AlertPrompt

The `AlertPrompt` function serves as a utility for error handling and messaging. It outputs error messages to the standard error stream (`stderr`). Depending on the `usePerror` flag, it either uses the `perror` function to provide detailed error messages based on the global `errno` value or simply outputs a custom error message.

b. locate, substring, myAtoi

These custom utility functions perform common string operations and conversions:
- locate : Finds the position of a specified delimiter within a string.
- substring : Extracts a substring from a given string based on a start position and length.
- myAtoi : Converts a string to an integer, returning `0` if the conversion fails due to invalid input. It is my own version of stoi().

2. Data Structures

a. ArrayList Class

The `ArrayList` class is a template-based dynamic array implementation, just like done in programming language Java,it provides functionalities similar to `std::vector`. It manages a resizable array of elements of a specified type, supporting operations like addition, removal, retrieval, and dynamic resizing. The class ensures efficient memory management through constructors, destructors, and copy/move semantics.

b. User Class

The `User` class represents a user within the system, encapsulating user-related information and functionalities. It includes attributes such as `user_id`, `password`, `logged_in` status, and `creation_time`. Key methods allow for password verification, updating login status, and retrieving account creation timestamps.

c. Group Class

The `Group` class represents a group within the system, managing group-related data and operations. Attributes include `group_id`, `owner`, lists of `members` and `pending_requests`, and `creation_time`. Methods facilitate adding join requests and accepting them, thereby managing group membership dynamically.

d. File Class

The `File` class represents a file within a group, managing file metadata and sharing information. It includes attributes like `file_name`, `file_size`, `sha1` hash for integrity verification, lists of `chunk_sha1s`, and `shared_by_users`. Methods allow for adding chunk hashes, managing shared users, and verifying if a user is sharing the file.

3. Global Data Structures and Synchronization

The server maintains several global data structures to manage users, groups, client mappings, and group files:

- users : A map linking `user_id`s to `User` objects.
- groups : A map linking `group_id`s to `Group` objects.
- client_user_map : A map linking client socket descriptors to `user_id`s.
- group_files : A map linking `group_id`s to lists of `File` objects.

To ensure thread safety across multiple client threads, mutexes (`users_mutex`, `groups_mutex`, `clients_mutex`) are employed to synchronize access to these shared resources.

4. Command Handling

a. Command Enum and getCommand Function

An enumeration of possible commands (`CREATE_USER`, `LOGIN`, `LOGOUT`, etc.) is defined to categorize incoming client commands. The `getCommand` function maps command strings received from clients to the corresponding enum values, facilitating streamlined processing within the server.

b. handleCommand Function

The `handleCommand` function processes incoming commands by invoking the appropriate handler functions based on the identified command type. It returns a boolean indicating whether to continue processing commands from the client (`true`) or to terminate the client connection (`false`).

5. Command Handler Functions

Each command received from clients is managed by a dedicated handler function that performs the necessary operations and updates the server's data structures accordingly. Below is an overview of these handler functions:

a. handleCreateUser

Handles the creation of new user accounts. It validates the command format, ensures that the `user_id` is unique, and adds the new user to the `users` map if validation passes. It responds with appropriate success or error messages based on the outcome.

b. handleLogin

Manages user authentication. It verifies the existence of the `user_id`, checks the provided password, ensures the user isn't already logged in, updates the user's login status, and maps the client socket to the `user_id` upon successful login. It communicates success or error messages back to the client.

c. handleLogout

Facilitates user logout by updating the user's `logged_in` status and removing the mapping between the client socket and the `user_id`. It ensures that only logged-in users can perform logout operations and responds accordingly.

d. handleCreateGroup

Enables logged-in users to create new groups. It validates the command, ensures the `group_id` is unique, creates a new `Group` object with the user as the owner, and adds it to the `groups` map. Success or error messages are communicated based on the operation's result.

e. handleJoinGroup

Allows users to send join requests to existing groups. It verifies the user's login status, checks if the group exists, ensures the user isn't already a member or hasn't already sent a join request, and adds the user's request to the group's `pending_requests` if eligible. It provides appropriate feedback to the user.

f. handleLeaveGroup

Permits users to leave a group. If the user is the group owner, it handles ownership transfer to another member or deletes the group if no members remain. For regular members, it removes the user from the group's members list and updates file sharing permissions if necessary. It ensures the user is part of the group before performing the operation.

