

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



Lớp TN01 - Nhóm 8386

---

Bài tập lớn 1

# *Mạng máy tính (CO3094)*

---

*“Implement HTTP server and chat application”*

Giảng viên hướng dẫn: Hoàng Lê Hải Thanh

Sinh viên thực hiện: Lê Nguyễn Kim Khôi - 2311671  
Nguyễn Công Minh - 2312080  
Bùi Ngọc Phúc - 2312665  
Lê Trọng Thiện - 2313233  
Phạm Trần Minh Trí - 2313622

THÀNH PHỐ HỒ CHÍ MINH, 11/2025



## Mục lục

<b>Danh sách thành viên</b>	<b>2</b>
<b>1 Mở đầu</b>	<b>3</b>
<b>2 Kiến trúc hệ thống</b>	<b>4</b>
2.1 HTTP Server – Reverse Proxy & Cookie Session . . . . .	4
2.2 Chat P2P – Server điều phối & Peer truyền dữ liệu . . . . .	5
<b>3 HTTP server with cookie session</b>	<b>6</b>
3.1 Implement authentication handling . . . . .	6
3.1.1 Routing & Validation . . . . .	6
3.1.2 Invalid . . . . .	6
3.1.3 Valid . . . . .	6
3.2 Implement cookie-based access control . . . . .	7
3.2.1 Routing . . . . .	7
3.2.2 Cookie Check . . . . .	7
3.2.3 Invalid cookie . . . . .	7
3.2.4 Valid cookie . . . . .	7
<b>4 Hybrid chat application</b>	<b>8</b>
4.1 Chatting between 2 peer . . . . .	8
4.1.1 Giao tiếp trực tiếp 1-1 giữa peer . . . . .	8
4.1.2 Cơ chế broadcast . . . . .	9
4.2 Channel management . . . . .	10
<b>5 Kết luận</b>	<b>12</b>
<b>Tài liệu tham khảo</b>	<b>13</b>



## Danh sách thành viên

STT	Họ và tên	MSSV	Phân công	% hoàn thành
1	Lê Nguyễn Kim Khôi	2311671	- Chat P2P - Báo cáo	100%
2	Nguyễn Công Minh	2312080	- Chat P2P - Báo cáo	100%
3	Bùi Ngọc Phúc	2312665	- HTTP server - Báo cáo	100%
4	Lê Trọng Thiện	2313233	- HTTP server - Báo cáo	100%
5	Phạm Trần Minh Trí	2313622	- Chat P2P - Báo cáo	100%

Bảng 1: Danh sách thành viên và phân công

## 1 Mở đầu

Trong bối cảnh công nghệ thông tin hiện đại, các ứng dụng mạng đóng vai trò then chốt trong hầu hết mọi hoạt động, từ giao tiếp, làm việc đến giải trí. Mục tiêu của bài tập lớn là thiết kế và triển khai một server HTTP và ứng dụng chat Peer-to-Peer. BTL yêu cầu xây dựng hệ thống từ đầu, sử dụng các thư viện lập trình python socket TCP, mà không dựa trên các framework web có sẵn. Hệ thống này kết hợp hai mô hình mạng cơ bản: mô hình Client-Server và mô hình Peer-to-Peer (P2P).

Nội dung chính của BTL gồm:

1. **HTTP Server với Cookie Session:** Xây dựng một máy chủ HTTP có khả năng xử lý các yêu cầu, parse header, và quản lý nội dung static. Một yêu cầu quan trọng là triển khai cơ chế xác thực người dùng (authentication) và quản lý phiên (session) dựa trên cookie.
2. **Ứng dụng Chat Hybrid:** Phát triển một ứng dụng chat kết hợp cả hai mô hình. Mô hình client-server được sử dụng cho các tác vụ ban đầu như đăng ký, đăng nhập và đăng ký địa chỉ peer (peer registration). Sau đó, hệ thống chuyển sang mô hình P2P cho việc trò chuyện trực tiếp (direct peer communication) và trò chuyện nhóm (channel management).

