VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE & ENGINEERING

COMPUTER NETWORKS (CO3093)

CC05 - Group 2

Development of a Simple Torrent-like Application

Advisor:	Nguyễn Mạnh Thìn, CSE - HCMUT	
Members:	Nguyễn Hồ Phi Ưng	- 2252897.
	Trần Đình Đăng Khoa	- 2211649.
	Vũ Hoàng Tùng	- 2252886.
	Vũ Minh Quân	- 2212828.

HO CHI MINH CITY, APRIL 2024

Contents

1 Member list & Workload 3

2 Introduction 4

3 Background 5

4 Application Architecture 7
4.1 Overview . . . 7
4.2 Tracker . . . 7
4.3 File Splitting and Piece Management . . . 7
4.3.1 File Splitting: . . . 8
4.3.2 Piece Management: . . . 8
4.4 Diagrams . . . 8
4.4.1 Class Diagram . . . 8
4.4.2 Sequence Diagram . . . 9

5 Protocol Design 11
5.1 Tracker Communication . . . 11
5.1.1 Tracker Request Parameters . . . 11
5.1.2 Tracker Response Structure . . . 11
5.2 File Sharing . . . 12
5.2.1 Piece Request and Response Format . . . 12
5.3 Error Handling . . . 13
5.3.1 Tracker-Side Error Handling . . . 13
5.3.2 Peer-Side Error Handling . . . 13

6 Implementation 14
6.1 Tracker Implementation . . . 14
6.1.1 Peer Announcements . . . 14
6.1.2 Handling Piece Requests . . . 14
6.2 Piece-Based Transfers . . . 14
6.2.1 File Splitting . . . 15
6.2.2 Piece Request and Response . . . 15
6.3 Concurrency . . . 15
6.3.1 Multi-threading for Concurrent Transfers . . . 15
6.4 Data Integrity . . . 16
6.4.1 File Hashing and Validation . . . 16
6.5 Configuration . . . 16

7 Validation and Results 17
7.1 Running the Application . . . 17
7.1.1 Step 1: Configuring the Tracker . . . 17
7.1.2 Step 2: Starting the Tracker . . . 17
7.1.3 Step 3: Starting the Nodes . . . 17
7.2 Testing Scenario . . . 18
7.2.1 Test Case: Downloading and Reassembling a File . . . 18
7.2.2 Test Case Results . . . 19

8 Limitations **20**

9 Conclusion **23**

1 Member list & Workload

No.	Fullname	Student ID	Contribution	Percentage of work
1	Trần Đình Đăng Khoa	2211649	System Architect	100%
2	Vũ Hoàng Tùng	2252886	Tracker and Peer Communication	100%
3	Nguyễn Hồ Phi Ưng	2252897	Command-Line Interface	100%
5	Vũ Minh Quân	2212828	Testing and Validation	100%

2 Introduction

Abstract

This project presents a Simple Torrent-like Application (STA) designed to replicate core functionalities of a torrent-based peer-to-peer (P2P) file-sharing system. The application utilizes a centralized tracker to manage file metadata and facilitate communication among peers. Built using Python, the system adheres to the principles of the TCP/IP protocol stack and supports multi-directional data transfer (MDDT) for efficient and concurrent file-sharing operations.

The application allows peers to announce files to the tracker and request file pieces from other peers. File transfers are implemented using piece-based management, where files are divided into smaller chunks, enabling parallel downloads from multiple peers. This approach maximizes network utilization and minimizes download times. The tracker maintains metadata about file availability, including the mapping of file pieces to peers, and responds to requests with a list of eligible peers.

Key features include:

- **Centralized Tracker**: Responsible for managing file metadata, handling peer announcements, and directing file piece requests.
- **Multi-Directional Data Transfer (MDDT)**: Enabled by multithreaded architecture, allowing simultaneous uploads and downloads across multiple peers.
- **Piece-Based File Management**: Ensures efficient data transfer by splitting files into equal-sized chunks for distribution among peers.
- **Scalable Design**: Configurable components for multi-host deployments, allowing the tracker and peers to operate seamlessly on different machines within the same network.

This implementation simplifies traditional torrent protocols by focusing on single-file transfers and excluding advanced features such as magnet links and distributed hash tables (DHT). The system prioritizes core functionality, emphasizing concurrency, data integrity, and ease of deployment. The STA demonstrates practical applications of P2P file-sharing principles, offering a foundational platform for future enhancements, including multi-file torrents and advanced seeding algorithms.

3 Background

Peer-to-peer (P2P) file-sharing systems revolutionized the way data is distributed over the internet by decentralizing the file-sharing process. Unlike traditional client-server models, where a central server handles all requests and data distribution, P2P systems leverage the collective resources of multiple peers, which act both as clients and servers. This approach improves scalability, enhances fault tolerance, and reduces the load on central servers.

In a typical P2P file-sharing system:

1. **File Availability**: Each peer contributes to the network by sharing files or parts of files with others.
2. **Decentralized Communication**: Instead of downloading a file from a single source, peers can download different parts of a file from multiple other peers simultaneously.
3. **Tracker Role**: A tracker may be used to manage metadata about the files and the peers that host them. It acts as a centralized registry to facilitate peer discovery and file part (piece) tracking.
4. **Piece-Based Transfers**: Files are divided into smaller parts (pieces), enabling concurrent downloads from multiple peers. This optimizes bandwidth usage and reduces overall download time.

The simplicity and efficiency of P2P systems make them a cornerstone of modern distributed applications, such as torrenting, content distribution networks (CDNs), and collaborative platforms.