g. handleListRequests

Allows group owners to view pending join requests for their groups. It verifies the user's ownership of the group and retrieves the list of pending requests, formatting them into a response message for the client.

h. handleAcceptRequest

Enables group owners to accept join requests from users. It validates the command format, ensures the requester is the group owner, and moves the specified user from the `pending_requests` to the `members` list. It provides feedback based on whether the acceptance was successful.

i. handleListGroups

Provides a list of all available groups on the server. It retrieves the list of `group_id`s from the `groups` map and formats them into a response message for the client.

j. handleUploadFile

Facilitates the upload of files to specific groups. It validates the command format, ensures the user is a member of the target group, checks for file existence within the group, manages chunk SHA1 hashes for file integrity, and updates the `group_files` map with the new or updated `File` object. It communicates the outcome of the upload operation to the client.

On the client side, we tokenize the absolute file path to extract the file name, and send it to tracker along with the target group_id, SHA1 of whole file (extracted using functions in openssl library), followed by SHA1s of chunks of 512 KB as stated in question statement. Then the tracker takes over.

k. handleListFiles

Lists all files available within a specified group. It verifies the user's membership in the group, retrieves the list of files from the `group_files` map, and formats them into a response message for the client.

l. handleStopShare

Allows users to stop sharing a particular file within a group. It validates the command format, ensures the user is a member of the group, checks if the user is currently sharing the specified file, and updates the `shared_by_users` list accordingly. It provides appropriate feedback based on the operation's success.

6. Client Handling

a. clientHandler

The `clientHandler` function is responsible for managing communication with a connected client. Each client connection is handled by a separate thread running this function. Its primary responsibilities include:

1. Connection Management : Adds the client to the `connected_clients` list in a thread-safe manner.
2. Data Reception : Continuously receives data from the client using `recv()`.
3. Command Parsing : Parses the received data into command tokens.
4. Command Processing : Invokes `handleCommand` to process the command and generate an appropriate response.
5. Response Transmission : Sends the response back to the client using `send()`.
6. Session Termination : Detects when a client wishes to terminate the session (e.g., via the `quit` command) and performs cleanup operations, including updating user login status and removing the client from the `connected_clients` list.
7. Error Handling : Detects and handles errors during communication, ensuring that the client session is terminated gracefully in case of issues.

B. download chunks

After requesting the tracker for metadata about download of the file, the client gets the requested data after it has been authenticated. The client after receiving metadata, tries to establish connection with all the peers which are on the share list of the file. Once established

the piece selection algorithm (eg Round Robin) selects the piece and begins the download. Each chunk once downloaded gets verified with the calculated SHA1 of the piece to maintain the authenticity of the file. I have added debugging statements so that we can see the download in real time. Once all the chunks have been downloaded, the complete file is built and the SHA1 of the whole is checked. (Note : Only clients with complete file can send file data)(Note 2: There are limitations to file download, I have checked pdf, txt, jpg and png files. These all are getting downloaded while .mp4 have failed)

7. Server Command Handling

a. serverCommandHandler

The `serverCommandHandler` function operates in a separate thread to listen for server-side commands entered via the console. Its primary role is to handle administrative commands, most notably the `shutdown` command, which initiates a graceful shutdown of the server. Its workflow includes:

1. Command Listening : Continuously prompts the server administrator for commands.
2. Command Processing : Detects the `shutdown` command and initiates the shutdown sequence.
3. Client Notification : Sends a shutdown message to all connected clients to inform them of the impending termination.
4. Resource Cleanup : Closes all client sockets, clears the `connected_clients` list, and closes the main server socket to unblock the `accept()` call.
5. Shutdown Initiation : Sets the `server_running` flag to `false` to terminate the main server loop and ensure the server exits gracefully.
6. Feedback : Provides feedback to the server administrator regarding the status of the shutdown process.

8. Signal Handling

a. signalHandler

The `signalHandler` function is designed to catch interrupt signals (e.g., `SIGINT` triggered by `Ctrl+C`) to ensure that the server can terminate gracefully upon receiving such signals. Its responsibilities include:

1. Signal Detection : Identifies when an interrupt signal is received.
2. Shutdown Initiation : Displays a message indicating that a shutdown has been initiated.
3. Resource Cleanup : Closes the main server socket to unblock any pending `accept()` calls.
4. Flag Update : Sets the `server_running` flag to `false` to signal the main server loop to terminate.
5. Graceful Termination : Ensures that all resources are properly released and that the server exits without abrupt termination, preventing potential data corruption or resource leaks.

