Báo cáo

1. Mở đầu
2. Tổng quan

Trong thời đại phát triển mạnh mẽ của công nghệ thông tin hiện nay, việc trao đổi dữ liệu giữa các máy tính đã trở nên vô cùng cần thiết. Có nhiều phương pháp đã được nghiên cứu và áp dụng để đáp ứng nhu cầu này, trong đó, nổi bật là phương pháp truyền tệp ngang hàng (peer-to-peer). Phương pháp này cho phép dữ liệu được truyền trực tiếp giữa hai máy tính mà không cần phải qua máy chủ trung gian.

Truyền tệp ngang hàng (P2P - Peer-to-Peer) là một mô hình mạng máy tính, trong đó các thiết bị tham gia hoạt động đồng thời như cả máy khách lẫn máy chủ, cho phép chia sẻ dữ liệu trực tiếp giữa các thiết bị mà không cần đến một máy chủ trung tâm. Khái niệm P2P xuất hiện vào cuối những năm 1990 và nhanh chóng trở nên phổ biến nhờ vào các ứng dụng chia sẻ tệp tin như Napster, Kazaa, và sau này là BitTorrent.

Với P2P, dữ liệu được chia nhỏ thành các phần và phân phối giữa các thiết bị trong mạng. Điều này giúp tăng tốc độ truyền tải và giảm tải cho một máy chủ duy nhất, đồng thời tăng khả năng chịu lỗi của mạng vì dữ liệu có thể được tải xuống từ nhiều nguồn. P2P không chỉ được ứng dụng trong chia sẻ tệp tin mà còn được sử dụng trong các lĩnh vực như blockchain, truyền phát video, và cả các hệ thống truyền thông phân tán.

Tuy nhiên, vì tính chất phân tán và không có trung gian, P2P cũng gặp một số thách thức về bảo mật và quản lý nội dung. Mặc dù vậy, với sự phát triển liên tục, mô hình P2P đã và đang khẳng định vai trò quan trọng trong việc truyền tải dữ liệu và xây dựng các hệ thống phân tán hiện đại.

1. Mục tiêu

Ứng dụng cần đạt được các mục tiêu sau:

* Xây dựng một ứng dụng truyền tệp ngang hàng đơn giản, dễ sử dụng.
* Hỗ trợ truyền tệp giữa các máy tính một cách hiệu quả.
* Cho phép tải xuống đồng thời nhiều tệp torrent từ nhiều node khác nhau.
* Cho phép tải lên đồng thời nhiều tệp torrent đến nhiều node.
* Cung cấp số liệu thống kê về lượt tải xuống/tải lên, hỗ trợ kiểm tra thông tin chi tiết về các node và tệp torrent đang hoạt động.

Để đạt được các mục tiêu trên, nhóm đã thực hiện các công việc sau:

* 1. Đối với Client
* Hiện thực đăng kí, đăng nhập và đăng xuất cho phép lưu thông tin sử dụng của người dùng.
* Cho phép người dùng download file từ nhiều peer khác nhau và publish thông tin file từ máy local lên server.
* Xây dựng giao diện dòng lệnh (CLI) giúp dễ dàng theo dõi hoạt động giữa các client và hỗ trợ truyền tải tệp.
  1. Đối với tracker
* Phát triển hệ thống cơ sở dữ liệu để lưu trữ thông tin về các client, peer và danh sách tệp của từng client.
* Có khả năng xử lí và phản hồi các yêu cầu fetch, publish của người dùng
  1. Đối với yêu cầu chung của hệ thống
* Hiện thực và kiểm thử các chức năng truyền tải dữ liệu giữa các client nhằm đảm bảo độ ổn định và hiệu quả của ứng dụng.
* Hiện thực thống có chức năng tương tự giao thức Torrent

1. Công nghệ sử dụng

* Python: Client, Client\_CLI, Tracker\_CLI.
* Typescript ứng dụng NestJS: Tracker\_server.
* MySQL cho database và Sequelize cho công cụ ORM.

1. Nội dung
   * 1. Cơ sở lí thuyết
   1. Khái niệm về giao thức

Giao thức được định nghĩa là định dạng, thứ tự của các thông điệp được gửi và nhận giữa các thực thể mạng, cũng như các hành động khi thực hiện truyền và nhận thông điệp.

(hình của giao thức)

* 1. Giao thức TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) là một bộ giao thức mạng thiết yếu, đóng vai trò là nền tảng chính cho việc kết nối và truyền tải dữ liệu trong môi trường Internet. Ra đời vào những năm 1970, TCP/IP đã trở thành tiêu chuẩn toàn cầu, cho phép các thiết bị và hệ thống khác nhau giao tiếp với nhau bất kể nền tảng hoặc kiến trúc của chúng.

Bộ giao thức này bao gồm nhiều thành phần, nhưng hai phần quan trọng nhất là TCP (Transmission Control Protocol) và IP (Internet Protocol). IP chịu trách nhiệm về việc định địa chỉ và định tuyến các gói dữ liệu, đảm bảo rằng chúng được gửi đến đúng đích trong mạng. Mỗi thiết bị trên mạng được xác định bằng một địa chỉ IP duy nhất, giúp cho việc quản lý và tổ chức các kết nối trở nên hiệu quả.

Trong khi đó, TCP đảm bảo tính toàn vẹn và chính xác của dữ liệu trong quá trình truyền tải. Nó thực hiện điều này thông qua việc chia nhỏ dữ liệu thành các gói tin, đánh số và đảm bảo rằng chúng được gửi và nhận theo đúng thứ tự. TCP còn có khả năng kiểm tra lỗi và yêu cầu truyền lại các gói tin bị mất, giúp đảm bảo rằng người nhận nhận được dữ liệu chính xác và đầy đủ.

Sự kết hợp giữa TCP và IP tạo ra một hệ thống mạng mạnh mẽ và linh hoạt, cho phép việc truyền tải thông tin diễn ra một cách hiệu quả và an toàn. Bộ giao thức này không chỉ hỗ trợ cho việc giao tiếp giữa các máy tính mà còn làm nền tảng cho nhiều ứng dụng và dịch vụ trực tuyến hiện nay, từ việc duyệt web đến truyền tải video, email và nhiều hình thức giao tiếp khác. TCP/IP đã và đang tiếp tục định hình cách mà chúng ta kết nối và chia sẻ thông tin trên toàn cầu.

* 1. Giao thức socket

Giao thức Socket là một công nghệ quan trọng trong mạng máy tính, cho phép giao tiếp giữa hai thiết bị qua mạng. Socket hoạt động như một điểm cuối trong kết nối mạng, thông qua đó các thiết bị có thể gửi và nhận dữ liệu. Về cơ bản, một socket là sự kết hợp của địa chỉ IP và cổng (port), giúp xác định duy nhất một ứng dụng hoặc dịch vụ cụ thể trên một thiết bị. Socket thường dựa trên các giao thức như TCP hoặc UDP. Với TCP, socket đảm bảo kết nối ổn định, tuần tự và đáng tin cậy, phù hợp với các ứng dụng yêu cầu tính chính xác cao như truyền file hay email. Ngược lại, với UDP, socket hỗ trợ kết nối không đồng bộ, tốc độ nhanh, nhưng có thể bị mất dữ liệu, phù hợp cho các ứng dụng thời gian thực như truyền phát video hoặc trò chơi trực tuyến. Giao thức Socket đóng vai trò là cầu nối linh hoạt giữa các ứng dụng, cung cấp cơ sở cho việc xây dựng các ứng dụng mạng đa dạng và hiệu quả trong môi trường phân tán.

Trong khuổn khổ dự án, nhóm sử dụng phương thức socket TCP với hằng số SOCK\_STEAM trong thư viện socket của Python để thiết lập kết nối tuần tự và đáng tin cậy giữ các peer với nhau và với tracker.

* 1. Giao thức HTTP

HTTP (HyperText Transfer Protocol) là giao thức tiêu chuẩn cho việc trao đổi thông tin trên World Wide Web, cho phép các máy tính kết nối với nhau và truyền tải dữ liệu trong môi trường Internet. Đây là giao thức nền tảng giúp trình duyệt web có thể tải và hiển thị trang web từ các server web, và cũng là giao thức chính được sử dụng trong các ứng dụng web.

HTTP hoạt động theo mô hình client-server, trong đó **client** (thường là trình duyệt hoặc ứng dụng) gửi một yêu cầu (request) đến **server** và server sẽ phản hồi (response) với dữ liệu.

Một yêu cầu HTTP bao gồm **phương thức HTTP** (như GET, POST), **đường dẫn URL** đến tài nguyên, và **các header** chứa thông tin về yêu cầu.

HTTP là giao thức không trạng thái, có nghĩa là mỗi yêu cầu được gửi đến server đều độc lập và không chứa thông tin về các yêu cầu trước đó.

Các phương thức HTTP phổ biến:

* **GET**: Truy xuất dữ liệu từ server (dùng để lấy trang web hoặc tài nguyên khác).
* **POST**: Gửi dữ liệu lên server (ví dụ, gửi dữ liệu từ một form đăng ký).
* **PUT**: Cập nhật tài nguyên hiện có trên server.
* **DELETE**: Xóa tài nguyên trên server.
* Các phương thức này cho phép HTTP thực hiện nhiều loại tác vụ trên dữ liệu, giúp tổ chức các hoạt động web linh hoạt và dễ dàng quản lý.

Như vậy, trong dự án, để client thực hiện 1 yêu cầu truy vấn thông tin, phía client sẽ liên lạc với server tracker thông qua giao thức HTTP để yêu cầu phía server và nhận hồi đáp từ server thông qua giao thức trên.

* + 1. Yêu cầu chức năng và phi chức năng
  1. Yêu cầu chức năng

1. Đăng ký và Đăng nhập

* Đăng ký tài khoản: Người dùng có thể tạo tài khoản với tên đăng nhập và mật khẩu.
* Đăng nhập: Người dùng đăng nhập để truy cập các tính năng của hệ thống.
* Xác thực người dùng

2. Quản lý Tệp

* Chia sẻ tệp: Người dùng có thể chia sẻ tệp mới lên mạng P2P. Tệp sẽ được băm (hash) để tạo ra một mã định danh duy nhất.
* Tìm kiếm tệp: Người dùng có thể tìm kiếm các tệp đã được chia sẻ bởi những người dùng khác.
* Danh sách tệp chia sẻ: Hiển thị danh sách các tệp mà người dùng đã chia sẻ hoặc đang chia sẻ.

3. Tạo và Quản lý Tệp Torrent

* Tạo tệp torrent: Cho phép người dùng tạo tệp torrent từ các tệp chia sẻ. Tệp torrent sẽ chứa thông tin về tệp, kích thước, và danh sách các phần chia nhỏ của tệp.
* Quản lý các tệp torrent: Người dùng có thể xem và quản lý các tệp torrent đã tạo, bao gồm việc xóa, chỉnh sửa hoặc dừng chia sẻ.

4. Truyền Tệp (Download và Upload)

* Download tệp từ nhiều nguồn (Multi-source Download): Cho phép người dùng tải xuống tệp từ nhiều người dùng khác nhau cùng lúc, nhằm tăng tốc độ tải xuống.
* Upload tệp cho nhiều người dùng: Cho phép người dùng tải lên tệp cho nhiều peer cùng lúc, tối ưu tốc độ chia sẻ.
* Chia nhỏ tệp (Chunking): Mỗi tệp sẽ được chia nhỏ thành các phần, và người dùng sẽ tải hoặc tải lên từng phần riêng biệt.
* Tái cấu trúc tệp (Reassembly): Hệ thống sẽ tái cấu trúc các phần tải xuống thành tệp hoàn chỉnh trên máy người dùng.

5. Theo dõi Kết nối và Quản lý Peer

* Danh sách các peer: Hệ thống sẽ duy trì danh sách các peer đang có sẵn từng phần hoặc toàn bộ tệp.
* Tự động kết nối lại: Nếu kết nối tới một peer bị gián đoạn, hệ thống sẽ tự động thử lại và tìm các nguồn khác cho phần tệp còn thiếu.
* Theo dõi tiến độ tải xuống/tải lên: Người dùng có thể xem tiến độ của các phiên tải xuống/tải lên, bao gồm phần trăm hoàn thành và tốc độ truyền tải.

