AOS Assignment 3
Peer-to-Peer Distributed File Sharing System
Monsoon 2025

International Institute of Information Technology, Hyderabad
Assignment Deadline: September 30, 2025 11:59 PM

1. Introduction:

This assignment focuses on implementing a **Peer-to-Peer (P2P) file-sharing client** with centralized tracker support. The client allows users to create accounts, login, manage groups, upload files, and download files from other peers in a secure and reliable manner. It communicates with a central tracker to authenticate users, maintain metadata about shared files, and coordinate group-based file-sharing activities.

The client also functions as a **peer server**, enabling direct file transfers between users by dividing files into fixed-size pieces and supporting parallel downloads from multiple seeders. It incorporates features like session caching, automatic tracker failover, and hash-based integrity verification to ensure **reliability, security, and efficient file distribution**.

Overall, the system combines the advantages of centralized control (for authentication and coordination) with decentralized file sharing (for speed and scalability), providing a robust and practical P2P file-sharing solution.

2. Tracker Synchronization:

The tracker achieves synchronization primarily through a **persistent log file** called sync_log.txt. Every operation that modifies the tracker’s state, such as user management, group updates, or file metadata changes, is first appended to this log. By recording all state-changing operations in a sequential log, the tracker ensures that its state can be fully reconstructed after a crash or restart. This design provides a **durable and consistent record** of the system’s state, serving as the backbone for failover resilience.

To maintain an up-to-date in-memory state, the tracker continuously **replays the log**. At startup, the tracker reads the entire log file and applies each recorded operation to its internal data structures. For ongoing operations, a dedicated **watcher thread** monitors the log for newly appended entries and incrementally applies them. This is coordinated using a **mutex** to protect shared data, ensuring that concurrent client interactions do not lead to race conditions or inconsistent state updates. The last_offset variable tracks the position in the log up to which operations have already been processed, allowing the tracker to efficiently handle only new changes.

The log-based approach also facilitates **session and state management**. Active user sessions, group memberships, and file sharing information are all persisted through the log. If the tracker restarts, it can restore both the system state and active sessions, maintaining a seamless experience for clients. The design inherently supports **idempotent updates**, meaning that reapplying operations from the log does not corrupt the state, which is crucial for reliability in a multi-client or multi-tracker environment.

Overall, the tracker’s synchronization mechanism combines **persistent logging, incremental log replay, mutex-protected state updates, and background monitoring** to provide a robust, fault-tolerant system. This ensures that the in-memory state is always consistent with the recorded operations, allows recovery from crashes, and supports concurrent access by multiple clients without conflicts.

3. Tracker Management at client side:

The client maintains a list of trackers and dynamically selects one for communication. Trackers are read from a configuration file (tracker_info.txt) and stored as IP-port pairs. Connection to a tracker is attempted in a failover sequence: if the initial tracker is unavailable, the client automatically attempts to connect to alternative trackers in the list.

When a tracker becomes unreachable or a session token becomes invalid, the client retries the command on the next available tracker. This mechanism ensures that operations such as login, file upload, and group management are resilient to tracker failures.

The client maintains cached credentials (cached_uid, cached_session) to wrap commands with session tokens, providing seamless authentication for subsequent requests. During failover, the client can automatically re-login using cached credentials to restore session validity.

All tracker communication is protected by a mutex (sock_lock). This guarantees that simultaneous send or receive operations do not interfere with each other, preventing session corruption or partial command execution.

4. Command Handling:

The client is capable of handling a wide range of commands, covering both **user management and file sharing functionalities**. For example:

User Management: create_user, login, logout

Group Management: create_group, join_group, leave_group, list_groups, list_requests, accept_request

File Management: upload_file, download_file, list_files, stop_share, show_downloads

Each command is first validated locally to ensure the correct number of arguments and proper authentication. Invalid commands are rejected immediately with descriptive error messages. Commands are then sent to the tracker using session-wrapped messages, ensuring that all operations are executed within the context of the authenticated session. Tracker responses are read fully using a marker **END_OF_REPLY**, which guarantees that multi-line responses are processed completely before moving to the next command. This ensures both data consistency and command reliability.

5. User Management:

Account Creation: Users can create an account by providing a username and password. The client validates the input locally and sends a request to the tracker, which stores the credentials securely. This ensures that only valid and unique users can register.

Login and Session Handling: Logging in requires the username and password, which the tracker verifies. Upon successful login, the tracker returns a session token, which the client caches along with the user ID. This session token is automatically included in all subsequent commands to authenticate the user, allowing seamless interaction with the tracker without repeated credential entry.

Failover and Auto-login: If the tracker is unavailable or the session is lost, the client automatically attempts to reconnect to an alternative tracker and re-login using cached credentials. This ensures uninterrupted service and maintains the user’s session consistently across multiple trackers.

Logout: Logging out clears the cached session and user ID in the client, effectively ending the authenticated session. Any subsequent commands require re-login to validate the user.

6. Group Management:

Group Creation: Authenticated users can create new groups by specifying a group name. The client sends the request to the tracker, which records the group details and assigns the creator as the owner. This ensures proper authorization and tracking of group ownership.

