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

Computer Networks (CO3093)

ASSIGNMENT 1

DEVELOP A NETWORK APPLICATION

Advisor: Nguyễn Lê Duy Lai
Students: Phạm Đức Hoàng - 2211111
Đỗ Huy Minh Dũng - 2210568

Ho Chi Minh city, April 2025

Contents

1 Application Overview

The system is built around a hybrid model that combines a centralized tracker server with decentralized peer-to-peer (P2P) communication between clients.

At the center of the architecture is the tracker, a centralized server responsible for managing metadata and peer coordination. It maintains an up-to-date registry of all active peers in the network, including which peers are currently seeding specific files. The tracker also stores and serves .torrent files (metadata files) that describe the contents of shared files and how they are divided into pieces.

When a peer wants to seed a file, it has the option to publish the corresponding .torrent file. If the peer chooses to make it public, the .torrent file is uploaded to the tracker. Other clients can then retrieve a list of all publicly available torrents and download the one they need. The downloaded .torrent file contains the metadata required to join the appropriate swarm and begin file transfers.

According to the tracker protocol, a peer announces to the tracker which content it is making available. This announcement includes metadata but not the actual file content. When a client wants to download a file, it contacts the tracker to obtain a list of peers currently seeding that file.

Once the peer list is received, the client initiates multiple direct connections to these peers using the P2P protocol. File pieces are then downloaded in parallel from different peers in the swarm. In our implementation, each peer is capable of simultaneously uploading and downloading multiple files across multiple connections, enabling efficient and scalable file distribution.

2 Component Functional

2.1 Tracker

The main roles of the tracker in our system include:

- Peer Management: The tracker keeps a current list of all peers actively seeding each shared file. These lists are continuously updated as new "join" announcements are received, indicating that a peer has begun seeding a particular file.

- Peer Discovery: When a client wants to download a file, it sends a request to the tracker. The tracker maintains a list of peers currently seeding that file

and responds with the IP addresses and port numbers of available peers from whom the file can be downloaded.

- Metadata File Hosting: The tracker stores metadata files (specifically, .torrent files in our application) for files that have been made publicly available. These metadata files are accessible to any peer that requests them.

2.2 Peer(s)

In our system, peers are equipped with the following capabilities:

- Torrent File Generation: Peers can generate .torrent files for individual files or folders. These metadata files contain the necessary information—such as file size, piece hashes, and tracker address—to enable sharing with other peers in the network.

- Seeding (Uploading) Content: Once a peer has the complete file, it can act as a seed, distributing file pieces to other peers requesting them. Seeding plays a critical role in maintaining the file's availability within the network.

- Leeching (Downloading) Content: Peers can initiate downloads by connecting to other peers based on the active peer list provided by the tracker. During this process, the peer requests and receives pieces of the desired file.

- Piece Management: Files are divided into smaller blocks, or pieces, for transmission. Peers maintain internal records of each piece's status—whether it is downloaded, in progress, or pending—and coordinate piece requests to minimize duplication and ensure complete file assembly.

- Piece Verification: After receiving a piece, the peer verifies its integrity by comparing its hash against the expected value from the .torrent file. This validation step prevents corrupted or tampered data from being accepted.

- Concurrent Connections: Peers can connect to multiple other peers at the same time. This allows for parallel downloading of different file pieces from various sources, while also uploading pieces to multiple peers simultaneously. This concurrency significantly boosts overall transfer speed and efficiency.

- Post-Download Seeding: Once a download is complete, a peer may transition to seeding mode. By sharing pieces of the file with others, the peer supports file availability and helps sustain the swarm.

- Status Monitoring: Peers continuously monitor and update the status of their downloads and uploads, enabling accurate tracking of transfer progress and connection performance.

3 System Specification

3.1 Data pieces

In the BitTorrent protocol, files are split into smaller, equally sized segments to improve the efficiency of both downloading and uploading among multiple peers. Each segment, or piece, represents a fixed portion of the original file, although the actual size of these pieces may vary based on the torrent's configuration. The meta-info file defines these pieces.

Each piece is uniquely identified by a 20-byte SHA1 hash, and all such hashes are concatenated into a single string stored in the pieces field of the meta-info file. When a peer receives a piece, it verifies the data by comparing its hash to the expected value. This process ensures the piece is valid and has not been altered.

The piece length field in the meta-info file specifies the size of each piece in bytes, providing the necessary structure for consistent data verification and transfer.

3.2 Metadata (.torrent) File

The metadata file, commonly known as a .torrent file, encompasses crucial information about the files being shared and provides instructions for identifying peers within the torrent network.

