

TRƯỜNG ĐẠI HỌC BÁCH KHOA
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



MẠNG MÁY TÍNH (CO3094)

BÀI TẬP LỚN

Implement HTTP server and chat application

Giáo viên hướng dẫn: Bùi Xuân Giang, CSE-HCMUT

Sinh viên: Nguyễn Tấn Phát - 2352888 (CN04)
Vũ Hà Như Ngọc - 2352818 (CN04)
Lê Diệu Quỳnh - 2353036 (CN04)
Lương Đức Huy - 2352384 (CN04)

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 11 NĂM 2025



Mục lục

Danh sách Ký hiệu	3
Danh sách Từ viết tắt	3
Danh sách Hình ảnh	5
Danh sách Bảng	5
Danh sách thành viên & khối lượng công việc	5
 Chương 1: Mở đầu	 6
1. Đặt vấn đề và mục tiêu	6
2. Phạm vi thực hiện	6
3. Bố cục báo cáo	7
 Chương 2: Cơ sở lý thuyết	 8
1. Mô hình giao tiếp mạng	8
2. Lập trình Socket và Luồng (Socket & Threading)	9
3. Giao thức HTTP và Cookie	10
4. Giao thức WebSocket	11
5. Quản lý phiên và đồng bộ hóa tin nhắn	12
 Chương 3: Phân tích và thiết kế hệ thống	 14
1. Kiến trúc tổng thể	14
2. Thiết kế các Module chính	15
3. Thiết kế Giao thức	17
 Chương 4: Hiện thực chương trình	 20
1. Môi trường và Công cụ	20
2. HTTP Server với Cookie Session	21
3. Ứng dụng Chat Hybrid (Client-Server kết hợp P2P)	22
4. “Put It All Together” – Tích hợp Hệ thống Hoàn chỉnh	25
 Chương 5: Kiểm thử và kết quả	 28
1. Kịch bản và Môi trường kiểm thử	28
2. Kết quả thực nghiệm	28



Chương 6: Kết luận	31
6.1. Tổng kết kết quả đạt được	31
6.2. Hạn chế của hệ thống	31
6.3. Hướng phát triển	31



Danh sách Ký hiệu

Danh sách Từ viết tắt



Danh sách Hình ảnh

Danh sách Bảng

1	Danh sách thành viên & khối lượng công việc	5
---	---	---



Danh sách thành viên & khối lượng công việc

STT	Họ Tên	MSSV	Vai trò	% Hoàn thành
1	Nguyễn Tấn Phát	2352888	Code	100%
2	Vũ Hà Như Ngọc	2352818	Code	100%
3	Lê Diệu Quỳnh	2353036	Code	100%
4	Lương Đức Huy	2352384	Code	100%

Bảng 1: Danh sách thành viên & khối lượng công việc

Chương 1: Mở đầu

1. Đặt vấn đề và mục tiêu

Trong bối cảnh Internet và các ứng dụng mạng ngày càng phát triển mạnh mẽ, việc hiểu rõ và áp dụng được các nguyên lý cơ bản của mạng máy tính là điều cần thiết đối với sinh viên ngành Khoa học máy tính. Các mô hình giao tiếp như **Client-Server** và **Peer-to-Peer (P2P)**, cùng với kỹ thuật lập trình socket, đóng vai trò nền tảng giúp kết nối các hệ thống và tạo nên các dịch vụ mạng hiện nay.

Bài tập lớn này được thực hiện nhằm áp dụng những kiến thức lý thuyết đã học trong môn *Mạng máy tính* vào việc xây dựng một hệ thống phần mềm thực tế, qua đó giúp người học củng cố và hiểu sâu hơn cách các tiến trình mạng tương tác với nhau trong môi trường phân tán.

Cụ thể, đề tài tập trung vào hai nhiệm vụ chính sau:

- **Xây dựng HTTP Server có cơ chế quản lý phiên bằng Cookie:** Nhiệm vụ này yêu cầu sinh viên hiện thực một máy chủ web sử dụng thư viện *socket* cơ bản của Python, có khả năng xử lý các yêu cầu HTTP (GET/POST) và đặc biệt là triển khai cơ chế xác thực người dùng. Hệ thống cần duy trì trạng thái đăng nhập thông qua việc sử dụng *HTTP Cookie*, giúp làm rõ cách thức vượt qua tính chất *stateless* của giao thức HTTP.
- **Xây dựng ứng dụng chat lai (Hybrid Chat Application):** Ứng dụng này kết hợp linh hoạt hai mô hình Client-Server và P2P. Trong đó, mô hình *Client-Server* được dùng để quản lý danh sách người dùng và kênh chat thông qua một *Tracker Server*, còn mô hình *P2P* cho phép các người dùng (Peer Client) kết nối trực tiếp và trao đổi tin nhắn mà không cần qua máy chủ trung gian. Mục tiêu là giúp sinh viên hiểu rõ cách thiết kế một hệ thống giao tiếp hai chiều có tính mở rộng cao và phản hồi thời gian thực.

Thông qua việc hoàn thành hai nhiệm vụ trên, đề tài hướng đến mục tiêu xây dựng một ứng dụng mạng hoàn chỉnh, minh họa rõ cách phối hợp giữa các tiến trình *client*, *server* và *tracker* trong một kiến trúc hybrid hiện đại.

2. Phạm vi thực hiện

Để đáp ứng các mục tiêu đã đề ra, dự án tập trung thiết kế và hiện thực một hệ thống hoàn chỉnh với các chức năng cốt lõi sau:

- **HTTP Server và Proxy cơ bản:** Xây dựng thành công máy chủ HTTP từ thư viện *socket*, có khả năng xử lý yêu cầu GET và POST. Hệ thống cũng bao gồm Proxy Server đơn giản để chuyển tiếp request HTTP, minh họa cho kiến trúc nhiều lớp trong ứng dụng web thực tế.
- **Cơ chế xác thực và quản lý phiên:** Hiện thực quy trình xác thực người dùng. Sau khi đăng nhập thành công, máy chủ thiết lập HTTP Cookie trên trình duyệt để duy trì trạng thái phiên – một yếu tố cơ bản trong các ứng dụng web hiện đại.
- **Kiến trúc Ứng dụng Chat Hybrid:** Gồm hai thành phần chính:
 - *Tracker Server*: Đóng vai trò máy chủ trung tâm, quản lý việc đăng ký, xác thực và khám phá người dùng.
 - *Peer Client*: Vừa là client của Tracker Server, vừa là server trong các kết nối P2P.
- **Giao thức Client-Server:** Cho phép Peer Client gửi thông tin (địa chỉ IP, port) lên Tracker Server để đăng ký, đồng thời truy vấn danh sách các peer khác đang trực tuyến.
- **Giao thức Peer-to-Peer để giao tiếp trực tiếp:** Hai Peer Client có thể thiết lập kết nối TCP trực tiếp sau khi biết địa chỉ của nhau. Cơ chế “bắt tay” (handshake) được triển khai để xác nhận kết nối trước khi trao đổi dữ liệu.

- **Hỗ trợ đa dạng hình thức Chat:**
 - Chat riêng giữa hai peer.
 - Chat nhóm qua các kênh chung (`#general`, `#test`, ...).
 - Kênh phát sóng (`#broadcast`) gửi tin đến toàn bộ người dùng trực tuyến.
- **Tích hợp giao diện Web:** Xây dựng giao diện người dùng (User Interface - UI) bằng HTML, CSS và JavaScript, mang lại trải nghiệm chat trực quan và thân thiện.
- **Giao tiếp thời gian thực qua WebSocket:** Sử dụng giao thức WebSocket để kết nối hai chiều giữa backend Python và giao diện người dùng trên trình duyệt, đảm bảo phản hồi tức thì khi có tin nhắn mới.
- **Duy trì trạng thái đăng nhập phía Client:** Lưu thông tin phiên đăng nhập, giúp người dùng tự động kết nối lại mà không cần thao tác thủ công, nâng cao trải nghiệm sử dụng.

3. Bố cục báo cáo

Để trình bày một cách hệ thống và rõ ràng quá trình thực hiện đề tài, nội dung báo cáo được cấu trúc thành các chương sau:

- **Chương 1 – Mở đầu:** Giới thiệu tổng quan về đề tài, xác định các mục tiêu, phạm vi chức năng đã được hiện thực và trình bày bố cục tổng thể của báo cáo.
- **Chương 2 – Cơ sở lý thuyết:** Trình bày các kiến thức nền tảng và công nghệ cốt lõi được sử dụng trong dự án, bao gồm các mô hình mạng Client-Server và Peer-to-Peer, lập trình Socket đa luồng, giao thức HTTP, cơ chế hoạt động của Cookie và vai trò của giao thức WebSocket.
- **Chương 3 – Phân tích và Thiết kế hệ thống:** Phân tích kiến trúc tổng thể của ứng dụng, mô tả chi tiết thiết kế của các thành phần chính như Tracker Server và Peer Client, cùng các giao thức giao tiếp giữa chúng.
- **Chương 4 – Hiện thực chương trình:** Trình bày quá trình hiện thực hệ thống, mô tả các đoạn mã quan trọng và giải thích cách triển khai các chức năng như xử lý đa luồng, kết nối P2P, và truyền thông real-time giữa backend và frontend.
- **Chương 5 – Kiểm thử và Kết quả:** Giới thiệu các kịch bản kiểm thử, phân tích kết quả thực nghiệm thông qua hình ảnh giao diện và log hệ thống, minh chứng cho tính chính xác và ổn định của chương trình.
- **Chương 6 – Kết luận:** Tổng kết những kết quả đạt được, đối chiếu với mục tiêu ban đầu và đề xuất hướng phát triển trong tương lai.