6. Tracker

* Tracker: Một server trung tâm sẽ đóng vai trò Tracker, theo dõi danh sách các client và các tệp họ đang chia sẻ, giúp các client kết nối nhanh chóng với nhau.

7. Hệ thống Kiểm tra và Sửa lỗi

* Kiểm tra tính toàn vẹn của tệp (Integrity Check): Khi tải xuống xong mỗi phần của tệp, hệ thống sẽ kiểm tra mã băm của phần đó để đảm bảo không có lỗi.
* Khôi phục khi lỗi: Nếu phát hiện lỗi trong một phần tải xuống, hệ thống sẽ tự động tải lại phần đó từ một peer khác.

8. Cơ chế Ưu tiên

* Ưu tiên tải lên/tải xuống: Người dùng có thể ưu tiên tải xuống hoặc tải lên cho một số tệp cụ thể.

10. Giao diện Người dùng (UI/UX)

* Giao diện quản lý phiên tải xuống/tải lên: Người dùng có thể dễ dàng quản lý, tạm dừng, tiếp tục hoặc hủy các phiên.
* Thông báo trạng thái: Hệ thống hiển thị các thông báo về trạng thái tải xuống/tải lên, tình trạng kết nối với các peer, và thông tin phiên hiện tại.

11. Thống kê và Báo cáo

* Thống kê chi tiết: Hệ thống cung cấp các thông tin về số lần tải xuống/tải lên, tốc độ trung bình, dung lượng dữ liệu đã chia sẻ.
* Lịch sử chia sẻ: Người dùng có thể xem lại lịch sử các tệp đã tải lên hoặc tải xuống, bao gồm các peer đã kết nối và các phiên đã hoàn thành.
  1. Yêu cầu phi chức năng

1. Hiệu suất (Performance)

* Độ trễ thấp: Hệ thống phải đáp ứng yêu cầu tải xuống và tải lên với độ trễ tối thiểu để cung cấp trải nghiệm người dùng tốt.
* Tốc độ truyền tải ổn định: Đảm bảo tốc độ truyền tải giữa các peer duy trì ổn định ngay cả khi có nhiều người dùng truy cập đồng thời.

2. Khả năng mở rộng (Scalability)

* Hỗ trợ nhiều kết nối đồng thời: Hệ thống phải có khả năng xử lý hàng ngàn kết nối từ nhiều peer mà không ảnh hưởng đến hiệu suất.
* Khả năng mở rộng linh hoạt: Khi lượng người dùng hoặc số lượng tệp tăng lên, hệ thống phải dễ dàng mở rộng mà không làm suy giảm hiệu năng.

3. Tính sẵn sàng và Độ tin cậy (Availability and Reliability)

* Duy trì kết nối: Trong trường hợp một peer mất kết nối, hệ thống phải có khả năng tìm kiếm peer thay thế để tiếp tục phiên tải xuống/tải lên mà không bị gián đoạn.

5. Khả năng sử dụng (Usability)

* Giao diện thân thiện: Hệ thống phải cung cấp giao diện đơn giản, dễ hiểu để người dùng dễ dàng thao tác, quản lý các phiên tải xuống/tải lên.
* Thông báo trạng thái và cảnh báo: Hiển thị rõ ràng các thông báo về trạng thái của phiên tải xuống/tải lên, và cảnh báo người dùng khi có lỗi xảy ra.

6. Tối ưu hóa tài nguyên (Resource Optimization)

* Tối ưu dung lượng lưu trữ: Các tệp và dữ liệu cần được lưu trữ một cách hiệu quả để không lãng phí không gian lưu trữ.

7. Khả năng mở rộng tính năng (Extensibility)

* Hỗ trợ tính năng mới: Hệ thống nên được thiết kế mở để có thể dễ dàng bổ sung tính năng mới mà không gây ảnh hưởng đến các chức năng hiện có.
* Hỗ trợ tích hợp API: Hệ thống nên có khả năng tích hợp với các API khác để mở rộng chức năng và giao tiếp với các ứng dụng hoặc dịch vụ khác.
  + 1. Mô hình hệ thống

1. Usecase Diagram
2. Usecase table
3. Sequence Diagram
   * 1. Hiện thực hệ thống

Tracker server

1. Định nghĩa tracker server

Trong hệ thống chia sẻ tệp ngang hàng (P2P) như torrent, tracker đóng vai trò quan trọng trong việc kết nối các peer (các thiết bị hoặc máy tính tham gia vào quá trình chia sẻ tệp) với nhau. Mặc dù bản thân tracker không chứa nội dung tệp, nó giúp các peer tìm thấy nhau và thiết lập các kết nối cần thiết để chia sẻ tệp một cách hiệu quả.

Khi một peer mới tham gia và yêu cầu tải xuống một tệp, tracker sẽ cung cấp cho nó danh sách các peer khác đang sở hữu các phần của tệp đó để nó có thể bắt đầu tải xuống. Đối với các mạng P2P mới, tracker giúp các peer mới tham gia mạng tìm thấy các peer khác, từ đó bắt đầu quá trình tải xuống. Một khi các kết nối giữa các peer được thiết lập, các peer có thể truyền dữ liệu trực tiếp với nhau mà không cần sự tham gia của tracker nữa.

Dự án lần này, nhóm sử dùng 1 tracker trung tâm đống vài trò kết nối các peer thành viên, lưu trữ và truy vấn các thông tin cần thiết trong quá trình tải file.

1. Kiến trúc tracker server

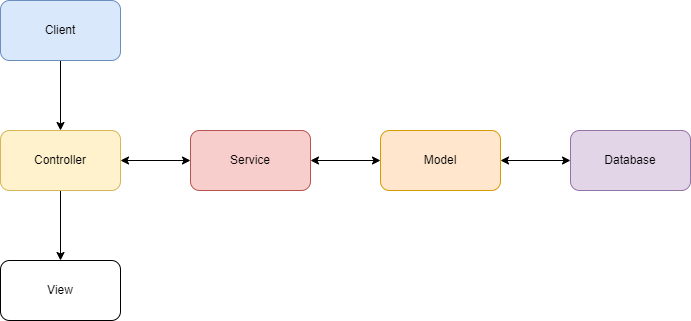
Server tracker sử dụng trong dự án được xây dựng dựa trên mô hình MVC theo kiến trúc Monolithic hay MVC Monolithic.

Khác với kiến trúc Microservice, kiến trúc MVC (Model-View-Controller) Monolithic là một kiểu thiết kế phần mềm được tổ chức theo mô hình MVC, trong đó toàn bộ ứng dụng được xây dựng và triển khai như một khối đơn lẻ (monolith), thay vì phân tán thành các dịch vụ độc lập như trong kiến trúc microservices. Kiến trúc MVC monolithic thường được sử dụng trong các ứng dụng web truyền thống và có cấu trúc rõ ràng, phân chia thành ba thành phần chính:

 **Model**: Quản lý dữ liệu, logic nghiệp vụ và các quy tắc của ứng dụng. Đây là phần kết nối trực tiếp với cơ sở dữ liệu, thực hiện các thao tác như lưu trữ, truy xuất và xử lý dữ liệu.

 **View**: Đóng vai trò là giao diện người dùng (UI) của ứng dụng, hiển thị dữ liệu từ Model và nhận dữ liệu đầu vào từ người dùng. View không chứa logic nghiệp vụ mà chỉ tập trung vào việc hiển thị dữ liệu.

 **Controller**: Là trung gian giữa Model và View, tiếp nhận các yêu cầu từ người dùng qua View, xử lý và cập nhật dữ liệu bằng cách tương tác với Model, sau đó trả về kết quả cho View để hiển thị cho người dùng.



Đối với hệ quản trị cơ sở dữ liệu, nhóm sử dụng MySQL là một DBMS hỗ trợ truy vấn có cấu trúc SQL để lưu trữ dữ liệu.

1. Cấu hình tracker

Để đáp ứng các chức năng cần thiết cho quá trình tải xuống, tracker cần lưu thông của User (người dùng), File và Peer.

Thuộc tính của User (định nghĩa trong user.entity.ts):

@Table

export class User extends Model<User> {

    @PrimaryKey

    @Default(UUIDV4)

    @Column(DataType.UUID)

    id: *UUID*;

    @AllowNull(false)

    @Column(DataType.STRING)

    username: string;

    @AllowNull(false)

    @Column(DataType.STRING)

    password: string;

    @AllowNull(false)

    @Column(DataType.STRING)

    fullName: string;

    @Default(0)

    @Column(DataType.INTEGER)

    point: number;

}

Thuộc tính Peer (định nghĩa trong peer.entity.ts):

@Table

export class Peer extends Model<Peer> {

    @PrimaryKey

    @Default(UUIDV4)

    @AllowNull(false)

    @Column(DataType.UUID)

    id: *UUID*;

    @Column(DataType.STRING)

    IPaddress: string;

    @Column(DataType.INTEGER)

    port: number;

    @BelongsToMany(()=> File, () => PeerHoldFile)

    files: File[];

}

Thuộc tính File (định nghĩa trong file.entity.ts):

@Table

export class File extends Model<File> {

    @PrimaryKey

    @Default(UUIDV4)

    @Column(DataType.UUID)

    id: *UUID*;

    @Column(DataType.STRING)

    name: string;

    @Column(DataType.INTEGER)

    size: number;

    @Unique

    @Column(DataType.STRING(255)) // or another suitable length for your data

    infoHash: string;

    @ForeignKey(() => User)

    @Column(DataType.UUID)

    userId: *UUID*;

    @BelongsTo(() => User)

    user: User;

    @BelongsToMany(() => Peer, () => PeerHoldFile)

    peers: Peer[];

}

Đối với quan hệ giữa Peer và File là quan hệ nhiều-nhiều (1 peer giữ nhiều file và 1 file có thể thuộc về nhiều peer chiếm giữ) ta định nghĩa bảng trung gian PeerHoldFile (peerHoldFile.entity.ts) để mapping 2 đối tượng:

@Table

export class PeerHoldFile extends Model<PeerHoldFile> {

    @PrimaryKey

    @Default(UUIDV4)

    @AllowNull(false)

    @Column(DataType.UUID)

    id: *UUID*;

    @ForeignKey(() => File)

    @Column(DataType.UUID)

    fileId: *UUID*;

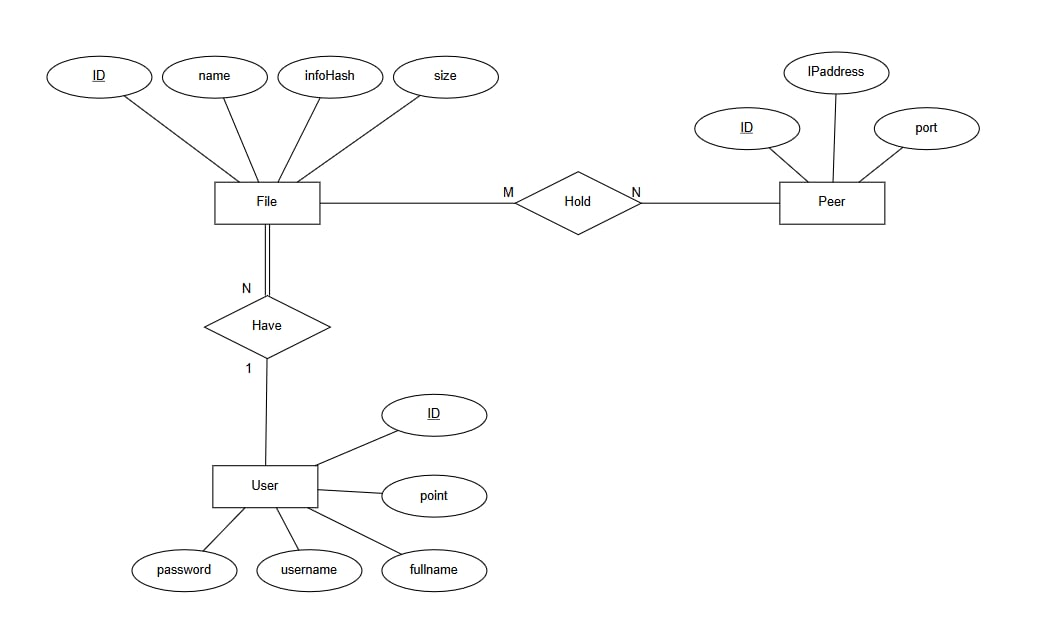
    @ForeignKey(() => Peer)

    @Column(DataType.UUID)

    peerId: *UUID*

}

Mô hình ERD:



Mô tả:

Hệ thống cần lưu trữ các thông tin về file, user và peer trong mạng lưới. Đối với file, cần lưu các thông tin sau: mã file (danh định), tên file, kích thước và chuỗi hash.

