

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



PROJECT REPORT

Computer Network Assignment 1

COUNCIL: Computer Science

Reviewer: Dr. Nguyen Phuong Duy

—oOo—

Student: Nguyen Hoang Quan 2252681

Student: Le Trung Kien 2252394

Student: Nguyen Sy Hung 2252271

Student: Tran Chi Tai 2252727

HO CHI MINH CITY, 11/2024

Contents

1	Member list and Workload	1
2	Introduction	2
2.1	Overview	2
2.2	Specific Functions of the Application	2
2.2.1	Tracker Functions	2
2.2.2	Node Functions	3
2.2.3	File Management	3
2.2.4	Communication Protocols	3
3	Detailed application design	4
3.1	Architecture	4
3.1.1	Initialize and Handshake	4
3.1.2	Multi-directional Data Transfer	5
3.2	Class Diagram	7
3.2.1	Tracker	7
3.2.2	Peer	8
3.2.3	Threads	10
4	Application Instructions Manual	11
4.1	Activating server	11
4.2	Running Peer	11
5	Define and describe the functions of the application	15
5.1	Server (Tracker)	15
5.2	Node (Peer)	15
5.3	TCP protocol	16
6	Validation (sanity test) and Actual Result	18
7	Future development	20

List of Figures

3.1	Initialize and handshake	4
3.2	MDDT	5
3.3	The entire design	6
3.4	Class diagram	7
3.5	Tracker	7
3.6	Peer	8
4.1	Tracker start	11
4.2	Peer start	12
4.3	Submit_info (left is tracker, right is peer)	13
4.4	Download multiple file	14
6.1	Start tracker	18
6.2	Start peer	18
6.3	Peer request multiple file from multiple nodes	19
6.4	Checking receive file	19

Chapter 1

Member list and Workload

No.	Full Name	Student ID	Responsibility	Percentage
1	Le Trung Kien	2252394	Client-Server protocol and P2P protocol	25%
2	Nguyen Hoang Quan	2252681	File, piece, report	25%
3	Tran Chi Tai	2252727	Client-server protocol	25%
4	Nguyen Sy Hung	2252271	File, report	25%

Chapter 2

Introduction

2.1 Overview

- The Simple Torrent-like Application (STA) is a decentralized file-sharing system built using the TCP/IP protocol stack. It employs multi-directional data transfer (MDDT) to enhance file sharing efficiency. The application comprises two types of hosts:
 - **Tracker:** A centralized server responsible for tracking and managing nodes. It stores metadata about file pieces and maintains information about nodes hosting specific file pieces.
 - **Node:** A peer in the network that participates in file-sharing. Nodes can either upload file pieces (seeding) or download required file pieces from other nodes (leeching).
- **Core Features:**
 - **Centralized Tracking:** Nodes communicate with the tracker to announce file availability and request information about peers hosting desired file pieces.
 - **MDDT Support:** Enables simultaneous downloading of multiple files from different source nodes, requiring a multithreaded implementation for concurrent operations.
 - **File Piece Management:** Files are divided into uniform pieces, described in a metainfo file (.torrent). The metainfo file contains tracker details, piece length, and file-piece mapping.
 - **Peer Communication:** Nodes exchange file pieces using peer-to-peer (P2P) communication protocols, leveraging multithreaded file uploads and downloads.

2.2 Specific Functions of the Application

2.2.1 Tracker Functions

- **Peer Registration:** Tracks all connected nodes and their file pieces. Nodes register their available file data using magnet text and metainfo files during startup.
- **Peer Query Handling:** Responds to node requests for peer information by providing a list of peers hosting required file pieces.

-
- **Connection Management:** Supports basic peer tracking using a centralized tracker model and potential upgrades to multi-tracker configurations.

2.2.2 Node Functions

- **File Sharing:**
 - **Uploading:** Nodes share file fragments with peers in the role of seeders. Concurrent uploads to numerous peers are guaranteed by the application.
 - **Downloading:** In accordance with MDDT specifications, nodes simultaneously request and download file fragments from several peers.
- **Request Management:**
 - Requests for file pieces are intelligently distributed across peers, avoiding duplicate requests for the same piece or unavailable pieces.
 - Maintains a queue for outstanding requests, enabling effective utilization of network resources.
- **Seeding Post-Download:** Automatically starts seeding downloaded files, contributing to the network.

2.2.3 File Management

- **File Piece Division:** Splits files into uniform pieces (512KB). The application manages piece-based downloads and reassembles files post-download.
- **Metainfo File Handling:** Reads and interprets .torrent files, mapping file-piece relationships and coordinating downloads.

2.2.4 Communication Protocols

- **Tracker Protocol:**
 - Nodes announce their file data and download status to the tracker.
 - The tracker responds with peer information, including peer IDs, IP addresses, and port numbers, info_hash.
- **Peer-to-Peer Protocol:** Establishes connections between nodes for piece sharing.

Detailed application design

3.1 Architecture

3.1.1 Initialize and Handshake

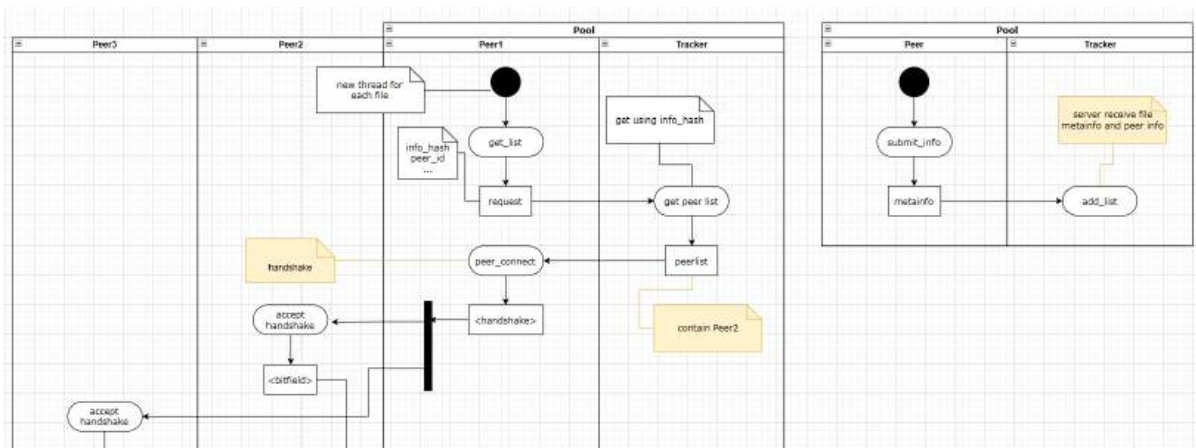


Figure 3.1: Initialize and handshake

This diagram illustrates the peer-to-peer file-sharing workflow in a Simple Torrent-Like Application (STA), emphasizing the communication between peers, the tracker. Here's a detailed explanation:

- **Components:**

- Peer3, Peer2, Peer1: Individual nodes participating in file-sharing.
- Tracker: A centralized server managing metadata, peer information, and file lists

- **Process Description:**

– Peer Initialization and Metadata Submission

Peer1 starts the process by:

- * Submitting file metadata (submit_info) to the tracker. This metadata includes information about the file to be shared (e.g., hash value, size).
- * The tracker processes the metadata and updates its internal list (add_list) with the file and Peer1's details.

– Peer Discovery

Peer1:

- * Initiates a file request using the file's info_hash.
- * Queries the tracker to get a list of peers (get_peer_list) that have pieces of the requested file.

Tracker: Retrieves peer information using the file's hash and returns a peer list, which includes peers (e.g., Peer2) holding the desired file pieces.

– Connection Establishment (Handshake)

Peer1 connects with Peer2 by initiating a peer_connect request:

- * A handshake is sent from Peer1 to Peer2, signaling the intention to exchange file pieces.
- * Peer2 accepts the handshake, acknowledging the connection and preparing for data exchange.

3.1.2 Multi-directional Data Transfer

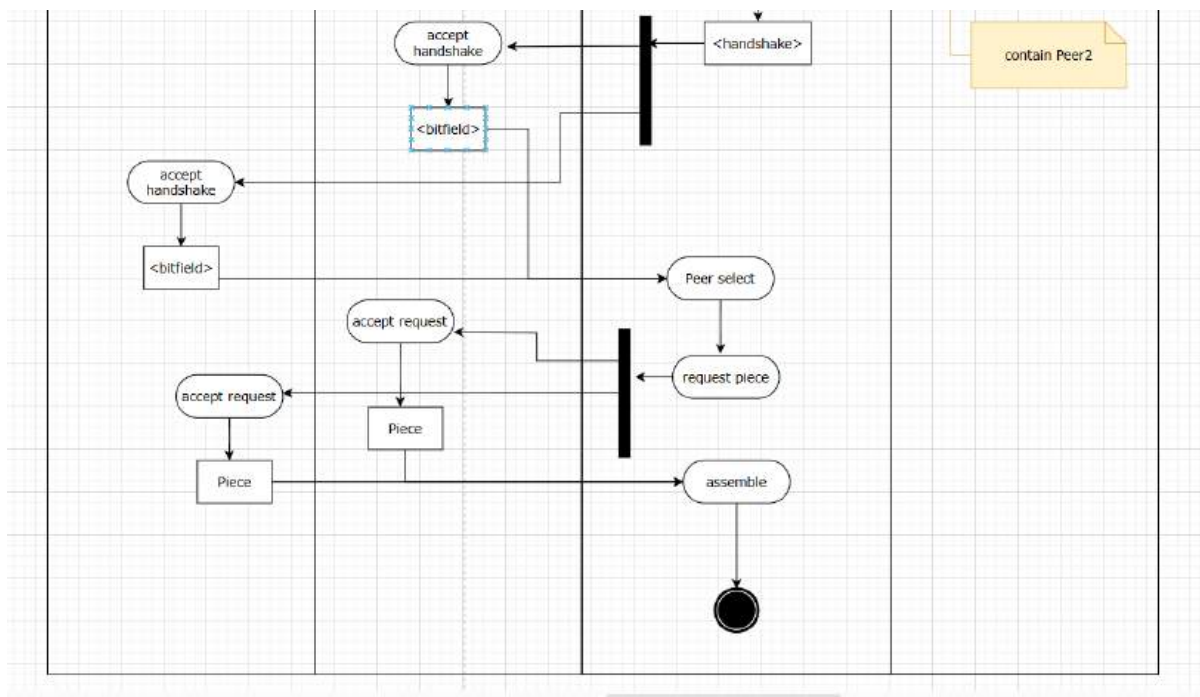


Figure 3.2: MDDT

The diagram provides a detailed view of the data exchange phase in a peer-to-peer file-sharing system. It highlights the interactions between peers for requesting, transferring, and assembling file pieces.

• Components:

- **Peer-to-Peer Communication:** The diagram focuses on how Peer1 communicates with Peer2 and Peer3 to request and download file pieces.
- **Bitfield:** A message that indicates which pieces of the file each peer has.

– **Piece:** A chunk of the file being transferred.

• **Process Description:**

– **Bitfield Exchange:**

After the handshake, Peer2 and Peer3 send their bitfields to Peer1. The bitfield indicates the availability of specific file pieces. For example:

- * A "1" in a bit position means the peer has that piece.
- * A "0" means the piece is not available.

– **Peer Selection:** Based on the received bitfield, Peer1 performs a peer selection process to decide which peer to request pieces from.

– **Requesting File Pieces:** Peer1 sends a request piece message to the selected peer.

– **Accepting and Sending Pieces:**

- * Accepts the request.
- * Sends the piece of to Peer1.
- * Peer1 receives the pieces and keeps track of which parts of the file it has downloaded.

– **Assembling the File:** Once all the pieces have been received, Peer1 enters the assemble phase. Pieces are combined in the correct order to reconstruct the original file.