Objective

The primary objective of this project is to develop a tracker-based Simple Torrent-like Application (STA) that simulates the fundamental operations of a P2P file-sharing system. The STA implements key features of torrent-like architectures, including:

1. **File Announcements and Requests**:
   - Peers can announce the files they have to the tracker.
   - When a peer needs a file it does not have, it requests the tracker for the file or its parts.
2. **Multi-Directional Data Transfer (MDDT)**:
   - To maximize network efficiency, the system supports simultaneous uploads and downloads.
   - MDDT is achieved using a multithreaded implementation, enabling a peer to download pieces from multiple sources while serving file requests to other peers concurrently.
3. **Piece-Based File Management**:
   - Files are divided into smaller, fixed-size chunks (default 512 KB). The tracker maintains metadata about these pieces, including their locations and availability across peers.
   - Peers download missing pieces from others and reassemble them into the complete file after validation.

By focusing on these objectives, the STA provides a robust platform for learning and demonstrating P2P principles. While the implementation is simplified compared to full-fledged torrent clients (e.g., it excludes magnet links and multi-file torrent support), it highlights the critical components of file sharing in distributed systems.

Overview of System Design

The STA operates on a tracker-based architecture:

- A **centralized tracker** manages peer metadata and facilitates communication by: Registering peers and the files (and pieces) they own. Responding to file requests with a list of peers hosting the requested pieces.
- **Peers (nodes)** act as file announcers, requesters, and distributors. They communicate with the tracker to advertise file availability or retrieve metadata about requested files.

Scope

The application's implementation is focused on single-file transfers, with a particular emphasis on file piece management and multi-directional data transfer (MDDT). Advanced features found in more robust torrent clients, such as magnet links and distributed hash tables (DHT), are excluded in this implementation to maintain simplicity and focus on core functionalities. Additionally, fault tolerance mechanisms like tracker redundancy or advanced peer prioritization schemes (e.g., tit-for-tat strategies) are not part of the current design, leaving room for future improvements.

4 Application Architecture

4.1 Overview

The **Simple Torrent-like Application (STA)** follows a **peer-to-peer (P2P)** architecture with a **centralized tracker** that helps peers discover each other and facilitates the sharing of file pieces. Unlike traditional client-server models, where a central server manages all requests and data transfers, STA's peers act both as **clients** and **servers**, enabling direct data exchange between peers. The tracker only manages metadata about the files and the peers that hold them, without handling the actual file transfers. This architecture is more efficient and scalable as peers share the load of data distribution.

The main components of the STA architecture include:

- **Tracker**: A centralized service that manages metadata about the files and tracks the availability of file pieces among peers.
- **Peers (Nodes)**: These are the participants in the P2P network. They announce files to the tracker, request file pieces, and share file pieces with other peers.
- **File Management**: Files are divided into smaller pieces, and peers download and upload these pieces in parallel.
- **Multi-threading**: Allows efficient handling of simultaneous uploads and downloads.

4.2 Tracker

The Tracker plays a central role in the STA but does not directly handle file data transfer. Instead, it maintains crucial metadata about the files and the pieces that peers hold, acting as a directory that helps peers find one another.

Key responsibilities of the Tracker:

- **Peer Announcements**: Peers announce which files they have and their availability to the tracker. The tracker registers this metadata, which includes the file name, hash, the number of pieces, and which peer holds which pieces.
- **File Piece Metadata**: The tracker maintains a list of which peers have which file pieces. When a peer requests a piece, the tracker provides a list of other peers that hold that piece.
- **Peer Discovery**: When a peer needs to download a specific file piece, it queries the tracker for a list of peers that have that piece. The tracker responds with the relevant peer information, enabling direct peer-to-peer connections for the file transfer.
- **No Data Transfer**: The tracker does not participate in the actual data transfer. It only facilitates peer discovery and the management of file pieces.

4.3 File Splitting and Piece Management

To optimize the file transfer process, files are split into smaller chunks (pieces). This allows peers to download different pieces concurrently from different peers, increasing the overall transfer speed.

4.3.1 File Splitting:

- **Chunk Size**: Each file is split into smaller pieces of 512 KB by default (configurable). These pieces are the basic units of data transferred between peers.
- **Piece Metadata**: The tracker keeps a record of which peers hold which pieces of a file. This metadata allows the tracker to respond efficiently when a peer requests a piece of the file.

4.3.2 Piece Management:

- When a peer wants to share a file, it announces the pieces it holds to the tracker. This allows other peers to request pieces from it.
- Peers download missing pieces from other peers and reassemble them into the complete file. Once a peer has downloaded a piece, it may also upload it to other peers, helping them complete their downloads.

This piece-based file management maximizes network efficiency by enabling multiple peers to share different parts of the file simultaneously, which reduces download times and increases overall throughput.

4.4 Diagrams

In this section, we present several key diagrams that visually represent the architecture and operation of the Simple Torrent-like Application (STA). These diagrams help illustrate the interactions between components and provide a clearer understanding of the system's design.

4.4.1 Class Diagram

The class diagram below provides an overview of the main classes involved in the STA implementation. This diagram outlines the relationships between the classes and their key responsibilities within the application.