Below is a breakdown of the structure and role of the metadata file. For the purposes of this assignment, our system employs a simplified version of the standard structure, by including only the most relevant elements. The file is formatted to a bencoded dictionary format, which includes the following key fields:

- announce: A string specifying the tracker's announce URL.

- announce-list: A list of alternative tracker URLs (represented as a list of strings) used as fallback options.

- creation date: A timestamp indicating when the torrent file was generated.
- created by: The name or identifier of the peer that created the torrent file.
- comment: A textual note or remark associated with the torrent file.
- info: A dictionary containing details about the shared file or directory. This field differs based on whether the torrent represents a single file or multiple files in a folder:
  - piece length: The size of each piece in bytes (typically set to 256KB).
  - name: The name of the file or folder being shared.
  - length (single-file mode only): The total size of the file in bytes.
  - files (multi-file mode only): A list of dictionaries, each representing an individual file. Each dictionary includes:
    - length: File size in bytes.
    - path: A list describing the relative path and filename within the directory.
  - pieces: A byte string made by concatenating the 20-byte SHA-1 hash of each piece. This enables the client to verify the integrity of downloaded pieces by comparing their hashes.

3.3 Bencode

3.3.1 What is Bencode?

Bencode is the encoding format used by the BitTorrent protocol to represent and transmit loosely structured data. It supports four basic data types: byte strings, integers, lists, and dictionaries (also known as associative arrays).

Bencoding is primarily used in .torrent files, and is an integral part of the BitTorrent specification. These metadata files are simply dictionaries encoded using the Bencode format.

One of Bencode's advantages is its simplicity. Since numeric values are stored as plain text in decimal form, the format is independent of system endianness—an essential feature for cross-platform compatibility.

Bencode is also relatively extensible. As long as clients ignore unrecognized dictionary keys, new fields can be added without breaking compatibility with existing applications.

3.3.2 Bencoding Algorithm

Bencode represents data using a minimal and compact format based on ASCII characters and digits. It defines specific rules for encoding data types

Integers are encoded using the format: i<integer in base 10>e:

- The integer must be in base 10 and may be negative, indicated by a leading minus sign (-).
- Leading zeros are not permitted unless the value is zero.
- Examples: 0 ⟶ "i0e", 24 ⟶ "i24e", -71 ⟶ "i-71e".

Byte strings are encoded as <length>:<content>:

- The length (in bytes) is specified in base 10 and is followed by a colon ':'.
- The content is a raw byte sequence exactly matching the declared length.
- Examples: "" ⟶ "0:", "Hello World " ⟶ "12:Hello World ", "mddepzai" ⟶ "8:mddepzai".

Lists begin with l and end with e:

- All elements are Bencoded values placed sequentially without separators.
- Examples:
  - [ ] ⟶ "le"
  - [4, -1, 9] ⟶ "li4ei-1ei9ee"
  - [["system", "verilog"], ["veryhard"]] ⟶ "ll6:system7:verilogel8:veryhardee"
  - ["mmtkhoqua", -972, "chuaduwow"] ⟶ "l9:mmtkhoquai-972e9:chuaduwowe"

Dictionaries start with d and end with e:

- Each entry is a key-value pair where keys are byte strings.
- Keys are byte strings and must be in lexicographical order.
- Values can be of any Bencoded type.
- Examples:

– { } ⟶ "de"

– {
"y3s2" : "18tinchi",
"IoT" : 3,
"tkvm" : [4, "gk", "lab", "btl", "ck"],
"mmt" : {
"lab" : 10,
"thi" : "60%"
}
}

⟶ "d4:y3s28:18tinchi3:IoTi3e4:tkvmli4e2:gk3:lab3:btl2:cked3:labi10e3:thi3:60%ee"

4 Protocols

4.1 Tracker Protocol

The tracker functions as an HTTP-based service that processes client requests made via the GET and POST methods. Clients use query parameters to inform the tracker of their status in the swarm and specify the content they want to share or download.

To manage these interactions, the tracker exposes a set of HTTP endpoints, each serving a specific purpose—such as announcing a peer's presence or retrieving metadata. The following endpoints are applied to maintain swarm activity and support peer discovery:

Endpoint "/":

- The default endpoint when running a tracker server.
- Returns the current status of the server, helping users quickly verify whether the server is online.
- URL Example: http://tracker0.vn.com/

Endpoint "/torrents":

- Returns a list of all torrents stored on the tracker. Each torrent is represented by its info hash, name, and a short description.