Chương 2: Cơ sở lý thuyết

Chương này trình bày các khái niệm và công nghệ nền tảng được sử dụng làm cơ sở để thiết kế và hiện thực hệ thống ứng dụng chat.

1. Mô hình giao tiếp mạng

Trong kiến trúc ứng dụng mạng, có hai mô hình giao tiếp chính là **Client-Server** và **Peer-to-Peer (P2P)**. Hệ thống được xây dựng trong đề tài này là một ứng dụng lai (hybrid) kết hợp sức mạnh của cả hai mô hình.

Mô hình Client-Server

Mô hình Client-Server là kiến trúc phổ biến nhất trong các ứng dụng mạng, được đặc trưng bởi sự phân chia vai trò rõ rệt giữa các thành phần tham gia:

- **Server (Máy chủ):** Là tiến trình luôn hoạt động, có địa chỉ cố định (IP và port), lắng nghe các yêu cầu từ bên ngoài và cung cấp tài nguyên hoặc dịch vụ cho client.
- **Client (Máy khách):** Là tiến trình chủ động kết nối đến server để yêu cầu tài nguyên hoặc dịch vụ. Client không giao tiếp trực tiếp với các client khác.

Áp dụng trong dự án: Giai đoạn khởi tạo và khám phá của ứng dụng chat được xây dựng dựa trên mô hình Client-Server. Trong đó:

- **Tracker Server** đóng vai trò là Server. Nó có địa chỉ cố định, luôn trực tuyến và cung cấp dịch vụ quản lý, đăng ký và cung cấp danh sách người dùng.
- **Peer Client** đóng vai trò là Client. Mỗi khi khởi động, nó chủ động kết nối đến Tracker Server để đăng ký thông tin và lấy danh sách các peer khác đang hoạt động.

Mô hình Peer-to-Peer (P2P)

Trái ngược với mô hình Client-Server, mô hình P2P không có một máy chủ trung tâm cố định. Thay vào đó, các máy tính trong mạng (peer) có vai trò tương đương nhau. Mỗi peer vừa có thể là client (khởi tạo kết nối) vừa có thể là server (chấp nhận kết nối).

Áp dụng trong dự án: Giai đoạn giao tiếp chính của ứng dụng chat dựa trên mô hình P2P. Cụ thể, sau khi nhận thông tin địa chỉ từ Tracker Server, các Peer Client sẽ:

- Mở một cổng (port) để lắng nghe, hoạt động như server đối với các peer khác.
- Chủ động sử dụng địa chỉ IP và port của peer khác để thiết lập kết nối TCP trực tiếp, hoạt động như client.
- Sau khi kết nối được thiết lập, trao đổi tin nhắn trực tiếp mà không cần sự can thiệp của Tracker Server.

Mô hình lai (Hybrid)

Mô hình lai kết hợp các ưu điểm của cả Client-Server và P2P. Nó sử dụng máy chủ trung tâm cho các tác vụ đòi hỏi sự tin cậy và quản lý tập trung, đồng thời tận dụng khả năng kết nối trực tiếp của P2P cho các tác vụ cần hiệu suất cao và khả năng mở rộng.

Áp dụng trong dự án: Ứng dụng chat là hệ thống hybrid điển hình:

- **Giai đoạn khám phá (discovery):** Sử dụng mô hình Client-Server. Tracker Server giúp peer mới tìm thấy các peer khác trong mạng.
- **Giai đoạn giao tiếp (communication):** Sử dụng mô hình P2P. Trao đổi tin nhắn trực tiếp giúp giảm độ trễ, tăng tính riêng tư và giảm tải cho Tracker Server, nâng cao khả năng mở rộng.

2. Lập trình Socket và Luồng (Socket & Threading)

Để hiện thực các mô hình giao tiếp mạng đã nêu, dự án sử dụng hai công nghệ nền tảng: **Socket API** để tạo kênh truyền và **đa luồng (Multithreading)** để quản lý các kết nối đồng thời.

Socket là một khái niệm trừu tượng của hệ điều hành, đại diện cho một *điểm cuối* trong kênh giao tiếp hai chiều qua mạng. Lập trình viên có thể thao tác với Socket mà không cần quan tâm chi tiết phức tạp của tầng giao vận (Transport Layer) bên dưới. Trong Python, thư viện `socket` cung cấp giao diện cấp cao để làm việc với các cơ chế mạng này.

Trong dự án, Socket API được sử dụng để thiết lập kết nối dựa trên giao thức TCP (Transmission Control Protocol), đảm bảo dữ liệu truyền đi đáng tin cậy và đúng thứ tự. Một số thao tác chính được sử dụng gồm:

Phía Server (Lắng nghe)

- `socket.socket()`: Tạo một đối tượng socket mới.
- `socket.bind((ip, port))`: Gán socket với địa chỉ IP và port cụ thể trên máy chủ.
- `socket.listen()`: Đặt socket vào trạng thái lắng nghe, sẵn sàng chấp nhận các kết nối đến.
- `socket.accept()`: Hàm khóa (blocking), tạm dừng chương trình cho đến khi có client kết nối. Trả về một socket mới đại diện cho kết nối và địa chỉ của client.

Phía Client (Kết nối)

- `socket.socket()`: Tạo một đối tượng socket mới.
- `socket.connect((server_ip, server_port))`: Chủ động kết nối tới server.

Trao đổi dữ liệu (Cả hai phía)

- `socket.sendall(data)`: Gửi toàn bộ dữ liệu (dạng bytes) qua kết nối.
- `socket.recv(buffer_size)`: Hàm khóa (blocking), nhận dữ liệu từ phía bên kia.

Một vấn đề cơ bản với server đơn luồng là không thể phục vụ nhiều client cùng lúc. Ví dụ, khi server đang chờ client A kết nối, các client khác phải chờ; nếu server xử lý yêu cầu của A lâu, client B, C... sẽ bị trì hoãn, ảnh hưởng trải nghiệm người dùng.

Đa luồng (Threading)

Để khắc phục, dự án sử dụng thư viện `threading` trong Python:

- Mỗi kết nối mới được xử lý trong một **luồng riêng** thay vì luồng chính.
- Luồng chính luôn sẵn sàng để tiếp nhận các kết nối mới.
- Giúp server phục vụ nhiều client đồng thời mà không bị treo.
- Tận dụng tối đa tài nguyên hệ thống, có thể xử lý hàng chục hoặc hàng trăm kết nối cùng lúc.

Áp dụng trong dự án: Cơ chế này được sử dụng trong cả:

- Tracker Server: xử lý nhiều peer cùng lúc.
- P2P Server bên trong mỗi Peer Client: chấp nhận kết nối từ nhiều peer khác.

3. Giao thức HTTP và Cookie

HTTP (HyperText Transfer Protocol) là giao thức nền tảng của World Wide Web, hoạt động dựa trên mô hình Client-Server. Trong dự án, giao thức này được sử dụng cho việc giao tiếp giữa Peer Client và Tracker Server. Thay vì sử dụng các thư viện HTTP cấp cao, dự án hiện thực việc tạo và phân tích các thông điệp HTTP một cách thủ công qua socket để hiểu rõ bản chất của giao thức.

Một phiên giao tiếp HTTP gồm một **request** từ client và một **response** từ server. Cả hai đều là các đoạn văn bản thô (plain text) tuân thủ định dạng nghiêm ngặt.

HTTP Request

Một request từ client gửi lên server gồm ba phần chính:

- **Start Line (Dòng bắt đầu):** Chứa phương thức (GET, POST), đường dẫn (Path) và phiên bản HTTP.
- **Headers (Các dòng tiêu đề):** Cung cấp thông tin bổ sung, dưới dạng Key: Value. Header quan trọng nhất là **Host**, cho biết server nào đang được yêu cầu. Các header khác như **Content-Type** và **Content-Length** được dùng trong request POST để mô tả dữ liệu gửi đi.
- **Body (Phần thân):** Chứa dữ liệu gửi kèm, ví dụ thông tin đăng nhập. Chỉ tồn tại với các request như POST.