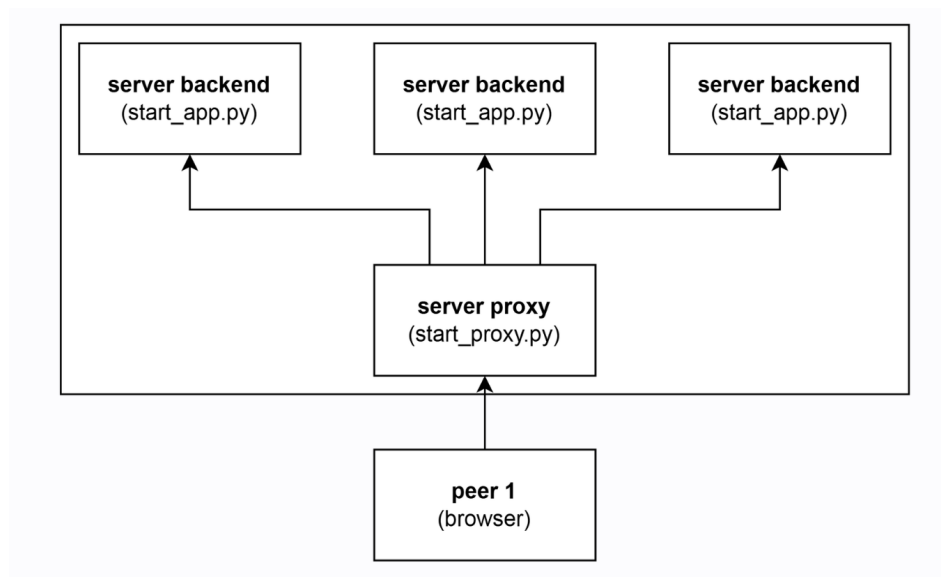
Nhóm sẽ trình bày chi tiết về kiến trúc hệ thống, các quyết định thiết kế và triển khai code, từ luồng xử lý của HTTP server, logic quản lý cookie, đến giao thức giao tiếp P2P.

Repo code của BTL: <https://github.com/phamtranminhtri/8386-C03094-weaprous>

## 2 Kiến trúc hệ thống

WeApRous là một ứng dụng mạng **hybrid** (chat tương tự Skype), kết hợp **client-server** và **peer-to-peer (P2P)**. Hệ thống hỗ trợ **quản lý channel** và **đồng bộ trạng thái** giữa các peer phân tán. Kiến trúc gồm hai phần đúng theo Hình 1 và Hình 2: (i) **HTTP Server theo Reverse Proxy** cho web/điều phối và xác thực phiên; (ii) **Chat P2P có server điều phối** để thiết lập kết nối và duy trì thông tin channel/peer.

### 2.1 HTTP Server – Reverse Proxy & Cookie Session

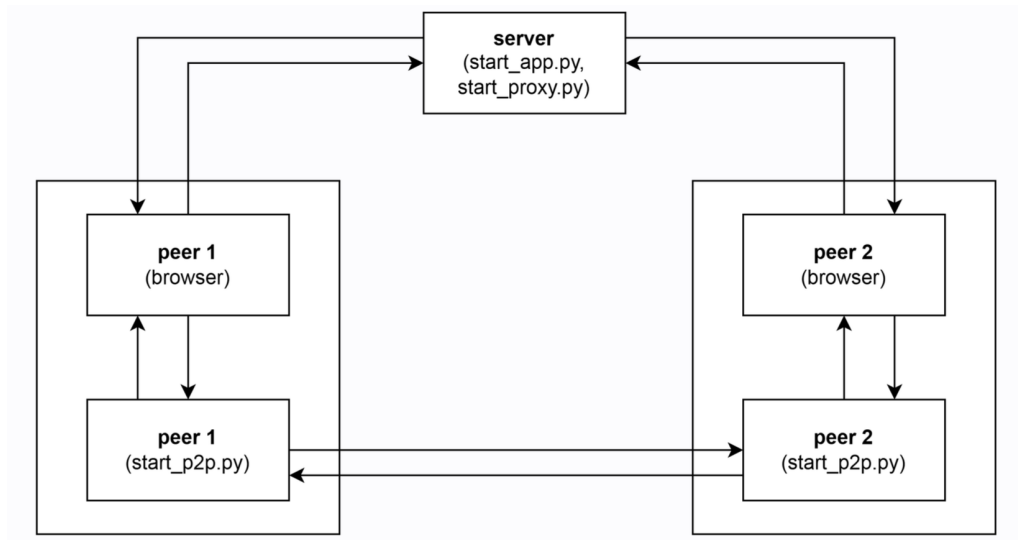


Hình 1: Kiến trúc HTTP Server với cơ chế xác thực cookie

Như Hình 1, mọi request từ **peer (browser)** đi vào **server proxy (start\_proxy.py)** và được **forward** đến một **server backend (start\_app.py)** trong backend pool.