- Allows clients to view available torrents without exposing internal file paths, which are intentionally hidden for security and privacy.
- URL Example: http://trackerjp11.com/torrents
- Response Example: {
  "3f786850e387550fdab836ed7e6dc881de23001b": {
  "name": "Computer_Organization_And_Design_RISC-V_Edition.pdf.torrent",
  "description": "Computer Architecture for RISC-V"
  },
  "7a1a1f5de77e46b7a02b26ce139e4fd7ec5807ab": {
  "name": "DazaiOsamu.mp4.torrent",
  "description": "Van hao luu lac"
  }
  }

Endpoint "/torrents/{info_hash}":

- When a user knows the info hash of a torrent, they can use this endpoint to retrieve the corresponding .torrent file from the tracker.
- If the tracker has the file, it will return it to the client. If not, it responds with a **Bad Request** error.
- URL Example: http://tracker.en.com/torrents/t3ra1lal3ero7t6ra0lal4at4r6ipp-4itr6o1ppi. This URL is used to request the torrent file for the info hash "t3ra1lal3ero7t6ra0lal4at4r6ipp4itr6o1ppi".

Endpoint: "/announce"

- **1. Get Request:** The peer announces its status and requests a peer list by appending parameters to the URL. Key parameters include:
  - info_hash: A 20-byte SHA1 hash that uniquely identifies the torrent.
  - event: The client's current action (e.g., started for joining the swarm, stopped for leaving it).
  - port: Port number that the peer is listening on the upcoming connect.

    Example URL: http://trackerA.com/announce?info_hash=A0F866E4871-C59D1A5224F1857897CCB788870BE&port=6419&event=started.

This URL is used to request the peer list for the info hash "A0F866E4871-C59D1A5224F1857897CCB788870BE", the listening peer port is 6419 and the event is "started".

- **2. Response Format:** The tracker replies with a JSON object containing a list of peers. Each peer is represented as a dictionary with an IP address and port number.

  Response Example:"peers": [{"ip": "192.168.1.81", "port": 6708},
  {"ip": "172.28.182.108", "port": 4239}]

- **3. Post Request:** In addition to GET, the tracker also supports POST requests for uploading metadata files. This allows a client to share a new torrent by sending:
  - info_hash: A 20-byte SHA1 hash identifies the torrent.
  - event: The client's current action.
  - file: The actual .torrent file content.

4.2 Peer Protocol

The peer-to-peer protocol plays a key role in the file-sharing system by allowing clients to communicate and share data directly, without the need for a central server.

4.2.1 HTTP

To handle peer interactions, the peer server provides a set of HTTP endpoints, each designed for a specific function—such as seeding files or leeching them from other peers.
The following endpoints are applied to support the operations:

- **Endpoint "/":** Checks whether the peer server is running.
  Response Example: { "status": "OK" }
- **Endpoint "/status":** Returns the list of torrents the peer is currently involved with, including:
  - Seeding: Files or folders being uploaded to other peers.
  - Leeching: Files or folders being downloaded from other peers.

Response: JSON object containing lists of info hash values for both seeding and leeching torrents.

Response Example: {

```
"seeding": [
    {
      "info_hash": "8a9f57d92b3d1b2cddf9a1c16f1799ac11223344":
      "name": "Chapter_8_v8.0.pdf"
    },
    {
      "info_hash": "4f8e1a2bd8f8b6548b2f58a763e9fda5bb998877":
      "name": "smile.jpg"
    }
 ]
"leeching": [
    {,
      "info_hash": "1c7c44cdb4a71eb9be43b9184365012da09292bf":
      "name": "Chapter_7_v8.0.pdf"
    },
    {
      "info_hash": "3e2ee3c610d21369eb371aec92def804079fda4c":
      "name": "Chapter_5_v8.0.pdf"
    }
 ]
}
```

- **Endpoint "/torrents":**
  - Retrieves all torrents managed by the peer.
  - Response: List of torrent metadata (excluding file paths, which are hidden for privacy and security reasons).
  - Response Example: {

```
"6a9d88e0c13e927f3f1e1e5a4fa3c4eacbda9b21": {
    "name": "IoT_Lab3_OTA_Firmware_Update.pdf.torrent",
     "description": None
},
 "f1a3e926b4f102d2b7c6cf6dfd79807dd2e88713": {
```

"name": "Do_An_Da_Nganh.torrent",
"description": "Do an da nganh"
}
}

- **Endpoint "/leech":** Requests the peer to start downloading a file based on a given .torrent file.
- **Endpoint "/torrents/<info_hash>":** Fetches the .torrent file corresponding to the specified info hash.

4.2.2 TCP Message

Peer connections are symmetrical. Messages sent in both directions look the same, and data can flow in either direction.
The following message types are designed for the BitTorrent communication:

- Choke: Temporarily stop sending data
- Unchoke: Allow data to be sent
- Interested: Peer is interested in receiving data
- NotInterested: Peer is not interested in receiving data
- Have: Notify that a data piece has been received
- BitField: Binary map of available data pieces
- Request: Request a specific piece of data
- Piece: Send a piece of the requested data
- Cancel: Cancel a previous data request

To simplify the peer communication, we will assume that all peers are unchoked by default. Instead of waiting to receive piece availability information via BitField or exchanging Interested/NotInterested messages, peers will directly send Request messages for the pieces they need.