Ví dụ: Một HTTP Request được tạo thủ công trong `peer_client.py` để đăng ký với tracker:

```
POST /submit-info HTTP/1.1
Host: 127.0.0.1:8001
Content-Type: application/json
Content-Length: 68

{"username": "alice", "ip": "192.168.1.6", "port": 9101}
```

HTTP Response

Một response từ server trả về cũng có cấu trúc tương tự:

- **Status Line (Dòng trạng thái):** Phiên bản HTTP, mã trạng thái (200 OK, 404 Not Found, ...), và thông điệp trạng thái.
- **Headers:** Ví dụ **Content-Type** cho biết loại dữ liệu trả về, **Content-Length** cho biết kích thước phần thân.
- **Body:** Chứa nội dung thực sự của tài nguyên, ví dụ code HTML hoặc dữ liệu JSON.

Cookie

Một đặc tính cơ bản của HTTP là nó **vô trạng thái**, tức server không lưu thông tin về các request trước đó. Mỗi request được xử lý độc lập. Điều này gây khó khăn khi xây dựng các ứng dụng yêu cầu duy trì phiên đăng nhập.

Để khắc phục, cơ chế **Cookie** được sử dụng. Cookie là mẩu thông tin nhỏ mà server gửi đến trình duyệt và yêu cầu lưu lại. Với mỗi request tiếp theo, trình duyệt sẽ tự động gửi lại cookie đến server.

Vai trò trong dự án: Cookie được dùng để minh họa khả năng duy trì trạng thái đăng nhập:

- **Thiết lập Cookie (Server gửi):** Sau khi người dùng đăng nhập thành công, server trả về response chứa header **Set-Cookie**, ví dụ:

```
Set-Cookie: session_token=session_for_alice; Path=/; HttpOnly
```

Trình duyệt sẽ lưu cookie này.

- **Gửi lại Cookie (Client gửi):** Với mỗi request tiếp theo, trình duyệt tự động đính kèm header:

```
Cookie: session_token=session_for_alice
```

Server đọc cookie này để xác định người dùng đã đăng nhập hay chưa, quyết định có cho phép truy cập tài nguyên bảo vệ hay không.

Như vậy, Cookie đóng vai trò như "thẻ căn cước", giúp server nhận ra người dùng qua nhiều request khác nhau, tạo ra một phiên làm việc có trạng thái trên một giao thức vốn vô trạng thái.

4. Giao thức WebSocket

Trong khi HTTP là giao thức lý tưởng cho mô hình yêu cầu-phản hồi (request-response) truyền thống của web, nó bộc lộ hạn chế đối với các ứng dụng đòi hỏi tính tương tác cao và thời gian thực như ứng dụng chat. Để khắc phục, dự án đã tích hợp giao thức WebSocket làm kênh giao tiếp chính giữa backend logic (`peer_client.py`) và giao diện người dùng trên trình duyệt (`chat.html`).

Mô hình HTTP hoạt động theo dạng *half-duplex*, do client khởi tạo. Client gửi request, server trả về response, rồi kết nối có thể bị đóng. Trong ứng dụng chat, điều này dẫn đến hai vấn đề chính:

- **Độ trễ cao:** Client phải liên tục gửi request "hỏi thăm" server để kiểm tra tin nhắn mới (Polling), tạo ra nhiều request thừa, tốn tài nguyên và tăng độ trễ.
- **Server không thể chủ động gửi tin:** Server không thể đẩy dữ liệu xuống client, luôn phải chờ client gửi request.

WebSocket giải quyết vấn đề trên bằng cách tạo kênh giao tiếp *full-duplex* và bền vững (persistent) giữa client và server:

- **Giao tiếp hai chiều:** Cả client và server có thể gửi dữ liệu bất cứ lúc nào mà không cần request mới.
- **Thời gian thực:** Dữ liệu được gửi gần như ngay lập tức, loại bỏ độ trễ của việc tạo kết nối HTTP mới.
- **Hiệu quả:** Giảm overhead so với HTTP, sau một lần handshake ban đầu, các frame dữ liệu rất nhỏ gọn.

Áp dụng trong dự án

WebSocket là cầu nối giữa "bộ não" P2P (`peer_client.py`) và "giao diện điều khiển" (`chat.html`). Khi một peer nhận tin nhắn P2P từ peer khác, nó có thể ngay lập tức đẩy tin nhắn lên giao diện trình duyệt thông qua kết nối WebSocket, giúp người dùng thấy tin nhắn gần như tức thời.

So sánh HTTP và WebSocket

Tiêu chí	HTTP	WebSocket
Mô hình	Yêu cầu - Phản hồi (Request - Response)	Song công toàn phần (Full-Duplex)
Khởi tạo	Luôn do client khởi tạo	Cả client và server đều có thể gửi dữ liệu sau khi kết nối
Kết nối	Không bền vững (có thể tạo kết nối mới cho mỗi request)	Bền vững (một kết nối duy nhất suốt phiên)
Độ trễ	Cao (do overhead cho mỗi request)	Rất thấp (sau handshake ban đầu)
Phù hợp cho	Tải tài nguyên, gọi API RESTful	Ứng dụng thời gian thực (chat, game online, bảng giá)

Luồng thiết lập kết nối WebSocket

Quá trình bắt đầu bằng một request HTTP từ client chứa header **Upgrade: websocket**. Nếu server hỗ trợ, nó trả về response HTTP với mã trạng thái 101 **Switching Protocols**. Kể từ đó, kết nối TCP ban đầu được nâng cấp thành kết nối WebSocket và cả hai bên bắt đầu trao đổi dữ liệu theo giao thức mới. Thư viện **websockets** trong Python và đối tượng WebSocket trong JavaScript tự động xử lý quá trình này.

5. Quản lý phiên và đồng bộ hóa tin nhắn

Một ứng dụng chat hiện đại không chỉ yêu cầu giao tiếp thời gian thực mà còn phải đảm bảo rằng mỗi người dùng được nhận dạng đúng và tin nhắn được đồng bộ chính xác giữa các peer và giao diện người dùng. Trong dự án, hai cơ chế chính được áp dụng để giải quyết vấn đề này: quản lý phiên (session management) và đồng bộ hóa tin nhắn.

Quản lý phiên (Session Management)

Vì HTTP vốn vô trạng thái, việc xác định ai đang đăng nhập và có quyền truy cập các tài nguyên bảo vệ là rất quan trọng. Dự án sử dụng kết hợp Cookie và localStorage trên trình duyệt để quản lý phiên làm việc của người dùng:

- **Cookie trên HTTP:** Sau khi người dùng đăng nhập thành công, server tạo một cookie chứa thông tin phiên, ví dụ `session_token`, gửi về trình duyệt. Mỗi request tiếp theo từ trình duyệt sẽ kèm theo cookie này, giúp server nhận biết người dùng.
- **LocalStorage trên trình duyệt:** Khi kết nối WebSocket được thiết lập, thông tin phiên cũng được lưu trong localStorage. Khi người dùng tải lại trang, client tự động gửi lại thông tin này để phục hồi kết nối, tránh phải đăng nhập lại thủ công.

Kết hợp hai cơ chế trên, ứng dụng có thể:

- Nhận biết và xác thực người dùng qua nhiều request HTTP.
- Duy trì kết nối thời gian thực ngay cả khi trang web được tải lại.

Đồng bộ hóa tin nhắn

Trong môi trường P2P, các peer trao đổi trực tiếp, vì vậy việc đồng bộ hóa tin nhắn là cần thiết để đảm bảo rằng:

- Tin nhắn gửi đi được hiển thị đầy đủ trên cả người gửi và người nhận.
- Các kênh chat nhóm (`#general`, `#broadcast`,...) luôn đồng bộ với tất cả các thành viên.

Cơ chế đồng bộ hóa trong dự án:

- Mỗi peer lưu một danh sách tin nhắn cục bộ cho từng kênh hoặc chat riêng.

- Khi nhận tin nhắn mới từ một peer khác, peer sẽ cập nhật danh sách tin nhắn cục bộ và đồng thời đẩy tin nhắn này lên giao diện trình duyệt qua WebSocket.
- Trong trường hợp chat nhóm hoặc broadcast, peer gửi tin nhắn đến tất cả peer trong kênh, đảm bảo mọi người đều nhận được nội dung gần như cùng lúc.

Nhờ cơ chế quản lý phiên kết hợp với đồng bộ hóa tin nhắn, ứng dụng chat đạt được các tiêu chí:

- Tin nhắn được gửi và nhận chính xác, đầy đủ.
- Người dùng không phải đăng nhập lại sau khi reload trang.
- Trải nghiệm thời gian thực gần như tức thì, hạn chế độ trễ và đảm bảo tính riêng tư.

Như vậy, dự án đã thiết lập được nền tảng lý thuyết vững chắc cho việc hiện thực hệ thống chat hybrid với khả năng quản lý phiên, đồng bộ hóa tin nhắn, và giao tiếp thời gian thực.

Chương 3: Phân tích và thiết kế hệ thống

1. Kiến trúc tổng thể

Hệ thống được thiết kế theo một kiến trúc đa thành phần, phân tách rõ ràng các vai trò và trách nhiệm nhằm đảm bảo tính module, dễ bảo trì và khả năng mở rộng. Sơ đồ dưới đây minh họa mối quan hệ và luồng tương tác giữa các thành phần chính của hệ thống.

