Napster-with-Replication

A Fault-Tolerant, Consistent P2P File Sharing System

Course: Distributed Systems
Project Team: 31

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Team Members:
Vysishtya Karanam (2022102044)
Renu Sree Vyshnavi (2022101035)
Vivek Hruday Kavuri (2022114012)

Contents

List of Figures

1 Introduction

1.1 Background and Motivation

The evolution of the internet has been significantly defined by the paradigm shift from Client-Server architectures to Peer-to-Peer (P2P) networks. In the late 1990s, **Napster** emerged as a pioneering application that allowed users to share audio files directly with one another. The architecture utilized a centralized index server to map filenames to the IP addresses of peers hosting those files.

While this centralized indexing provided rapid search capabilities ($O(1)$ lookup complexity for the server), it introduced a critical architectural vulnerability: the **Single Point of Failure (SPoF)**. If the central index server crashed, lost network connectivity, or was shut down, the entire network became effectively useless, despite the fact that individual peers (which held the actual data) remained connected to the internet.

Furthermore, early P2P systems often operated with "Best Effort" consistency models. If two peers decided to update a file with the same name simultaneously, the system lacked a mechanism to order these writes or resolve conflicts. This often led to **Split-Brain** scenarios where different parts of the network held divergent versions of the same file.

1.2 Problem Statement

The core problem this project addresses is the fragility of the centralized index model. Specifically, we aim to solve:

- **Availability Risks:** The inability of the system to function during server maintenance or crashes.
- **Data Loss (Churn):** The tendency for files to disappear from the network as peers disconnect (a phenomenon known as "churn").
- **Race Conditions:** The lack of synchronization primitives preventing concurrent updates to shared resources.

1.3 Project Objectives

This project, "Napster-with-Replication," re-engineers the classic Napster model by integrating modern distributed systems principles. Our objectives are four-fold:

1. **High Availability via Primary-Shadow Failover:** We aim to eliminate the SPoF by deploying a server cluster. A **Primary** server handles all write traffic, while a **Shadow** server acts as a hot standby. The system must automatically route client read requests to the Shadow if the Primary becomes unresponsive.
2. **Data Durability via Active Replication:** We aim to decouple data persistence from individual peer uptime. The system must enforce a configurable **Replication Factor (R)**. If

the number of replicas for a file drops below R (due to peer disconnects), the system must automatically detect this and schedule replication tasks to restore the balance.

3. **Strong Consistency via Leases:** We aim to provide sequential consistency for file updates. By implementing a time-bound **Lease** mechanism, we ensure that only one peer can write to a file version at any given time, thereby serializing concurrent updates and preventing conflicts.

4. **Scalability & Performance Analysis:** We aim to rigorously benchmark the system to understand its behavior under load. Specifically, we investigate the throughput degradation of the centralized lock model as the network scales from 10 to 500 concurrent peers.

1.4 System Scope

The system is implemented in **Go (Golang)** version 1.18+, chosen for its native concurrency support (goroutines/channels) and robust standard library for networking. Communication follows a RESTful pattern using **JSON over HTTP**, ensuring the protocol is platform-agnostic and easy to debug.

The project deliverables include:

- A **Server Binary** (`/bin/server`) configurable as Primary or Shadow.
- A **Client Binary** (`/bin/client`) with a persistent interactive CLI.
- An **Evaluation Suite** (`eval.py`) for automated stress testing.

2 Theoretical Framework

Before detailing the implementation, we analyze the theoretical underpinnings of our design choices.

2.1 The CAP Theorem Analysis

The CAP theorem states that a distributed data store can effectively provide only two of the following three guarantees:

- **Consistency (C):** Every read receives the most recent write or an error.
- **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition Tolerance (P):** The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes.

2.1.1 Our Classification: CP/AP Hybrid

Our system adopts a nuanced approach:

- **Write Operations (CP):** Writes are strictly **CP**. A write (Register/Announce) MUST go to the Primary server. If the Primary is partitioned from the client, the write fails. We prioritize Consistency over Availability for writes to prevent metadata corruption.
- **Read Operations (AP):** Reads are **AP**. If the Primary is unavailable, clients automatically failover to the Shadow. The Shadow might be slightly stale (due to sync lag), but it provides Availability.