Peers will also skip sending Have messages after completing a piece. Instead, they will immediately proceed to request the next needed piece. Additionally, if a

peer is in the process of downloading a piece it already has or no longer needs, it will simply stop downloading that piece without sending a Cancel message.

As a result, the communication logic will focus primarily on the Request and Piece messages, along with the Handshake message (Handshake is not really part of the messages). The handshake is required to authenticate peers and verify that the receiving peer holds the info hash corresponding to the desired torrent:

The following is a breakdown of the messages used in peer communication:

- Handshake: Establishes the initial connection between two peers and verifies that both are participating in the same torrent.
  Fields:
  - info_hash (20 bytes): SHA-1 hash of the info section from the .torrent file. This uniquely identifies the torrent being requested or served.
- Request: Request a specific portion of data (a piece).
  Fields:
  - index (4 bytes): The zero-based index of the piece.
  - begin (4 bytes): The byte offset within the piece where the requested block begins.
  - length (4 bytes): The length of the piece in bytes (default is 256KB).
- Piece: Delivers a block of data in response to a Request message.
  Fields:
  - index (4 bytes): The zero-based index of the piece.
  - begin (4 bytes): The byte offset within the piece.
  - block (4 bytes): The actual block of raw data bytes being transferred.

5 System Design

5.1 Class Diagram

The class diagram presents the primary components of the system and the relationships between them:

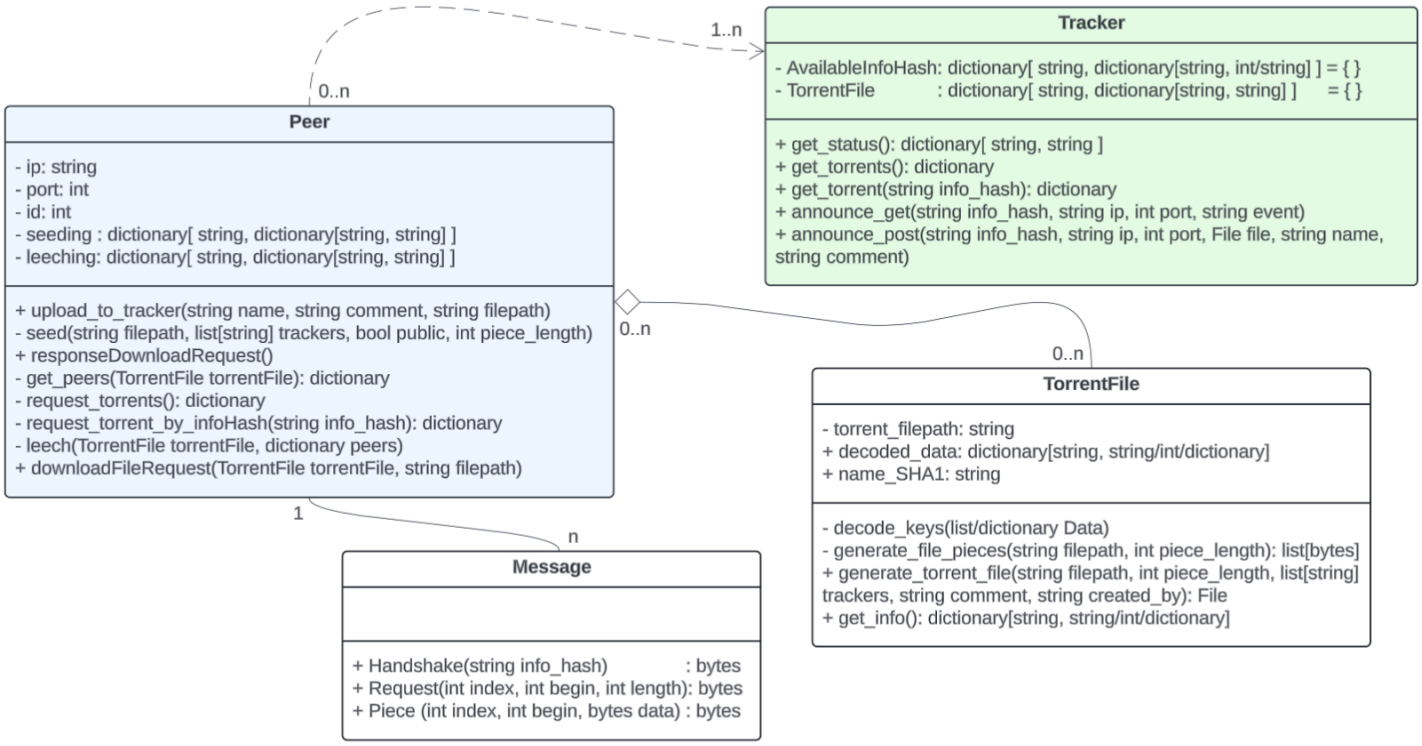
- **Tracker Class**
  - Manages peer information.