Kiến trúc này gồm ba thành phần chính, mỗi thành phần có vai trò riêng nhưng phối hợp chặt chẽ với nhau:

Tracker Server

Đây là thành phần trung tâm, hoạt động như một trung tâm điều phối trong giai đoạn Client-Server. Nó là một tiến trình độc lập, lắng nghe trên một port cố định và có các vai trò chính sau:

- **Quản lý định danh và xác thực:** Tracker Server duy trì cơ sở dữ liệu người dùng đơn giản (`users_credentials`) và cung cấp API `/login` để xác thực danh tính của các peer khi tham gia mạng.
- **Quản lý trạng thái peer:** Tracker giữ danh sách động (`peers_lock`) của tất cả peer đang trực tuyến, bao gồm username, địa chỉ IP và port P2P. Việc đăng ký và cập nhật thông tin được thực hiện qua API `/submit-info`.
- **Quản lý kênh chat:** Tracker Server quản lý danh sách các kênh và thành viên thông qua API `/add-list`.
- **Hỗ trợ khám phá peer (Peer Discovery):** API `/get-list` cho phép peer truy vấn và lấy danh sách các peer khác, làm tiền đề cho việc thiết lập kết nối P2P.

Peer Client

Đây là thành phần cốt lõi và phức tạp nhất, được thiết kế như một tiến trình daemon chạy trên máy của người dùng. Mỗi peer client thực hiện đồng thời ba vai trò:

- **Vai trò Client (với Tracker):** Chủ động kết nối TCP đến Tracker Server để xác thực, đăng ký thông tin và tham gia kênh.
- **Vai trò Server P2P:** Mở một socket TCP trên port do người dùng chỉ định (`peer_port`) và lắng nghe các kết nối từ các peer khác.
- **Vai trò Server giao diện (UI Server):** Chạy trên một luồng riêng, gồm:
 - **HTTP Server:** Chạy trên port `http_port = peer_port + 10000`, phục vụ file `chat.html` và tài nguyên tĩnh (CSS, hình ảnh), đồng thời cung cấp endpoint `/confirm-login` để thiết lập cookie.
 - **WebSocket Server:** Chạy trên port `websocket_port = peer_port + 20000`, mở kênh giao tiếp hai chiều thời gian thực với giao diện web, nhận các lệnh từ người dùng và đẩy cập nhật tin nhắn lên trình duyệt.

Giao diện người dùng (Web UI)

Đây là lớp hiển thị của ứng dụng, đóng vai trò như "bộ điều khiển từ xa" cho peer client. Nhiệm vụ của nó gồm:

- **Hiển thị trạng thái:** Hiển thị danh sách kênh, danh sách peer và nội dung chat dựa trên dữ liệu được đẩy xuống qua WebSocket.
- **Thu thập tương tác người dùng:** Ghi nhận hành động gõ tin nhắn, nhấn gửi, chọn cuộc trò chuyện.

- **Gửi lệnh:** Chuyển hóa các tương tác thành lệnh đơn giản (ví dụ: `send bob hello world`) gửi đến peer client để xử lý.
- **Quản lý session phía client:** Sử dụng `localStorage` để lưu thông tin phiên, cho phép tự động kết nối lại khi tải lại trang, mang lại trải nghiệm liền mạch.

2. Thiết kế các Module chính

Hệ thống được cấu thành từ ba module chính, hoạt động độc lập nhưng tương tác chặt chẽ với nhau: **Tracker Server**, **Peer Client** và **Giao diện Web**. Thiết kế của mỗi module được tối ưu hóa cho vai trò cụ thể của nó.

Tracker Server

Tracker Server được xây dựng như một dịch vụ API RESTful đơn giản bằng framework `WeApRous`. Nó quản lý trạng thái tập trung của toàn bộ mạng lưới P2P. Để đảm bảo an toàn dữ liệu khi nhiều peer truy cập đồng thời, các biến toàn cục như `peers_list` và `channels_list` được bảo vệ bằng `threading.Lock`.

Các API chính của Tracker Server bao gồm:

- **API /login**

Phương thức: POST

Chức năng: Xác thực danh tính người dùng dựa trên `username` và `password`.

Input (JSON Body):

```
{
  "username": "<tên>",
  "password": "<mật khẩu>"
}
```

Output (JSON):

– Thành công:

```
{
  "status": "success",
  "token": "token_<tên>"
}
```

– Thất bại:

```
{
  "status": "failed",
  "message": "Invalid credentials"
}
```

- **API /submit-info**

Phương thức: POST

Chức năng: Cho phép một peer đăng ký hoặc cập nhật thông tin với Tracker, bao gồm các kênh tham gia.

Input (JSON Body):

```
{
  "username": "<tên>",
  "ip": "<địa chỉ IP>",
  "port": <số port>,
}
```

```
"channels": ["<tên kênh>"]
}
```

Output (JSON):

```
{
  "status": "success",
  "peer_id": <id>,
  "total_peers": <số lượng>
}
```

- **API /add-list**

Phương thức: POST

Chức năng: Cho phép peer thông báo tham gia một kênh chat mới.

Input (JSON Body):

```
{
  "username": "<tên>",
  "channel": "<tên kênh>"
}
```

Output (JSON):

```
{
  "status": "success",
  "channel": "<tên kênh>",
  "members": ["<thành viên>"]
}
```

- **API /get-list**

Phương thức: GET hoặc POST

Chức năng: Cung cấp danh sách các peer đang trực tuyến, có thể lọc theo kênh.

Input (JSON Body, tùy chọn):

```
{
  hoặc
  {"channel": "<tên kênh>"}
}
```

Output (JSON):

```
{
  "status": "success",
  "peers": [{"username": ..., "ip": ..., "port": ...}]
}
```

Peer Client

Peer Client là module phức tạp nhất, được thiết kế để chạy như một tiến trình nền đa nhiệm. Khi `peer_client.py` được thực thi, hàm `start()` sẽ khởi tạo ba server độc lập, mỗi server chạy trên một luồng riêng để tránh block lẫn nhau:

- **_run_p2p_server**: mở socket TCP trên `peer_port`, lắng nghe kết nối P2P từ các peer khác.
- **_run_http_server**: mở TCP Server trên `http_port`, phục vụ giao diện web `chat.html` và xử lý các yêu cầu HTTP đặc biệt như tạo cookie.
- **_run_websocket_server**: mở server WebSocket trên `websocket_port`, sẵn sàng nhận kết nối từ giao diện web để giao tiếp hai chiều.

Luồng kết nối P2P (Handshake)

Quy trình thiết lập kết nối trực tiếp giữa hai peer (ví dụ Alice và Bob) được thiết kế theo cơ chế "handshake":

1. **Alice (khởi tạo)**: Gọi hàm `connect_peer` để tạo socket TCP mới và gửi request connect đến Bob.
2. **Bob (lắng nghe)**: Hàm `_handle_peer_connection` nhận kết nối, đọc tin nhắn đầu tiên.
3. **Alice gửi handshake**: Gửi JSON:

```
{"type": "handshake", "username": "alice"}
```

4. **Bob phản hồi**: Nhận handshake, lưu socket và gửi JSON:

```
{"type": "handshake_ack", "username": "bob"}
```

5. **Alice nhận ACK**: Xác nhận kết nối thành công, lưu socket.
6. **Bắt đầu lắng nghe**: Mỗi kết nối tạo luồng `_listen_to_peer` để nhận tin nhắn tương lai.

Giao diện Web (chat.html)

Giao diện được thiết kế như một *Single-Page Application* (SPA) đơn giản, nhận dữ liệu từ `peer_client.py`.

- **Tải trang và tự động điền**: Peer Client "tiêm" username và `peer_port` vào HTML. JavaScript điền form tự động.
- **Tự động kết nối (Session)**: Kiểm tra `localStorage`. Nếu tìm thấy phiên hợp lệ, bỏ qua màn hình đăng nhập và tự khởi tạo WebSocket.
- **Kết nối thủ công**: Nếu không có session, người dùng nhập thông tin, nhấn "Connect", lưu session và mở WebSocket.
- **Giao tiếp qua WebSocket**:
 - **Gửi lệnh**: Hành động người dùng (ví dụ gõ tin nhắn) được đóng gói thành lệnh dạng chuỗi và gửi qua WebSocket.
 - **Nhận dữ liệu**: JavaScript lắng nghe sự kiện `onmessage` để nhận tin nhắn mới hoặc danh sách peer cập nhật.
 - **Cập nhật giao diện**: Dựa trên loại dữ liệu, các hàm `displayMessages`, `updatePeerList` cập nhật giao diện liên tục.

3. Thiết kế Giao thức

Để các thành phần của hệ thống có thể giao tiếp và hiểu lẫn nhau, hai giao thức ứng dụng đơn giản đã được thiết kế dựa trên các kênh truyền TCP và WebSocket.