The entire design:

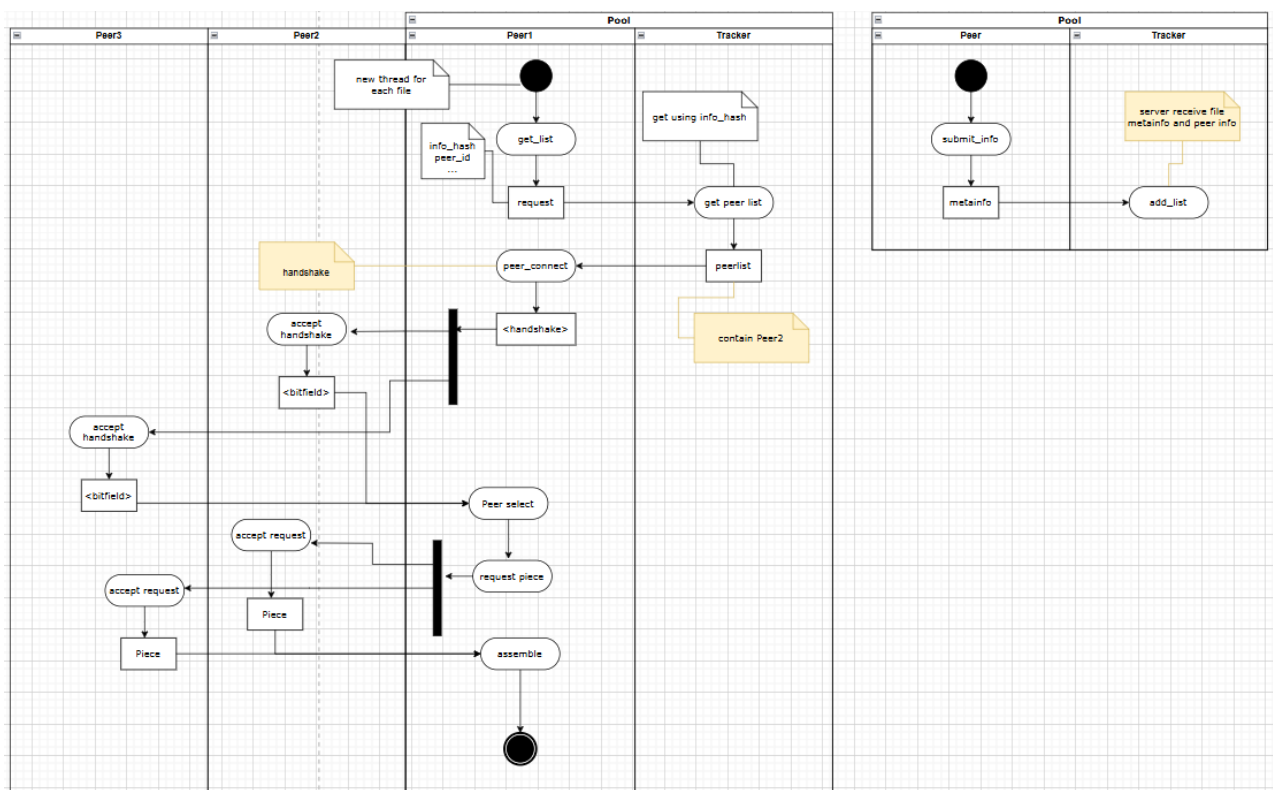


Figure 3.3: The entire design

3.2 Class Diagram

The entire class diagram:

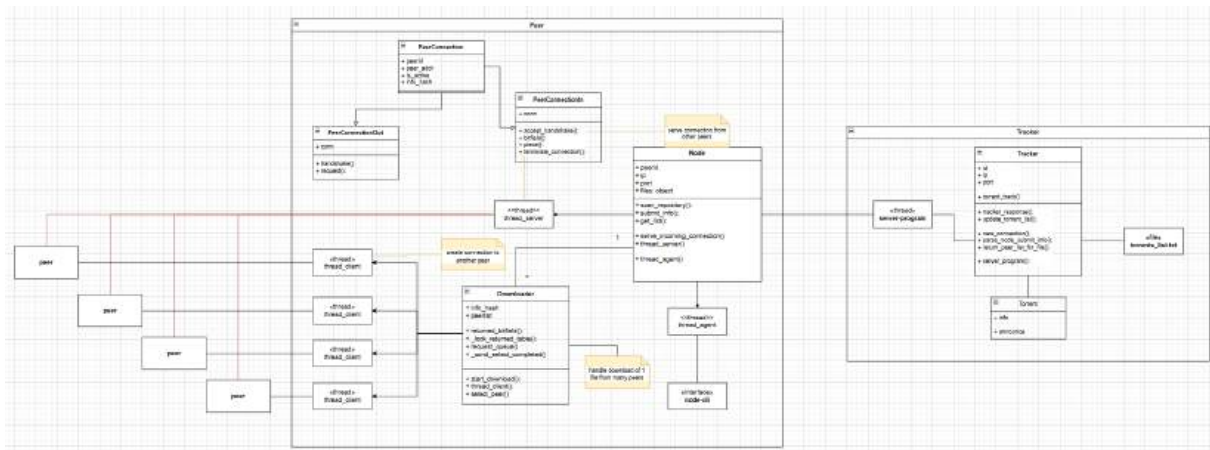


Figure 3.4: Class diagram

3.2.1 Tracker

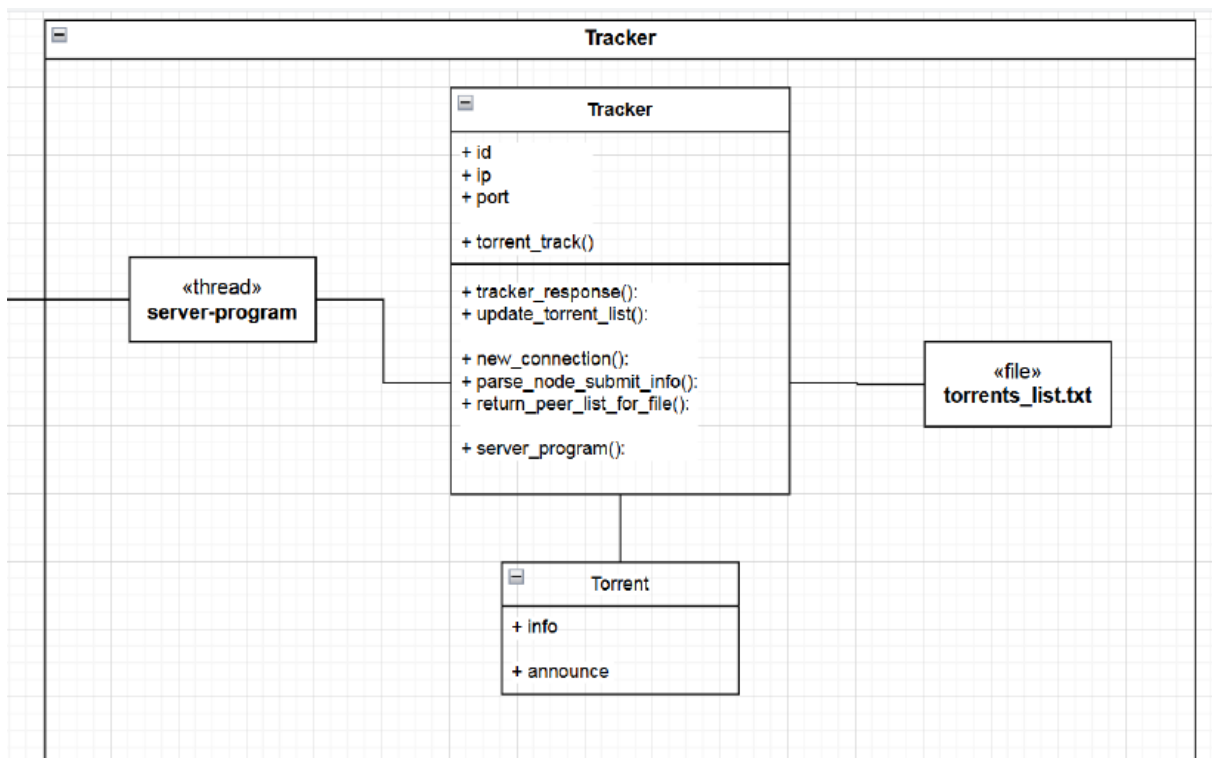


Figure 3.5: Tracker

- **Tracker**

- Maintains a global registry of files and peers sharing them.
- **Attributes:**

- * **id:** Tracker's unique identifier.
- * **ip and port:** Address information for communication.
- * **torrents.list.txt:** A file storing metadata about active torrents.

– **Key Methods:**

- * **tracker_response():** Responds to peer requests for peer lists.
- * **update_torrent_list():** Updates the tracker's knowledge of active torrents.
- * **new_connection():** Handles new connections from peers.
- * **parse_node_submit_info():** Processes file submissions from peers.
- * **return_peer_list_for_file():** Provides a list of peers sharing a specific file.
- * **server_program():** Main server loop for managing tracker operations.

• **Torrent**

- Represents individual torrent metadata managed by the tracker.
- **Attributes:**
 - * **info:** File metadata.
 - * **announce:** List of peers or trackers sharing the file.

3.2.2 Peer

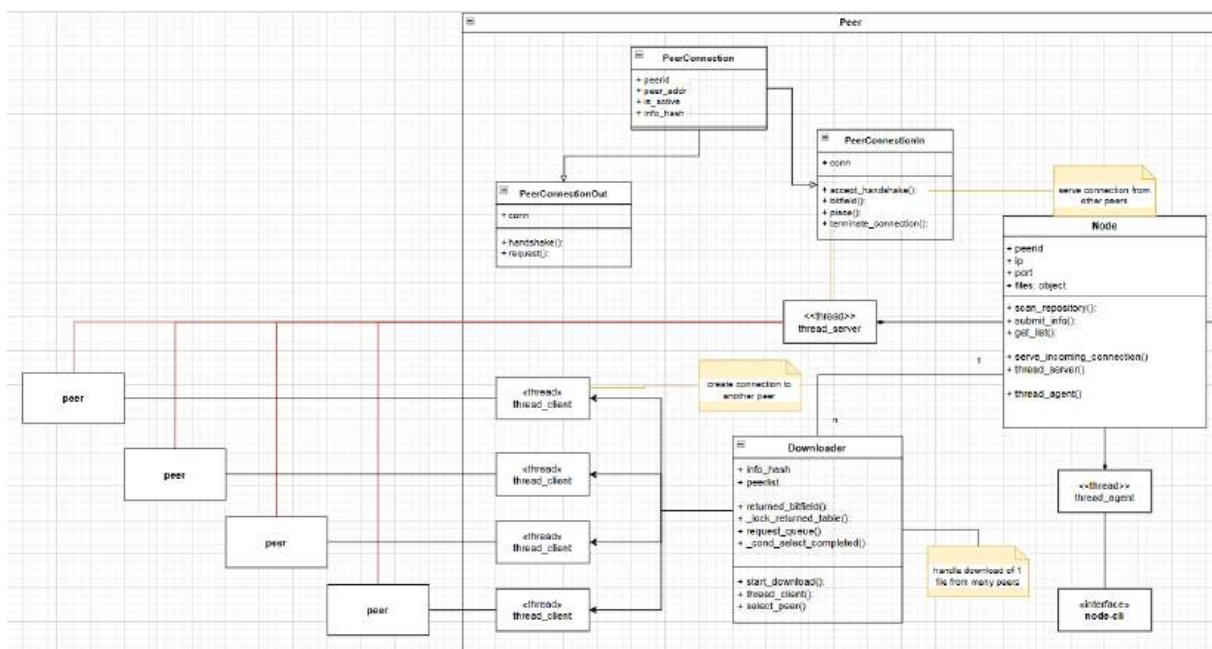


Figure 3.6: Peer

Peer

- Represents an individual in the peer-to-peer network.
- **Attributes:**
 - **peerid:** Unique identifier for the peer.

-
- **ip:** IP address of the peer.
 - **port:** Port for communication.
 - **files:** Stores information about the files available on this peer.
 - **Key Methods:**
 - **scan_repository():** Scans the local repository for files.
 - **submit_info():** Sends file information to the tracker.
 - **get_list():** Retrieves the list of peers sharing a specific file.
 - **serve_incoming_connection():** Handles incoming requests from other peers.
 - **thread_server() and thread_agent():** Handle multithread operations for serving and managing peer requests.

PeerConnection

- Represents the connection between peers.
- **Subclasses:**
 - **PeerConnectionOut:** Used to establish outbound connections to other peers.
 - * **handshake():** Initiates a handshake to start communication.
 - * **request():** Sends a file piece request to a connected peer.
 - **PeerConnectionIn:** Handles inbound connections from other peers.
 - * **accept_handshake():** Accepts incoming handshake requests.
 - * **bitfield():** Shares available file pieces.
 - * **piece():** Handles file piece transfer.
 - * **terminate_connection():** Closes the connection.