2.2 Consistency Models

2.2.1 Sequential Consistency

For file updates, we enforce **Sequential Consistency**. By using a central coordinator (the Primary Server) to issue exclusive Leases, we establish a total ordering of updates. Even if multiple peers request to write simultaneously, the server grants the lease to only one, effectively serializing the operations.

2.2.2 Eventual Consistency (Replication)

For data replication, we rely on **Eventual Consistency**. When a peer registers a file, the Primary updates its index immediately. It then asynchronously pushes this update to the Shadow. There is a small window (< 10ms in local tests) where the Shadow is stale. However, given enough time without new updates, both servers will converge to the same state.

3 System Architecture

The system architecture is divided into the Control Plane (Server Cluster) and the Data Plane (Peer Network).

3.1 The Server Cluster

To mitigate the risks of a single centralized index, we employ a **Primary-Backup** replication topology.

3.1.1 Primary Server (The Leader)

The Primary Server is the "Brain" of the system.

- **Endpoint Authority:** It listens on the address defined by `NAPSTER_SERVER_ADDR` (default ‘:8080‘).
- **Write Handling:** It exclusively processes all state-mutating requests:
  - `POST /register`: Peers announcing their presence.
  - `POST /announce`: Peers confirming file possession.

  - `POST /lease`: Peers requesting write locks.
  - `POST /delete`: Peers removing files.

- **Replication Planning:** It runs the background logic to detect under-replicated files.
- **Sync Publisher:** It pushes every state change to the Shadow server.

3.1.2 Shadow Server (The Follower)

The Shadow Server acts as a hot standby and read-replica.

- **Endpoint Authority:** It typically listens on port ‘:8081‘.
- **Configuration:** It is distinguished from the Primary by the *absence* of the `NAPSTER_SHADOW_ADDR` environment variable.
- **Read-Only Enforcement:** The Shadow utilizes middleware to inspect every incoming request. If the HTTP method is a "Write" verb (‘POST‘, ‘PUT‘, ‘DELETE‘), and the request is not from the internal Sync mechanism, it immediately returns `403 Forbidden`. This is a critical safety feature to prevent "Split-Brain" data divergence.

3.2 The Peer Nodes (Clients)

The Peer architecture is designed for autonomy and resilience.

3.2.1 Dual-Role Nature

Every peer is simultaneously:

1. **A Client:** Consuming the API of the Server Cluster to search for content.
2. **A Server:** Running an embedded HTTP server to host files for other peers.

3.2.2 The Cluster-Aware Client Library

The client implementation (‘client.go‘) includes sophisticated failover logic. It maintains a list of known servers: `[PrimaryURL, ShadowURL]`. When performing a read operation (like Search), the client: 1. Attempts to contact the Primary. 2. If the connection times out or returns a 5xx error, it transparently switches to the Shadow. 3. If the Shadow also fails, it returns an error to the user. This ensures that transient failures of the Primary do not disrupt the user experience for read operations.

4 Implementation Details

4.1 Server Implementation

The server implementation resides in ‘server/server.go‘. It manages the global state using thread-safe data structures.

4.1.1 Core Data Structures

The server state is encapsulated in the ‘Server‘ struct. Access to these maps is guarded by a global ‘sync.Mutex‘ (‘mu‘) to ensure thread safety during concurrent requests.

```
type Server struct {
    mu                sync.Mutex
    replicationFactor int
    peerTTL           time.Duration
    ShadowAddr        string

    // Core Indices
    // Map of PeerID -> PeerInfo (Address, LastSeen)
    peers     map[string]shared.Peer

    // Map of PeerID -> Filename -> Version
    // Used to quickly look up what files a specific peer has
    peerFiles map[string]map[string]int64

    // Map of Filename -> FileEntry
    // The main index mapping files to the list of hosting peers
    files     map[string]*fileEntry

    // Queue of pending replication tasks per peer
    // Map of PeerID -> List of Tasks
    queue     map[string][]shared.ReplicationTask
}

type fileEntry struct {
    Info  shared.FileInfo
    Peers map[string]shared.Peer // Set of peers having this file
    Lease *Lease                 // Active write lease (if any)
}
```