Giao thức P2P giữa các Peer

Giao thức này được sử dụng để trao đổi dữ liệu trực tiếp giữa các tiến trình `peer_client.py`. Toàn bộ dữ liệu được đóng gói dưới dạng JSON và gửi qua kết nối TCP. Mỗi thông điệp JSON đều chứa trường `type` để xác định mục đích của nó.

Các loại thông điệp chính:

- **Handshake**

Mục đích: Peer khởi tạo kết nối gửi thông điệp này để tự giới thiệu và bắt đầu quá trình "bắt tay".

Định dạng JSON:

```
{
  "type": "handshake",
  "username": "<tên của người gửi>"
}
```

- **Handshake Ack**

Mục đích: Peer nhận kết nối phản hồi lại handshake, xác nhận kết nối đã được chấp nhận.

Định dạng JSON:

```
{
  "type": "handshake_ack",
  "username": "<tên của người phản hồi>"
}
```

- **Chat**

Mục đích: Gửi tin nhắn riêng tư đến một peer cụ thể.

Định dạng JSON:

```
{
  "type": "chat",
  "from": "<tên người gửi>",
  "channel": "direct",
  "message": "<nội dung tin nhắn>",
  "time": "<dấu thời gian ISO 8601>"
}
```

- **Broadcast**

Mục đích: Gửi tin nhắn đến một kênh chat (ví dụ: **general** hoặc **broadcast**). Peer gửi sẽ lặp qua tất cả các kết nối P2P hiện có.

Định dạng JSON:

```
{
  "type": "broadcast",
  "from": "<tên người gửi>",
  "channel": "<tên kênh>",
  "message": "<nội dung tin nhắn>",
  "time": "<dấu thời gian ISO 8601>"
}
```

Giao thức WebSocket giữa UI và Backend

Giao thức này được thiết kế để giao tiếp giữa giao diện web (`chat.html`) và tiến trình `peer_client.py`. Nó hoạt động theo mô hình *command-based*, trong đó UI gửi các lệnh dạng văn bản thô, và backend trả về các thông điệp JSON.

Lệnh từ UI đến Backend (JavaScript → Python)

Các lệnh được gửi dưới dạng chuỗi, phân tách bằng khoảng trắng:

- `login <username> <password>`: Xác thực với Tracker Server.
- `join <channel_name>`: Tham gia một kênh chat.
- `list`: Lấy danh sách peer mới nhất từ Tracker.
- `connect <username> <ip> <port>`: Kết nối thủ công đến một peer (dành cho gỡ lỗi/nâng cao).
- `send <username> <message>`: Gửi tin nhắn riêng.
- `sendchannel <channel_name> <message>`: Gửi tin nhắn đến một kênh.

Thông điệp từ Backend đến UI (Python → JavaScript)

Backend gửi các thông điệp JSON để UI dễ phân tích và cập nhật giao diện:

- **Kết quả đăng nhập**

```
{
  "type": "login_result",
  "result": {"status": "success", "token": ...}
}
```

- **Danh sách Peer**

```
{
  "type": "peer_list",
  "peers": [{"username": ..., "ip": ..., "port": ...}, ...]
}
```

- **Tin nhắn Chat hoặc Broadcast**

Chuyển tiếp nguyên vẹn thông điệp JSON nhận được từ peer khác:

```
{
  "type": "chat",
  "from": "bob",
  "message": "Hello Alice!",
  ...
}
```

- **Thông báo hệ thống / Lỗi**

```
{
  "type": "system",
  "message": "P2P link established with bob."
}

{
  "type": "error",
  "message": "Connection to charlie timed out."
}
```

Chương 4: Hiện thực chương trình

1. Môi trường và Công cụ

Chương này trình bày chi tiết các khía cạnh kỹ thuật trong quá trình xây dựng hệ thống, bao gồm môi trường phát triển, công cụ sử dụng, cũng như mã nguồn then chốt hiện thực hai chức năng cốt lõi: **HTTP Server với Cookie Session** và **Ứng dụng Chat Hybrid (P2P + WebSocket)**.

Môi trường và Công cụ Phát triển

Để đảm bảo hệ thống vừa đáp ứng đúng yêu cầu kỹ thuật, vừa có tính ổn định và dễ mở rộng, nhóm đã lựa chọn bộ công cụ và thư viện phổ biến, mạnh mẽ nhưng vẫn giữ mức độ “tự hiện thực” cao ở tầng mạng (network layer).

Ngôn ngữ lập trình

- **Python 3.x**: Được chọn làm ngôn ngữ chính cho phần **backend** bao gồm **Proxy**, **Backend**, **Tracker Server** và **Peer Client**. Python cung cấp cú pháp trong sáng, thư viện chuẩn phong phú và khả năng xử lý mạng mạnh mẽ, giúp nhóm dễ dàng xây dựng từ tầng socket thấp mà vẫn đảm bảo hiệu năng và khả năng mở rộng.
- **HTML5, CSS3, JavaScript (ES6)**: Dùng để xây dựng toàn bộ **frontend** — giao diện web của hệ thống. Kết hợp HTML và CSS để tạo giao diện hiện đại, trong khi JavaScript (với WebSocket và API Fetch) đảm nhận phần logic tương tác thời gian thực.

Các thư viện Python chính sử dụng

- **socket**: Thư viện nền tảng của Python, được sử dụng để xây dựng các kết nối mạng TCP cấp thấp. Đây là công cụ cốt lõi để hiện thực **HTTP Server**, **Proxy**, **Tracker** và các kết nối P2P giữa các peer.
- **threading**: Được sử dụng để triển khai kiến trúc *đa luồng* (multithreading). Mỗi server (Backend, Tracker, P2P) hoạt động trong luồng riêng, đảm bảo khả năng phục vụ đồng thời nhiều kết nối mà không làm nghẽn (blocking) luồng chính.
- **argparse**: Cung cấp giao diện dòng lệnh (CLI) thân thiện, cho phép cấu hình linh hoạt các tham số như địa chỉ IP, port, hoặc đường dẫn file cấu hình khi khởi chạy server.
- **http.server** và **socketserver**: Hai thư viện tiêu chuẩn được kết hợp để xây dựng một *HTTP Server tùy chỉnh* bên trong `peer_client.py`. **socketserver** cung cấp khung xử lý socket đa luồng, trong khi **http.server** định nghĩa các lớp cơ sở giúp dễ dàng xử lý request và tạo response HTTP.
- **websockets**: Thư viện hiện thực giao thức WebSocket trong Python, cho phép xây dựng kênh giao tiếp hai chiều (full-duplex) giữa backend và trình duyệt. Đây là nền tảng cho phần ứng dụng chat thời gian thực.
- **json**: Dùng để *serialize* và *deserialize* dữ liệu giữa các module. JSON là định dạng truyền thông tin chính trong API Tracker và trong giao thức P2P.
- **functools.partial**: Được sử dụng để “gắn” tham chiếu đối tượng **PeerClient** vào lớp **CustomHandler** trong HTTP Server một cách an toàn, giúp lớp xử lý request có thể truy cập và thao tác trực tiếp trên trạng thái nội bộ của client hiện tại.

Công cụ phát triển phía Client (Trình duyệt)

- **WebSocket API:** Được hỗ trợ sẵn trong các trình duyệt hiện đại, cho phép JavaScript khởi tạo và quản lý kết nối WebSocket đến backend. Đây là cầu nối chính để gửi và nhận dữ liệu chat thời gian thực.
- **Fetch API:** Được sử dụng để gửi các yêu cầu HTTP không đồng bộ (asynchronous requests), ví dụ như trong quá trình xác thực `/confirm-login` hoặc khởi tạo session cookie.
- **localStorage:** Cơ chế lưu trữ phía client của trình duyệt. Dự án tận dụng localStorage để duy trì **phiên làm việc** (session) của người dùng, cho phép họ tự động kết nối lại sau khi tải lại trang hoặc đóng mở tab.
- **Developer Tools (F12):** Công cụ gỡ lỗi tích hợp trong trình duyệt — đặc biệt là:
 - Tab **Console:** Quan sát log JavaScript và lỗi runtime.
 - Tab **Network:** Theo dõi các request HTTP và luồng WebSocket.
 - Tab **Application:** Kiểm tra cookie, sessionStorage và localStorage.