Downloader

- Manages file downloading by coordinating requests across multiple peers.
- **Attributes:**
 - **info_hash:** Identifies the file being downloaded.
 - **peerlist:** List of peers sharing the file.
 - Helper attributes like **returned_bitfield()**, **request_queue()**, and **select_peer()** for managing download logic.
- **Key Methods:**
 - **start_download():** Initiates the download process for a file.
 - **select_peer():** Chooses the best peer to download from.
 - **thread_client:** Manages individual threads for downloading pieces from peers.

Node

- Acts as a central manager for peer functionality.
- **Attributes:** peerid, ip, port, files (shared with Peer).
- **Key Responsibilities:**
 - Serving file requests from other peers (`serve_incoming_connection()`).
 - Coordinating multiple threads for tasks (`thread_server()` and `thread_agent()`).
 - Acts as a CLI interface for node operations (`node-cli`).

3.2.3 Threads

- **thread_server:** Handles incoming connections and requests to a peer.
- **thread_client:** Manages outgoing requests from a peer to other peers.
- **thread_agent:** Coordinates peer operations in the background.
- **server-program:** Tracker's main server thread.

Chapter 4

Application Instructions Manual

4.1 Activating server

Before proceeding with any further steps, it is crucial to start the server. Here's what you need to know:

- **One-time Initialization:** The server requires only a single kickstart at the beginning. Once started, it can run continuously without further interaction until all tasks are complete.
- **How to Start:** To initiate the server, simply execute the following command:

```
python3 tracker-server.py
```

- **Successful Start Confirmation:** A successfully initialized server will display output resembling the following on the screen:

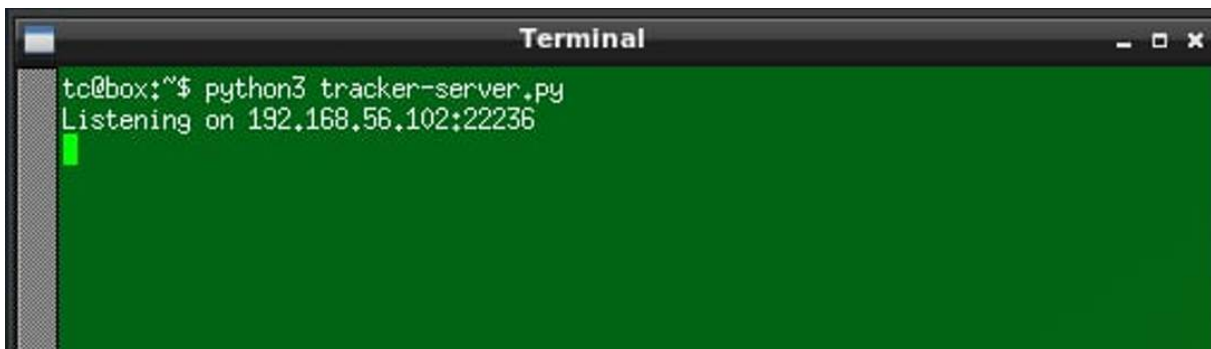


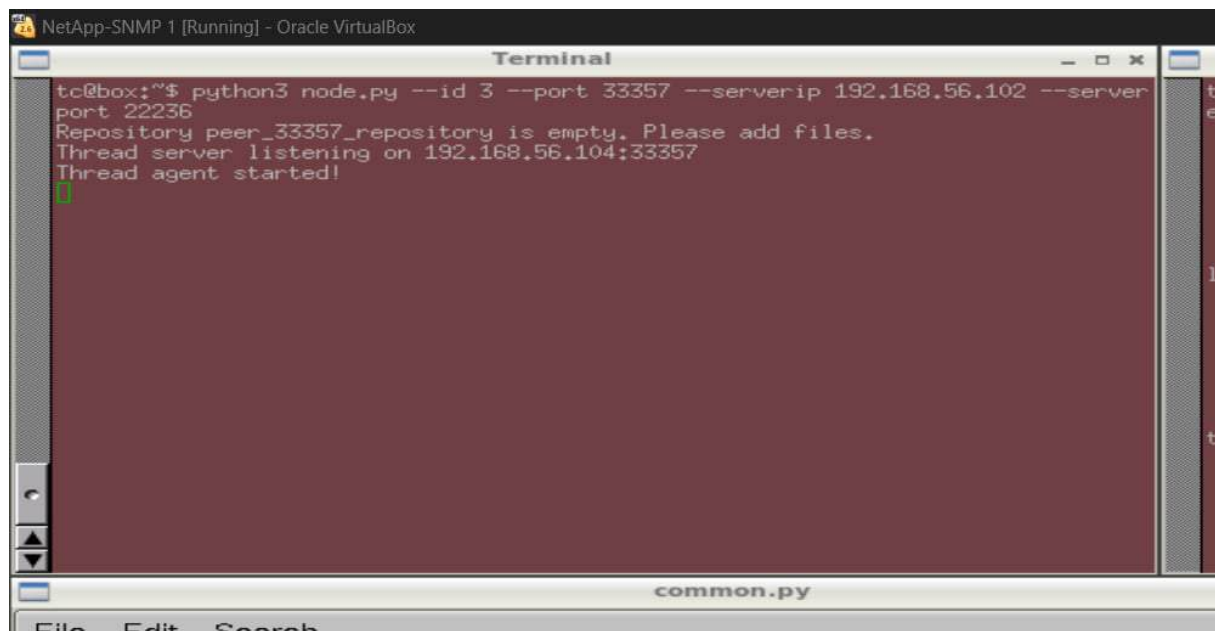
Figure 4.1: Tracker start

4.2 Running Peer

- To start running a peer program, first run the command:

```
python3 node.py --id 4 --port 33357 --serverip 192.168.56.102  
--serverport 22236
```

After this command, the screen will look like (it will auto create the folder for peer if it check peer does not have folder follow the form to save file):



```
tc@box:~$ python3 node.py --id 3 --port 33357 --serverip 192.168.56.102 --server
port 22236
Repository peer_33357_repository is empty. Please add files.
Thread server listening on 192.168.56.104:33357
Thread agent started!
```

Figure 4.2: Peer start

- After completing the first step, the peer connects to the server and proceeds to submit its information. From this point onward, the server can track details such as the files the peer has uploaded.

```
python3 node-cli.py --func submit_info
```