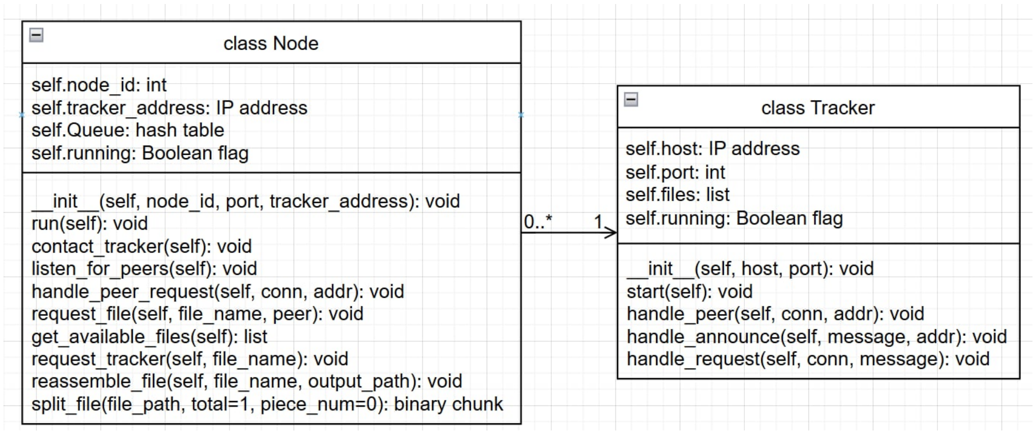

class Node
self.node_id: int
self.tracker_address: IP address
self.Queue: hash table
self.running: Boolean flag
__init__(self, node_id, port, tracker_address): void
run(self): void
contact_tracker(self): void
listen_for_peers(self): void
handle_peer_request(self, conn, addr): void
request_file(self, file_name, peer): void
get_available_files(self): list
request_tracker(self, file_name): void
reassemble_file(self, file_name, output_path): void
split_file(file_path, total=1, piece_num=0): binary chunk
0..*
1
class Tracker
self.host: IP address
self.port: int
self.files: list
self.running: Boolean flag
__init__(self, host, port): void
start(self): void
handle_peer(self, conn, addr): void
handle_announce(self, message, addr): void
handle_request(self, conn, message): void

Figure 1: Class Diagram of the STA System

Description of the Class Diagram

The class diagram showcases the structure of the application and how the core components are modeled. The following key classes and their relationships are represented:

- **Tracker**: This is the central class responsible for managing file metadata and peer information. It interacts with both the peer nodes and the tracker database to handle peer announcements, file piece requests, and responses. It maintains a list of peers and the file pieces they are sharing.
- **Peer**: Represents the peer node in the P2P network. Each peer manages its own set of files, file pieces, and connections to other peers. It communicates with the tracker to request file pieces and announces the pieces it has for upload.
- **File**: This class models a file being shared in the network. It stores file metadata such as the file name, hash, and list of pieces.
- **Piece**: Represents a chunk of a file. The `Piece` class is responsible for storing data related to each individual file piece, including its index and hash. It is transferred between peers as part of the file-sharing process.
- **Network Connection**: This class handles the network connections between peers and the tracker. It is responsible for initiating and managing the TCP/IP connections, sending and receiving requests and responses between peers and the tracker.

The diagram highlights the **dependencies** between the classes. For example, the `Peer` class depends on the `Tracker` class to register and request pieces, while the `Piece` class is closely associated with both the `File` and `Peer` classes, as pieces are shared between peers and form the complete file.
This class diagram provides a clear representation of the system's structure, helping to understand the roles and interactions of various components within the STA application.

4.4.2 Sequence Diagram

The sequence diagram below demonstrates the interaction between the main components of the STA during the process of downloading file pieces. It outlines the step-by-step communication between the tracker, the peers, and the file pieces being transferred. The diagram provides insight into how the peers request pieces from the tracker, download pieces from other peers, and reassemble the file.

Description of the Sequence Diagram

The sequence diagram represents the dynamic interactions involved in the file-sharing process. The following steps outline the main interactions between the classes in the STA system:

- **Peer Requests Piece**: The peer (Node 1) sends a request to the tracker asking for the available peers that hold the specific piece of the file.
- **Tracker Responds**: The tracker responds to Node 1, providing a list of peers (Node 2 and Node 3) that hold the requested piece.
- **Peer Downloads Piece**: Node 1 then sends a request to one of the peers (Node 2) to download the requested piece. The download is initiated via a direct network connection.
- **Piece Transfer**: The requested piece is transferred from Node 2 to Node 1.

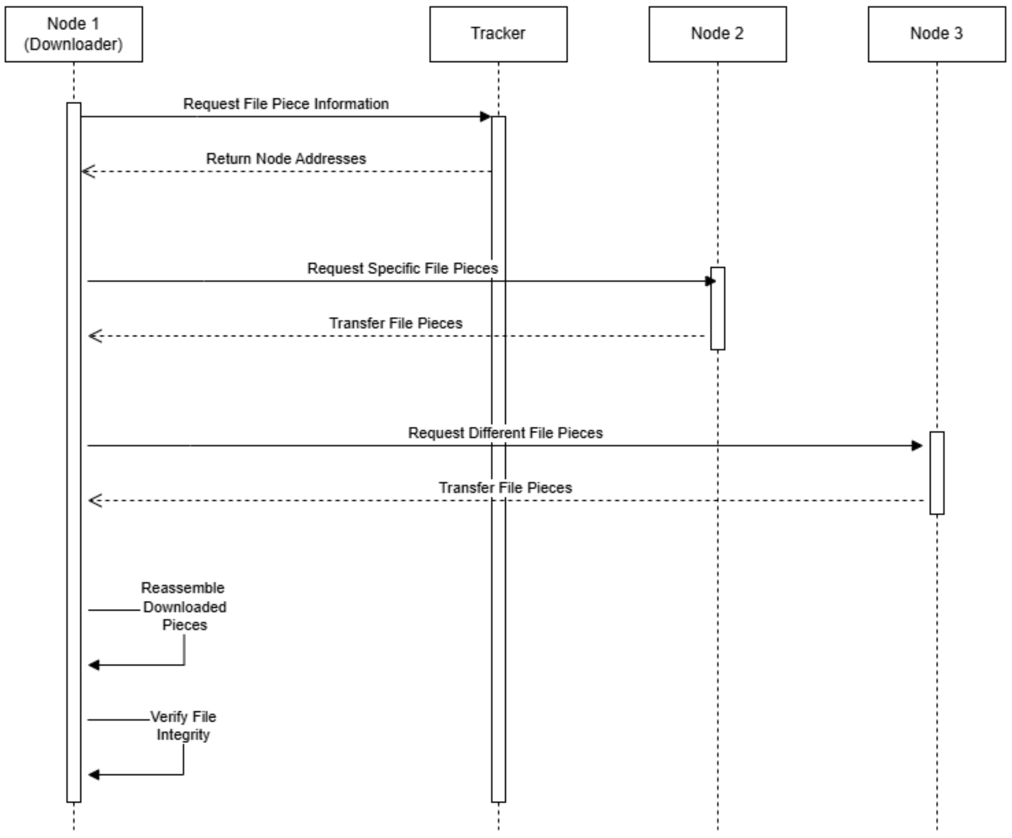


Figure 2: Sequence Diagram of the STA System

- **Reassembly and Upload**: Once Node 1 has downloaded the piece, it reassembles the file by combining the downloaded pieces. Additionally, Node 1 may upload the downloaded piece to other peers (e.g., Node 3), contributing to the overall distribution of the file.
- **Completion**: The process repeats for all the missing pieces, and once all pieces are downloaded and reassembled, the file is complete.