2. HTTP Server với Cookie Session

Phần này trình bày chi tiết cách các chức năng trong đề bài đã được hiện thực, kèm các đoạn mã nguồn minh họa cho các giải pháp kỹ thuật đã áp dụng. Máy chủ HTTP cơ bản được xây dựng hoàn toàn bằng **socket** và **threading**, không sử dụng framework, giúp hiểu rõ cơ chế hoạt động của HTTP ở mức nền tảng. Logic xử lý request được đóng gói trong lớp `HttpAdapter` (file `daemon/httpadapter.py`), trong khi việc khởi tạo server và quản lý luồng được triển khai trong `daemon/backend.py`.

a. Khởi tạo Server và Xử lý Đa luồng

Nền tảng của máy chủ là khả năng xử lý nhiều kết nối đồng thời. Hàm `run_backend` chịu trách nhiệm lắng nghe kết nối mới và tạo luồng riêng biệt cho mỗi client. **Đoạn mã 4.1: Khởi tạo server và tạo luồng mới cho mỗi client trong `daemon/backend.py`**

Listing 1: Khởi tạo server và xử lý đa luồng

```
def run_backend(ip, port, routes):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        server.bind((ip, port))
        server.listen(50)
        print("[Backend] Listening on port {}".format(port))

        while True:
            conn, addr = server.accept()
            client_thread = threading.Thread(
                target=handle_client,
                args=(ip, port, conn, addr, routes)
            )
            client_thread.daemon = True
            client_thread.start()

    except socket.error as e:
        print("Socket error: {}".format(e))
```

b. Xử lý Yêu cầu Đăng nhập (Task 1A)

Khi người dùng gửi yêu cầu POST đến /login, hàm `handle_login` trong `HttpAdapter` sẽ được gọi để phân tích dữ liệu form, kiểm tra thông tin đăng nhập và thiết lập cookie nếu hợp lệ.

Đoạn mã 4.2: Logic xử lý đăng nhập và thiết lập cookie trong `daemon/httpadapter.py`

Listing 2: Xử lý đăng nhập và tạo cookie

```
def handle_login(self, req, resp):
    raw_body = req.body.decode("utf-8", "ignore")
    creds = {}
    for pair in raw_body.split("&"):
        if "=" in pair:
            k, v = pair.split("=", 1)
            creds[k] = v

    if creds.get("username") == "admin" and creds.get("password") == "password":
        req.path = "/index.html"
        raw_response = resp.build_response(req)
        body = raw_response.split(b"\r\n\r\n", 1)[1]

        headers = {
            "Content-Type": "text/html; charset=utf-8",
            "Set-Cookie": "auth=true; Path=/",
        }
        return ("200_OK", headers, body)
    else:
        e = RESP_TEMPLATES["login_failed"]
        return (e["status"], {"Content-Type": e["content_type"]}, e["body"])
```

c. Kiểm soát Truy cập dựa trên Cookie (Task 1B)

Để bảo vệ các tài nguyên nhạy cảm, hàm `cookie_auth_guard` được triển khai để kiểm tra tính hợp lệ của cookie trong mỗi request. Nếu không có hoặc cookie sai, người dùng sẽ bị chặn truy cập và nhận trang lỗi 401.

Đoạn mã 4.3: Logic kiểm tra cookie trong `daemon/httpadapter.py`

Listing 3: Kiểm soát truy cập bằng cookie

```
def cookie_auth_guard(self, req):
    if req.path in ("/", "/index.html"):
        if req.cookies.get("auth") != "true":
            e = RESP_TEMPLATES["unauthorized"]
            return (e["status"], {"Content-Type": e["content_type"]}, e["body"])

    if req.path == "/":
        req.path = "/index.html"

    return None
```

Thông qua các thành phần trên, hệ thống đã hoàn chỉnh chức năng của một **HTTP Server** có quản lý phiên người dùng bằng **Cookie Session**.

3. Ứng dụng Chat Hybrid (Client-Server kết hợp P2P)

Nhiệm vụ này yêu cầu xây dựng một hệ thống chat hoàn chỉnh, kết hợp cả hai mô hình **Client-Server** và **Peer-to-Peer (P2P)**. Hệ thống được chia thành hai thành phần chính: **Tracker Server** (`start_chatapp.py`) và **Peer Client** (`peer_client.py`).

a. Tracker Server – Trung tâm điều phối

Tracker Server đóng vai trò là “bộ não” trung tâm của mạng lưới. Thành phần này được xây dựng dựa trên framework **WeApRous** được cung cấp sẵn. Nó sử dụng các biến toàn cục được bảo vệ bởi **threading.Lock** để đảm bảo an toàn khi nhiều luồng truy cập và ghi dữ liệu đồng thời. Các chức năng chính được hiện thực thông qua các API endpoint.

Đoạn mã 4.4: Khai báo biến trạng thái và định nghĩa route trong start_chatapp.py

Listing 4: Tracker Server - quản lý trạng thái mạng lưới

```
peers_list = []
channels_list = {}
users_credentials = { "admin": "password", "alice": "alice123", ... }

peers_lock = threading.Lock()
channels_lock = threading.Lock()

app = WeApRous()

@app.route('/submit-info', methods=['POST'])
def submit_info(headers="guest", body="anonymous"):
    try:
        data = json.loads(body)
        # ...
        with peers_lock:
            # ...
            return json.dumps(response)
    except Exception as e:
        # ...
```

Đoạn mã trên minh họa cách **WeApRous** được dùng để định nghĩa endpoint một cách trực quan. Mỗi khi có yêu cầu POST `/submit-info`, hàm `submit_info()` sẽ được gọi để xử lý. Khóa `peers_lock` đảm bảo rằng việc cập nhật danh sách `peers_list` là an toàn, ngay cả khi nhiều peer đăng ký cùng lúc.

b. Peer Client – Kiến trúc Đa Server

PeerClient là thành phần phức tạp nhất của hệ thống, được thiết kế như một tiến trình đa nhiệm có thể đảm nhiệm nhiều vai trò đồng thời. Khi được khởi động, hàm `start()` sẽ tạo ra ba luồng chạy song song, tương ứng với ba server độc lập:

- **Server P2P** (`_run_p2p_server`): Lắng nghe và chấp nhận các kết nối TCP từ các peer khác.
- **Server HTTP** (`_run_http_server`): Phục vụ giao diện web `chat.html` và các tài nguyên tĩnh.
- **Server WebSocket** (`_run_websocket_server`): Duy trì kênh giao tiếp thời gian thực giữa backend và trình duyệt.

Đoạn mã 4.5: Khởi chạy ba server song song trong peer_client.py

Listing 5: PeerClient - khởi chạy ba server đồng thời

```
class PeerClient:
    # ...
    def start(self):
        self.running = True

        p2p_thread = threading.Thread(target=self._run_p2p_server, daemon=True)
        p2p_thread.start()
```

```
http_thread = threading.Thread(target=self._run_http_server, daemon=True)
http_thread.start()

ws_thread = threading.Thread(target=self._run_websocket_server, daemon=True)
ws_thread.start()
```

Cấu trúc đa luồng này đảm bảo rằng giao diện người dùng có thể hoạt động mượt mà, trong khi các kết nối mạng được xử lý song song mà không làm nghẽn tiến trình chính.

c. Giai đoạn Client-Server: Khám phá Peer

Trong giai đoạn này, một **Peer Client** đóng vai trò là “Client”, gửi yêu cầu đến **Tracker Server** để đăng ký thông tin và khám phá các peer khác trong cùng kênh.

Quy trình được thực hiện trong hàm `join_channel()`, gồm ba bước chính:

1. Gửi yêu cầu `POST /submit-info` đến Tracker để thông báo việc tham gia kênh.
2. Gửi yêu cầu `/get-list` để lấy danh sách các peer hiện đang trong cùng kênh.
3. Với mỗi peer nhận được, khởi tạo một luồng kết nối TCP riêng (P2P).

Đoạn mã 4.6: Logic tham gia kênh và khám phá peer trong `peer_client.py`

Listing 6: Tham gia kênh và khởi tạo kết nối P2P

```
def join_channel(self, channel):
    try:
        # ...

        print(f"[P2P-Auto-Connect]_Getting_peers_for_channel_{channel}...")
        peers_in_channel = self.get_peer_list(channel=channel)

        for peer in peers_in_channel:
            if peer.get("username") != self.username:
                connect_thread = threading.Thread(
                    target=self.connect_peer,
                    args=(peer.get("username"), peer.get("ip"), peer.get("port")),
                    daemon=True
                )
                connect_thread.start()

        return True
    except Exception as e:
        # ...
```

Việc sử dụng luồng riêng cho mỗi kết nối đảm bảo rằng quá trình khám phá và kết nối nhiều peer không gây treo giao diện hoặc nghẽn mạng.

d. Giai đoạn Peer-to-Peer: Giao tiếp trực tiếp

Sau khi quá trình khám phá hoàn tất, các peer có thể giao tiếp trực tiếp với nhau mà không cần qua Tracker. Hàm `connect_peer()` (bên chủ động) và `_handle_peer_connection()` (bên bị động) hiện thực quy trình “bắt tay” (handshake) và thiết lập kết nối TCP ổn định. Mỗi kết nối P2P sau đó được gắn với một luồng riêng `_listen_to_peer()` để lắng nghe tin nhắn đến.

Đoạn mã 4.7: Vòng lặp lắng nghe tin nhắn P2P trong `peer_client.py`

Listing 7: Lắng nghe tin nhắn từ peer khác

```
def _listen_to_peer(self, conn, peer_username):
    print(f"[P2P-Listen]_Started_listening_thread_for_{peer_username}.")
    conn.settimeout(10.0)
```

```
while self.running:
    try:
        data = conn.recv(4096)
        if not data:
            break

        message = json.loads(data.decode('utf-8'))
        self._process_peer_message(message)

    except socket.timeout:
        continue
    except Exception as e:
        print(f"[P2P_Listen] Connection_error_with_{peer_username}:{e}")
        break
```

Nhờ cơ chế này, hệ thống cho phép các peer gửi và nhận tin nhắn trong thời gian thực mà không cần máy chủ trung gian, đạt được mục tiêu **kết hợp mô hình Client-Server và P2P trong cùng một ứng dụng**.