Joining and Leaving Groups: Users can request to join existing groups. The tracker maintains pending join requests, which must be approved by the group owner. Users can also leave a group voluntarily, and the tracker updates membership records accordingly.

Request Management: Group owners can view pending join requests for their groups and accept or reject them. This mechanism ensures controlled access to groups and maintains the integrity of membership.

Listing Groups: Users can retrieve a list of all available groups from the tracker, allowing them to explore and choose groups to join. This operation requires the user to be authenticated to maintain security.

7. Peer-to-Peer communication:

In the implemented system, each client functions as both a peer and a downloader, enabling true peer-to-peer communication. To facilitate this, every client runs a dedicated peer server thread that listens on a dynamically allocated port, allowing it to accept incoming requests from other peers at any time. When another peer requests a specific piece of a file, the client first retrieves the requested piece directly from its local storage and then transmits it over the network to the requesting peer. This design allows multiple peers to connect and request pieces concurrently, which enhances the availability and distribution of shared files across the network. To maintain data integrity and prevent inconsistencies during simultaneous file access, all read and write operations on local files are carefully synchronized using mutexes, ensuring that no race conditions occur. Additionally, the dynamically assigned peer server port is communicated to the tracker during initial uploads as well as periodic re-announcements, making the client discoverable to other peers in the network and enabling efficient piece exchange.

8. Upload Mechanism:

The upload process in the system is designed to ensure both consistency and data integrity between the client and the tracker. When a user initiates a file upload, the client first divides the file into pieces and computes the SHA-1 hash for each piece as well as for the entire file. This hashing process provides a unique fingerprint for each piece, enabling verification of data integrity during downloads. Once the hashes are generated, the client announces the file to the tracker, providing essential metadata including the file size, the SHA-1 hashes, the client's IP address, and the dynamically assigned peer server port. To maintain persistent awareness of uploads, the client stores all upload information in the .uploads_registry file. This allows the client to re-announce files automatically after logging in or reconnecting to the tracker, ensuring that previously uploaded files remain available to other peers even after a session restart. By maintaining detailed metadata and hashes, the system guarantees that all file pieces can be verified by requesting peers, thereby preserving the integrity and reliability of the shared files throughout the network.

9. Download Mechanism:

The client incorporates a robust parallel download mechanism to ensure efficient, reliable, and secure file retrieval. Each file to be downloaded is divided into fixed-size pieces, and each piece is individually verified using the SHA-1 hashes provided by the tracker. To maximize download speed, multiple threads—configurable by the user, with a default of eight—are employed to fetch pieces concurrently from multiple seeders. In the event of a failure or a corrupted transfer, each piece is automatically retried a configurable number of times, with a default of five attempts, to ensure successful completion. All disk operations are carefully synchronized to prevent race conditions, ensuring that concurrent threads do not overwrite or corrupt each other's data when writing pieces to storage. Once all pieces have been successfully downloaded, the client computes the SHA-1 hash of the entire file to verify end-to-end integrity. This comprehensive strategy ensures not only fast and efficient downloads but also guarantees that corrupted or incomplete pieces are detected and corrected immediately, maintaining the reliability and security of the file-sharing process.

10. Piece Selection Strategy:

Adaptive Piece Selection: The client divides files into fixed-size pieces, each with a unique SHA-1 hash, and dynamically selects pieces based on availability from multiple seeders. Instead of downloading sequentially, pieces are assigned to threads as they become available, enabling efficient and adaptive retrieval while maximizing download speed.

Parallelism and Thread Coordination: Multiple threads (configurable, default eight) handle the downloads concurrently. Each thread accesses shared metadata that tracks downloaded and pending pieces. An atomic counter (next_piece) ensures that each thread retrieves a unique piece index, preventing conflicts and maintaining safe concurrency.

Piece Verification and Safe Writing: Once a thread receives a piece using the GET_PIECE command, it verifies the data against the SHA-1 hash. Only verified pieces are written to disk at the correct offset, with a mutex ensuring that concurrent writes do not overwrite or corrupt data. This guarantees both correctness and thread-safe operations.

Dynamic Work Distribution: Threads operate in a load-balanced manner: faster threads pick up new pieces as they finish their current tasks, while slower threads continue processing existing pieces. Simultaneous downloads from multiple seeders further optimize bandwidth utilization and reduce dependency on any single peer, improving overall reliability and efficiency.

End-to-End Integrity: After all pieces are successfully downloaded, the client computes the complete file hash and compares it with the tracker-provided hash. This final verification step ensures that the downloaded file is fully complete and accurate, maintaining the integrity of the peer-to-peer transfer process.

11. Conclusion:

The client implementation provides a robust, fault-tolerant, and fully-featured P2P file sharing solution. It handles authentication, group management, file uploads, downloads, failover, and peer-to-peer sharing, while ensuring data integrity and synchronization with multiple trackers. Multi-threading, session management, persistent local state, and SHA-1 verification ensure high reliability, efficiency, and consistency. This system demonstrates an in-depth understanding of distributed systems, networking, and concurrency management, providing a solid foundation for scalable P2P file sharing applications.