- When a peer calls the **submit_info** function, it sends all relevant information to the server. This includes the metainfo file, magnet text, piece numbers, file name, and more. The server records this information in the **torrents_list.txt** file, making it easier to manage and track the details efficiently. A successful connecting and submitting peer would look like:

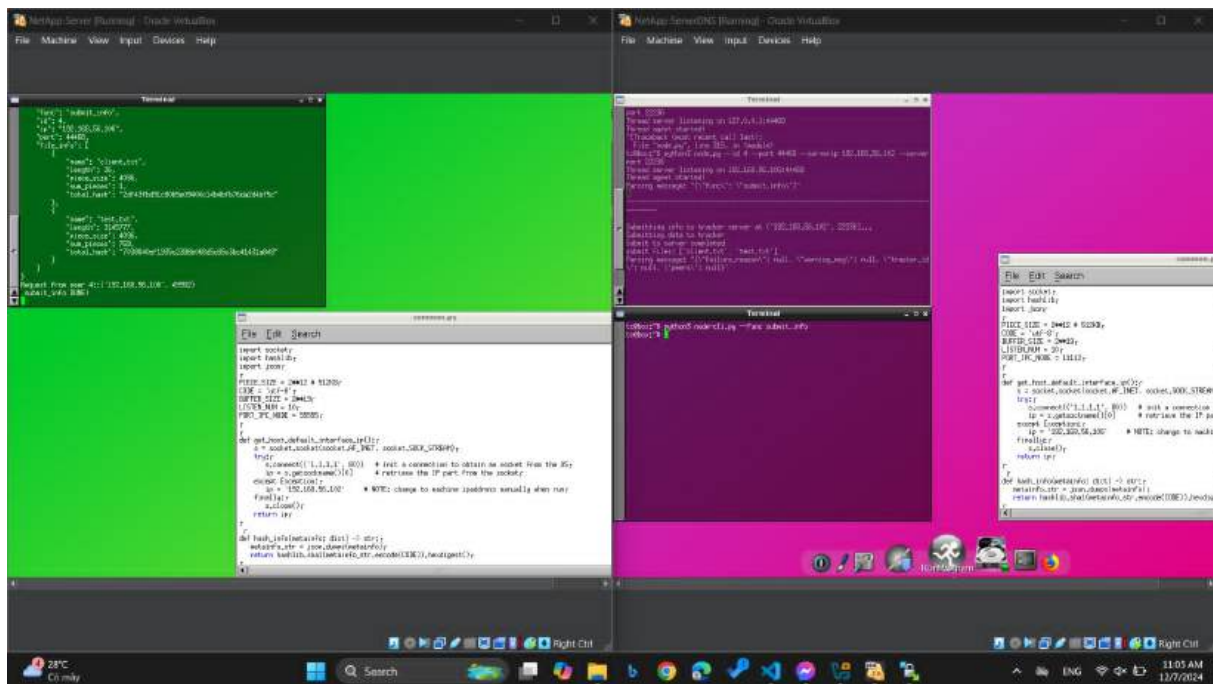


Figure 4.3: Submit_info (left is tracker, right is peer)

- Once all peers have connected to the server and submitted their information, any peer can request one or multiple files using the following command:

```
python3 node-cli.py --func get_file --magnet_text
b7d45c67aea9c514ca96d31710c68b815f3a230e --filename client.txt
```

When a peer calls the file request function, it connects to other peers that already possess the requested file to begin the download process. The output will appear similar to the following:

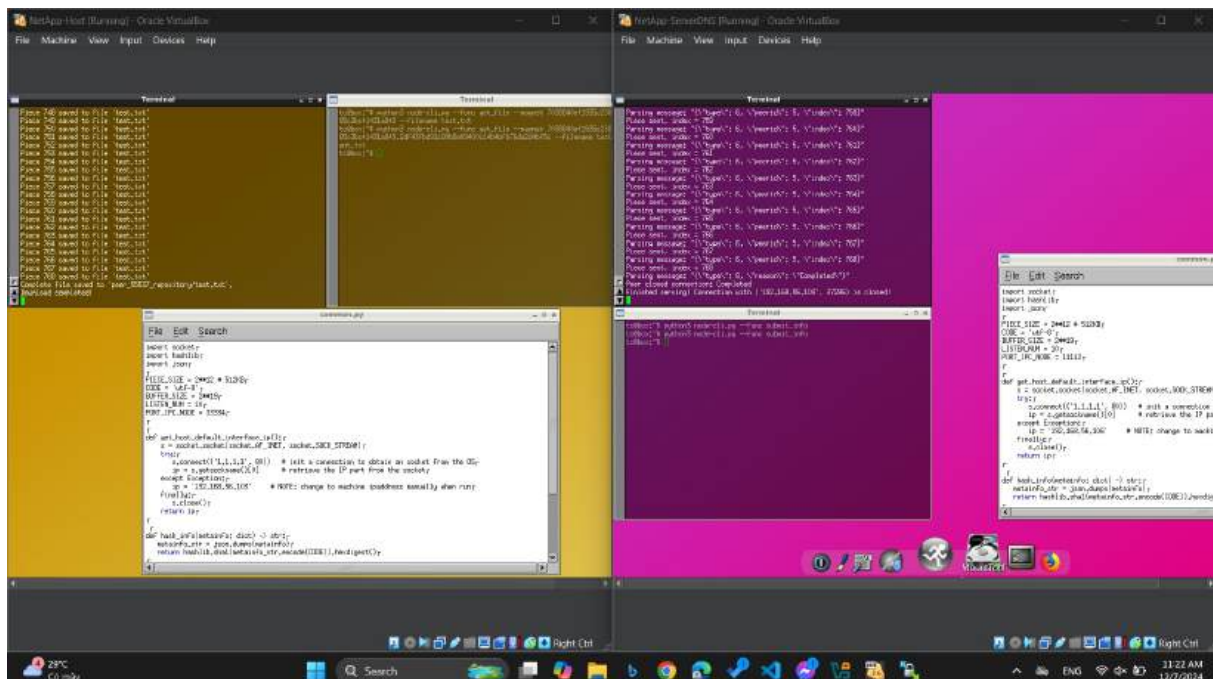


Figure 4.4: Download multiple file

Chapter 5

Define and describe the functions of the application

In this section, we try to provide summarized usage of each function in the application. Note that every communication-purposed function in this application is implemented via TCP protocol.