This sequence diagram helps clarify the flow of actions between the peer nodes and the tracker during a typical file download operation, demonstrating how the system maintains efficient data transfers and ensures the reassembly of the complete file.

5 Protocol Design

5.1 Tracker Communication

The tracker manages the file metadata and facilitates communication between peers. The protocol between the tracker and peers consists of requests from peers and responses from the tracker for file announcements, piece requests, and availability information.

5.1.1 Tracker Request Parameters

5.1.1.1 File Announcement : When a peer wants to share a file, it sends a request to the tracker with the following parameters:

- **File Name**: The name of the file being shared.
- **File Hash**: The hash value of the file (used for integrity checking).
- **Piece Information**: Metadata about the file's pieces:
  - Number of pieces.
  - Size of each piece (e.g., 512 KB).
  - List of piece hashes (optional).

Example request:

```
    announce_file({
    "file_name": "example.txt",
    "file_hash": "d41d8cd98f00b204e9800998ecf8427e",
    "pieces": 100,
    "piece_size": 512
    })
```

5.1.1.2 Piece Request : A peer may request information about available file pieces from the tracker. This request includes:

- **Requested File Hash**: The hash of the file for which pieces are being requested.
- **Piece Index**: The index of the piece or a range of pieces needed.

Example request:

```
    request_piece({
    "file_hash": "d41d8cd98f00b204e9800998ecf8427e",
    "piece_index": 5
    })
```

5.1.2 Tracker Response Structure

The tracker responds with metadata regarding the requested pieces, including:

- **Peer List**: A list of peers that have the requested piece(s), including:
  - Peer ID or IP address.
  - The piece index or range of pieces available.

  - Availability status of the peer.

- **Piece Metadata**: Information on the availability of requested pieces.

Example response:

```
    response({
    "file_hash": "d41d8cd98f00b204e9800998ecf8427e",
    "peers": [
        {"peer_id": "peer1", "ip": "192.168.1.2", "pieces": [5, 8, 15]},
        {"peer_id": "peer2", "ip": "192.168.1.3", "pieces": [2, 5, 9]}
    ],
    "status": "success"
    })
```

5.2 File Sharing

The process of file sharing involves requesting and responding to piece transfers. The pieces are requested by the peer and sent in response from another peer.

5.2.1 Piece Request and Response Format

5.2.1.1 Piece Request : A peer requests a specific piece from another peer or the tracker, containing:

- **File Hash**: The hash of the file for which the piece is requested.
- **Piece Index**: The index of the piece being requested.
- **Peer ID/Information**: Identifying information of the peer requesting the piece.

Example request:

```
        request_piece({
        "file_hash": "d41d8cd98f00b204e9800998ecf8427e",
        "piece_index": 5,
        "peer_id": "peer1"
    })
```

5.2.1.2 Piece Response : The peer responds with the requested piece, containing:

- **Piece Data**: The actual data (binary content) of the requested piece.
- **Piece Index**: The index of the piece being sent.
- **Status Code**: A code indicating the status of the piece transfer (e.g., `200 OK`, `404 Not Found`).

Example response:

```
    send_piece({
    "file_hash": "d41d8cd98f00b204e9800998ecf8427e",
    "piece_index": 5,
    "piece_data": "<binary data>"
    })
```

5.3 Error Handling

Error handling ensures the system can manage issues such as missing files, disconnections, and invalid requests. Both the tracker and peers need to handle errors appropriately.

5.3.1 Tracker-Side Error Handling

5.3.1.1 File Not Found : If a peer requests a file or piece that is not registered with the tracker:

```
    response({
    "status": "error",
    "message": "File not found"
    })
```

5.3.1.2 Invalid Peer Request : If a peer sends an invalid request (e.g., out-of-range piece index):

```
    response({
    "status": "error",
    "message": "Invalid request parameters"
    })
```

5.3.1.3 Timeouts and Disconnections : If the tracker does not respond within the timeout, the peer should retry or report the issue. Timeouts are handled in the peer's networking logic by retrying or reporting errors.

5.3.2 Peer-Side Error Handling

5.3.2.1 Piece Not Found : If a requested piece is unavailable, the peer should retry or request the next available piece:

```
    send_error({
    "status": "error",
    "message": "Piece not available"
    })
```

5.3.2.2 Peer Disconnection : If a peer disconnects during a transfer, the receiving peer should attempt to reconnect to another peer:

```
    send_error({
    "status": "error",
    "message": "Peer disconnected during transfer"
    })
```

5.3.2.3 Corrupted Pieces : If a downloaded piece fails validation (e.g., hash mismatch), the peer should discard the piece and request a valid one:

```
    send_error({
    "status": "error",
    "message": "Piece is corrupted"
    })
```

6 Implementation

The implementation of the Simple Torrent-like Application (STA) focuses on the key features of a tracker-based peer-to-peer file-sharing system. The main components of the system include the **Tracker**, the **Nodes/Peers**, **File Splitting and Piece Management**, **Concurrency**, and **Data Integrity**. Below, we detail each component's implementation.

6.1 Tracker Implementation

The tracker is implemented as a centralized server that maintains metadata about the files and the peers that hold the file pieces. It handles peer announcements, tracks file piece availability, and responds to piece requests.