9. Main Function

The `main` function serves as the entry point of the Tracker Server. It orchestrates the initialization of server components, sets up socket communication, and manages incoming client connections. Its primary workflow includes:

1. Signal Registration : Registers the `signalHandler` to manage interrupt signals.
2. Argument Validation : Ensures that the correct number of command-line arguments (`<tracker_info.txt>` and `<tracker_no>`) are provided.
3. Socket Initialization : Creates a TCP socket for network communication.
4. Binding : Associates the socket with the loopback IP address (`127.0.0.1`) and a port number determined by `5000 + tracker_no`.
5. Listening : Puts the server socket into a listening state to accept incoming client connections.
6. Command Thread Creation : Spawns the `serverCommandHandler` thread to handle server-side commands.
7. Connection Acceptance Loop : Enters a loop where it continuously accepts new client connections using `accept()`.
For each accepted connection:
   - Logs the connection details.
   - Allocates memory for the new client socket descriptor.
   - Creates a detached thread running the `clientHandler` function to manage the client session.
8. Shutdown Handling : Detects when a shutdown has been initiated either via the `shutdown` command or an interrupt signal. Closes the main server socket and ensures that all resources are released before terminating the server.
9. Final Feedback : Provides confirmation that the server has been closed gracefully.

<u>Socket Programming Implementation</u>

The Tracker Server employs socket programming to establish and manage communication between clients and the server. Below is an in-depth explanation of the socket-related functionalities:

1. Socket Creation

The server initializes a TCP socket using the `socket()` function, specifying the IPv4 address family (`AF_INET`) and the TCP protocol (`SOCK_STREAM`). This socket serves as the primary endpoint for network communication, enabling the server to send and receive data over the network reliably.

2. Binding

Once the socket is created, it is bound to a specific IP address (`127.0.0.1`, representing the localhost) and a port number calculated as `5000 + tracker_no`. The `bind()` function associates

the socket with these network parameters, ensuring that the server listens for incoming connections on the designated address and port.

3. Listening

After binding, the server places the socket into a passive listening state using the `listen()` function. This prepares the socket to accept incoming client connections. The backlog parameter (`5` in this case) specifies the maximum number of pending connections that can be queued before new connections are refused.

4. Accepting Connections

The server enters a loop where it continuously accepts new client connections using the `accept()` function. For each incoming connection:
- The server retrieves the client's address information.
- It logs the connection details for monitoring purposes.
- Allocates memory for the client's socket descriptor.
- Spawns a new detached thread running the `clientHandler` function to manage the client session independently.

This design ensures that the server can handle multiple clients concurrently without blocking the main listening thread.

5. Data Transmission

a. Receiving Data

Within each `clientHandler` thread, the server uses the `recv()` function to receive data from the client. This data typically consists of commands issued by the client, which the server parses and processes accordingly. The received data is stored in a buffer for further processing.

b. Sending Data

After processing a client's command, the server formulates an appropriate response message. This response is sent back to the client using the `send()` function, ensuring that the client is informed of the outcome of their request (e.g., success, error messages).

6. Graceful Shutdown

The server incorporates mechanisms to ensure a graceful shutdown, preventing abrupt termination that could lead to data loss or resource leaks. There are two primary methods to initiate shutdown:

a. Server-Initiated Shutdown

Triggered by entering the `shutdown` command in the server console, the server performs the following steps:
1. Client Notification : Sends a shutdown message to all connected clients, informing them of the impending termination.
2. Socket Closure : Closes all client sockets to terminate their connections.
3. Resource Cleanup : Clears the list of connected clients and closes the main server socket to unblock any pending `accept()` calls.
4. Flag Update : Sets the `server_running` flag to `false` to exit the main server loop.

b. Signal-Based Shutdown

Triggered by interrupt signals (e.g., pressing `Ctrl+C`), the `signalHandler` function initiates a shutdown sequence by:
1. Display Message : Notifies that a shutdown has been initiated.
2. Socket Closure : Closes the main server socket to stop accepting new connections.
3. Flag Update : Sets the `server_running` flag to `false` to exit the main server loop.

In both cases, the server ensures that all resources are properly released and that clients are notified before termination.

