Performance Analysis Report: Peer-to-Peer File Sharing System

Roll no: 2025201093

Introduction

This report evaluates the performance of the implemented peer-to-peer (P2P) file sharing system in C++. The system consists of a tracker server and multiple clients that interact in a distributed manner. Clients serve dual roles as downloaders and seeders, while the tracker coordinates user management, groups, and file metadata.

The analysis is based on the implementation code and assignment specification. Experiments were conducted with **1 tracker and 4 clients**, handling files up to **1 GB** with **multiple concurrent downloads**.

Methodology

- **System Setup:** One tracker instance, four client instances, each running on separate ports.
- **Test Files:** Sizes ranging from <512 KB to 1 GB.
- **Metrics:** concurrency behavior, resource usage, and failure scenarios.

Observations

- **Concurrency:**
  - Clients use multiple download threads for parallel piece fetching.
  - Uploads handled by a pool of 4 worker threads per client.
  - Tracker spawns one thread per client connection.

- **File Integrity:**
  - Each 512 KB piece hashed using SHA-1.
  - Full file hash checked after reassembly.
  - Hash mismatches trigger retries (up to 5 attempts).

- **Large Files:**
  - Correct handling of offsets with 64-bit file pointers.
  - Low memory footprint (only per-thread buffers and piece-hash tables).

- **Failure Handling:**
  - Peer disconnection mid-download → retries with other peers.
  - Tracker crash halts system (single point of failure).

Analysis

- **Efficiency:** Parallel downloads + SHA-1 verification
- **Bottlenecks:**
    - Tracker is single-thread-per-client without synchronization could fail under high scale.
    - Fixed upload thread pool (4) limits seeding capacity.

Implementation Approach

The system was built modularly with two main executables:

- **Tracker (`tracker.cpp`)** – manages user login, groups, file metadata, peer discovery. Uses maps and structs for efficient lookup.

- **Client (`client.cpp`)** – runs both as a requester and a seeder. Uses threads for downloads and uploads, plus a command-driven interface for user interaction.

Design priorities included **simplicity and correctness** (e.g., mandatory SHA-1 hashing, TCP-based communication, concurrency).

Synchronization Algorithm Design

The assignment required tracker synchronization between two trackers. My implementation currently runs **a single tracker**, so **synchronization across trackers is not implemented**.

However, intra-client synchronization is achieved through:

- **Mutex locks** to protect shared download status and piece queues.
- **Condition variables** (where required) to manage concurrent access.
- Each tracker-client connection runs in its own thread, avoiding blocking operations.

Future extension: add periodic state exchange between two trackers using TCP, resolving conflicts by “latest update wins.”

Piece Selection Strategy

My implementation uses a **sequential queue-based selection**:

- Each thread dequeues the next available piece index.
- Pieces are distributed evenly among worker threads.
- On hash failure, the piece is re-queued and retried with another peer.

This approach is **simple but effective**.

Protocol Design

- **Transport:** TCP sockets chosen for reliability and in-order delivery.
- **Message Format:** Simple space-delimited text commands (e.g., `upload_file groupid filename …`). This was chosen for **ease of debugging** and **low implementation overhead**, instead of binary framing.
- **Error Handling:** Tracker replies with codes/messages.
- **Peer-to-Peer Communication:** Clients connect directly and exchange file piece requests in a minimal protocol (send piece index → receive piece data).

Challenges Encountered and Solutions

1. **Concurrent File Access:**
    - Issue: Multiple threads writing to the same file risk corruption.
    - Solution: Each thread seeks directly to its piece offset using `fseeko`. Mutex-protected metadata updates prevent inconsistencies.

2. **Piece Integrity Failures:**
    - Issue: Corrupt or incomplete pieces during transfer.
    - Solution: Immediate SHA-1 validation and re-download from another peer.

3. **Tracker as Single Point of Failure:**
    - Issue: No fault tolerance with only one tracker.
    - Solution: Left unimplemented, but design allows adding second tracker with periodic state sync.

4. **Socket Management:**
    - Issue: Large numbers of connections could exceed file descriptor limits.
    - Solution: Connections are short-lived for piece transfers, sockets closed promptly after use.

5. **Testing with Large Files (1 GB):**
    - Issue: Memory exhaustion risk if naively loaded.
    - Solution: File handled piecewise.

Conclusion

The system demonstrates a functioning P2P file sharing model with concurrency, integrity checking, and group-based sharing, simple in synchronization and piece selection.