Đối với peer, cần lưu: mã peer (danh định), IPaddress và port.

Đối với user, cần lưu: mã user (danh định), tài khoản, mật khẩu, tên đầy đủ và số điểm của user.

Mỗi peer có thể sở hữu nhiều file và mỗi file có nằm ở nhiều peer khác nhau nên quan hệ nắm giữ (Hold) là nhiều-nhiều.

Mỗi user có thể chứa nhiều file ở local nhưng 1 file chỉ thuộc về 1 user nên quan hệ có (Has) là một-nhiều.

1. Hiện thực dịch vụ Tracker
2. Fetch thông tin file bằng infoHash

Ở lớp Controller:

@Get('fetch')

    async fetch(@Res() res, @Query('infoHash') infoHash: string) {

        try {

            const file = await *this*.fileService.search(infoHash);

*this*.response.initResponse(true, "Tìm thông tin thành công", file);

            return res.status(*HttpStatus*.CREATED).json(*this*.response);

        } catch (error) {

*this*.logger.error(error.message, error.stack);

            if (error instanceof InternalServerErrorException) {

*this*.response.initResponse(false, error.message, null);

                return res.status(*HttpStatus*.INTERNAL\_SERVER\_ERROR).json(*this*.response);

            }

            if (error instanceof BadRequestException) {

*this*.response.initResponse(false, error.message, null);

                return res.status(*HttpStatus*.BAD\_REQUEST).json(*this*.response);

            }

*this*.response.initResponse(false, "Đã xảy ra lỗi. Vui lòng thử lại", null);

            return res.status(*HttpStatus*.INTERNAL\_SERVER\_ERROR).json(*this*.response);

        }

    }

Ở lớp Service:

async search(infoHash: string) {

        let findOption: *FindOptions* = {

            include: [

                {

                    model: Peer,

                    through: { attributes: [] },

                    attributes: ['IPaddress', 'port']

                },

                {

                    model: User,

                    attributes: ['id', 'fullname', 'point']

                }

            ]

        };

        if (infoHash) {

            findOption.where = { infoHash };

        }

        return await *this*.fileRepository.findAll(findOption);

    }

Phương thức fetch trong lớp Controller là một API endpoint xử lý yêu cầu HTTP GET tới đường dẫn /fetch với tham số truy vấn infoHash. Nó có nhiệm vụ:

* **Bước 1:** Lấy tham số infoHash từ truy vấn của yêu cầu.
* **Bước 2:** Sử dụng phương thức search của fileService để tìm kiếm thông tin tệp dựa trên infoHash đã được cung cấp.
* **Bước 3:** Sau khi nhận được kết quả từ fileService, phản hồi lại cho client với trạng thái thành công (CREATED) và dữ liệu của tệp được tìm thấy.

Nếu có lỗi xảy ra, phương thức sẽ bắt lỗi và trả về các phản hồi phù hợp:

* **Internal Server Error**: Nếu lỗi là InternalServerErrorException, trả về phản hồi với trạng thái **500 INTERNAL\_SERVER\_ERROR**.
* **Bad Request**: Nếu lỗi là BadRequestException, trả về phản hồi với trạng thái **400 BAD\_REQUEST**.
* **Các lỗi khác**: Trả về thông báo lỗi chung và trạng thái **500 INTERNAL\_SERVER\_ERROR**.

Phương thức fetch sử dụng **@Res()** để truy cập đối tượng phản hồi và **@Query('infoHash')** để lấy giá trị infoHash từ tham số truy vấn.

Phương thức search trong lớp Service thực hiện việc truy vấn thông tin từ cơ sở dữ liệu, dựa trên mã định danh infoHash:

* **Thiết lập tùy chọn truy vấn**: Sử dụng đối tượng FindOptions để chỉ định các bảng liên quan và các trường sẽ lấy từ mỗi bảng.
  + **Peer**: Bao gồm bảng Peer liên kết với File thông qua through, chỉ lấy các trường IPaddress và port.
  + **User**: Bao gồm bảng User với các trường id, fullname, và point.
* **Áp dụng điều kiện tìm kiếm**: Nếu infoHash có giá trị, thiết lập điều kiện tìm kiếm để tìm các tệp có infoHash tương ứng.
* **Truy vấn cơ sở dữ liệu**: Gọi this.fileRepository.findAll(findOption) để tìm tất cả các tệp phù hợp với điều kiện, sau đó trả về kết quả cho Controller.

1. Publish file (hay Upload file)

Ở lớp controller:

@Post('upload')

    async upload(@Req() req, @Body() uploadFileDto: UploadFileDto, @Res() res) {

        try {

            const createdFile = await *this*.fileService.create(uploadFileDto);

*this*.response.initResponse(true, "Tạo peer thành công", createdFile);

            return res.status(*HttpStatus*.CREATED).json(*this*.response);

        } catch (error) {

*this*.logger.error(error.message, error.stack);

            if (error instanceof InternalServerErrorException) {

*this*.response.initResponse(false, error.message, null);

                return res.status(*HttpStatus*.INTERNAL\_SERVER\_ERROR).json(*this*.response);

            }

            if (error instanceof BadRequestException) {

*this*.response.initResponse(false, error.message, null);

                return res.status(*HttpStatus*.BAD\_REQUEST).json(*this*.response);

            }

*this*.response.initResponse(false, "Đã xảy ra lỗi. Vui lòng thử lại", null);

            return res.status(*HttpStatus*.INTERNAL\_SERVER\_ERROR).json(*this*.response);

        }

    }

Ở lớp Peer service:

async create(dto: UploadFileDto) {

        const existedUser = await *this*.userService.findOneById(dto.userId);

        if (!existedUser) {

            throw **new** NotFoundException('Người dùng không tồn tại');

        }

        const [file] = await *this*.fileRepository.findOrCreate({

            where: {

                infoHash: dto.infoHash

            },

            defaults: {

                infoHash: dto.infoHash,

                name: dto.filename,

                size: dto.size,

                userId: dto.userId

            }

        });

        const [peer] = await *this*.peerRepository.findOrCreate({

            where: {

                IPaddress: dto.peerIPAddress,

                port: dto.peerPort

            },

            defaults: {

                IPaddress: dto.peerIPAddress,

                port: dto.peerPort

            }

        })

        await *this*.pHFService.create({ fileId: file.id, peerId: peer.id });

        return file;

    }

Ở lớp PeerHoldFile Service:

async create(createDto: CreatePHFDTO) {

        const existedRelation = await *this*.pHFRepository.findOne({

            where: {

                fileId: createDto.fileId,

                peerId: createDto.peerId

            }

        });

        if(existedRelation) {

            throw **new** ConflictException('Liên kết Peer-File đã được tạo từ trước');

        }

        return await *this*.pHFRepository.create({

            fileId: createDto.fileId,

            peerId: createDto.peerId

        });

    }

Phương thức upload trong lớp Controller là một API endpoint xử lý yêu cầu HTTP POST tới đường dẫn /upload. Nhiệm vụ của phương thức này là:

* **Bước 1:** Nhận dữ liệu từ yêu cầu thông qua uploadFileDto, chứa thông tin cần thiết để tạo một file và peer mới.
* **Bước 2:** Gọi phương thức create của fileService để lưu thông tin file và peer vào cơ sở dữ liệu.
* **Bước 3:** Sau khi file và peer được tạo thành công, trả về phản hồi với trạng thái **201 CREATED** cùng với dữ liệu của file vừa tạo.

Trong trường hợp xảy ra lỗi, upload sẽ bắt lỗi và trả về phản hồi phù hợp:

* **Internal Server Error**: Nếu lỗi là InternalServerErrorException, trả về phản hồi với trạng thái **500 INTERNAL\_SERVER\_ERROR**.
* **Bad Request**: Nếu lỗi là BadRequestException, trả về phản hồi với trạng thái **400 BAD\_REQUEST**.
* **Các lỗi khác**: Trả về thông báo lỗi chung và trạng thái **500 INTERNAL\_SERVER\_ERROR**.

Phương thức upload sử dụng **@Req()** để truy cập yêu cầu gốc và **@Body()** để nhận dữ liệu từ yêu cầu HTTP POST.

**2. Mô tả lớp Service:**

Phương thức create trong lớp Service thực hiện việc xử lý logic để tạo file và peer mới trong cơ sở dữ liệu. Quá trình này bao gồm các bước sau:

* **Kiểm tra người dùng**: Truy vấn người dùng dựa trên userId từ dto. Nếu người dùng không tồn tại, ném ra lỗi NotFoundException với thông báo “Người dùng không tồn tại”.
* **Tạo hoặc tìm file**:
  + Sử dụng findOrCreate để tìm hoặc tạo file mới dựa trên infoHash từ dto.
  + Nếu file chưa tồn tại, tạo mới file với các thuộc tính gồm infoHash, filename, size, và userId.
* **Tạo hoặc tìm peer**:
  + Sử dụng findOrCreate để tìm hoặc tạo peer mới dựa trên IPaddress và port từ dto.
  + Nếu peer chưa tồn tại, tạo mới peer với các thuộc tính IPaddress và port.
* **Tạo mối quan hệ giữa file và peer**:
  + Sử dụng pHFService.create để lưu quan hệ giữa fileId và peerId, cho phép liên kết giữa file và peer.
* **Trả về file**: Sau khi hoàn thành tất cả các bước, trả về đối tượng file đã tạo hoặc tìm thấy cho Controller.

Ngoài các dịch vụ chính trên còn đăng nhập, đăng kí, tìm và sửa đổi thông tin User, các chức năng trên cơ bản, tương với lớp Controller của User (use.controller.ts) và lớp Service của User (user.service.ts).

Client:

1. download

**Cách 1: Giải mã Magnet Link:** Đầu tiên, hàm bắt đầu bằng cách giải mã liên kết magnet để lấy các thông tin như info\_hash, tên file và kích thước file.

def decode\_magnet\_link(magnet\_link):

    # Parse the magnet link

    parsed = urllib.parse.urlparse(magnet\_link)

    params = urllib.parse.parse\_qs(parsed.query)

    # Extract components

    info\_hash = params.get("xt", [None])[0]

    if info\_hash and info\_hash.startswith("urn:btih:"):

        info\_hash = info\_hash[9:]  # Remove 'urn:btih:' prefix

    filename = params.get("dn", [None])[0]

    trackers = params.get("tr", [])

    web\_seeds = params.get("ws", [])

    file\_size = params.get("xl", [None])[0]

    if file\_size:

        file\_size = int(file\_size)

    # Return extracted info as a dictionary

    return {

        "info\_hash": info\_hash,

        "filename": filename,

        "trackers": trackers,

        "web\_seeds": web\_seeds,

        "file\_size": file\_size

    }

Hàm decode\_magnet\_link được thiết kế để giải mã một liên kết magnet nhằm trích xuất các thông tin quan trọng cần thiết cho quá trình tải xuống file trong hệ thống peer-to-peer. Cụ thể, hàm này lấy mã băm của file (info\_hash), tên file (filename), danh sách các tracker (trackers) để tìm kiếm peer, danh sách các web seeds (web\_seeds) nếu có, và kích thước file (file\_size).