7. Concurrency Management

To handle multiple clients efficiently, the server employs multi-threading:
- Client Threads : Each client connection is managed by a separate thread, allowing the server to process multiple client requests concurrently.
- Detached Threads : Client threads are detached, meaning they run independently without requiring the main thread to join them. This approach simplifies thread management and enhances scalability.
- Mutexes : To prevent race conditions and ensure data integrity, mutexes (`users_mutex`, `groups_mutex`, `clients_mutex`) are used to synchronize access to shared resources such as user data, group data, and the list of connected clients.

Flow Chart Description

Given the text-based medium, the flow chart is described step-by-step to illustrate the server's workflow. Below is a sequential outline representing the flow of operations within the Tracker Server.

1. Server Initialization

- Start : The server begins execution.
- Register Signal Handler : Sets up the `signalHandler` to manage interrupt signals like `SIGINT`.

- Parse Command-Line Arguments : Validates and extracts necessary arguments (`<tracker_info.txt>`, `<tracker_no>`).
- Create Socket : Initializes a TCP socket for network communication.
- Bind Socket : Associates the socket with the loopback IP address (`127.0.0.1`) and port (`5000 + tracker_no`).
- Listen on Socket : Start listening for incoming client connections.
- Start Server Command Thread : Launches the `serverCommandHandler` thread to handle server-side commands.

2. Accepting Client Connections

- Loop While Server Running:
  - Accept Connection : Uses `accept()` to accept a new client connection.
  - Log Connection Details : Records information about the connected client.
  - Spawn Client Thread : Creates a detached thread running `clientHandler` to manage the client session.
  - Repeat : Continues accepting new connections.

3. Handling Client Requests (clientHandler)

- Receive Data : Uses `recv()` to receive commands from the client.
- Parse Command : Splits the received data into tokens for processing.
- Process Command : Invokes `handleCommand` to execute the appropriate handler function.
- Send Response : Sends the outcome of the command processing back to the client using `send()`.
- Check for Termination :
  - If the `quit` command is received or an error occurs, terminate the client session.
- Cleanup:
  - Updates the user's login status if necessary.
  - Removes the client from the `connected_clients` list.
  - Closes the client socket.

4. Processing Commands (handleCommand and Handlers)

- Identify Command : Determines the command type using the `getCommand` function.
- Invoke Handler : Calls the corresponding handler function (e.g., `handleCreateUser`, `handleLogin`).
- Generate Response : Based on the handler's outcome, formulates an appropriate response message.
- Return Status : Indicates whether to continue the client session or terminate.

5. Server Command Handling (serverCommandHandler)

- Wait for Server Commands : Continuously listens for input from the server console.

- Process Commands :
  - If `shutdown` Command :
    - Notify Clients : Sends a shutdown message to all connected clients.
      - Close Client Sockets : Iterates through the `connected_clients` list and closes each client socket.
    - Clear Client List : Empties the `connected_clients` vector.
    - Close Server Socket : Closes the main server socket to unblock the `accept()` call.
    - Terminate Server Loop : Sets `server_running` to `false` to exit the main loop.
  - Else :
    - Displays an error message for unknown commands.
    - Continues listening.

6. Graceful Shutdown (signalHandler)

- Detect `SIGINT` Signal :
  - Displays a shutdown initiation message.
  - Closes the main server socket to stop accepting new connections.
  - Sets the `server_running` flag to `false` to terminate the main loop.
- Proceed with Shutdown :
  - The main loop detects the shutdown flag and proceeds to close remaining resources.
  - End : The server completes the shutdown process and terminates.

7. Shutdown Completion

- Close Main Socket : Ensures the main server socket is closed.
- Display Shutdown Message : Informs that the server has been closed gracefully.
- Terminate : Exits the server application.

<u>Concurrency and Threading</u>

Multi-Threading

The server employs multi-threading to handle multiple clients simultaneously:

- **Client Handler Threads**: Each client connection is managed by a separate thread running the `clientHandler` function. This allows the server to process multiple client requests concurrently without blocking the main listening thread.

- **Server Command Thread**: A dedicated thread (`serverCommandHandler`) listens for server-side commands (e.g., `shutdown`) from the console, ensuring that server management tasks do not interfere with client handling.

<u>Thread Safety</u>

To maintain data integrity across multiple threads, the server utilizes mutexes:

- users_mutex : Protects access to the `users` map, ensuring safe creation, login, and logout operations.