4. “Put It All Together” – Tích hợp Hệ thống Hoàn chỉnh

Nhiệm vụ này yêu cầu kết hợp các kỹ năng đã học từ việc xây dựng **HTTP Server (Task 1)** và các thành phần của **Ứng dụng Chat Hybrid (Task 2)** để tạo ra một hệ thống thống nhất, liền mạch và có trải nghiệm người dùng hoàn chỉnh. Hệ thống được hiện thực thông qua kiến trúc ba lớp rõ ràng:

- **Backend Logic (Python)** – xử lý nghiệp vụ mạng và giao tiếp P2P.
- **Kênh Giao tiếp Real-time (WebSocket)** – cầu nối hai chiều giữa backend và giao diện.
- **Giao diện Người dùng (JavaScript)** – cung cấp trải nghiệm trực quan và tương tác.

a. Tích hợp Giao diện Web và Backend Logic

Thay vì hoạt động như một ứng dụng dòng lệnh (CLI), tệp `peer_client.py` được nâng cấp để đóng vai trò **server backend** phục vụ cho giao diện web. Mục tiêu của việc tích hợp này là:

- Cung cấp trải nghiệm người dùng trực quan, hiện đại.
- Giải quyết giới hạn bảo mật của trình duyệt (trình duyệt không thể trực tiếp mở socket TCP).

HTTP Server tùy chỉnh

Thành phần HTTP Server trong `peer_client.py` không chỉ đơn thuần phục vụ file HTML tĩnh, mà còn chủ động “*tiêm dữ liệu cấu hình*” (*injection*) vào file `chat.html` trước khi gửi đến trình duyệt. Dữ liệu này giúp giao diện web nhận biết được thông tin khởi tạo (ví dụ: `username`, `peerPort`) để bắt đầu giao tiếp với backend.

Listing 8: Logic “tiêm” dữ liệu vào HTML trong `peer_client.py`

```
def do_GET(self):
    if self.path == '/www/chat.html' or self.path == '/':
        # ...
        injected_data = {
            "username": self.server.peer_client_ref.username,
            "peerPort": self.server.peer_client_ref.peer_port
        }
        content = content.replace(
```

```
'<script_id="config-placeholder"></script>',  
f'<script>window.PEER_CONFIG=_{json.dumps(injected_data)};</script>' )
```

WebSocket – Cầu nối hai chiều giữa giao diện và backend

WebSocket đóng vai trò là **cầu nối thời gian thực** giữa trình duyệt và backend Python. Khi người dùng gửi tin nhắn, JavaScript không truyền trực tiếp qua mạng, mà gửi một lệnh dạng chuỗi (**send**) đến backend thông qua WebSocket. Backend sau đó xử lý logic P2P, gửi tin nhắn đến các peer khác, rồi đẩy phản hồi trở lại trình duyệt.

Listing 9: Gửi lệnh từ giao diện đến backend qua WebSocket

```
function sendMessage() {  
    const text = messageInput.value.trim();  
    const command = 'send ${currentChatTarget} ${text}';  
    sendCommand(command);  
}
```

Listing 10: Xử lý lệnh gửi tin nhắn trong backend Python

```
def _handle_ui_command(self, cmd):  
    parts = cmd.strip().split()  
    command = parts[0].lower()  
    if command == "send":  
        username = parts[1]  
        message = "_".join(parts[2:])  
        self.send_peer(username, message)
```

b. Kết hợp Quản lý Session (Cookie và localStorage)

Để đáp ứng yêu cầu “HTTP Cookies work” trong Task 1 và đồng thời đảm bảo trải nghiệm người dùng liền mạch, hệ thống kết hợp cả hai cơ chế lưu trữ phía client: **Cookie** và **localStorage**.

Cookie

Sau khi người dùng đăng nhập thành công, JavaScript gửi một yêu cầu `fetch('/confirm-login')`. Phía backend sẽ phản hồi với header `Set-Cookie`, minh chứng khả năng xử lý cookie của server HTTP. Mặc dù cookie này không tham gia vào logic chat thực tế, sự xuất hiện của nó trong trình duyệt (tab **Application** → **Cookies**) xác nhận rằng chức năng đã được hiện thực đúng chuẩn.

localStorage – Quản lý phiên thực tế

Cơ chế lưu trữ chính của ứng dụng là **localStorage**. Nó giúp duy trì trạng thái đăng nhập và tự động kết nối lại khi tải lại trang.

- **Lưu Session:** Sau khi kết nối WebSocket thành công, JavaScript lưu thông tin phiên (`username`, `peerPort`, `websocketPort`) vào **localStorage**.
- **Tự động kết nối lại:** Khi tải lại trang, script sẽ kiểm tra **localStorage**. Nếu tồn tại session hợp lệ, hệ thống bỏ qua bước đăng nhập và kết nối lại tự động.
- **Xóa Session:** Khi backend dừng hoạt động, session sẽ bị xóa để đảm bảo tính bảo mật.

Listing 11: Kiểm tra và tự động kết nối lại bằng localStorage trong `chat.html`

```
document.addEventListener('DOMContentLoaded', () => {  
    const savedSession = localStorage.getItem('chatSession');
```

```
    if (savedSession) {  
        const session = JSON.parse(savedSession);  
        if (window.PEER_CONFIG && session.username === window.PEER_CONFIG.username) {  
            console.log('Found matching session. Auto-connecting...');  
            currentUser.username = session.username;  
            connectWebSocket(session.websocketPort);  
        }  
    }  
});
```

Tổng thể, sự kết hợp giữa **Python Backend**, **WebSocket**, **Cookie** và **localStorage** giúp ứng dụng đạt được cả hai mục tiêu:

1. Đáp ứng các yêu cầu kỹ thuật của từng nhiệm vụ riêng lẻ.
2. Cung cấp một hệ thống chat phân tán hoạt động ổn định, tiện dụng và có tính thực tiễn cao.

Chương 5: Kiểm thử và kết quả

1. Kịch bản và Môi trường kiểm thử

- **Môi trường:** Các kịch bản được thực hiện trên một máy tính duy nhất chạy hệ điều hành Windows, mô phỏng một môi trường mạng cục bộ (LAN). Các tiến trình server và client được khởi chạy trên các cửa sổ dòng lệnh (PowerShell) riêng biệt. Giao diện người dùng được truy cập thông qua trình duyệt web ở chế độ ẩn danh để đảm bảo không bị ảnh hưởng bởi cache.
- **Địa chỉ IP:**
 - Tracker Server chạy trên 0.0.0.0:8001.
 - Các Peer Client kết nối đến Tracker Server qua địa chỉ 127.0.0.1:8001.
 - Địa chỉ P2P được các Peer Client đăng ký với Tracker là địa chỉ IP cục bộ của máy (ví dụ: 192.168.1.6).

2. Kết quả thực nghiệm

2.1. Kiểm thử HTTP Server và Cookie Session

Kịch bản này nhằm kiểm tra khả năng xử lý **HTTP request**, **xác thực**, và **quản lý phiên bằng cookie** của máy chủ HTTP được tự xây dựng từ đầu.

Bước 1: Khởi chạy các Server

- **Mở Terminal 1**, khởi chạy *Backend Server*:

```
python start_backend.py --server-ip <local-ip> --server-port 9000
```
- **Mở Terminal 2**, khởi chạy *Proxy Server*:

```
python start_proxy.py --server-ip 0.0.0.0 --server-port 8080
```

Bước 2: Kiểm thử luồng xác thực trên trình duyệt

- Mở trình duyệt ở chế độ ẩn danh, truy cập địa chỉ:

`http://<local-ip>:8080/`

Kết quả mong đợi: Trang web trả về lỗi 401 Unauthorized vì chưa có cookie xác thực.

- Truy cập trang đăng nhập:

`http://<local-ip>:8080/login.html`

Kết quả mong đợi: Form đăng nhập hiển thị chính xác.

- Nhập thông tin:

```
username = admin, password = password
```

Sau đó gửi form.

Kết quả mong đợi:

- Đăng nhập thành công, trình duyệt được chuyển hướng đến `index.html`.
- Trong Developer Tools (F12), một cookie `auth=true` được thiết lập.
- Truy cập lại `http://<local-ip>:8080/`.

Kết quả mong đợi: Trang `index.html` được hiển thị vì trình duyệt đã gửi kèm cookie hợp lệ.

Kết luận:

Chức năng xử lý request, xác thực đăng nhập và quản lý phiên bằng cookie hoạt động chính xác theo thiết kế. Hệ thống có thể phục vụ nhiều client đồng thời và duy trì phiên làm việc ổn định.