**Cách 2: Giải mã file torrent**. Từ file torrent ta lấy thông tin infoHash, tên file và kích thước file.

def extract\_torrent\_info(torrent\_file):

    # Kiểm tra file torrent có tồn tại không

    if not os.path.isfile(torrent\_file):

        raise FileNotFoundError(f"File '{torrent\_file}' không tồn tại.")

    # Tạo đối tượng torrent từ file

    torrent = Torrent.from\_file(torrent\_file)

    # Trích xuất thông tin cần thiết

    info\_hash = torrent.info\_hash  # Lấy info hash

    filename = torrent.name          # Lấy tên file

    file\_size = sum(file.length for file in torrent.files)  # Tổng kích thước của tất cả các file

    trackers = torrent.announce\_urls # Lấy danh sách trackers

    web\_seeds = torrent.webseeds    # Lấy danh sách web seeds

    return {

        'info\_hash': info\_hash,

        'filename': filename,

        'file\_size': file\_size,

        'trackers': trackers,

        'web\_seeds': web\_seeds

    }

**Lấy thông tin về peers từ server**: Sử dụng get\_file\_info\_from\_server(info\_hash) để nhận danh sách các peers có khả năng chia sẻ file dựa trên info\_hash.

def get\_file\_info\_from\_server(info\_hash):

    response = requests.get(f'http://localhost:3000/v1/file/fetch?hash\_info={info\_hash}')

    try:

        # Try to parse the response as JSON regardless of the content type header

        json\_response = response.json()

        # Extract values with defaults in case fields are missing

        success = json\_response.get('success', False)

        data = json\_response.get('data', {})

        message = json\_response.get('message', "No message provided")

        return {'success': success, 'data': data, 'message': message}

    except ValueError:

        # If JSON decoding fails, handle it as a non-JSON response

        print("Received non-JSON response:", response.text)

        return {'success': False, 'data': {}, 'message': "Non-JSON response"}

Hàm get\_file\_info\_from\_server gửi một yêu cầu GET đến server với tham số infoHash để lấy thông tin về file. Hàm phân tích phản hồi từ server dưới dạng JSON.

Nếu thành công, nó sẽ trích xuất các trường success, data, và message từ phản hồi JSON. Các trường này có giá trị mặc định nếu không có trong phản hồi.

Nếu phản hồi không thể phân tích thành JSON (ví dụ: phản hồi không phải JSON), hàm sẽ trả về một thông báo lỗi và dữ liệu trống.

Hàm này cung cấp thông tin cần thiết về các peers, giúp hàm download xác định các nguồn có sẵn để tải file.

**Kiểm tra Trạng thái File từ Các Peers**: Hàm get\_file\_status\_in\_peer(ips[i], ports[i], info\_hash) được sử dụng để kiểm tra trạng thái file trên mỗi peer. Hàm này kết nối tới peer thông qua socket, gửi yêu cầu về trạng thái file, và nhận phản hồi về trạng thái các phần file mà peer đang sở hữu.

def get\_file\_status\_in\_peer(peer\_ip, peer\_port, info\_hash):

    try:

        with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

            s.connect((peer\_ip, peer\_port))

            print(f"Connected to {peer\_ip}:{peer\_port}")

            request = {

                'type': 'GET\_FILE\_STATUS',

                'info\_hash': info\_hash

            }

            s.sendall(json.dumps(request).encode('utf-8'))

            response\_data = s.recv(4096)

            response = json.loads(response\_data.decode('utf-8'))

            # print(f"and get pieces: {response['pieces\_status']}")

            if response['type'] == 'RETURN\_FILE\_STATUS' and response['info\_hash'] == info\_hash:

                pieces\_status = response['pieces\_status']

                point = response['point']

                return peer\_ip, peer\_port, pieces\_status, point

            else:

                return None, None, None, 0

    except (socket.error, ConnectionRefusedError, TimeoutError) as e:

        print(f"Connection error: {e}")

        return None, None, None, 0

**Quản lý Trạng thái Tải Xuống:** Hàm kiểm tra xem file đã được tải xuống hay chưa và tạo thư mục cùng file JSON để lưu trữ trạng thái của các phần file nếu chúng chưa tồn tại. Trạng thái hiện tại của các phần file được lưu trữ trong file status.json.

points = {}

    for i in range(len(ips)):

        points[(ips[i], ports[i])] = point[i]

    peers = [(ips[i], ports[i]) for i in range(len(ips))]

    selected\_chunks = {peer: [] for peer in peers}

    num\_chunks = len(pieces[0])

    currentStatus = []

    if not os.path.exists(f'files/{info\_hash}'):

        os.makedirs(f'files/{info\_hash}')

    if not os.path.exists(f'files/{info\_hash}/status.json'):

        with open(f'files/{info\_hash}/status.json', 'w') as f:

            json.dump({"fileName": fileName, "piece\_status": [

                0 for \_ in range(fileSize // PIECE\_SIZE)

            ]}, f)

    if not os.path.exists(f'storage/{fileName}'):

        with open(f'storage/{fileName}', 'w') as f:

            pass

    with open(f'files/{info\_hash}/status.json', 'r') as f:

        data = json.load(f)

    if data != None:

        currentStatus = data["piece\_status"]

**Lựa chọn Peer Tốt Nhất:** Trong quá trình tải xuống, hàm xác định peer tốt nhất cho từng phần file dựa trên trạng thái hiện tại và điểm số (priority) của từng peer. Mỗi lần người dùng tải xuống một hoặc nhiều chunk, điểm số của họ sẽ được cộng thêm tương ứng với số lượng chunk đã tải xuống. Ngược lại, khi một peer chia sẻ chunk cho người khác, điểm số của họ sẽ bị trừ đi bấy nhiêu.

for chunk\_index in range(num\_chunks):

        if data != None and currentStatus[chunk\_index] == 1:

            continue

        best\_peer = None

        best\_priority = -1000

        for peer\_index, peer\_chunks in enumerate(pieces):

            if peer\_chunks[chunk\_index] == 1:

                peer = peers[peer\_index]

                priority = points[peer]

                if priority > best\_priority:

                    best\_priority = priority

                    best\_peer = peer

        if best\_peer is not None:

            selected\_chunks[best\_peer].append(chunk\_index)

Hệ thống sử dụng một mảng để theo dõi trạng thái của các phần file (piece status) và điểm số của từng người dùng. Dựa trên các điểm số này, một hàng đợi ưu tiên (priority queue) sẽ được sử dụng để sắp xếp mức độ ưu tiên tải cho các peer. Điều này có nghĩa là những peer có điểm số cao hơn sẽ được ưu tiên tải xuống nhiều hơn, từ đó tạo ra mảng chunk\_list cho từng peer, giúp tối ưu hóa quá trình tải xuống và khuyến khích người dùng chia sẻ file hiệu quả hơn.

**Tải Xuống File:** Sử dụng multithreading, hàm khởi động nhiều luồng để tải xuống các phần file từ các peers đã được chọn. Mỗi luồng thực hiện chức năng download\_file\_chunk\_from\_peer. Chức năng này thực hiện việc tải xuống một hoặc nhiều phần của file từ một peer trong mạng peer-to-peer. Đầu tiên, hàm tạo một kết nối socket tới địa chỉ IP và cổng của peer, sau đó gửi yêu cầu dưới dạng JSON để lấy các phần file cần thiết. Khi nhận được phản hồi từ peer, hàm kiểm tra xem phản hồi có hợp lệ hay không; nếu đúng, nó trích xuất dữ liệu của các phần và ghi chúng vào vị trí chính xác trong file trên đĩa, đồng thời cập nhật trạng thái của từng phần đã tải xuống thành công. Cuối cùng, hàm ghi lại trạng thái mới vào tệp status.json, giúp theo dõi quá trình tải xuống. Nếu phản hồi không hợp lệ, hàm sẽ in ra thông báo cảnh báo. Hàm này giúp tối ưu hóa quá trình tải xuống bằng cách phân chia file thành các phần nhỏ và tải từ nhiều peer khác nhau.

threads = []

    for (ip, port), chunks in selected\_chunks.items():

        peer\_thread = threading.Thread(target=download\_file\_chunk\_from\_peer, args=(ip, port, info\_hash, chunks, f"storage/{fileName}"),daemon=True)

        peer\_thread.start()

        threads.append(peer\_thread)

    for thread in threads:

        thread.join()

def download\_file\_chunk\_from\_peer(peer\_ip, peer\_port, info\_hash, chunk\_list, file\_path):

    with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

        s.connect((peer\_ip, peer\_port))

        # print(f"Connected to {peer\_ip}:{peer\_port}")

        request = {

            'type': 'GET\_FILE\_CHUNK',

            'info\_hash': info\_hash,

            'chunk\_list': chunk\_list

        }

        s.sendall(json.dumps(request).encode('utf-8'))

        response\_data = s.recv(4096)

        response = json.loads(response\_data.decode('utf-8'))

        # print(f"and get chunks: {response['chunk\_data']}")

        if response['type'] == 'RETURN\_FILE\_CHUNK' and response['info\_hash'] == info\_hash:

            chunk\_data = response['chunk\_data']

            with file\_lock:

                with open(f"files/{info\_hash}/status.json", "r") as file:

                    data = json.load(file)

                    currentStatus = data["piece\_status"]

                    fileName = data["fileName"]

                with open(file\_path, "r+b") as f:

                    for i, chunk in enumerate(chunk\_data):

                        f.seek(chunk\_list[i] \* PIECE\_SIZE)

                        f.write(chunk.encode('latin1'))

                        currentStatus[chunk\_list[i]] = 1

                print(currentStatus)

                with open(f"files/{info\_hash}/status.json", "w") as file:

                    json.dump(

                        {

                            "fileName": fileName,

                            "piece\_status": currentStatus

                        },

                        file

                    )

        else:

            print("Has been received invalid response from peer")

Cập nhật Trạng thái Người Dùng: Sau khi quá trình tải xuống hoàn tất, hàm đọc trạng thái từ file status.json, tính toán điểm số người dùng mới và cập nhật thông tin người dùng trong hệ thống.

dataInJsonFile = {}

    with open(f"files/{info\_hash}/status.json", "r") as file:

        data = json.load(file)

        dataInJsonFile = data

        if all(status == 0 for status in data["piece\_status"]):

            return

1. upload\_full\_file

Hàm này được thiết kế để tải lên một file hoàn chỉnh từ thư mục lưu trữ vào hệ thống quản lý torrent.

def upload\_full\_file(filename):

    filesize = os.path.getsize(f"storage/{filename}")

    infohash = helper.generate\_hash\_info(f"storage/{filename}")

    os.makedirs(f"files/{infohash}", exist\_ok=True)

    with open(f'files/{infohash}/status.json', 'w') as file:

        json.dump(

            {

                "fileName": filename,

                "piece\_status": [1 for \_ in range(filesize // PIECE\_SIZE) ]

            },

            file

        )

    f = open("userId.txt", "r")

    userId = f.read()

    helper.upload\_file(infohash, filename, filesize, self\_ip\_address, self\_port, userId)

    helper.file\_to\_torrent(f"storage/{filename}","http://localhost:3000/v1",filesize)

    return helper.generate\_magnet\_link(infohash, filename, filesize, "http://localhost:3000/v1", None)

Quá trình thực hiện hàm diễn ra theo các bước sau:

**Xác định kích thước file**: Hàm bắt đầu bằng việc lấy kích thước của file thông qua os.path.getsize, với đường dẫn là storage/{filename}. Điều này giúp xác định dung lượng của file cần tải lên.

**Tạo infohash**: Tiếp theo, hàm gọi generate\_hash\_info để tạo ra một (infohash) cho file, giúp xác định duy nhất file trong hệ thống.

**Tạo thư mục cho file**: tạo ra một thư mục mới tương ứng với infohash trong thư mục files. Nếu thư mục đã tồn tại, lệnh này sẽ không gây lỗi.