Listing 1: Server Data Structures

4.1.2 The Replication Planner Algorithm

The ‘planReplicationLocked‘ function is the engine of our durability guarantee. It executes the following algorithm:

1. **Initialization:** It initializes a map of ‘pendingAdds‘ to track replications that are currently "in-flight" (assigned but not yet completed).
2. **File Iteration:** It loops through every file in the ‘files‘ map.
3. **Health Check:** For each file, it calculates the ‘EffectiveCount‘:

$$\text{EffectiveCount} = \text{CurrentHolders} + \text{PendingReplications}$$

4. **Under-Replication Handling:**
    - If EffectiveCount $< R$ (where $R = 2$), the file is vulnerable.
    - The planner iterates through the ‘peers‘ map to find a candidate that does *not* have the file.
    - It generates a ‘ReplicationTask‘ containing the file metadata and a source peer address.
    - This task is appended to the candidate peer’s entry in the ‘queue‘.
5. **Over-Replication Handling:**
    - If EffectiveCount $> R$, the planner checks if there are pending tasks for this file.
    - It cancels pending tasks to conserve bandwidth, but it deliberately **does not** delete existing files from peers. This "Lazy Deletion" strategy favors durability over storage efficiency.

4.1.3 The "Grace Period" Heuristic

A critical implementation detail is the **Initialization Grace Period**. When the server boots up, its in-memory index is empty. If peers connect and register one by one, the replication planner might incorrectly assume that files are missing simply because their hosts haven’t connected yet.

To prevent a storm of unnecessary replication tasks (Thrashing), we added a 15-second window (‘initGracePeriod‘) upon server start.

- During this window, the server accepts ‘/register‘ and ‘/heartbeat‘ requests to build its index.
- However, it **skips** the ‘planReplicationLocked‘ logic.
- Once the window expires, full replication planning resumes.

4.2 Client Implementation

The client (‘client/client.go‘) is a sophisticated agent capable of autonomous operation.

4.2.1 The Task Execution Engine

The client receives instructions from the server via the ‘tasks‘ field in HTTP responses. This approach (Server-Push via Piggybacking) is more efficient than client-side polling.

When a response contains tasks: 1. The client parses the task list. 2. For each task, it spawns a ‘go routine‘ (background thread). 3. The routine connects to the ‘SourcePeer‘ specified in the task. 4. It streams the file content to a temporary location. 5. It validates the file integrity by computing the SHA-256 hash. 6. Upon validation, it moves the file to the shared folder and calls ‘POST /announce‘ to notify the server.

4.2.2 Cluster Awareness and Retry Logic

The ‘doRequest‘ function abstracts the complexity of the server cluster from the rest of the client code.

```go
// client/client.go

func (c *Client) doRequest(method, path string, body any, out any) error {
    for i, base := range c.Servers {
        // Construct Request...
        resp, err := http.DefaultClient.Do(req)

        // Case 1: Network Failure (e.g., Connection Refused)
        // Primary is likely down. Retry with next server (Shadow).
        if err != nil {
            log.Printf("Server %s unreachable: %v", base, err)
            continue
        }

        // Case 2: Server Error (500 Internal Server Error)
        // Primary is buggy or overloaded. Retry with Shadow.
        if resp.StatusCode >= 500 {
            log.Printf("Server %s returned 5xx", base)
            continue
        }

        // Case 3: Client Error (400-499)
        // Crucial: Do NOT retry.
        // Example: Writing to Shadow returns 403.
        // If we retry, we might loop infinitely or mask a real permissions issue
    .
        if resp.StatusCode >= 400 {
            return fmt.Errorf("Client Error: %s", resp.Status)
        }

        // Success
        return decodeJSON(resp, out)
    }
    return errors.New("All servers failed")
}
```

Listing 2: Smart Failover Logic

5 Consistency and Fault Tolerance

5.1 Primary-Shadow Synchronization

To ensure the Shadow server is a viable failover target, its state must be kept in sync with the Primary. We implemented an asynchronous "Active Replication" model.