5.1 Server (Tracker)

- **__init__**: Initializes the tracker with its IP address, port, a unique ID, and an empty dictionary to track torrent information.
- **tracker_response**: Generates a dictionary response for a peer, including failure reasons, warnings, tracker ID, and a list of peers.
- **update_torrents_list**: Writes the list of tracked torrents and their details into a text file for reference.
- **parse_node_submit_info**: Parses and records information submitted by a peer about shared files, updating the torrent tracking dictionary.
- **return_peer_list_for_file**: Retrieves the list of peers holding a file based on its info hash. Returns None if the file is not found.
- **new_connection**: Handles a single peer connection by processing requests (e.g., submitting file info or fetching peer lists) and sending appropriate responses.
- **server_program**: Starts the tracker server, listens for peer connections, and handles each connection in a separate thread.

5.2 Node (Peer)

- **Downloader Class**:
 - **__init__**: Initializes the downloader with its peer ID, file info hash, peer list, repository path, filename, and synchronization mechanisms for multithreaded downloads.

-
- **start_download:** Starts the download process by creating threads to connect to peers, obtaining bitfields, selecting pieces, and assembling the complete file.
 - **thread_client:** Manages communication with a single peer: establishes a connection, retrieves bitfields, waits for peer selection, and downloads pieces.
 - **select_peer:** Determines which peer to request each file piece from, balancing load among peers and ensuring all pieces are assigned.
- **Node Class:**
 - **__init__:** Initializes the node, creates a local repository, loads files into memory, and starts threads for serving connections and handling commands.
 - **scan_repository:** Scans the local repository for files and returns a list of filenames.
 - **submit_info:** Submits the node's file information to the tracker for registration, returning the tracker's response.
 - **get_list:** Sends a magnet text to the tracker to request a list of peers holding the specified file.
 - **serve_incoming_connection:** Handles incoming connections from other peers to serve file piece requests or manage connections.
 - **thread_server:** Listens for incoming connections and starts a thread for each connection to serve peers.
 - **thread_download:** Creates a Downloader object to handle downloading a file and runs its start_download method.
 - **thread_agent:** Handles communication with the node's CLI or other external commands, processing requests such as submitting info or downloading files.

5.3 TCP protocol

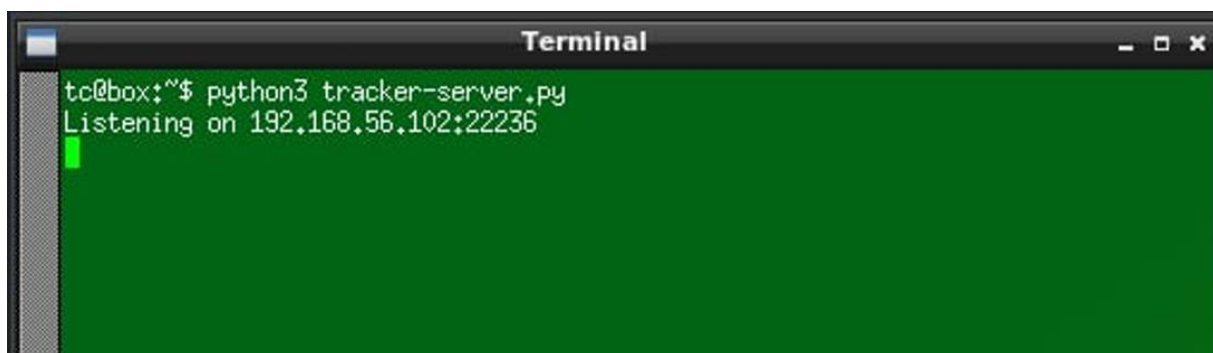
- **PeerConnection Class:**
 - **__init__:** Initializes a connection object with peer information (selfid, peerid, peer_addr) and file metadata (info_hash).
 - **close_connection:** Marks the connection as inactive.
- **PeerConnectionIn Class:**
 - **__init__:** Sets up an incoming peer connection by accepting a handshake and initializing the parent PeerConnection.
 - **accept_handshake:** Receives and validates a handshake message, sends a bitfield message in response, and returns the peer's ID and info_hash.
 - **piece:** Sends a specific piece of the file (based on the given index) to the connected peer.
 - **terminate_connection:** Ends the connection with a reason, sending an END message to the peer.
- **PeerConnectionOut Class:**

-
- **__init__**: Establishes an outgoing connection to a peer and performs a handshake.
 - **handshake**: Sends a handshake message to the peer and receives the bitfield message in response, indicating the pieces the peer has.
 - **request**: Sends a request message for a specific piece (based on index) and returns the received data.
 - **terminate.connection**: Ends the connection with a reason, sending an END message to the peer and closing the socket.
- **General Utility Functions:** MsgType Enum: Defines message types used for peer communication (e.g., HANDSHAKE, BITFIELD, REQUEST, PIECE, END). These functions and classes are part of a peer-to-peer protocol implementation, supporting communication and data exchange between peers.