**Lưu trữ trạng thái file**: Trong thư mục vừa tạo, hàm mở hoặc tạo file status.json và lưu trữ tên file cùng với trạng thái các phần (pieces) của file. Trạng thái được khởi tạo với giá trị 1 cho mỗi phần, chỉ ra rằng tất cả các phần đã được tải lên.

**Đọc userId**: Hàm mở file userId.txt để đọc userId, xác định người dùng hiện tại đang thực hiện upload.

**Gửi file lên server**: Tiếp theo, hàm gọi upload\_file(infohash, filename, filesize, self\_ip\_address, self\_port, userId để gửi yêu cầu tải lên một file đến server thông qua HTTP POST. Trong quá trình thực hiện, hàm gửi yêu cầu đến địa chỉ http://localhost:3000/v1/file/upload, kèm theo thông tin như infoHash, tên file, địa chỉ IP và cổng của peer, kích thước file và ID người dùng. Sau khi gửi yêu cầu, hàm cố gắng phân tích phản hồi từ server dưới dạng JSON để xác định xem yêu cầu có thành công hay không, đồng thời trích xuất dữ liệu và thông điệp từ phản hồi. Nếu phản hồi không phải là JSON, hàm sẽ trả về thông tin về lỗi. Kết quả cuối cùng là một từ điển chứa thông tin về trạng thái tải lên, giúp đảm bảo rằng quá trình tải lên diễn ra suôn sẻ và thông tin phản hồi được xử lý hiệu quả.

**Chuyển đổi file thành định dạng torrent**: Sau khi upload thành công, hàm gọi file\_to\_torrent để chuyển đổi file thành định dạng torrent và gửi nó đến địa chỉ URL chỉ định.

**Tạo liên kết magnet**: Cuối cùng, hàm trả về một liên kết magnet thông qua generate\_magnet\_link Liên kết này cho phép người dùng chia sẻ file trong hệ thống torrent.

1. Handle\_client

Hàm handle\_client được thiết kế để xử lý các yêu cầu từ client trong một hệ thống chia sẻ file. Khi một client kết nối, hàm bắt đầu bằng cách nhận dữ liệu từ socket của client và giải mã nó từ định dạng JSON. Tùy thuộc vào loại yêu cầu (type), hàm thực hiện các thao tác khác nhau.

Hàm bắt đầu bằng cách nhận dữ liệu từ socket của client thông qua client\_socket.recv(1024), sau đó giải mã dữ liệu thành định dạng JSON bằng json.loads(data).

**Xử lý Yêu cầu GET\_FILE\_STATUS**:

if request['type'] == 'GET\_FILE\_STATUS':

            info\_hash = request['info\_hash']

            response = {

                'type': 'RETURN\_FILE\_STATUS',

                'info\_hash': info\_hash,

                'fName': None,

                'pieces\_status': [],

                'point': 0

            }

            print(info\_hash)

            with open(f'files/{info\_hash}/status.json', 'r') as f:

                data = json.load(f)

            if not data:

                client\_socket.sendall(json.dumps(response).encode('utf-8'))

                return

            file\_name = get\_filename\_in\_folder(f"files/{info\_hash}")

            f = open("userId.txt", "r")

            userId = f.read()

            point = helper.search\_by\_id(userId)["data"]["point"]

            response = {

                'type': 'RETURN\_FILE\_STATUS',

                'info\_hash': info\_hash,

                'fName': file\_name,

                'pieces\_status': data['piece\_status'],

                'point': point

            }

            client\_socket.sendall(json.dumps(response).encode('utf-8'))

Khi nhận được yêu cầu GET\_FILE\_STATUS, hàm sẽ lấy giá trị info\_hash từ yêu cầu.

Một phản hồi mặc định được khởi tạo, chứa thông tin về file, trạng thái các phần (pieces), và điểm số (point).

Hàm mở file status.json nằm trong thư mục tương ứng với info\_hash để đọc trạng thái của file. Nếu không tìm thấy file, hàm gửi phản hồi mặc định về cho client.

Nếu file tồn tại, hàm sẽ gọi hàm get\_filename\_in\_folder để lấy tên file và đọc userId từ file userId.txt.

Điểm số của người dùng được truy xuất bằng cách gọi search\_by\_id(userId), và một phản hồi chi tiết chứa tên file, trạng thái các phần, và điểm số được gửi về cho client.

**Xử lý Yêu cầu GET\_FILE\_CHUNK:**

Khi yêu cầu là GET\_FILE\_CHUNK, hàm sẽ lấy info\_hash và danh sách các chunk từ request.

elif request['type'] == 'GET\_FILE\_CHUNK':

            info\_hash = request['info\_hash']

            chunk\_list = request['chunk\_list']

            chunk\_data = []

            response = {

                'type': 'RETURN\_FILE\_CHUNK',

                'info\_hash': info\_hash,

                'chunk\_data': []

            }

            with open(f'files/{info\_hash}/status.json', 'r') as f:

                data = json.load(f)

            if not data:

                client\_socket.sendall(json.dumps(response).encode('utf-8'))

                return

            file\_name = get\_filename\_in\_folder(f"files/{info\_hash}")

            try:

                with open(f"storage/{file\_name}", "rb") as f:

                    for chunk\_index in chunk\_list:

                        f.seek(chunk\_index \* PIECE\_SIZE)

                        data = f.read(PIECE\_SIZE)

                        chunk\_data.append(data.decode('latin1'))

                f = open("userId.txt", "r")

                userId = f.read()

                userPoint = helper.search\_by\_id(userId)["data"]["point"]

                newPoint = userPoint - (len(chunk\_list))

                helper.update\_user(newPoint, userId)

            except FileNotFoundError:

                print(f"File {file\_name} does not exit.")

                client\_socket.sendall(json.dumps(response).encode('utf-8'))

                return

            response['chunk\_data'] = chunk\_data

            client\_socket.sendall(json.dumps(response).encode('utf-8'))

Một phản hồi mặc định chứa thông tin về chunk được khởi tạo.

File status.json được mở để đọc trạng thái của file. Nếu không tìm thấy file, phản hồi mặc định sẽ được gửi về cho client.

Hàm sẽ lấy tên file bằng cách gọi get\_filename\_in\_folder và mở file trong thư mục lưu trữ để đọc dữ liệu cho các chunk đã yêu cầu.

Dữ liệu từ các chunk được lưu trữ trong một danh sách và điểm số của người dùng được cập nhật bằng cách trừ đi số lượng chunk đã tải xuống.

Cuối cùng, hàm gửi dữ liệu chunk đã đọc trở lại client.

**Xử lý Yêu cầu PING:**

Nếu yêu cầu từ client là PING, hàm sẽ gửi lại phản hồi PONG để xác nhận kết nối và sự hiện diện của server.

elif request['type'] == 'PING':

