Client.cpp

The client.cpp is designed to handle client-side communication with a tracker and peers in a distributed file-sharing system. It includes functionality for uploading and downloading files, computing file integrity checks, and communicating with other clients (seeders) using sockets. Below is a summary of the main components:

1. **Socket Creation & Management**:
    - The program sets up TCP sockets using the `socket()`, `bind()`, and `connect()` system calls. The client first connects to a tracker to send commands like uploading and downloading files.
    - A `socket_binding()` function is used to bind the client's socket for listening to incoming connections from peers.
2. **File Hashing**:
    - The program uses SHA-1 hashing to compute file integrity. The file is split into chunks (default size 512KB), and each chunk is hashed. The function `compute_sha_hash()` reads chunks of the file and computes their SHA-1 hash, which is later used for verifying file integrity after download.
3. **Commands Execution**:
    - The program processes user commands in the `exe_cmnds()` function, including the ability to upload or download files.
    - **Upload**: The client calculates SHA-1 hash keys for the file chunks and sends metadata (file size, hash keys) to the tracker.
    - **Download**: The client requests the file from a peer (seeder), verifies the file integrity by checking the downloaded file’s hash with the original, and then updates the tracker once the download is successful.
4. **Active Tracker Detection**:
    - The client can detect an active tracker by reading from a tracker information file. The function `get_all_tracker_info()` helps in selecting an active tracker from a list of IP addresses and ports.
5. **Error Handling**:
    - Extensive error handling is implemented, with checks for failures during socket creation, file reading, and SHA-1 computations.
6. **Concurrency**:
    - Threads are used to allow the client to handle multiple tasks simultaneously, like uploading/downloading files while keeping the tracker communication alive.

Tracker.cpp

The tracker.cpp implements a Tracker Server for a peer-to-peer (P2P) file-sharing system. The tracker acts as a central coordinator, managing user accounts, groups, and file metadata. It facilitates communication between clients (peers) by handling user authentication, group management, and file upload/download requests.

1. **Libraries and Dependencies**
    - Includes headers like `<iostream>`, `<vector>`, `<unordered_map>`, `<thread>`, and others for basic functionalities (standard libraries).
    - Utilizes `<netinet/in.h>`, `<arpa/inet.h>`, and `<sys/socket.h>` for socket programming (network libraries).
    - `<thread>` and `<mutex>` to handle multiple client connections simultaneously. Uses `<openssl/sha.h>` for SHA hashing, ensuring data integrity.
2. **Data Structures**
    - **User Management**:
        - `log_creds`: Maps user IDs to passwords.
        - `isloggedin`: Tracks the login status of users.
        - `user_details`: Stores user IP addresses and ports.
    - **Group Management**:
        - `grp_owners`: Associates group IDs with their owners.
        - `grp_mems`: Maintains members of each group.
        - `grp_pend_reqs`: Keeps track of pending join requests for groups.
        - `user_grps`: Lists groups each user belongs to.
    - **File Management**:
        - `files_info` Structure: Contains metadata about uploaded files, including file name, path, seeder details, SHA keys, and size.
        - `grp_files`: Lists files available in each group.
        - `user_files`: Tracks files each user has uploaded or downloaded.
3. **Core Functionalities**
    - **User Operations**:
        - **Create User (`create_user`)**: Registers a new user by adding their credentials to `log_creds`.
        - **Login (`login`)**: Authenticates users and updates their login status in `isloggedin`.
        - **Logout (`logout`)**: Logs out users and updates their status.
    - **Group Operations**:
        - **Create Group (`create_group`)**: Allows logged-in users to create new groups and assigns ownership.
        - **Join Group (`join_group`)**: Lets users request to join existing groups, adding their requests to `grp_pend_reqs`.

        - **Leave Group (`leave_group`)**: Enables users to leave groups, updating group memberships and handling ownership transfer if necessary.
        - **List Groups (`list_groups`)**: Displays all available groups.
        - **List Pending Requests (`list_requests`)**: Shows pending join requests for a group to the group owner.
        - **Accept Request (`accept_request`)**: Allows group owners to accept user join requests, adding them to the group members.
    - **File Operations**:
        - **Upload File (`upload_file`)**: Enables users to upload files to a group. It records file metadata, including SHA keys for integrity.
        - **Download File (`download_file`)**: Provides users with file details and seeder information to facilitate downloading from other peers.
        - **List Files (`list_files`)**: Lists all files available within a specific group.
    - **Utility Functions**:
        - **SHA Hashing (`compute_sha_hash`)**: Although not fully shown in this snippet, it likely computes SHA hashes for files to ensure data integrity.
        - **Command Management (`manage_client_cmnds`)**: Parses and executes commands received from clients.
        - **Tracker Connection (`connect_tracker`)**: Establishes a connection to the tracker based on provided IP and port.
        - **Argument Evaluation (`evaluate_args`)**: Validates and processes command-line arguments for tracker configuration.
4. **Concurrency and Thread Safety**
    - The tracker handles multiple clients simultaneously by spawning a new thread (`manage_client_cmnds`) for each incoming client connection.
    - **Mutexes**: Utilizes several `std::mutex` objects (e.g., `mt_log_creds`, `mt_logged_in`) to protect shared data structures from concurrent access, ensuring thread safety.
5. **Error Handling**
    - The code includes comprehensive error checks after critical operations like socket creation, binding, reading from sockets, and file operations.
    - Informative messages are sent back to clients in case of errors, such as invalid commands, authentication failures, or insufficient permissions.
6. **User Interaction**
    - **Commands**: The tracker supports various commands sent by clients, including user management (`create_user`, `login`, `logout`), group management (`create_group`, `join_group`, etc.), and file operations (`upload_file`, `download_file`).
    - **Help Command (`help`)**: Provides clients with a list of available commands and their usage.

Workflow

1. **Startup**:
    - The tracker reads its configuration from a file (`tracker_info.txt`), which includes IP addresses and ports of available trackers.
    - It selects the appropriate tracker based on the provided tracker number and establishes a listening socket to accept client connections.
2. **Client Connection Handling**:
    - For each incoming client connection, the tracker spawns a new thread to handle that client's commands independently.
    - This ensures that multiple clients can interact with the tracker concurrently without blocking each other.
3. **Command Processing**:
    - The `manage_client_cmnds` function reads commands from the client, parses them, and invokes the corresponding handler functions (e.g., `func_create_user`, `func_login_user`).
    - Responses are sent back to the client based on the outcome of each command.
4. **Group and File Management**:
    - Users can create and manage groups, upload files to groups, and request to download files from other group members.
    - The tracker maintains detailed metadata about groups and files to facilitate these operations.