  - Stores and serves metadata (.torrent) files.
  - Maintains a list of active peers for each torrent.
  - Responds to peer requests for peer lists and meta-info files.

- **Peer Class**
  - Handles both uploading (seeding) and downloading (leeching) of files.
  - Communicates with the Tracker to:
    * Announce its seeding status.
    * Retrieve peer lists and the .torrent file required for downloading.

- **TorrentFile Class**
  - Generates meta-info files (.torrent) from local files or directories.
  - Provide information from existing .torrent files to extract metadata.

- **Message Class**
  - Generates and processes protocol messages used for peer-to-peer communication.



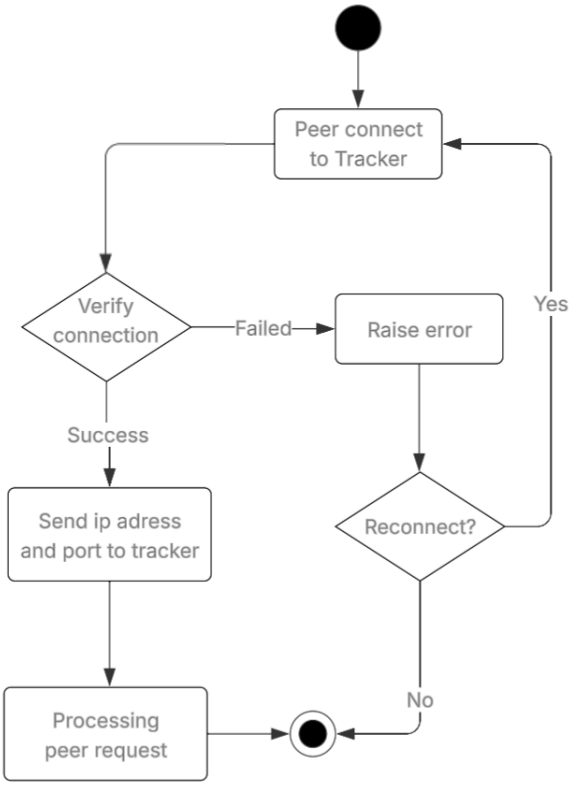
5.2 Activity Diagram

5.2.1 Peer Connection to Tracker

This activity diagram illustrates the process a peer follows when attempting to connect to the tracker server:

- Start: The process begins when a peer attempts to connect to the tracker.
- Peer Connects to Tracker: The peer initiates a connection request to the tracker server.
- Verify Connection: The system checks whether the connection is successfully established.
  If the connection fails:
  - An error is raised.
  - The system prompts whether the peer wants to reconnect.
    * If Yes, the process loops back to the "Peer connects to Tracker" step.
    * If No, the process terminates.

  If the connection is successful: The peer sends its IP address and port number to the tracker.
- Processing Peer Request: After the peer successfully provides its network information, the tracker proceeds to process the peer's request.
- End: The process concludes.



5.2.2 Torrent File Creation and Upload to Tracker

This activity diagram describes the workflow for generating a .torrent file from a given file or folder and uploading it to the tracker server:

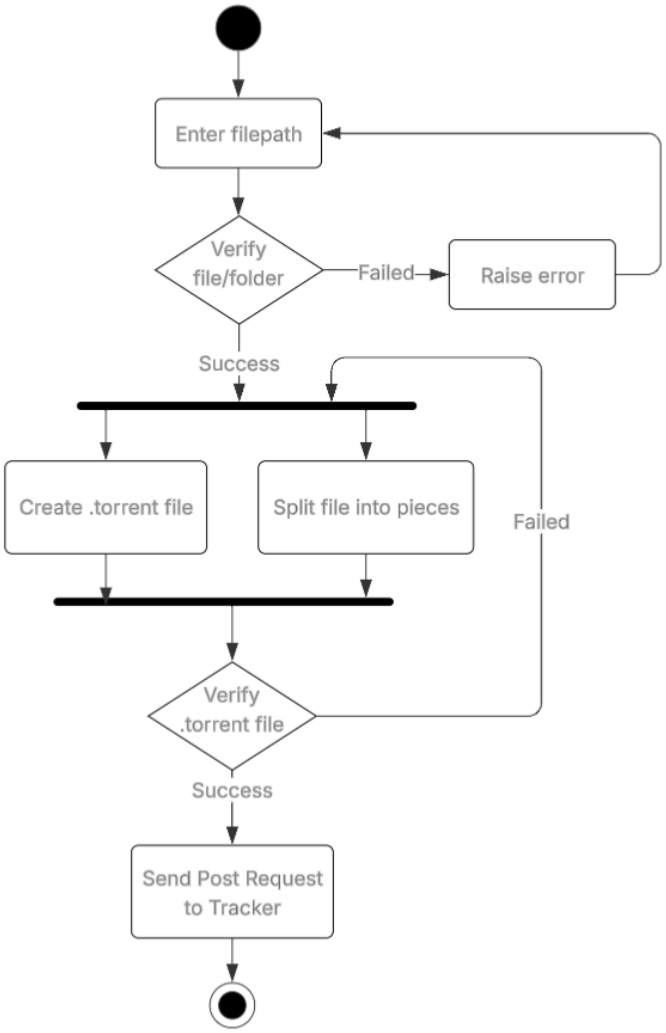
- Start: The process begins when the user initiates the torrent creation operation.
- Enter file path: The user provides the path to the target file or folder.
- Verify File/Folder: The system checks whether the specified file or folder exists and is accessible:
  - If the verification fails, an error is raised and the user is prompted to re-enter the filepath.
  - If verification succeeds, the process continues.
- Create .torrent File and Split File into Pieces (Parallel Actions). These two actions occur in parallel:
  - The system creates a .torrent metadata file.
  - The original file is divided into fixed-size pieces for transfer.

  - If splitting into pieces fails, the system raises an error and loops back to the initial input step.

- Verify .torrent File: The generated .torrent file is verified to ensure it meets required specifications and contains valid data.
  - If verification fails, the process terminates or prompts for correction.
  - If verification succeeds, the system proceeds.

- Send POST Request to Tracker: A POST request is sent to the tracker server, containing the .torrent file and necessary metadata.

- End: The process concludes successfully.