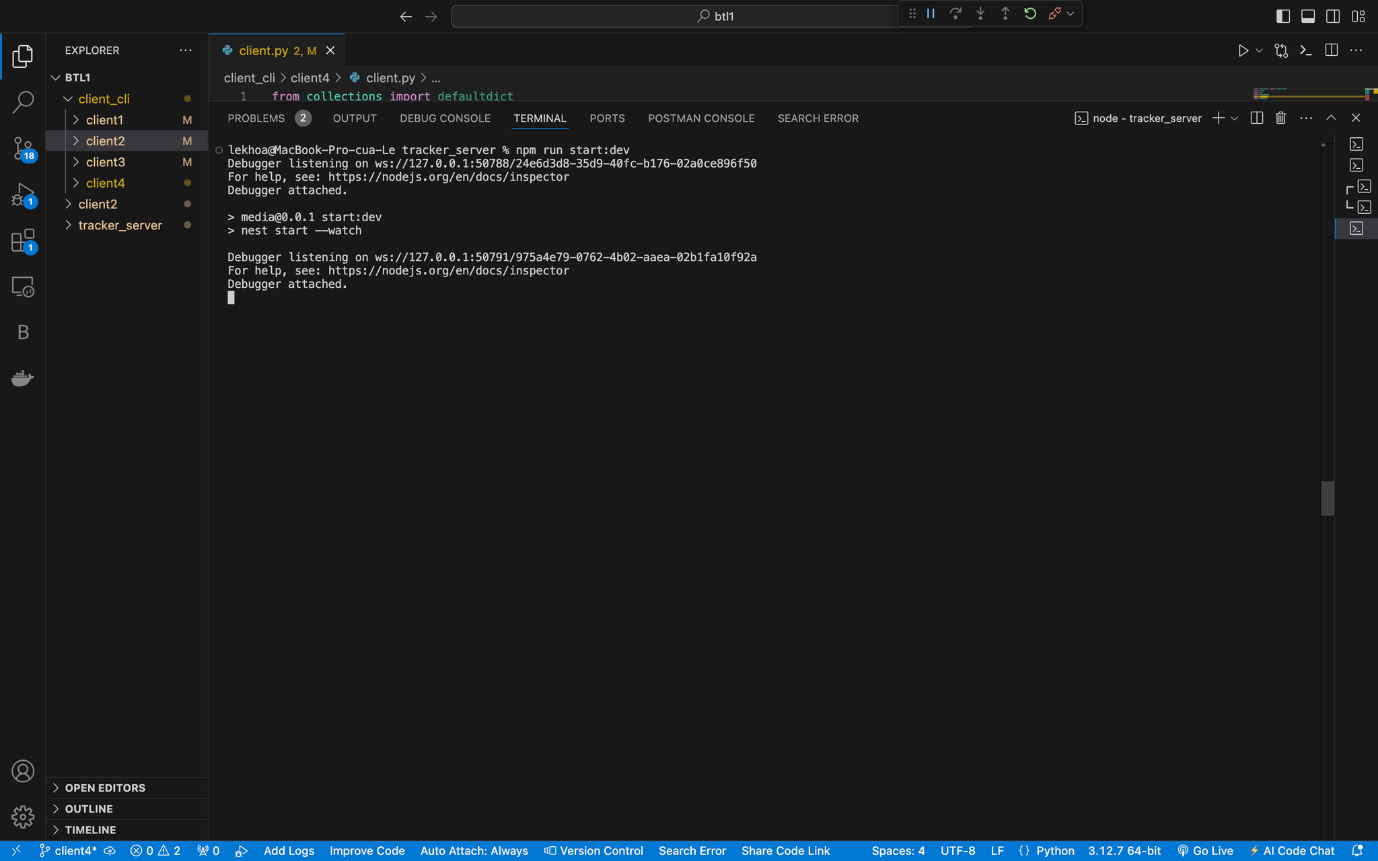
            response = {

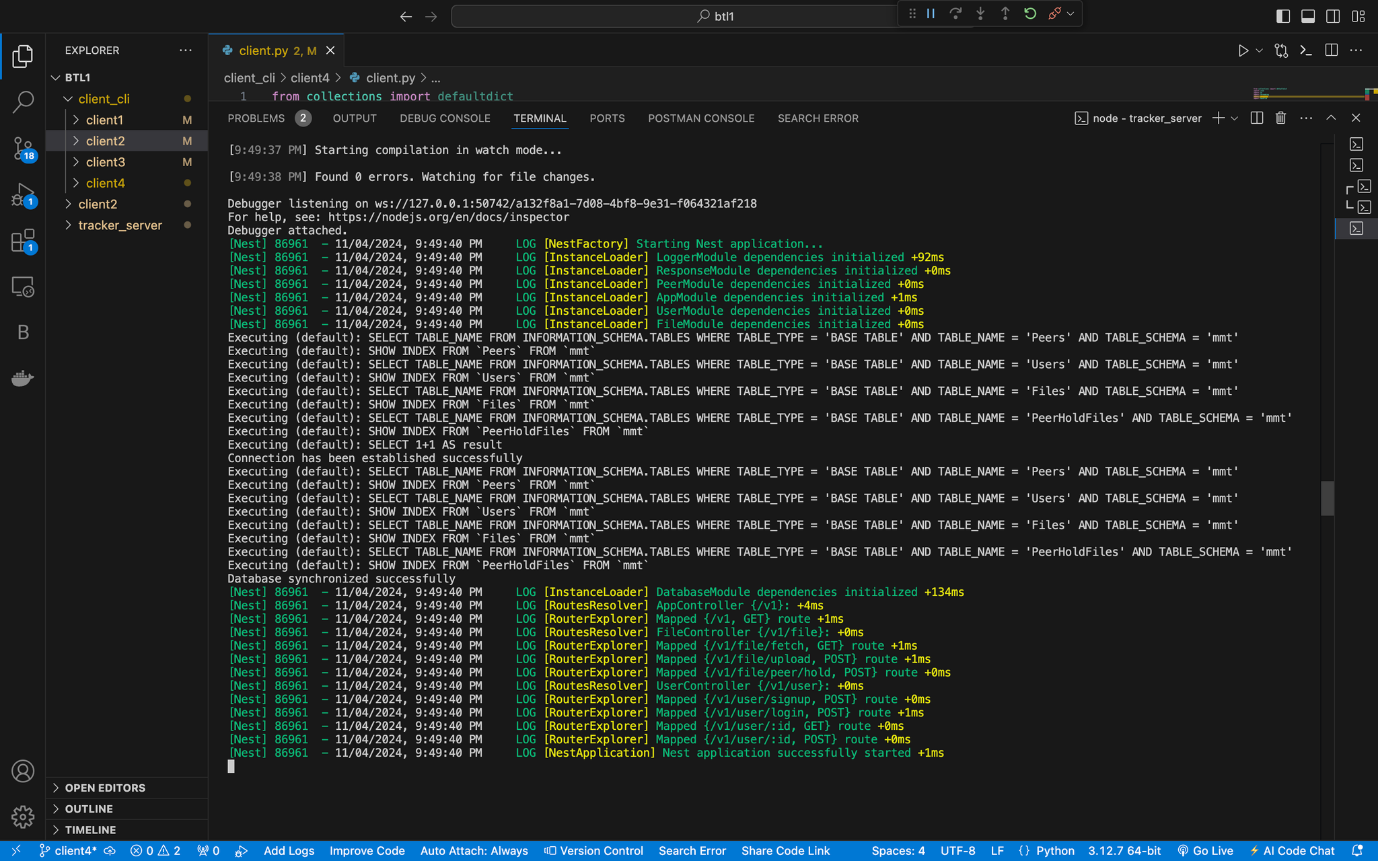
                'type': 'PONG'

            }

            client\_socket.sendall(json.dumps(response).encode('utf-8'))

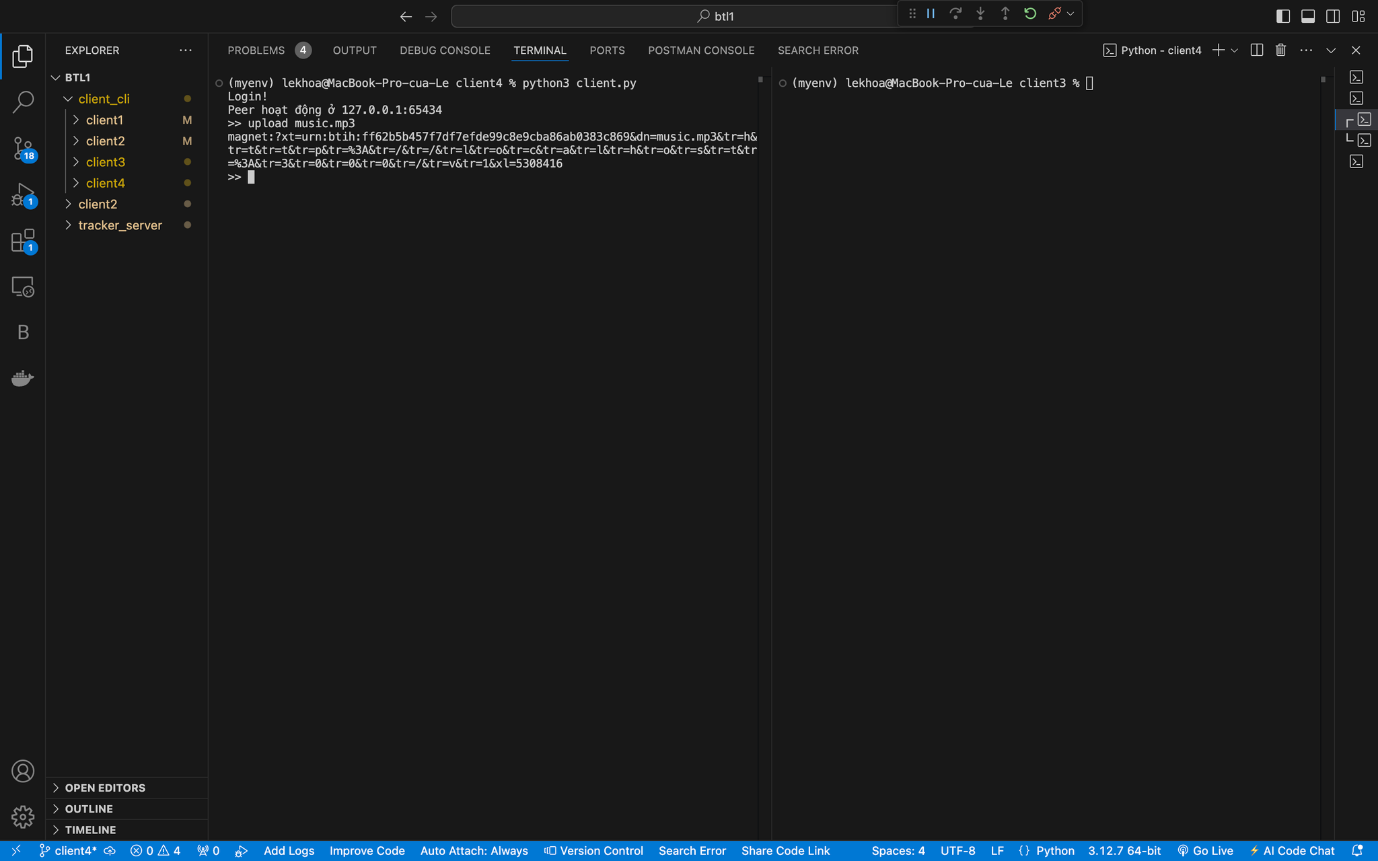
1. Hướng dẫn sử dụng
2. Khởi động server tracker



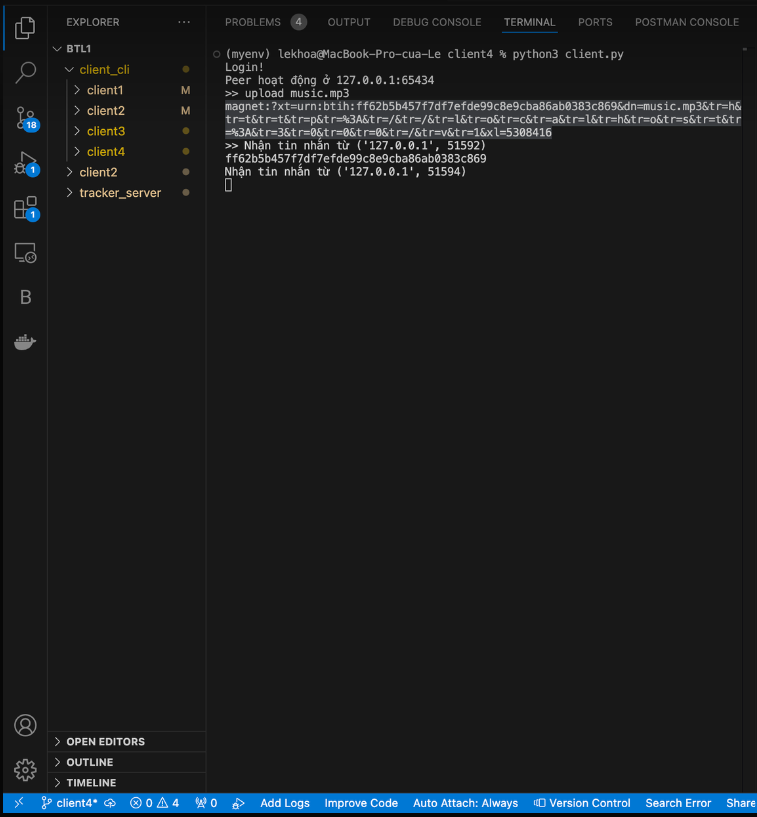


Khi terminal hiển thị như trên, server đã sẵn sàng phục vụ. Nếu server hiện lỗi chưa có package, sử dụng lệnh “npm i” để tải các packages được định nghĩa trong package.json.

