Proposal Report: Peer-to-Peer (P2P) File-Sharing System

1. Introduction

This project aims to develop a peer-to-peer (P2P) file-sharing system inspired by the BitTorrent protocol. The system enhances file distribution efficiency by dividing files into chunks, allowing parallel downloads from multiple peers. It consists of a **tracker server** to manage active peers and a **torrent client** for initiating downloads, sharing chunks, and facilitating peer communication. The project will also simulate multiple peers and compare performance with a centralized file transfer model to demonstrate the scalability benefits of P2P networking.

2. Objectives

The primary objectives of this project are:

1. **Enable File Sharing**: Allow users to upload and download files efficiently across a distributed network.
2. **Group Management**: Implement group-based sharing to provide controlled access to shared files.
3. **User Authentication**: Ensure secure access through user registration and login functionalities.
4. **Scalability**: Design the system to support multiple concurrent users and large file transfers.
5. **Efficiency**: Optimize communication between clients and the tracker for faster file operations

3. System Architecture

The proposed system includes two primary components:

- **Tracker:**
    - Maintains a registry of active peers and the files they share.
    - Manages user authentication, group membership, and request handling.
    - Facilitates peer discovery by providing client information for requested files
- **Client:**
    - Handles file chunking, uploading, and downloading.
    - Supports both seeder (sharing) and leecher (downloading) modes.
    - Manages peer-to-peer communication via handshakes and chunk requests.
    - Implements logging for tracking user activity and system events.

4. Functional Specifications

The system supports the following operations:

1. User Management

- User registration
- User login
- User logout

2. **Group Management**

- Create a group
- Join a group
- Leave a group
- List all groups
- List pending join requests
- Accept group join requests

3. **File Sharing**

- Upload a file to a group
- Stop sharing a file
- List available files in a group
- Download a file from a group

4. **System Commands**

- Shutdown of the tracker

5. Technical Specifications

- **Programming Language**: Go (Golang)
- **Networking**: TCP/IP for communication between the tracker and clients.
- **Concurrency**: Managed using Go routines and sync package for safe concurrent access.
- **Data Structures will be used to**:
    1. Stores user credentials.
    2. Maps active connections to users.
    3. Maintains group memberships.

4. Tracks pending join requests.
5. Stores file-sharing metadata.
6. Records file integrity using hashes.
7. Holds file size information.

6. Workflow

1. **User Registration and Login**
   Users register with the tracker and log in to access the system. Session information is managed via active sockets.
2. **Group Management**
   Users can create and join groups. Group admins approve new members, allowing for controlled file sharing.
3. **File Sharing**
   - Upload: Users share files by providing metadata (file name, size, hash, etc.).
   - Download: Users retrieve files by querying the tracker for available seeders with whom it can connect directly and download files.
4. **Synchronization**
   All file-sharing information is kept synchronized using mutex locks for safe concurrent operations.

7. Error Handling

- Invalid commands or parameters return clear error messages.
- Incorrect login attempts prevent unauthorized access.
- Concurrent access is controlled using sync. Mutex to prevent data races.
- A dedicated logging function records user activities and system events in persistent log files for future analysis.

As the project moves forward, we will make improvements to enhance performance, scalability, and reliability. New features may be added based on testing and feedback. We will keep optimizing the system to make it more efficient and better.

If we encounter performance or concurrency challenges during implementation in Go, we may consider transitioning to C++ for its advanced low-level optimizations and fine-grained control over system resources.