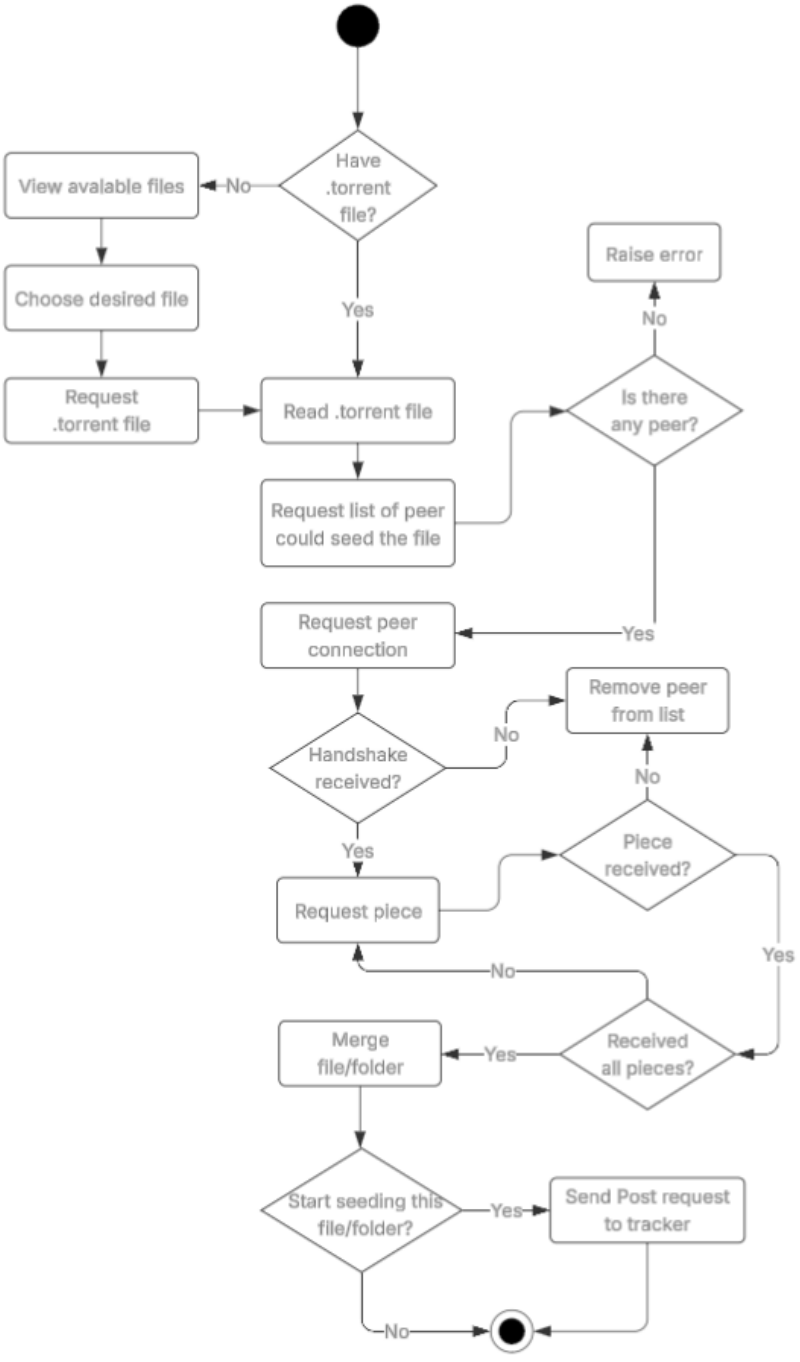


5.2.3 Downloading Workflow

This diagram illustrates the process of downloading a file using a .torrent file and transitioning to seeding once the download is complete.

- Start: The process begins when the user initiates a download operation.
- Check for .torrent File: The system checks whether the user already has a .torrent file for the desired content.
  - If not, the user is shown a list of available files, selects a desired file, and requests the associated .torrent file.
  - If the file is available, proceed to read it.
- Read .torrent File: The client reads and parses the .torrent file to retrieve metadata and info hash.
- Request Peer List: The client requests a list of peers from the tracker that can seed the specified file.
- Check for Available Peers:
  - If no peers are available, an error is raised and the process stops.
  - If peers are found, the client attempts to establish a connection.
- Request Peer Connection: The client sends a connection request to one of the peers.
- Perform Handshake: The client performs a handshake to authenticate and verify that the peer has the correct file.
  - If the handshake fails, the peer is removed from the list and another connection attempt is made.
  - If the handshake succeeds, proceed.
- Request File Piece: The client requests a specific piece of the file from the connected peer.
- Check if Piece Received:
  - If the piece is not received, the system retries or tries another peer.
  - If received, check whether all pieces have been downloaded.
- Verify Completion: If all pieces have been received, merge them into the original file or folder.

- Merge File/Folder: The system reconstructs the original file/folder from the downloaded pieces.
- Seeding Decision: The client checks if it should begin seeding the file to other peers.
  - If yes, a POST request is sent to the tracker to announce availability as a seeder.
  - If no, the process ends.
- End: The process completes successfully.



6 Implementation

6.1 Source code

- Link: https://github.com/dohuyminhdung/ComputerNetwork_Assignment1
- Peer Messages References: https://github.com/eliasson/pieces

6.2 Programming Language and Library

The system is primarily developed using the Python programming language due to its simplicity, flexibility, and strong support for asynchronous programming and networking.

Tracker Component

The tracker operates as an HTTP server, implemented with the FastAPI framework—a modern and high-performance Python web framework built on top of Starlette and Pydantic. By utilizing Python's type hinting system, FastAPI offers automatic data validation and serialization, significantly simplifying request handling logic. The tracker is responsible for managing peer metadata, responding to peer announcements, and providing peer lists associated with specific torrents.

Peer Communication and Concurrency

Each peer in the system functions both as a client and a lightweight server, capable of handling simultaneous upload and download operations. To support this dual role efficiently, the system adopts a fully asynchronous design based on Python's asyncio framework and the Quart web framework.

The peer application is designed to be non-blocking by default, enabling multiple I/O-bound tasks—such as network communication and file operations—to be executed concurrently. This is achieved through:

- **asyncio**, which provides an event loop, task scheduling, and coroutine management. It allows the peer to process incoming requests, initiate connections to other peers, and manage file piece transfers concurrently within a single-threaded context.
- **aiofiles**, used for asynchronous file system access, ensures that reading and writing large data chunks (file pieces) from disk does not block the event loop,

thus maintaining application responsiveness even under heavy load.

To handle incoming connections from other peers, each peer runs a built-in asynchronous web server using Quart—a Flask-compatible, asyncio-based framework. Quart enables the peer to:

- Expose endpoints for receiving requests for file pieces.
- Handle incoming handshakes to validate connections and info hashes.
- Respond to metadata requests or piece transfers via HTTP routes.

By integrating Quart, each peer operates as a lightweight server capable of serving multiple requests concurrently, leveraging the same asyncio event loop that manages its client-side tasks.

Supporting Utilities: Hashing, Encoding, and Logging

To ensure file integrity, interoperability with the BitTorrent protocol, and maintainability of the system during development and execution, we integrate several essential libraries that support core functionalities behind the scenes.