1. Khởi động các client

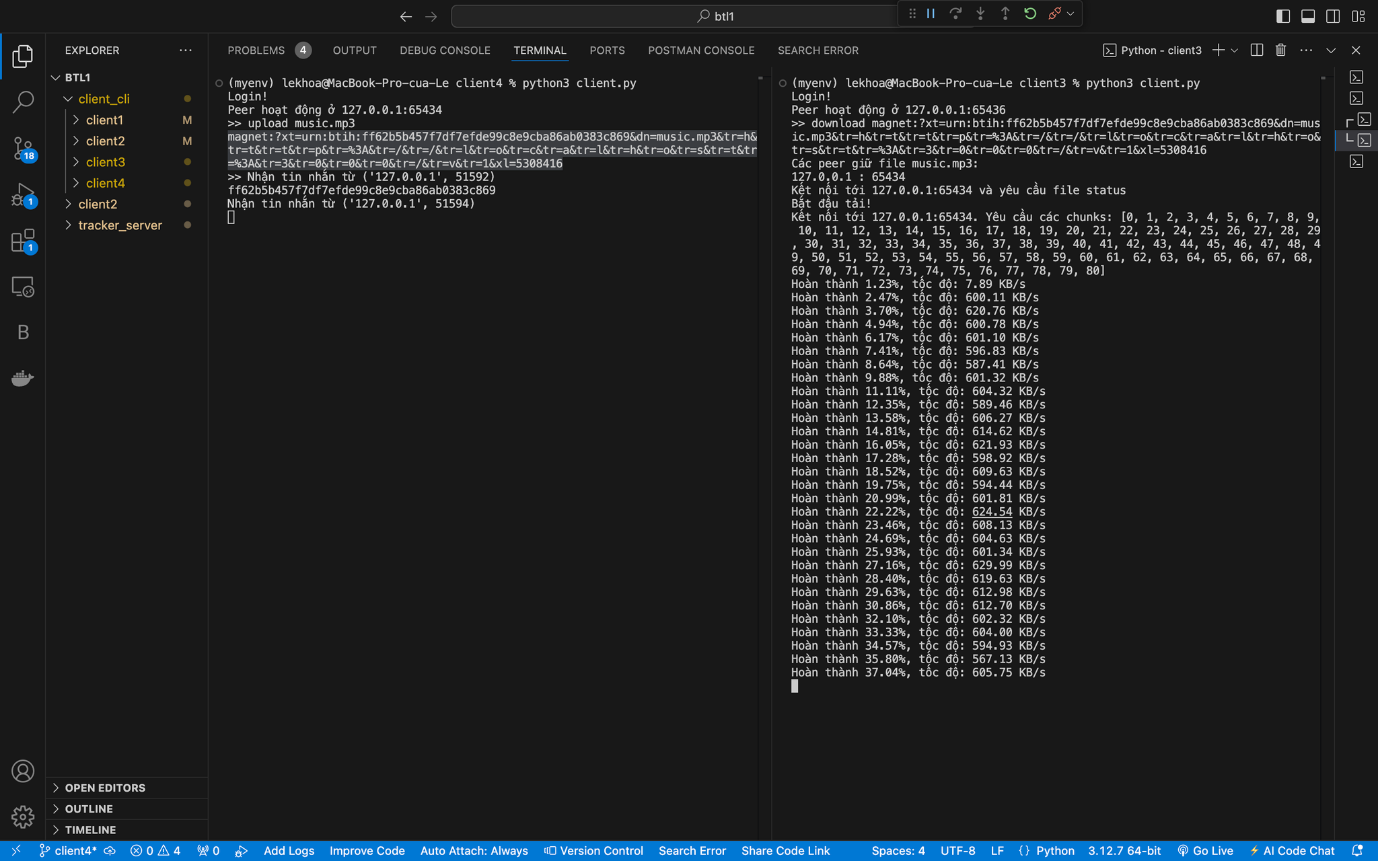


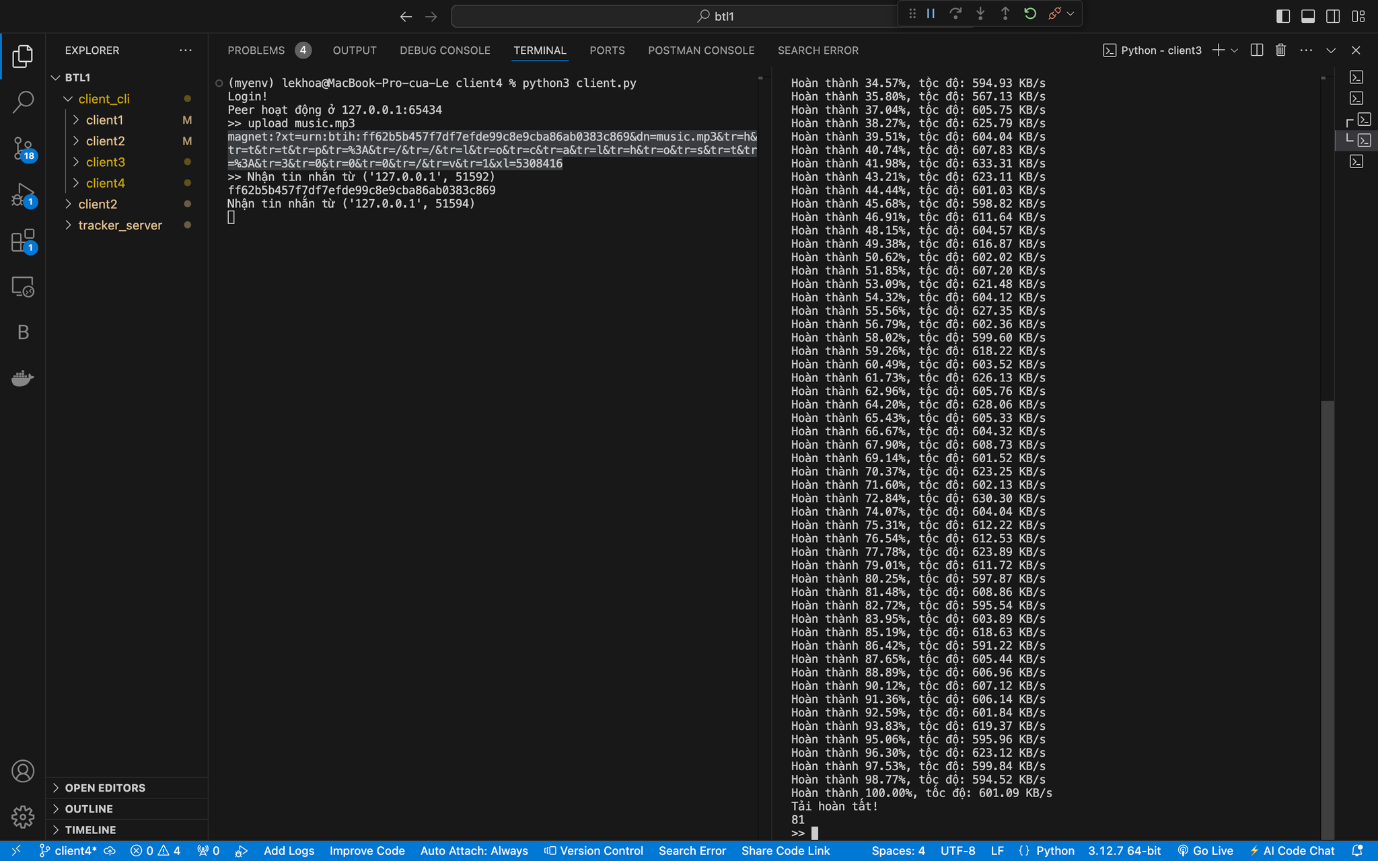
1. Tải file lần đầu cho một peer để đăng kí thông tin file cho tracker



Sau khi upload, chương trình trả về một magnet link để người dùng khác có thể sử dụng để download.

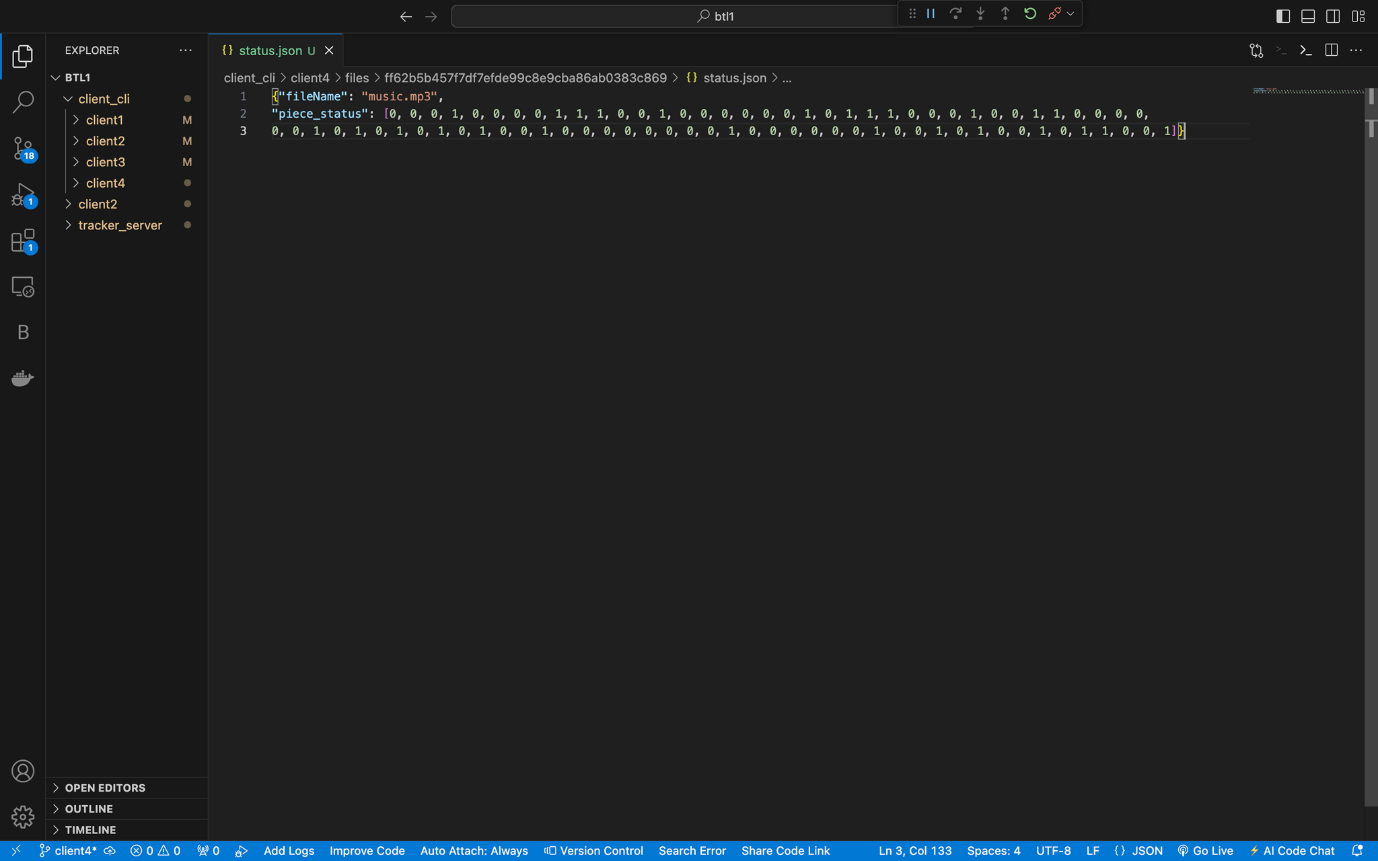
1. Download file



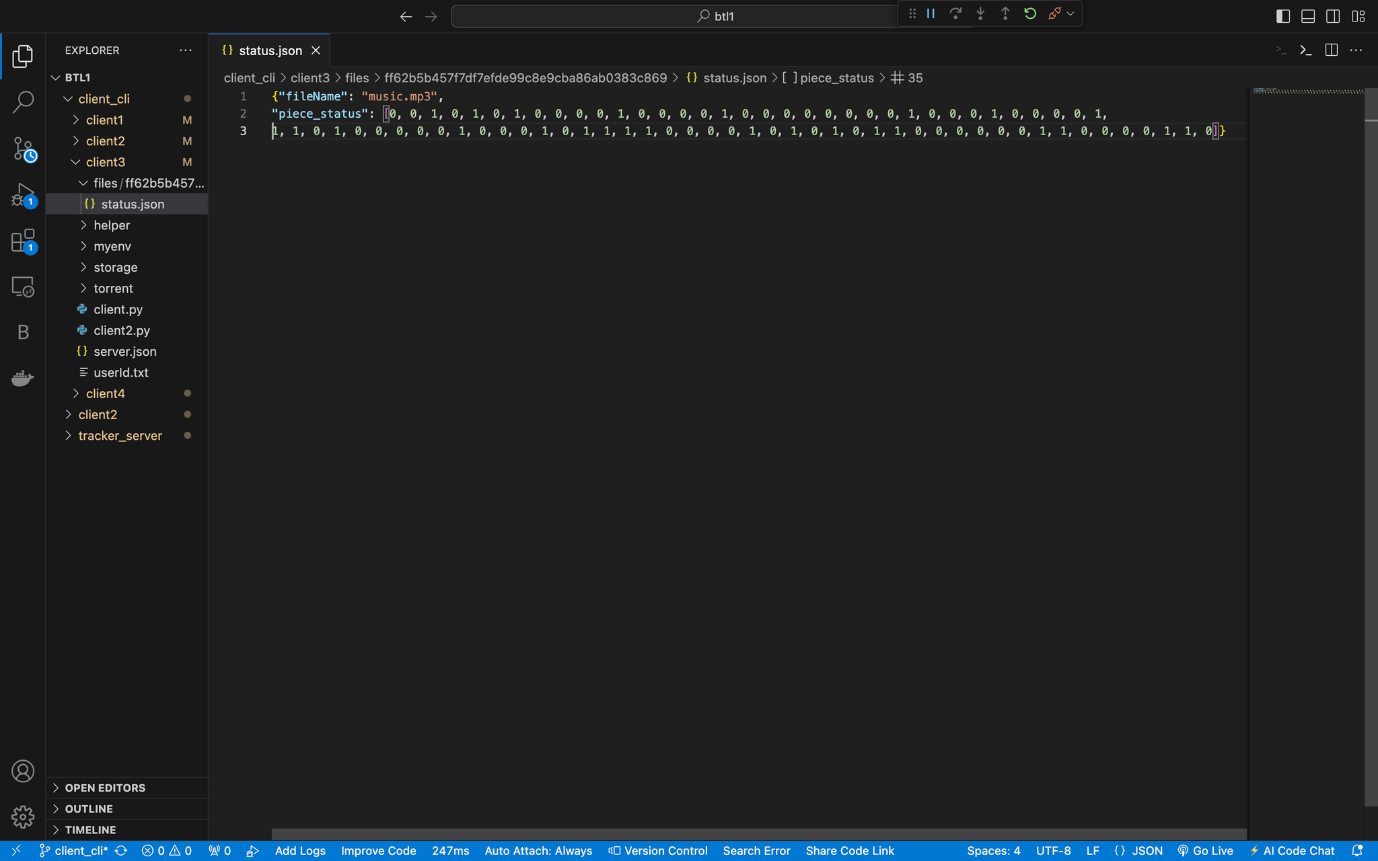


Nhập lệnh “download {magnet\_link}” chương trình sẽ tiến hành download và hiển thị tuần tự quá trình tải xuống cho người dùng.