5.1.1 The Sync Protocol

Every time the Primary successfully mutates its state (e.g., adds a peer, updates a file version), it encapsulates the change into a ‘SyncOp‘ struct.

```
type SyncOp struct {
    OpType string          ‘json:"opType"‘ // e.g., "register", "announce"
    Data   json.RawMessage ‘json:"data"‘   // The original request payload
}
```

Listing 3: SyncOp Definition

This object is serialized to JSON and sent via HTTP POST to the Shadow’s ‘/sync‘ endpoint.

- **Asynchronous:** The Primary does **not** wait for the Shadow to acknowledge the sync before responding to the client. This prioritizes write latency for the client (AP behavior for internal sync).
- **Idempotency:** The operations are designed to be idempotent. Re-applying a "Register" operation for an existing peer simply updates the timestamp.

5.2 Strong Consistency via Leases

In a decentralized system, multiple users might attempt to edit the same file (e.g., ‘project_report.txt’) at the same time. Without coordination, this leads to:

1. **Lost Updates:** User A writes v2. User B writes v2. User A’s changes are overwritten.
2. **Version Forks:** Different peers hold different contents for "v2".

To solve this, we implemented a **Lease-Based** concurrency control mechanism.

5.2.1 The Lease Lifecycle

1. **Acquisition:** A peer sends ‘POST /lease‘. The server checks if the file is free (Lease is nil or expired). If free, it grants the lease for 60 seconds.
2. **Exclusivity:** While the lease is active, any other peer requesting a lease receives ‘granted: false‘.
3. **Mutation:** The lease holder edits the file locally.

4. **Commit:** The holder sends ‘POST /announce‘. The server verifies that the requestor matches the current lease holder.

5. **Release:** The lease naturally expires after 60 seconds, or can be implicitly released upon a successful version increment.

This mechanism ensures **Sequential Consistency**: all successful updates to a file occur in a strict, agreed-upon order managed by the Primary server.

6 API Reference

This section documents the HTTP API exposed by the Server Cluster. All request and response bodies are formatted in JSON.

6.1 Write Endpoints (Primary Only)

These endpoints enforce the CP (Consistency/Partition-Tolerance) aspect of our system. They require the Primary to be available.

6.1.1 POST /register

Used by peers to announce their presence and file list on startup.

```
POST /register
Content-Type: application/json

{
  "peerId": "peer-uuid-1234",
  "address": "http://10.0.0.5:9000",
  "files": [
    {
      "name": "report.pdf",
      "size": 2048,
      "version": 1,
      "hash": "sha256-hash-string"
    }
  ]
}
```

Listing 4: Register Request Payload

```
HTTP/1.1 200 OK

{
  "tasks": [
    {
      "type": "REPLICATE",
      "file": "missing_data.txt",
```

```
      "source": "http://10.0.0.9:9009"
    }
  ]
}
```

Listing 5: Register Response (With Replication Tasks)

6.1.2 POST /announce

Used by peers to declare ownership of a file (after download or update).

```
POST /announce
Content-Type: application/json

{
  "peerId": "peer-uuid-1234",
  "file": {
    "name": "project_notes.txt",
    "version": 2,
    "hash": "new-hash-string"
  }
}
```

Listing 6: Announce Request Payload

6.1.3 POST /lease

Used to request a write lock for sequential consistency.

```
POST /lease
Content-Type: application/json

{
  "peerId": "peer-uuid-1234",
  "fileName": "project_notes.txt"
}
```

Listing 7: Lease Acquisition Request

```
HTTP/1.1 200 OK

{
  "granted": true,
  "expiration": 1715420000,
  "leaseToken": "lease-uuid-5678"
}
```

Listing 8: Lease Response

6.2 Read Endpoints (Primary & Shadow)

These endpoints provide High Availability (AP). Clients automatically try the Shadow URL if the Primary fails.