- Hashing for Piece Identification and Verification: File pieces are uniquely identified and verified using SHA-1 hashes. The hashlib module in Python is used to compute SHA-1 digests for each piece of the file being shared or downloaded.
  These hashes are:
  - Embedded into the .torrent file during its creation.
  - Used by peers to verify the integrity of each received piece by comparing the hash of the downloaded block against the expected SHA-1 value.
- Torrent File Encoding with bencodepy: To construct and parse .torrent files according to the BitTorrent specification, we use the bencodepy library. Bencode is a lightweight encoding format used to serialize the torrent metadata. The use of bencodepy ensures that our .torrent files remain compatible with other BitTorrent clients and adhere strictly to protocol standards. During download, peers parse the .torrent content to extract relevant information for identifying files and locating other peers.

- To improve observability and simplify debugging, we employ Python's built-in logging module. Logging is integrated throughout the system to:
  - Record key events such as peer connections, piece requests, uploads/downloads, and tracker interactions.
  - Log warnings and errors, including failed connections, invalid responses, or file verification failures.
  - Support optional log level configuration (e.g., DEBUG, INFO, ERROR), allowing developers to control verbosity during testing or production.

6.3 Command-Line Interface

To simulate and control the execution of the system, we have implemented a comprehensive set of command-line interface (CLI) commands using the **click** library in Python. This approach provides a structured and user-friendly interface for interacting with various components of the BitTorrent system, including connectivity checking, torrent file creation, seeding, leeching, and system status monitoring.

Each command encapsulates a specific task within the system and is equipped with meaningful options to offer flexibility and ease of use. Below is a description of the available CLI commands and their functionalities:

- **hello** – Connectivity Check
  This command is used to test whether a connection can be established with a given host and port. It is helpful for verifying whether peer servers are reachable.
  Options:
  - host <STRING>: Host address to check connectivity. (Default: 127.0.0.1)
  - port <INTEGER>: Port number to check connectivity [Required].

- **create** – Torrent File Creation
  This command generates a .torrent file from a specified input path (file or directory). It supports multiple tracker URLs and allows customization of metadata. Options:
  - input <STRING>: Path to the file or directory to include in the torrent [Required].

  - tracker <STRING>: One or more tracker URLs. Can be specified multiple times.
  - output <STRING>: Path to save the generated .torrent file.
  - piecelen <INTEGER>: Piece length in bytes (must be a power of 2). (Default: 262144 bytes)
  - cmt <STRING>: Optional comment embedded in the torrent metadata.
  - cre <STRING>: Creator name or information about the author of the content.

- **seed** – Start Seeding a File
  This command initializes a peer server to begin seeding the specified file(s) to the network. It can either create a new torrent or use an existing .torrent file. Options:
  - port <INTEGER>: Port to run the peer server [Required].
  - input <STRING>: Path to the file(s) you wish to seed [Required].
  - tracker <STRING>: Tracker URL(s). Can be specified multiple times.
  - torrent <STRING>: Optional path to an existing .torrent file.
  - piecelen <INTEGER>: Piece length in bytes (power of 2). (Default: 262144 bytes)
  - cmt <STRING>: Optional comment for the torrent metadata.
  - cre <STRING>: Creator name or information about the author of the content.
  - name <STRING>: Custom name for the .torrent file.
  - private: Private flag, use this option with no parameter to indicate that the torrent should not be publicly discoverable.

- **show-info** – Torrent File Inspection
  This command displays the content and metadata of a specified .torrent file, including piece hashes, length, tracker info, and other embedded data. Options:
  - torrent <STRING>: Path to the .torrent file to inspect [Required].

- **get-torrent** – Retrieve a Torrent File
  This command allows a peer to fetch a .torrent file from another peer by connecting to its server via a specified port. Options:

  - port <INTEGER>: Port of the peer server to connect to [Required].

- **leech** – Download a File from Peers
  This command launches a peer client that connects to the network and downloads the file(s) described in the provided .torrent file.
  - port <INTEGER>: Port of the peer server to connect to [Required].
  - torrent <STRING>: Path to the .torrent file used for downloading [Required].

- **leech** – Download a File from Peers
  This command launches a peer client that connects to the network and downloads the file(s) described in the provided .torrent file.
  - port <INTEGER>: Port of the peer server to connect to [Required].
  - torrent <STRING>: Path to the .torrent file used for downloading [Required].

- **status** – Peer Server Status Check
  This command displays the current state of a peer, including active uploads/-downloads and connected peers.
  - port <INTEGER>: Port of the peer server to query [Required].

References

[1] *Computer Networking: A Top-Down Approach* (8th Edition). Jim Kurose, Keith Ross Pearson (2020).

[2] BitTorrent Specification: https://wiki.theory.org/BitTorrentSpecification

[3] The lecture slides by Nguyễn Lê Duy Lai.