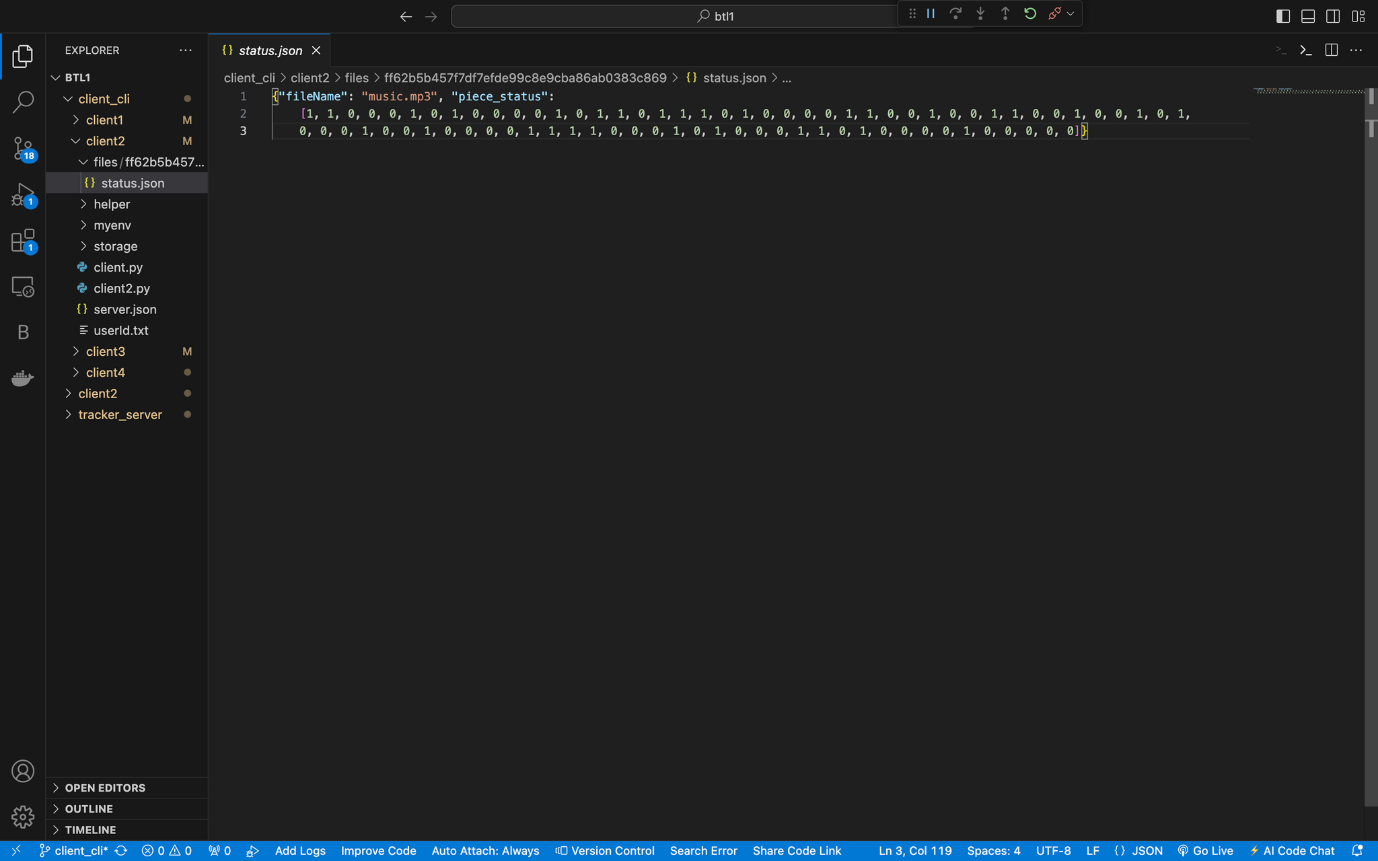
1. Tải file từ nhiều đồng thời từ nhiều peer



Trạng thái file piece của peer 1.

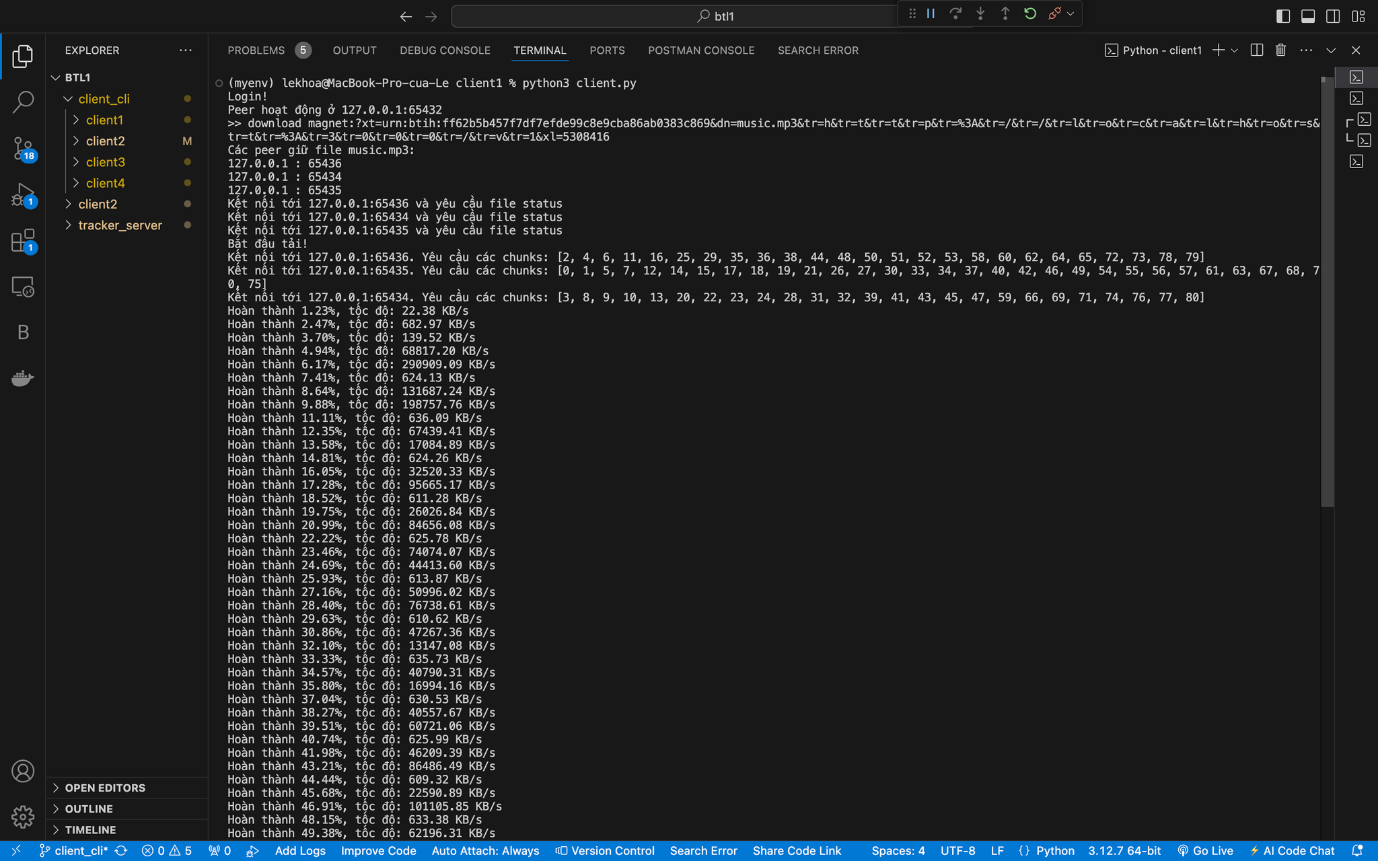


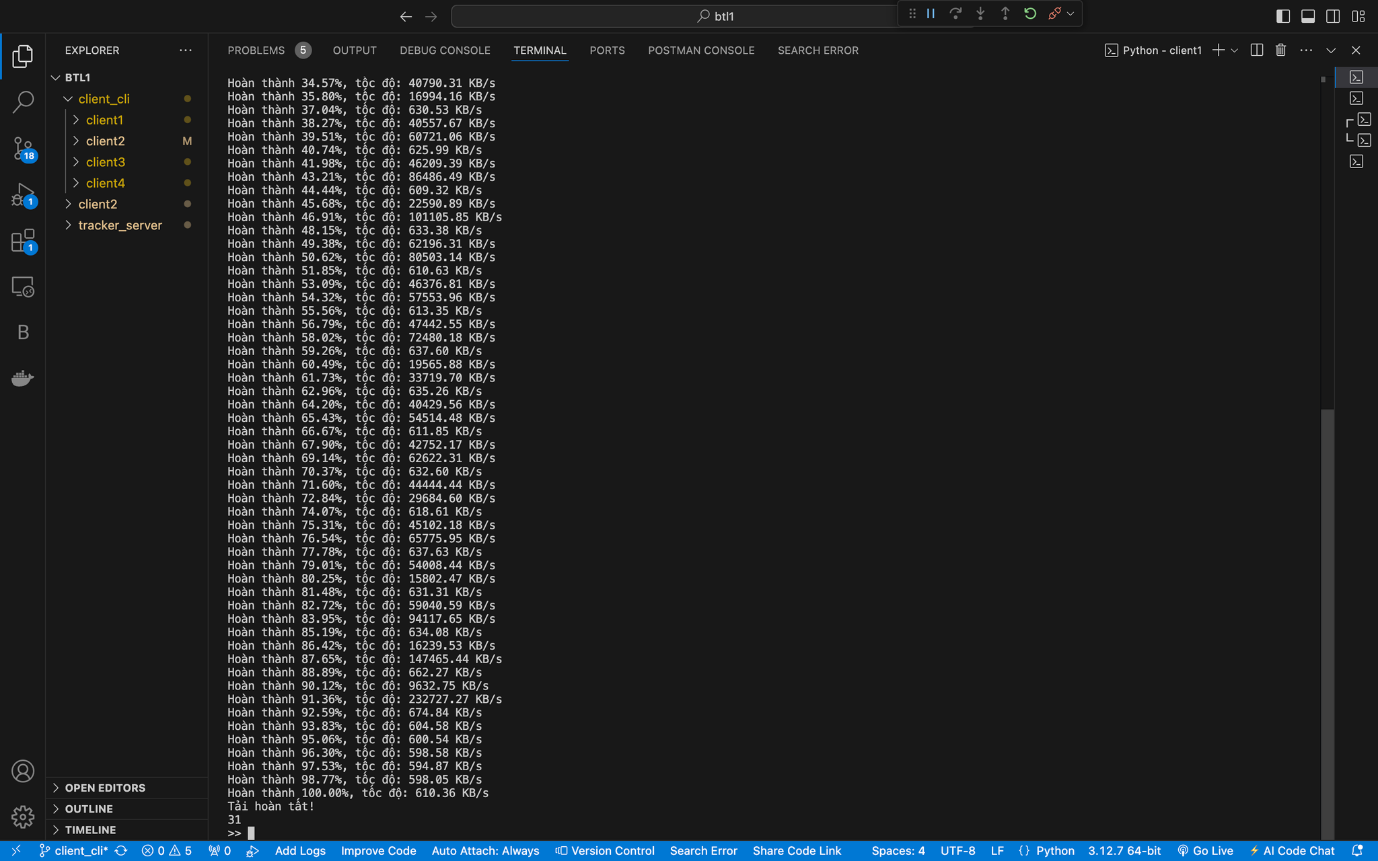
Trạng thái file piece của peer 2.



Trạng thái file piece của peer 3.

Peer 4 tiến hành download từ 3 peer trên:





Người dùng sau khi download, kiểm tra thư mục storage/{file\_name} để xem nội dung file.