- groups_mutex : Secures the `groups` map and related group operations, preventing race conditions during group creation, joining, or leaving.

- clients_mutex : Guards the `connected_clients` list, ensuring accurate tracking of active client connections.

Detached Threads

Client handler threads are detached using `pthread_detach`, allowing them to run independently without requiring explicit joins. This approach simplifies thread management and enhances scalability by preventing resource leaks associated with joinable threads.

Graceful Shutdown Mechanisms

The server supports two primary methods for graceful shutdown:

1. Server-Initiated Shutdown

Triggered by entering the `shutdown` command in the server console.

Steps :
1. Receive Command : The `serverCommandHandler` thread detects the `shutdown` command.
2. Notify Clients : Sends a shutdown message to all connected clients.
3. Close Client Sockets : Iterates through the `connected_clients` list and closes each client socket.
4. Clear Client List : Empties the `connected_clients` vector.
5. Close Server Socket : Closes the main server socket to unblock the `accept()` call.
6. Terminate Server Loop : Sets `server_running` to `false` to exit the main loop.
7. Finalize Shutdown : Ensures all resources are released and the server terminates gracefully.

2. Signal-Based Shutdown

Triggered by interrupt signals, such as pressing `Ctrl+C` (`SIGINT`).

Steps :
1. Catch Signal : The `signalHandler` function intercepts the `SIGINT` signal.
2. Initiate Shutdown : Displays a shutdown initiation message.
3. Close Server Socket : Closes the main server socket to unblock the `accept()` call.

4. Set Shutdown Flag : Updates `server_running` to `false` to terminate the main loop.
5. Proceed with Shutdown : The main server loop detects the shutdown flag and proceeds to close remaining resources.

Benefits :
- Ensures that all client connections are properly terminated.
- Prevents abrupt termination, which could lead to data corruption or resource leaks.

Error Handling and Validation

Robust error handling mechanisms are embedded throughout the server to ensure stability and reliability.

1. Command Validation

- Argument Checks : Each command handler verifies the correct number of arguments.
- Authorization Checks : Ensures that operations are performed by authorized users (e.g., only group owners can accept join requests).
- Data Validation : Validates data types and formats (e.g., ensuring `file_size` is a positive integer, `sha1` hashes are of correct length).

2. Socket Operations

- Return Value Checks : Verifies the return values of socket functions (`socket()`, `bind()`, `listen()`, `accept()`, `recv()`, `send()`) for errors.
- Error Logging : Utilizes the `AlertPrompt` function to log errors and provide informative messages.
- Specific Error Handling : Handles specific errors like `EBADF` to suppress unnecessary error messages during shutdown.

3. Resource Management

- Memory Management : Ensures that dynamically allocated memory (e.g., for client sockets) is properly freed.
- RAII Principles : Employs Resource Acquisition Is Initialization (RAII) where applicable to manage resource lifetimes.
- Memory Leak Prevention : Deletes dynamically allocated objects (e.g., `User`, `Group`) to prevent memory leaks.

4. User Feedback

- Clear Messaging : Sends clear and concise error messages back to clients for invalid operations.

- Confirmation Messages : Provides confirmation messages upon successful operations (e.g., "User created.", "Login successful.").

Conclusion

The Tracker Server exemplifies a robust and scalable network application, effectively leveraging socket programming and multi-threading to manage complex functionalities such as user authentication, group management, and file sharing. Its modular design, comprehensive command set, and meticulous error handling ensure efficient and reliable operations. The integration of mutexes for thread safety and graceful shutdown mechanisms further enhance its resilience and maintainability. This architecture serves as a solid foundation for applications requiring centralized coordination and management of users and shared resources.

For further enhancements, consider integrating persistent storage (e.g., databases) to maintain user and group data across server restarts, implementing more sophisticated authentication mechanisms, and expanding file management capabilities.

References

1. Think & Learn Youtube Channel (For hands-on socket programming)
2. GeeksForGeeks Socket Programming in C++ (https://www.geeksforgeeks.org/socket-programming-in-cpp/)
3. P2P Networking with Bit-Torrent (Jahn Arne, Lars Erik & Birkeland)
4. Other resource materials for brief introduction to sub-topics and implementations
5. SHA1 knowledge (https://www.geeksforgeeks.org/sha-1-hash-in-java/)