Chapter 6

Validation (sanity test) and Actual Result

1. Start tracker



```
tc@box:~$ python3 tracker-server.py
Listening on 192.168.56.102:22236
```

Figure 6.1: Start tracker

2. Start peer

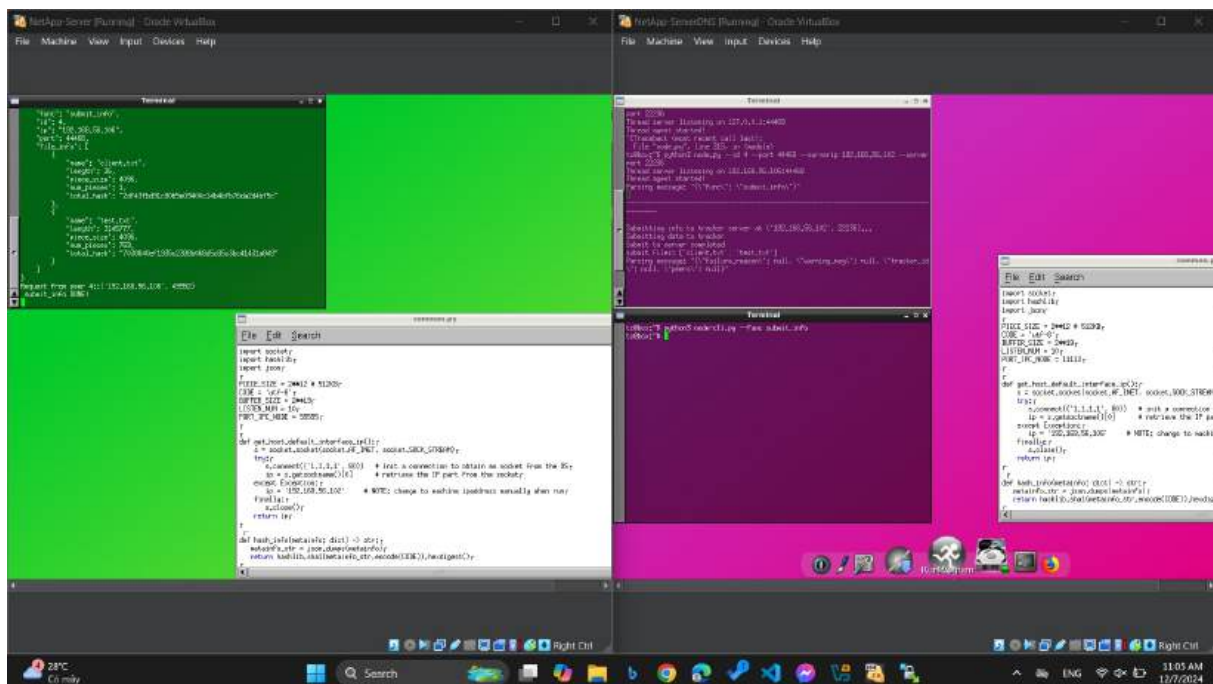


Figure 6.2: Start peer

The peer submits information about all the files it currently possesses to the server. In response, the server provides metadata for files available from other clients, including the hash code, file size, number of pieces, and file name for each file. The peer can then request to download specific files by specifying their hash code and file name.

3. **Peer request multiple file from multiple nodes.** Peer use the command:

```
python3 node-cli.py --func get_file --magnet_text
7038840ef1935c2388b068d5c85c3bc41431a849,
2df43fbd91c80b5e09400c14b4bfb76da2d4bf5c --filename
test.txt,client.txt
```

to request file from 2 peer that already have 2 these files. After that the result would be:

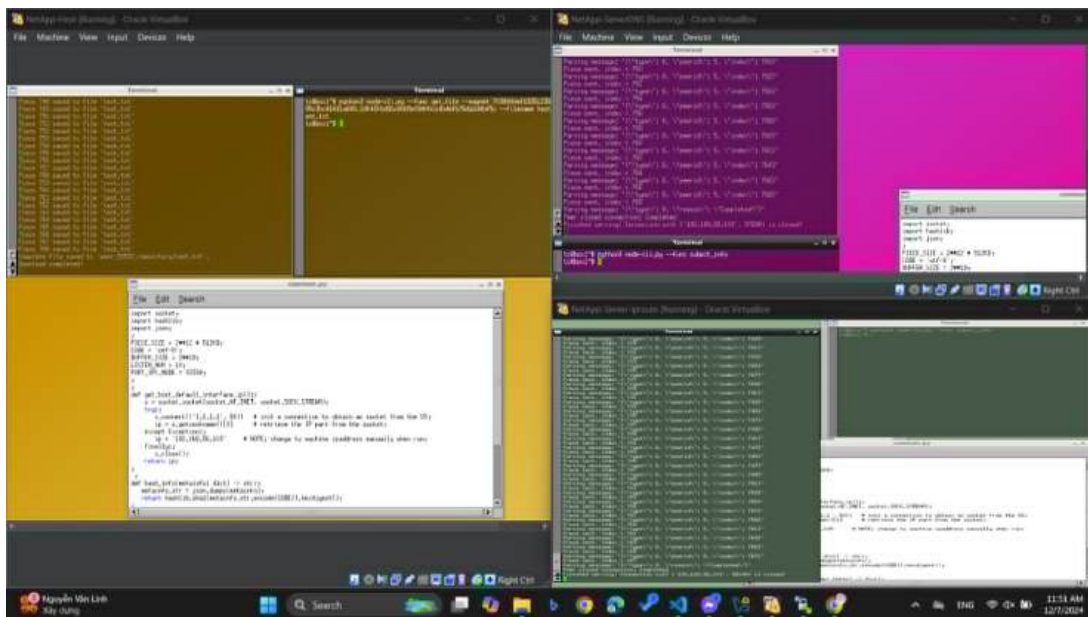


Figure 6.3: Peer request multiple file from multiple nodes

The peer successfully downloads the multi-downloading files from many peers

4. **Checking file**

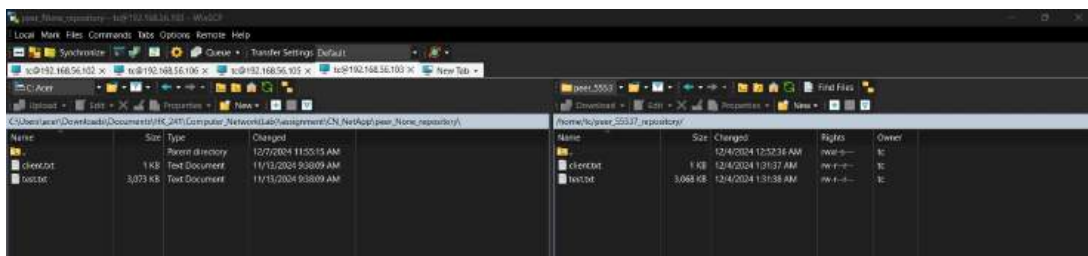


Figure 6.4: Checking receive file

We are use WinSCP app to check the size of 2 files with the origin. As the result, there is no missing piece that is not downloaded.

Chapter 7

Future development

- **UI Design:** Develop an intuitive and user-friendly interface to improve the overall user experience while ensuring there is room to accommodate future functionalities and enhancements.
- **Security Enhancements:** The current application lacks a mechanism to verify whether a peer is authorized to share or download files from another peer, which introduces potential security vulnerabilities. To address this, we plan to implement a dedicated authentication system in future updates.
- **Algorithm Optimization:** The current algorithm for requesting file pieces from other peers is inefficient and could be further optimized. Enhancing this algorithm will significantly improve the system's performance and resource utilization.