Peer-to-Peer Distributed File Sharing System

Course: Advanced Operating Systems (AOS)
Language: C/C++
Modules: Client and Tracker
Features: Synchronized Trackers, Parallel Downloading, Piece Selection Algorithms, File Integrity Checking (SHA1)

---

1. Overview

This report presents the development of a peer-to-peer (P2P) distributed file-sharing system implemented in C/C++. The system is designed to facilitate file sharing among clients through a tracker-based architecture. The primary components of the system are the **Client** and **Tracker** modules.

Client Module: Responsible for interacting with the tracker and other peers, handling file uploads and downloads.
Tracker Module: Manages client connections, user validation, and maintains metadata about files and peers.

2. Key Features

- **Synchronized Trackers**: The system supports multiple trackers for redundancy. Clients can connect to different trackers based on configuration details provided at runtime.
- **Parallel Downloading**: Clients download files by connecting to multiple peers simultaneously, improving download speed and reliability.
- **Custom Piece Selection Algorithm**: An algorithm optimizes piece selection for downloading, reducing the likelihood of bottlenecks and ensuring file integrity.
- **File Integrity Verification**: The system employs SHA1 hashing for file pieces and entire files, ensuring data consistency and security.

---

3. Modules and Function Overview

3.1 Client Module (`client.cpp`)

The client module is the core of the system's P2P interactions, handling both the download and upload processes, maintaining connections with peers, and verifying data integrity.

1. **`splitString`**:
   Splits a given string based on a delimiter and returns a vector of substrings. It is used throughout the module for parsing commands and file paths.

2. **`getIPAndPortFromFileName`**:
   Reads a configuration file to extract the IP address and port of the tracker. This function allows the client to establish connections dynamically based on the tracker's details.

3. **`file_size`**:
   Computes the size of a given file, essential for managing file transfers and ensuring that the correct amount of data is sent or received.
4. **`singlehash`**:
   Computes the SHA1 hash for a single data block. This function is vital for validating the integrity of file pieces during download.
5. **`bitvector`**:
   Manages a bit vector representing the availability of file pieces. This aids in implementing the custom piece selection algorithm by tracking which parts of the file are available or required.
6. **`combinehash`**:
   The combinehash function computes a combined hash for a file by reading it in chunks. It first checks if the file exists and determines its size. The file is read in segments, accumulating data, and calculating a hash for each segment using the singlehash function. After processing all segments, it trims any trailing characters from the final hash and returns the combined hash string.
7. **`peer_connection`**:
   The peer_connection function connects to a peer using a specified IP address and port. It sends commands and receives data related to file chunks, handling errors during communication. The function also ensures data integrity by checking the received chunks against expected hash values. Finally, it updates the status of the file chunks and closes the socket connection.
8. **`upload_file`**:
   The upload_file function handles the file upload process by first checking if the file is already uploaded. If not, it marks the file as uploaded and generates both a piecewise hash and a SHA256 hash of the file content. It then constructs a detailed string containing file information, including its size and hashes, and sends this data to the server. After sending, it reads the server's response and updates the chunk information accordingly.
9. **`thread_func_2`**:
   Manages multi-threaded tasks related to file transfers. It handles parallel upload or download processes to optimize performance.
10. **`thread_func`**:
    Another multi-threaded function that manages specific client-server interactions, helping the client manage simultaneous tasks efficiently.
11. **`piecewiseAlgo`**:
    The piecewiseAlgo function manages the downloading of a file in chunks from multiple peers. It first initializes threads for each peer connection and checks if all chunks are available. After creating a destination file, it randomly selects peers to download missing chunks in parallel. The function waits for all download threads to complete, verifies the download status, and sends a request to a peer for the current file path. It ensures efficient utilization of available peers to reconstruct the entire file.