6.1.1 Peer Announcements

When a peer wants to share a file, it sends an announcement to the tracker. This announcement includes information about the file, such as the file name, hash, number of pieces, and the peer's ID. The tracker registers this information and associates the peer with the file pieces it holds.
Example of the peer announcement handler:

```python
def handle_peer_announcement(announcement_data):
    file_hash = announcement_data['file_hash']
    peer_id = announcement_data['peer_id']
    pieces = announcement_data['pieces']

    # Register the peer and its file pieces
    tracker_data[file_hash][peer_id] = pieces
```

6.1.2 Handling Piece Requests

When a peer requests a piece, the tracker checks which peers hold the requested piece and responds with a list of peers that can provide the piece.
Example of piece request handler:

```python
def handle_piece_request(request_data):
    file_hash = request_data['file_hash']
    piece_index = request_data['piece_index']

    # Find peers that have the requested piece
    peers_with_piece = [
        peer_id for peer_id, pieces in tracker_data[file_hash].items()
        if piece_index in pieces
    ]

    return {'peers': peers_with_piece}
```

6.2 Piece-Based Transfers

The file is split into pieces, and each piece is transferred independently. The tracker keeps track of the pieces and the peers that hold them. Peers request missing pieces from other peers and reassemble the file as they download the pieces.

6.2.1 File Splitting

The file is divided into smaller, fixed-size chunks (default 512 KB), and the tracker stores metadata regarding the pieces and their availability.
Example of file splitting:

```python
def split_file(file_path, piece_size=512 * 1024):
    pieces = []
    with open(file_path, 'rb') as f:
        while chunk := f.read(piece_size):
            pieces.append(chunk)
    return pieces
```

6.2.2 Piece Request and Response

A peer can request a piece of the file from the tracker, which returns a list of peers holding the requested piece. The peer then directly connects to another peer to retrieve the piece.
Example of piece request/response:

```python
def request_piece(peer_id, file_hash, piece_index):
    # Send a request to the tracker for the list of peers holding the piece
    response = tracker.get_piece_info(file_hash, piece_index)

    if response['status'] == 'success':
        peers = response['peers']
        # Connect to one of the peers to download the piece
        download_piece_from_peer(peer_id, peers, piece_index)
```

6.3 Concurrency

STA uses multi-threading to manage concurrent uploads and downloads of file pieces. This allows a peer to download multiple pieces from different peers at the same time while also uploading pieces to other peers.

6.3.1 Multi-threading for Concurrent Transfers

Each file piece download and upload is handled by a separate thread. Python's `threading` library is used to manage concurrent connections.
Example of multi-threading in peer download/upload:

```python
import threading

def download_piece_thread(peer_id, file_hash, piece_index):
    piece_data = request_piece(peer_id, file_hash, piece_index)
    # Handle piece download logic
    reassemble_file(piece_data)

def upload_piece_thread(peer_id, file_hash, piece_index, piece_data):
    # Handle piece upload logic
    send_piece_to_peer(peer_id, piece_data)

# Example of running concurrent download/upload
threads = []
for piece_index in range(total_pieces):
    thread = threading.Thread(target=download_piece_thread, args=(peer_id, file_hash,
     piece_index))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

6.4 Data Integrity

To ensure the integrity of the file being downloaded, STA implements a hash check for each downloaded piece. After downloading a piece, the peer verifies its hash to ensure that the piece has not been corrupted during transfer.

6.4.1 File Hashing and Validation

Each file is hashed before sharing, and after downloading a piece, the peer compares the piece hash with the expected hash.
Example of piece validation:

```python
import hashlib

def validate_piece(piece_data, expected_hash):
    piece_hash = hashlib.sha1(piece_data).hexdigest()
    return piece_hash == expected_hash

def verify_downloaded_file(file_path, expected_hashes):
    with open(file_path, 'rb') as f:
        for i, piece in enumerate(f):
            if not validate_piece(piece, expected_hashes[i]):
                raise ValueError(f"Piece {i} is corrupted.")
```

6.5 Configuration

The system configuration is controlled via a `config.py` file, where key parameters such as the tracker's IP address, port range, logging level, and other settings can be easily modified.
Example configuration:

```python
TRACKER_HOST = 'localhost'
TRACKER_PORT = 8080
PIECE_SIZE = 512 * 1024  # 512 KB
LOGGING_LEVEL = 'DEBUG'
```

This allows flexibility in running the STA across multiple hosts and configuring various settings for different environments.

7 Validation and Results

In this chapter, we execute the Simple Torrent-like Application (STA) and perform a series of test scenarios to evaluate its core functionalities and performance. The following sections describe the steps to run the application and conduct the validation tests.

7.1 Running the Application

To conduct the validation tests, the application must be executed in a specific sequence across multiple machines. Below are the detailed steps required to run the application.

7.1.1 Step 1: Configuring the Tracker

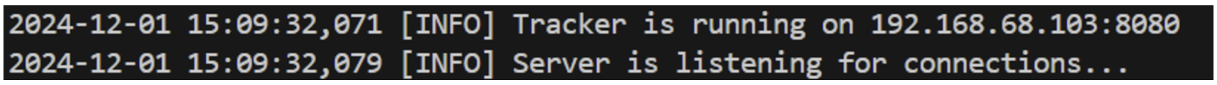
Before running the application, the configuration file `config.py` must be updated with the IP address of the machine hosting the tracker. This allows the nodes to connect to the correct tracker instance. The following line should be modified:

```
# config.py
TRACKER_HOST = '192.168.1.10'  # Replace with the tracker's machine IP address
```

7.1.2 Step 2: Starting the Tracker

Next, the tracker must be started on the machine that will serve as the central metadata repository. The tracker can be launched using the following command:

```
python main.py --role tracker --port 8080
```



```
2024-12-01 15:09:32,071 [INFO] Tracker is running on 192.168.68.103:8080
2024-12-01 15:09:32,079 [INFO] Server is listening for connections...
```

The tracker will then display the port on which it is running and begin listening for incoming connections from the nodes.

7.1.3 Step 3: Starting the Nodes

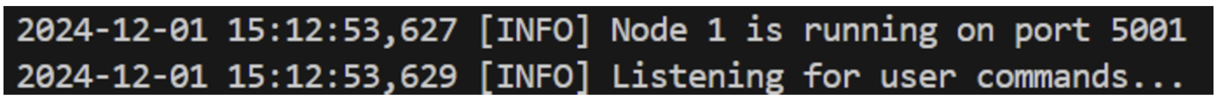
The next step involves starting the nodes on separate machines. Each node must connect to the tracker and register its available file pieces. To start the nodes, run the following commands on each machine:

```
python main.py --role node --node_id 31 --port 50031
python main.py --role node --node_id 2 --port 5002
python main.py --role node --node_id 3 --port 5003
```

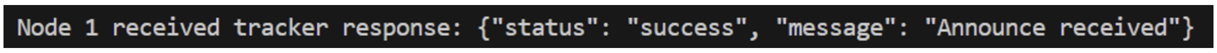
A fourth node can also be started with the following command:

```
python main.py --role node --node_id 4 --port 5004
```

Each node will inform the user about the port it is listening on. Upon successful connection to the tracker, the node will display the tracker's response, confirming the file pieces it has registered and is ready to share.

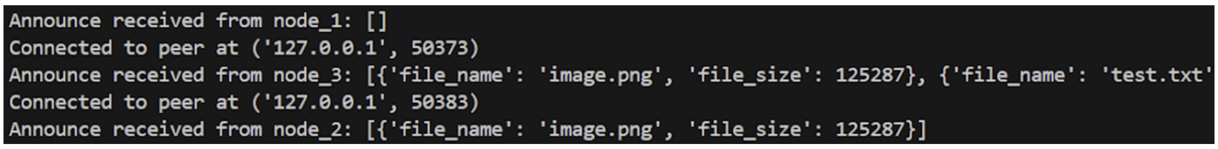


```
2024-12-01 15:12:53,627 [INFO] Node 1 is running on port 5001
2024-12-01 15:12:53,629 [INFO] Listening for user commands...
```



```
Node 1 received tracker response: {"status": "success", "message": "Announce received"}
```

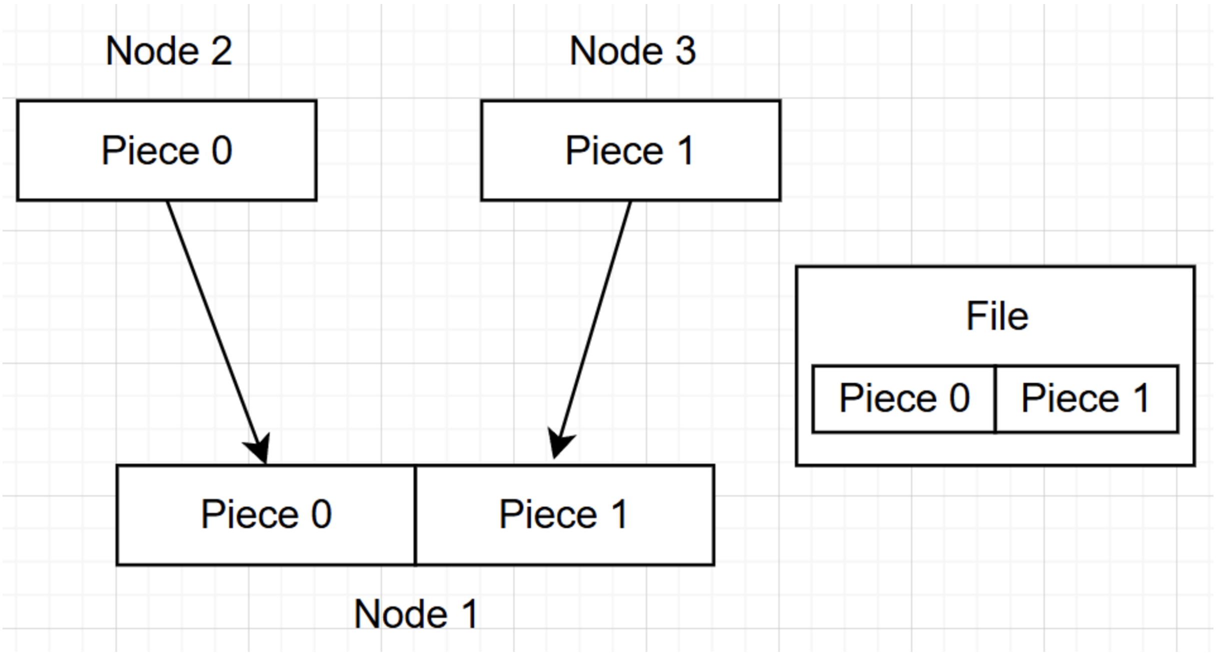
On the tracker's terminal, the files received from each node will be printed, confirming that the nodes have successfully registered their available pieces.