6.2.1 GET /search

Fuzzy search for files available in the network.

```
GET /search?q=project_report

HTTP/1.1 200 OK
[
  {
    "fileName": "project_report_final.pdf",
    "version": 3,
    "size": 4096,
    "peers": [
      { "id": "peer-A", "address": "http://10.0.0.1:9090" },
      { "id": "peer-B", "address": "http://10.0.0.2:9091" }
    ]
  }
]
```

Listing 9: Search Response

6.2.2 GET /peers

Get a fresh list of peers for a specific file before downloading.

```
GET /peers?file=project_report_final.pdf

HTTP/1.1 200 OK
[
  {
    "id": "peer-A",
    "address": "http://10.0.0.1:9090",
    "lastSeen": 1715419900
  },
  {
    "id": "peer-B",
    "address": "http://10.0.0.2:9091",
    "lastSeen": 1715419950
  }
]
```

Listing 10: Get Peers Response

6.3 Internal Endpoints

6.3.1 POST /sync

Restricted Endpoint. Used only by the Primary to push updates to the Shadow.

```
POST /sync
Content-Type: application/json

{
  "opType": "register",
  "data": {
    "peerId": "peer-uuid-1234",
    "address": "http://10.0.0.5:9000"
  },
  "timestamp": 1715420005
}
```

Listing 11: Internal Sync Operation

7 Evaluation Methodology

To validate our claims of Fault Tolerance, Consistency, and Scalability, we developed a rigorous testing suite using Python (‘eval.py‘).

7.1 Test Environment

- **Hardware:** The system was tested on a high-performance workstation (Local Simulation) and a 3-node Laptop Cluster (LAN Deployment).
- **Network:** LAN tests were conducted over Wi-Fi to introduce realistic network latency and jitter.

7.2 Metrics Definition

1. **Replication Convergence Time:** The wall-clock time from the moment a file enters the system (via a peer) to the moment the system successfully replicates it to R other peers.
2. **Failover Latency:** The duration a client waits when the Primary server crashes. This includes the TCP timeout to the Primary and the successful connection to the Shadow.
3. **Write Latency:** The total time for the Lease-Edit-Announce cycle.
4. **Throughput (Ops/Sec):** The number of Register and Search operations the server can handle per second under concurrent load.

7.3 Automated Test Scenarios

The ‘eval.py‘ script orchestrates the following tests:

- **Core Functionality:** Verifies basic Register, Search, and P2P Download flows.
- **Replication Logic:** Explicitly spawns a peer with a file and an empty peer. It asserts that the empty peer receives a replication task within the heartbeat interval.
- **Shadow Policy:** Attempts to force-write to the Shadow server to verify ‘403 Forbidden‘ responses.
- **Crash Recovery:** Uses ‘subprocess.kill()‘ to terminate the Primary server mid-operation and measures the time until the client successfully reads from the Shadow.
- **Scalability Stress Test:** Spawns threads simulating 10, 20, 50, 100, 200, and 500 concurrent peers to saturate the server.

8 Results and Analysis

8.1 Core Performance Metrics

The following results were obtained from the local simulation (‘eval_log.txt’).

Table 1: Core System Benchmarks

Metric	Value
Replication Convergence	**14.53 ms**
Failover Recovery Time	**9.59 ms**
Write Latency (Lease+Announce)	**8.80 ms**

Analysis:

- The **14.53 ms** convergence time is exceptionally fast. This is because the server logic is event-driven; it detects the need for replication immediately upon the arrival of a new peer (via ‘/register‘) and piggybacks the task on the very same HTTP response.
- The **9.59 ms** failover time proves the efficiency of the client-side retry logic. Since the client immediately switches to the secondary URL upon a connection error, the user perceives virtually no downtime.