2.2. Kiểm thử Ứng dụng Chat Hybrid

Kịch bản này kiểm thử toàn bộ luồng hoạt động của **ứng dụng chat hybrid**, từ quá trình khám phá peer đến giao tiếp **P2P (Peer-to-Peer)** và quản lý session phía client.

Bước 1: Khởi chạy Tracker Server

```
python start_chatapp.py --server-ip 0.0.0.0 --server-port 8001
```

Kết quả mong đợi: Server khởi động thành công và lắng nghe trên port 8001.

Bước 2: Khởi chạy các Peer Client

- Mở **Terminal 2**, khởi chạy client cho người dùng **alice**:

```
py peer_client.py --username alice --peer-port 9101
```

- Mở **Terminal 3**, khởi chạy client cho người dùng **bob**:

```
py peer_client.py --username bob --peer-port 9102
```

Kết quả mong đợi:

- Mỗi client khởi động thành công.
- Mỗi client chạy **3 server** riêng biệt: P2P, HTTP và WebSocket.
- Các client tự động đăng ký với **Tracker Server**.

Bước 3: Kiểm thử Giao diện và Giao tiếp

- Mở hai cửa sổ trình duyệt ẩn danh riêng biệt cho Alice và Bob.
- Truy cập các URL tương ứng được in ra terminal, ví dụ:

```
http://localhost:19101/www/chat.html (Alice)
```

- Thực hiện đăng nhập cho cả hai người dùng.

Kết quả mong đợi:

- Cả hai đăng nhập thành công và tự động tham gia vào các kênh **#general** và **#broadcast**.
- Giao diện chat chính được hiển thị đúng bố cục.

Kiểm thử Chat Nhóm (Kênh #general)

- Trên giao diện của Alice, chọn kênh **#general** và gửi một tin nhắn.
- **Kết quả mong đợi:** Tin nhắn của Alice hiển thị đồng thời trên giao diện của Alice và Bob (vì cả hai cùng trong kênh).

Kiểm thử Chat Riêng (P2P)

- Trên giao diện của Alice, chuyển sang tab **Peers**, danh sách hiển thị bob.
- Chọn bob để mở cửa sổ chat riêng và gửi tin nhắn.
- **Kết quả mong đợi:** Giao diện của Bob hiển thị chấm thông báo bên cạnh tên **alice**. Khi Bob mở cửa sổ chat, lịch sử tin nhắn được hiển thị đầy đủ.

Kiểm thử Broadcast

- Trên giao diện của Alice, chọn kênh **#broadcast** và gửi tin nhắn.
- **Kết quả mong đợi:** Tin nhắn xuất hiện ngay trên cửa sổ chat của Bob, bất kể Bob đang ở kênh hay cuộc trò chuyện nào.

Kiểm thử Duy trì Session

- Nhấn F5 để tải lại trang của Alice.
- **Kết quả mong đợi:** Trang web tự động quay lại giao diện chat mà không cần đăng nhập lại, nhờ thông tin lưu trong **localStorage**. Cookie “trình diễn” vẫn còn hiệu lực.

Kết quả thực nghiệm

Kết luận:

Toàn bộ luồng hoạt động của ứng dụng chat hybrid — bao gồm khám phá, giao tiếp P2P, chat nhóm, broadcast và duy trì session — đều hoạt động ổn định, đúng với yêu cầu của đề tài. Hệ thống chứng minh tính tích hợp hoàn chỉnh giữa các thành phần HTTP, WebSocket, và P2P.

Chương 6: Kết luận

6.1. Tổng kết kết quả đạt được

Sau quá trình phân tích, thiết kế và hiện thực, nhóm đã hoàn thành đầy đủ các mục tiêu đề ra, xây dựng thành công một ứng dụng mạng hoàn chỉnh, qua đó áp dụng hiệu quả các kiến thức nền tảng về mạng máy tính, giao thức truyền thông và lập trình socket.

- **Hoàn thiện HTTP Server và cơ chế Cookie:** Hệ thống đã hiện thực thành công máy chủ HTTP và Proxy ở mức socket thấp, đảm bảo xử lý chính xác luồng xác thực và quản lý phiên làm việc (session) thông qua cookie. Chức năng này đáp ứng đầy đủ yêu cầu của Nhiệm vụ 2.1.
- **Xây dựng kiến trúc Chat Hybrid:** Dự án đã triển khai một mô hình chat kết hợp (hybrid) giữa hai kiến trúc Client-Server và Peer-to-Peer, bao gồm hai thành phần chính là Tracker Server (điều phối và khám phá) và Peer Client (giao tiếp trực tiếp). Hệ thống hoạt động ổn định, có khả năng mở rộng linh hoạt.
- **Hiện thực các hình thức giao tiếp đa dạng:** Ứng dụng hỗ trợ ba chế độ liên lạc chính: chat nhóm (qua kênh #general), chat riêng P2P, và broadcast toàn hệ thống, đáp ứng đầy đủ nhu cầu trao đổi giữa các người dùng.
- **Phát triển giao diện Web hoàn chỉnh:** Giao diện người dùng được xây dựng bằng HTML5, CSS3 và JavaScript (ES6), kết nối trực tiếp với backend Python thông qua Web-Socket, mang lại trải nghiệm tương tác thời gian thực, mượt mà và hiện đại.
- **Tối ưu trải nghiệm và tính tiện dụng:** Hệ thống hỗ trợ lưu trạng thái đăng nhập bằng cơ chế *localStorage*, tự động đăng nhập lại sau khi tải lại trang, đồng thời đảm bảo sự ổn định trong quá trình thử nghiệm thực tế.

Tổng thể, đề tài đã được hoàn thiện và vận hành đúng theo mục tiêu ban đầu, vượt qua các yêu cầu cốt lõi về kỹ thuật và chức năng.

6.2. Hạn chế của hệ thống

Mặc dù hệ thống đã hoạt động ổn định, một số hạn chế vẫn tồn tại, chủ yếu ở khía cạnh bảo mật và khả năng mở rộng:

- **Bảo mật:** Các gói tin và thông tin người dùng (bao gồm mật khẩu và nội dung tin nhắn) hiện vẫn được truyền ở dạng văn bản thô, chưa được mã hóa. Việc lưu trữ mật khẩu trực tiếp trên Tracker Server cũng tiềm ẩn rủi ro an ninh.
- **Khả năng chịu lỗi và mở rộng:** Tracker Server đang là *Single Point of Failure*. Nếu máy chủ này gặp sự cố, toàn bộ mạng lưới sẽ không thể khám phá hoặc kết nối với nhau.
- **Độ tin cậy của giao thức P2P:** Cơ chế truyền tin giữa các peer còn đơn giản, chưa có biện pháp xử lý mất gói, xác nhận tin nhắn hay tái truyền dữ liệu khi xảy ra lỗi mạng.
- **Vấn đề NAT Traversal:** Hệ thống hiện chỉ hoạt động ổn định trong mạng nội bộ (LAN). Khi các peer nằm sau các bộ định tuyến NAT khác nhau, kết nối P2P trực tiếp sẽ thất bại do chưa tích hợp các kỹ thuật vượt NAT như STUN/TURN.

6.3. Hướng phát triển

Dựa trên những hạn chế trên, nhóm đề xuất một số hướng mở rộng và cải tiến cho hệ thống trong tương lai:

- **Tăng cường bảo mật:**

- Mã hóa đầu cuối (*End-to-End Encryption*): Áp dụng các thuật toán như RSA hoặc AES để mã hóa nội dung tin nhắn ngay tại phía người gửi, đảm bảo chỉ người nhận có thể giải mã.
- Băm mật khẩu (*Password Hashing*): Sử dụng các thuật toán như bcrypt hoặc Argon2 để lưu trữ mật khẩu an toàn thay vì dạng văn bản thuần.

- **Nâng cao khả năng mở rộng và tin cậy:**

- Triển khai mô hình *Multi-Tracker* hoặc phân tán bằng *Distributed Hash Table (DHT)* để loại bỏ sự phụ thuộc vào một server trung tâm.
- Bổ sung cơ chế đồng bộ và xác nhận tin nhắn (ACK), đảm bảo độ tin cậy khi truyền dữ liệu qua mạng không ổn định.

- **Cải thiện trải nghiệm người dùng:**

- Bổ sung chức năng gửi tập tin (File Transfer) qua giao thức P2P.
- Hiển thị trạng thái người dùng (online/offline/typing) và cho phép tùy chỉnh hồ sơ cá nhân.
- Mở rộng hệ thống thông báo và danh sách bạn bè, hướng tới trải nghiệm tương tự các ứng dụng chat chuyên nghiệp.

Kết luận chung: Đề tài không chỉ giúp nhóm củng cố kiến thức lý thuyết về mạng máy tính mà còn rèn luyện kỹ năng lập trình mạng ở cấp độ hệ thống. Kết quả đạt được chứng minh tính khả thi của việc xây dựng các ứng dụng mạng phân tán từ nền tảng socket, mở ra nhiều hướng phát triển chuyên sâu hơn trong tương lai.