- **Proxy** đóng vai trò **trung gian/entry point**, giúp quản lý traffic, tập trung định tuyến, và hỗ trợ forwarding/load distribution khi có nhiều backend, từ đó tăng khả năng mở rộng và độ ổn định.
- **Backend** là lớp xử lý nghiệp vụ: routing/API, phục vụ nội dung, và **xác thực & quản lý phiên** bằng **cookie session** (trả Set-Cookie khi đăng nhập và kiểm tra Cookie ở các request sau). Ngoài ra, backend cung cấp thông tin peer và metadata phục vụ điều phối P2P (peer online, tạo/join channel, thành viên).

## 2.2 Chat P2P – Server điều phối & Peer truyền dữ liệu



Hình 2: Kiến trúc ứng dụng Chat P2P

Như Hình 2, kiến trúc chat tách thành:

- **Client-server:** peer dùng **browser** làm việc với **server** (`start_app.py`, `start_proxy.py`) để đăng nhập/xác thực, đăng ký IP/port, lấy danh sách peer online và **quản lý channel**. Server đóng vai trò **tracker/registry** để hỗ trợ **đồng bộ trạng thái** (thành viên channel, trạng thái peer, thông tin kết nối).
- **P2P:** mỗi máy chạy **P2P client** (`start_p2p.py`), sau đó các peer **kết nối trực tiếp** qua TCP socket để gửi/nhận tin nhắn, không chuyển tiếp nội dung chat qua server.

Mỗi peer gồm **browser** (UI/điều phối) và **P2P client** (socket dữ liệu), đảm bảo server tập trung vào quản lý/đồng bộ, còn dữ liệu chat truyền trực tiếp giữa các peer.

## 3 HTTP server with cookie session

### 3.1 Implement authentication handling

Hệ thống WeApRous thực hiện việc xác thực thông qua sự phối hợp của 4 tệp: `start_app.py` (định nghĩa logic), `httpadapter.py` (điều phối), `request.py` (xử lý body), và `response.py` (tạo phản hồi).

#### 3.1.1 Routing & Validation

- Tệp `start_app.py` định nghĩa route: `@app.route('/login', methods=['POST'])` và liên kết nó với hàm `login_post`.
- `request.py` (trong `prepare_body`) phân tích cú pháp body của yêu cầu POST (đã được `httpadapter.py` truyền vào) để lấy `username` và `password`.
- Hàm `login_post` thực hiện kiểm tra logic: `if (username == "admin" and password == "password")` .... Điều này đáp ứng yêu cầu xác thực.

#### 3.1.2 Invalid

- Nếu if thất bại, hàm `login_post` trả về một từ điển: `{"auth": "false"}`.
- `httpadapter.py` (trong `handle_client`) nhận kết quả này (`hook_result`). Nó kiểm tra `if hook_result["auth"] == "false":`.
- Nó ngay lập tức gọi `response = resp.build_unauthorized()`.
- Hàm `build_unauthorized` trong `response.py` tạo ra một trang phản hồi HTTP/1.1 401 Unauthorized hoàn chỉnh.
- **Kết luận:** Yêu cầu "401 Unauthorized" được đáp ứng chính xác.

#### 3.1.3 Valid

- Nếu if thành công, hàm `login_post` tạo một `session_id` và trả về một từ điển phức tạp hơn: `{"auth": "true", "redirect": "/", "session_id": session_id}`.
- `httpadapter.py` nhận kết quả này. Nó **không** trả về trang `index` ngay lập tức. Thay vào đó, nó phát hiện khóa `"redirect"`.
- Nó gọi hàm `response = resp.build_redirect(redirect, req, new_session_id)`.
- Hàm `build_redirect` trong `response.py` tạo ra một phản hồi HTTP/1.1 302 Found (chuyển hướng).
- **Thiết lập Cookie:** Quan trọng nhất, bên trong `build_redirect`, nó chèn các header `Set-Cookie` vào phản hồi:
  - `Set-Cookie: auth=true; Path=`
  - `Set-Cookie: session_id=...; Path=`
- **Kết luận:** Yêu cầu được đáp ứng. Máy chủ đặt cookie `auth=true` thành công. Thay vì trực tiếp phục vụ trang `index`, nó sử dụng một thông báo chuyển hướng (Redirect), điều này khiến trình duyệt của máy khách tự động thực hiện một yêu cầu GET / mới, và yêu cầu này sẽ được xử lý bởi logic của Task 1B.

## 3.2 Implement cookie-based access control

### 3.2.1 Routing

Tập `start_app.py` định nghĩa route: `@app.route('/', methods=['GET'])` và liên kết nó với hàm `index`.