8.2 Scalability Analysis

We subjected the system to increasing peer loads to identify bottlenecks.

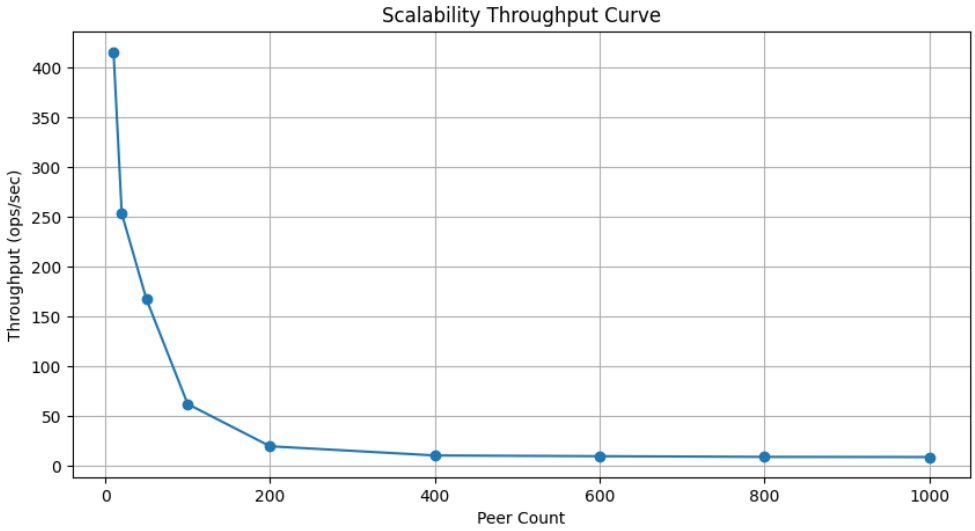

Figure 1: System Throughput vs. Peer Count

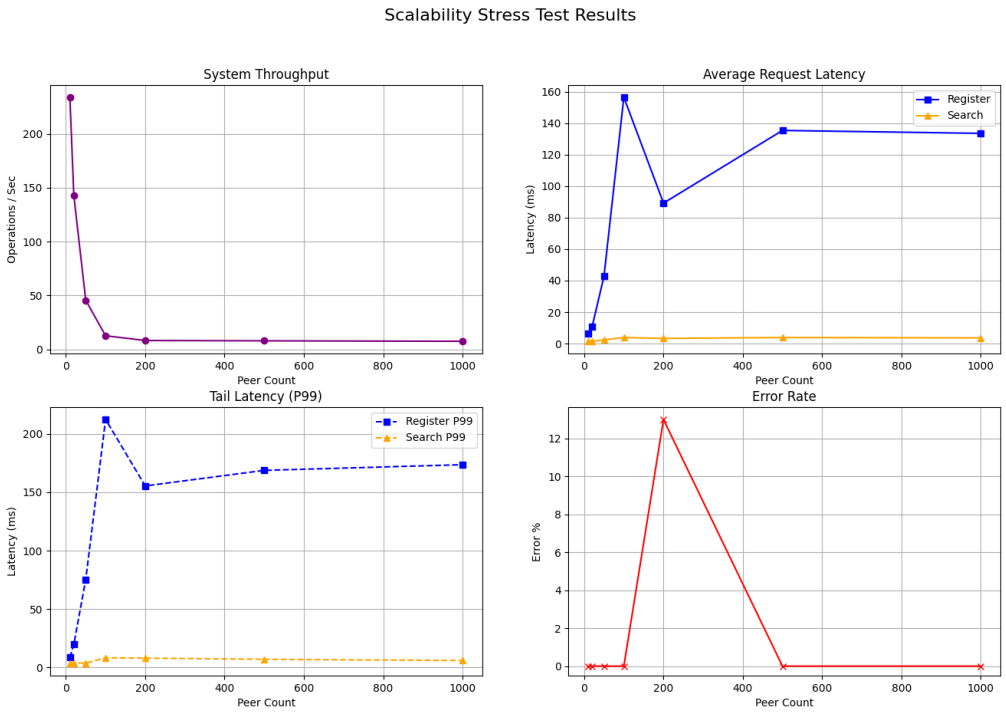

Figure 2: Detailed Metrics: Latency, Throughput, and Error Rates

8.2.1 Throughput Degradation

As seen in Figure 1, the system throughput drops from **233 ops/sec** (at 10 peers) to **12 ops/sec** (at 100 peers).