12. **`downloadFile`**:
    The downloadFile function initiates a file download request by preparing and sending the file details to the specified socket. Upon receiving a response, it checks if the file is available; if not, it outputs an error message. If the file is found, it extracts the list of peers that have the file and their corresponding piece hashes. The function then calls piecewiseAlgo to manage the downloading of the file in chunks from the available peers.
13. **`connection`**:
    The connection function manages communication with a server based on various commands. It reads the server's response and handles different commands such as login, logout, file upload, file download, and group management. Depending on the command, it displays relevant messages, processes file uploads and downloads, and lists groups or files. The function also checks for specific conditions, like invalid arguments or previously downloaded files, and responds accordingly. Overall, it orchestrates the interaction between the client and the server for file-sharing operations.
14. **`handleconnection`**:
    The handleconnection function processes client requests based on specific input commands. It reads the command from the client and splits it into components for further handling. If the command is "current_chunk," it retrieves the specified chunk of a file and sends it to the client. For the "current_chunk_vector_details" command, it sends the status of the file's chunks. If the command is "current_path_file," it returns the file path to the client. Finally, the function closes the client socket after processing the request.
15. **`server_func`**:
    Implements server-like behavior for the client when it acts as a file provider. It listens for incoming connections from other peers requesting file pieces.
16. **`main` Function**:
    The main function initializes a peer-to-peer client application by checking command-line arguments for the peer's IP and port and the tracker's address. It creates a socket for communication and starts a server thread for handling incoming connections. After establishing a connection to the tracker, it enters a loop to read user input, validate login status, and send commands to the server. Depending on the command, it invokes the connection function to handle server responses. Finally, it closes the socket upon exiting the loop.

3.2 Tracker Module (`tracker.cpp`)

The tracker module manages the overall coordination of the P2P network, maintaining user information, managing client connections, and facilitating file transfers.

1. **`splitString`**:
   A utility function similar to the client module, used for parsing configuration data and user input.
2. **`ispath`**:
   Verifies the validity of file paths, ensuring that files and directories exist before proceeding with read or write operations.

3. **`getIPAndPortFromFileName`**:
   Retrieves the IP address and port information from a configuration file, allowing the tracker to set up its network socket for client connections.
4. **`close_server`**:
   Manages the orderly shutdown of the tracker by closing sockets and terminating threads, ensuring that no resources are left hanging.
5. **`check_login_validate`**:
   Validates user login information, ensuring that only authenticated users can access the tracker and share files.
6. **`download_File`**:
   The download_File function handles download requests for files within a specified group. It first checks if the group exists, if the requested file path is valid, and if the client is a member of the group. If any of these checks fail, it sends an appropriate error message. If all checks pass, it sends a "Downloading" message, retrieves the file details, and checks if the file is available in the group's upload list. If the file is found, it constructs a response with the file's size and details of available peers, sending this information back to the client.
7. **`upload_File`**:
   The upload_File function manages file upload requests from clients within a specific group. It first checks if the group exists, the file path is valid, and if the client is a member of the group. If any check fails, it sends an appropriate error message. Upon successful validation, it reads the file data from the socket, processes the file details, and updates the group's upload list with the file information, including its size and hash. Finally, it confirms the upload with a success message.
8. **`connection`**:
   The connection function handles communication between the server and a client connected through a socket. It continuously reads client commands, such as creating a user, logging in, downloading or uploading files, joining groups, and managing group memberships. Based on the command received, it performs the corresponding actions, validates input, and responds with appropriate messages. The function also maintains the client's session state, such as checking if the user is logged in or part of a group. It manages group requests, file uploads, and downloads while ensuring that all operations adhere to the established rules for group membership and file sharing. If an invalid command is received or the connection is lost, it gracefully handles the situation by closing the socket and updating the user's session status.
9. **`main` Function**:
   The main function initializes a tracker server for a peer-to-peer file-sharing system. It retrieves the tracker IP and port from a provided filename, creates a TCP socket, and configures it for reuse. The server binds to the specified address and listens for incoming client connections, spawning new threads to handle each client. Additionally, it starts a separate thread to monitor for shutdown commands, ensuring the server can gracefully exit when required.

---

4. Implementation Highlights

- **Socket Programming**: The system extensively uses socket programming to establish and manage connections between the client, tracker, and peers. This ensures efficient data transfer and communication.
- **Multi-threading**: Both the client and tracker use threads to handle multiple connections simultaneously, improving the system's ability to scale and manage multiple tasks concurrently.
- **File Integrity (SHA1 Hashing)**: The system uses SHA1 hashing to validate file integrity, ensuring that any file shared or downloaded is complete and uncorrupted.
- **Configuration Files**: The client and tracker both read from configuration files for dynamic setup. This design choice enhances flexibility, allowing the system to adapt to different network environments and tracker configurations.

5. Conclusion

The developed P2P distributed file-sharing system demonstrates the integration of various advanced operating system concepts, such as socket programming, multi-threading, and file integrity checking. The modular structure and use of custom algorithms ensure the system is optimized for efficiency and reliability. This project showcases the complexities of building a distributed network system and the importance of designing robust error handling and modularized code for such applications.