Great job implementing the full assignment! Here's the **updated** `README.md` reflecting **all required features**, following the assignment specification you shared:

Peer-to-Peer Distributed File Sharing System

This project implements a **group-based peer-to-peer distributed file sharing system** using C++ and system calls. The system consists of **clients** and **synchronized trackers**, supporting multi-peer parallel downloads, user and group management, and SHA1-based integrity checks.

🏷️ Features

✅ Core Functionalities

- Two synchronized trackers
- User account creation & login (with basic authentication)
- Group creation and management (owner-based model)
- Group membership request and approval system
- File sharing (SHA1 based piecewise metadata)
- List available groups and files
- Parallel **multi-peer downloading** with custom **piece selection algorithm**
- File integrity verification (SHA1 hash comparison)
- Download progress monitoring
- Persistent file sharing across login sessions
- Logout disables sharing (until next login)

🧱 Prerequisites

- C++11 compiler
- OpenSSL library (for SHA1 hashing)
- Linux OS (uses POSIX system calls)

PROF

🛠️ Compilation

Use `make` or compile manually as follows:

Tracker

```
g++ tracker.cpp -o tracker
```

Client

```
g++ client.cpp -o client -lssl -lcrypto
```

> **Note**: OpenSSL is used only for SHA1 hashing. `-lssl -lcrypto` are necessary.

🚀 Running the Application

🖥️ Tracker

Start each tracker:

```
./tracker tracker_info.txt <tracker_no>
```

- `tracker_info.txt`: Contains IP and port of all trackers (one per line)
- `<tracker_no>`: 0-based index into the tracker list (e.g., 0 for the first one)

Example:

```
./tracker tracker_info.txt 0
```

💻 Client

```
./client <IP>:<PORT> tracker_info.txt
```

Example:

PROF

```
./client 127.0.0.1:10000 tracker_info.txt
```

👩‍💻 Client Commands

Command	Description
`create_user <user_id> <passwd>`	Register new user
`login <user_id> <passwd>`	Authenticate and login
`create_group <group_id>`	Create a group
`join_group <group_id>`	Request to join group
`leave_group <group_id>`	Leave group

Command	Description
`list_requests <group_id>`	View pending join requests (group owner only)
`accept_request <group_id> <user_id>`	Accept group join request
`list_groups`	List all groups
`list_files <group_id>`	List shareable files in a group
`upload_file <file_path> <group_id>`	Share file with group
`download_file <group_id> <file_name> <destination_path>`	Parallel download with multi-peer piece selection
`show_downloads`	Monitor file downloads
`stop_share <group_id> <file_name>`	Stop sharing specific file
`logout`	Logout and temporarily stop sharing

⚙ Implementation Highlights

- **Multi-threading**: Separate threads for communication, uploads, downloads.
- **Tracker Synchronization**: Trackers sync on user data, group info, file metadata.
- **File Division**: Files are split into **512KB** pieces.
- **SHA1 Hashing**:
  - For full file integrity check
  - For each 512KB piece (stored with tracker)
- **Piece Selection Algorithm**:
  - Prefer downloading different pieces from different peers
  - Fallback to other peers if any disconnects
- **Persistence**:
  - Shared files are automatically re-shared upon next login
- **Session Control**:
  - Logout halts sharing
  - Login resumes all previous file sharing states

PROF

🗂 File Structure

```
<ROLL_NO>_A3.zip
├── tracker/
│   ├── tracker.cpp
│   ├── tracker_functions.h
│   ├── read_tracker.h
│   └── tracker_info.txt
├── client/
│   ├── client.cpp
│   ├── client_functions.h
│   └── tracker_info.txt
└── README.md
```

> **Note:** No extra folders should be present outside `tracker/` and `client/`.

❗ Assumptions

- At least one tracker is always online.
- Clients reconnect automatically to alternate tracker if primary fails.
- `Single-threaded` tracker; `multi-threaded client`.
- Shared files are not physically copied to tracker—only metadata is.
- Only system calls are used. No banned libraries (like `filesystem`, `exec`, `system()`).
- Every client runs on a `fixed (static) IP and port` that does not change across restarts/logins.

📌 Limitations

- Communication is in plaintext (no encryption over sockets).
- Passwords stored/compared in plain form.
- No GUI; entirely command-line interface.

PROF

📚 References

- Linux `socket()`, `bind()`, `listen()`, `accept()`, `send()`, `recv()` system calls
- SHA1 hashing using OpenSSL (`SHA1()` API)
- Threading via POSIX `pthread` API