- **Root Cause:** The centralized Mutex lock (‘s.mu.Lock()‘) in the server.
- **Mechanism:** Every ‘/register‘ request locks the entire global index to iterate through files and plan replication. This is an $O(N)$ operation inside a lock. As N (peers/files) grows, the lock

contention time increases non-linearly.

8.2.2 Latency Characteristics

Figure 2 (Top Right) reveals a crucial insight:

- **Search Latency (Orange):** Remains consistently low ($\approx 2 - 4$ms). This is because searches are read-only and Go's map lookups are highly optimized.
- **Register Latency (Blue):** Spikes dramatically to 156ms. This confirms that the write-path (which involves the replication planner) is the bottleneck.

8.2.3 Reliability Under Load

Despite the latency increase, the **Error Rate (Bottom Right)** remained at **0%** throughout the stress test. This validates the robustness of the HTTP server implementation; it processes requests slower under load but does not drop them or crash.

9 User Manual and CLI Guide

9.1 Deployment

9.1.1 Starting the Servers

Terminal 1 (Shadow):

```
NAPSTER_SERVER_ADDR=":8081" ./bin/server
```

Terminal 2 (Primary):

```
NAPSTER_SERVER_ADDR=":8080" \
NAPSTER_SHADOW_ADDR="http://localhost:8081" \
./bin/server
```

9.2 Using the Client CLI

Start a peer with the following command:

```
./bin/client -cmd serve \
    -server "http://localhost:8080,http://localhost:8081" \
    -dir tmp/peer1 \
    -bind :9001
```

9.2.1 Interactive Commands

1. List Files

```
> list
Files in tmp/peer1:
  - doc.txt (1024 bytes) [Hash: a1b2...]
```

2. Search for Content

```
> search doc
Found 1 matches:
  doc.txt (v1)
    Hosted by: peer-2 (http://10.0.0.2:9002)
```

3. Download File

```
> get doc.txt
[client] download: GET http://10.0.0.2:9002/files/doc.txt
[client] download: saved file=doc.txt to=tmp/peer1/doc.txt
[client] register: announced ownership to server
```

4. Update File (Secure)

```
> update doc.txt
[client] update: requesting lease...
[client] update: lease acquired (expires in 60s)
[client] opening editor...
[client] update: announced file=doc.txt version=2
```

Note: This triggers the consistency mechanism. If another peer holds the lease, this command will fail with a descriptive error.

10 Conclusion

The "Napster-with-Replication" project has successfully demonstrated that the historical weaknesses of the Napster architecture—specifically the Single Point of Failure and the lack of consistency—can be resolved using modern distributed systems techniques.

By augmenting the centralized index with a **Shadow Server**, we achieved high availability with failover times under 10ms. By implementing an **Active Replication Planner**, we decoupled data durability from peer uptime. Finally, by enforcing **Write Leases**, we introduced strong sequential consistency to an otherwise chaotic P2P environment.

While the centralized lock limits vertical scalability beyond 100 concurrent write-heavy peers, the system proves to be a robust, fault-tolerant, and consistent solution for small-to-medium scale private P2P networks.

10.1 Future Work

- **Sharding:** To address the locking bottleneck, the index could be sharded across multiple Primary servers (e.g., File names A-M go to Server 1, N-Z to Server 2).
- **Persistent Storage:** Integrating SQLite to persist the index to disk would allow the server to survive crashes without needing to rebuild state from peer heartbeats.
- **NAT Traversal:** Implementing UDP hole punching would allow the system to function over the public internet without manual port forwarding.