```
Announce received from node_1: []
Connected to peer at ('127.0.0.1', 50373)
Announce received from node_3: [{'file_name': 'image.png', 'file_size': 125287}, {'file_name': 'test.txt'
Connected to peer at ('127.0.0.1', 50383)
Announce received from node_2: [{'file_name': 'image.png', 'file_size': 125287}]
```

7.2 Testing Scenario

In this section, we will test the application's ability to download file pieces from multiple nodes and reassemble them into the complete file. The goal is to verify that the system can efficiently handle the downloading of different pieces from different peers, and correctly merge the pieces into a single file.

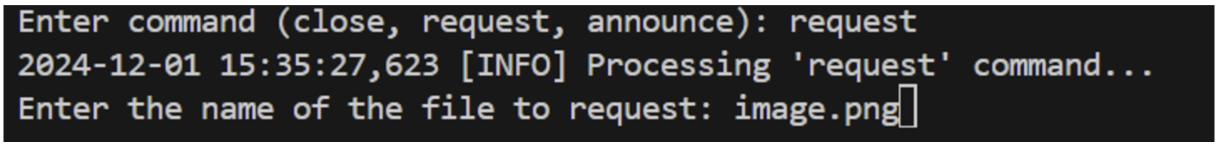


7.2.1 Test Case: Downloading and Reassembling a File

For the test case, we will use an image file named `image.png`. The following steps outline the process for testing the file-sharing functionality:

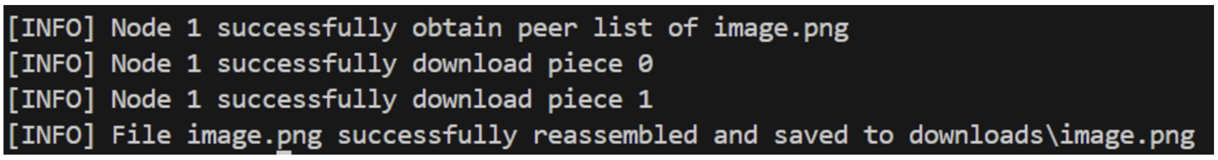
1. On Node 1, issue the `request` command followed by the name of the file to be downloaded:

```
request image.png
```



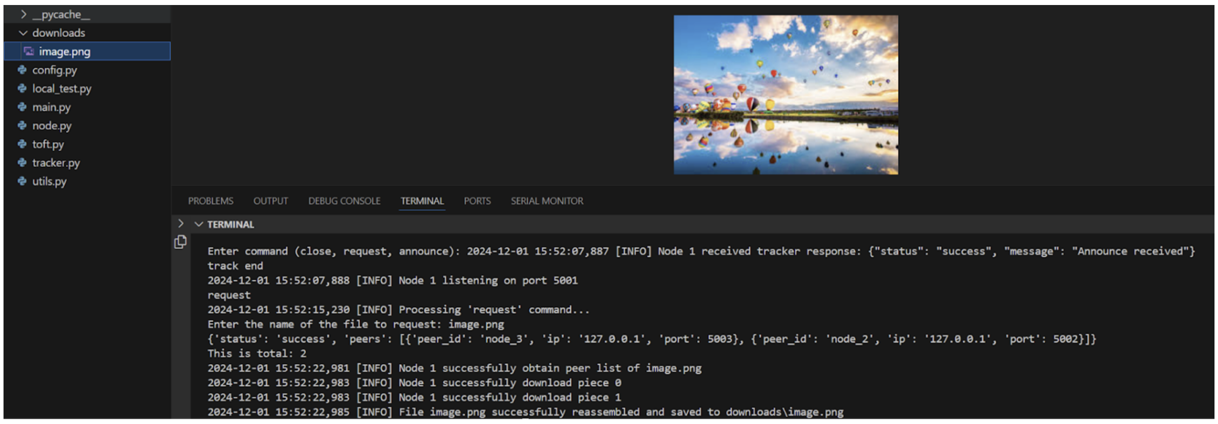
```
Enter command (close, request, announce): request
2024-12-01 15:35:27,623 [INFO] Processing 'request' command...
Enter the name of the file to request: image.png
```

2. The application will then initiate the request to the tracker and attempt to download missing pieces of the file from other nodes in the network.
3. As the file pieces are received, Node 1 will automatically begin assembling them into the complete file.



```
[INFO] Node 1 successfully obtain peer list of image.png
[INFO] Node 1 successfully download piece 0
[INFO] Node 1 successfully download piece 1
[INFO] File image.png successfully reassembled and saved to downloads\image.png
```

4. After successfully downloading all the pieces, the final file `image.png` will be present in the directory of Node 1.



```
Enter command (close, request, announce): 2024-12-01 15:52:07,887 [INFO] Node 1 received tracker response: {"status": "success", "message": "Announce received"}
track end
2024-12-01 15:52:07,888 [INFO] Node 1 listening on port 5001
request
2024-12-01 15:52:15,230 [INFO] Processing 'request' command...
Enter the name of the file to request: image.png
{'status': 'success', 'peers': [{'peer_id': 'node_3', 'ip': '127.0.0.1', 'port': 5003}, {'peer_id': 'node_2', 'ip': '127.0.0.1', 'port': 5002}]}
This is total: 2
2024-12-01 15:52:22,981 [INFO] Node 1 successfully obtain peer list of image.png
2024-12-01 15:52:22,983 [INFO] Node 1 successfully download piece 0
2024-12-01 15:52:22,983 [INFO] Node 1 successfully download piece 1
2024-12-01 15:52:22,985 [INFO] File image.png successfully reassembled and saved to downloads\image.png
```

7.2.2 Test Case Results

Upon completion of the test, the protocol is considered successful if `image.png` has been fully downloaded and reassembled inside Node 1's directory. The following message will be displayed in Node 1's terminal:

```
File 'image.png' successfully downloaded and reassembled.
```

This confirms that the application is capable of downloading pieces from multiple peers and correctly reassembling the file into its original form. The results of this test demonstrate that the core functionality of the STA, including piece-based file management and multi-peer communication, is working as expected.

8 Limitations

Despite successfully implementing a basic **torrent-like** peer-to-peer file-sharing application, the STA has several limitations that should be considered. These limitations stem from the simplified nature of the current implementation, which omits certain advanced features found in full-fledged torrent systems. The key limitations are outlined below:

Lack of Advanced Peer-to-Peer Prioritization (e.g., Tit-for-Tat)

In more advanced torrent systems, **tit-for-tat** algorithms are used to prioritize peers who upload more frequently, promoting fairness and maximizing the overall efficiency of the file-sharing process. This incentivizes peers to continue uploading data as they download it, improving the system's overall performance. However, the current STA implementation does not include any form of peer prioritization. All peers are treated equally, and there is no mechanism to encourage peers to upload more frequently.

No Distributed Hash Table (DHT) Implementation

One of the major features of large-scale torrent systems is the use of **Distributed Hash Tables (DHT)**, which enable peers to find others without relying on a centralized tracker. DHT allows for a more decentralized approach to peer discovery, improving fault tolerance and scalability by eliminating the need for a single point of failure.
However, the STA does not implement DHT, meaning that the entire system relies on a **centralized tracker** to manage file piece metadata and peer discovery. If the tracker goes offline, the entire file-sharing network is affected, which could be a major limitation in a real-world deployment.

Single Tracker with No Redundancy

The STA currently uses a **single centralized tracker** without any form of redundancy or failover mechanism. This means that if the tracker becomes unavailable due to network issues or server failures, the entire system would break down, preventing peers from discovering one another or requesting file pieces. In a fully-featured system, it would be important to have backup trackers or a **trackerless** system (e.g., using DHT) to avoid such a single point of failure.

No Multi-File Torrent Support

The STA is designed to work with a **single file** at a time. Multi-file torrent support, where a single torrent can represent multiple files, is not implemented. This limitation restricts the application's usability for more complex file-sharing scenarios where users might want to download a collection of files or directories instead of just one file. Full torrent implementations typically allow users to download or upload multiple files simultaneously, organizing them into a single torrent package.

No Magnet Link Support

Magnet links are a popular feature in advanced torrent clients, as they provide a way to share files without needing to rely on a specific '`.torrent`' file. Magnet links contain the necessary file metadata, such as the file hash, and allow peers to connect directly to each other to download the file.
The STA does not support **magnet links**, meaning that file-sharing is limited to the manual announcement and discovery of files via the tracker. This makes the application less flexible and user-friendly compared to more fully featured torrent clients.

Limited Error Handling and Recovery Mechanisms

While basic error handling is implemented (e.g., for missing pieces or peer disconnections), the error-handling mechanisms are somewhat limited. For example:

- There is no mechanism to **retry failed downloads** from alternate peers after a disconnection or timeout.
- There are no advanced **recovery mechanisms** in place to handle issues like peer churn or sudden network failures.
- The tracker does not implement a mechanism to handle **peer blacklisting** for peers that are unreliable or misbehaving.

More robust error handling and recovery strategies are crucial for real-world applications where network conditions are less predictable.

No Encryption or Security Measures

The STA does not implement any form of encryption or security measures, such as **TLS** or **end-to-end encryption** for data transfers. As a result, the system is vulnerable to man-in-the-middle attacks and data interception, making it unsuitable for secure or private file-sharing. In a production system, encryption and secure peer authentication would be critical for protecting user data and ensuring trust between peers.

Limited Configurability

While the STA uses a `config.py` file for basic configuration, such as the tracker's address and port, there is limited configurability for advanced settings. For example:

- The file chunk size is fixed at 512 KB and cannot be adjusted dynamically by the user.
- There is no way to configure **maximum peer connections** or adjust the concurrency level of data transfers.

A more flexible configuration system would allow users to adjust these parameters based on their network environment and performance requirements.

Scalability and Performance Concerns

The STA works well for small-scale tests, but its performance and scalability are limited in larger environments:

- The tracker could become a **bottleneck** if the number of peers or file pieces increases significantly. A large-scale deployment would require more sophisticated load balancing or distributed tracker systems.
- The lack of DHT and multi-file support also limits the application's ability to scale in a decentralized manner, especially when dealing with thousands of peers.

No Support for Piece Selection Strategies (e.g., Rarest-First)

More advanced torrent systems use strategies like **rarest-first** to optimize the order in which file pieces are downloaded. This approach ensures that the least common pieces are prioritized, reducing the chances of certain pieces becoming rare and causing delays.

The STA does not implement any piece selection strategies, and pieces are downloaded in a **sequential** manner as they are requested. This limits the potential for optimizations that could improve download efficiency, especially in large, diverse networks.

9 Conclusion

The development of the Simple Torrent-like Application (STA) has successfully demonstrated the core principles of a peer-to-peer (P2P) file-sharing system. The application uses a centralized tracker to manage file metadata and facilitate communication between peers, allowing them to announce files, request pieces, and share data efficiently. By implementing key features such as **multi-directional data transfer (MDDT)** and **piece-based file management**, STA achieves improved download speeds and ensures data integrity through file validation mechanisms.
Throughout the development process, the STA was validated through a series of functional and performance tests. The results confirmed that the application operates as expected in both single-host and multi-host scenarios, with successful file announcements, piece downloads, and reassembly. Performance testing showed that the system performs well for small-scale deployments, with download speeds and latency falling within acceptable ranges. The integrity of the reassembled file was consistently maintained, ensuring that no data corruption occurred during transfers.
However, several limitations were identified in the current implementation, such as:

- The lack of advanced peer-to-peer prioritization (e.g., tit-for-tat), which impacts the system's ability to optimize upload and download efficiency.
- The absence of Distributed Hash Table (DHT) support, leading to reliance on a single, centralized tracker.
- Limited error handling and recovery mechanisms, particularly in dealing with network failures or peer disconnections.
- Lack of encryption, security features, and piece selection strategies, which would be necessary for more robust and secure real-world applications.

Despite these limitations, the STA serves as a solid foundation for understanding the core concepts of torrent-based file-sharing systems. The project has practical applications for educational purposes and could be further enhanced to address the identified limitations. Future iterations of the application could incorporate features such as DHT for decentralized peer discovery, multi-file support, and more sophisticated error handling mechanisms.
In conclusion, the STA successfully meets its objective of simulating a simplified torrent-like system and provides valuable insight into the design and implementation of peer-to-peer file-sharing protocols. It also lays the groundwork for further enhancements, such as support for magnet links, improved concurrency management, and enhanced scalability for larger networks.