### 3.2.2 Cookie Check

- Hành động đầu tiên của hàm `index` là gọi `if not authenticate(headers): return {"auth": "false"}`. Hàm `authenticate` (trong `start_app.py`) là cơ chế kiểm soát truy cập.
- **Phân tích `authenticate(headers)`:**
  - Nó không đọc trực tiếp header `Cookie`: Thay vào đó, nó dựa vào `request.py`.
  - Khi yêu cầu `GET` / đến, `request.prepare` được gọi trước. `request.prepare` gọi `prepare_cookies`.
  - `prepare_cookies` đọc chuỗi cookie thô (ví dụ: `"auth=true; session_id=123"`) và phân tích nó thành một từ điển Python: `{'auth': 'true', 'session_id': '123'}`. Từ điển này được lưu trong `headers["cookie-pair"]`.
  - Hàm `authenticate` sau đó đọc từ điển đã được xử lý này: `cookie = headers.get("cookie-pair", None)`.
  - Nó kiểm tra cụ thể: `auth = cookie.get("auth", "")` và sau đó kiểm tra `if auth == "true" and session_id in session_to_account`.

### 3.2.3 Invalid cookie

- Nếu `authenticate` trả về `False` (do cookie không tồn tại, `auth` không phải là `"true"`, hoặc `session_id` không hợp lệ), hàm `index` sẽ trả về `{"auth": "false"}`.
- `httpadapter.py` bắt được giá trị này và nó gọi `resp.build_unauthorized()` để trả về `401 Unauthorized`.
- **Kết luận:** Yêu cầu được đáp ứng chính xác.

### 3.2.4 Valid cookie

- Nếu `authenticate` trả về `True`, hàm `index` tiếp tục thực thi.
- Nó trả về một từ điển: `{"auth": "true", "content": "index.html", "placeholder": ...}`.
- `httpadapter.py` nhận kết quả này. Nó thấy khóa `"content"` (và `"placeholder"`).
- Nó gọi `response = resp.build_content_placeholder(req, content, placeholder)`.
- Hàm này trong `response.py` sẽ tải tệp `index.html` từ thư mục `www/`, thay thế bất kỳ nội dung động nào (nếu có), và xây dựng một phản hồi `HTTP/1.1 200 OK` hoàn chỉnh với nội dung trang.
- **Kết luận:** Yêu cầu "phục vụ trang index" được đáp ứng.

## 4 Hybrid chat application

Thiết kế này áp dụng nguyên tắc phân tách trách nhiệm (Separation of Concerns) rõ ràng giữa hai thành phần chính:

- Web Server (`start_app.py`): Đóng vai trò là "người điều phối". Server chịu trách nhiệm quản lý việc xác thực (authentication) người dùng, phân quyền (authorization), tạo channel, quản lý thành viên (ai được tham gia) và lưu trữ mật khẩu (dưới dạng hash).
- P2P Client (`start_p2p.py`): Chịu trách nhiệm xử lý toàn bộ logic nhắn tin (messaging). Sau khi được Web Server "giới thiệu" và cung cấp danh sách thành viên, P2P Client sẽ thực hiện broadcast tin nhắn trực tiếp đến các thành viên khác trong channel mà không cần thông qua server.

### 4.1 Chatting between 2 peer

Hệ thống hỗ trợ hai chức năng chính trong mô hình P2P:

- Giao tiếp trực tiếp 1-1 giữa các peer.
- Broadcast tin nhắn đến tất cả peer trong mạng.

Luồng hoạt động chính:

- Gửi thông tin địa chỉ — `@app.route('/submit-info', methods=['POST'])`: Người dùng gửi thông tin địa chỉ (*chatting IP, port, local-port*) trước khi tham gia mạng P2P.
  1. Người dùng nhập thông tin địa chỉ thông qua form trên giao diện.
  2. Hệ thống kiểm tra tính hợp lệ của địa chỉ và thực hiện xác thực người dùng.
  3. Sau khi xác thực thành công, hệ thống lấy *username* và lưu thông tin địa chỉ tương ứng vào bảng `account_to_address`.
- Xem danh sách các người dùng — `@app.route('/get-list', methods=['GET'])`: Hỗ trợ người dùng xem danh sách peer đang online trong mạng P2P.
  1. Hệ thống xác thực người dùng và kiểm tra xem người dùng đã khai báo địa chỉ hay chưa.
  2. Duyệt toàn bộ bảng `account_to_address` để xây dựng danh sách peer đang hoạt động.
  3. Với mỗi peer, hệ thống sinh một form HTML cho phép kết nối trực tiếp 1-1 thông qua endpoint `/connect-peer`.
  4. Đồng thời tạo form broadcast, chứa danh sách toàn bộ peer còn lại để gửi tin nhắn diện rộng.

#### 4.1.1 Giao tiếp trực tiếp 1-1 giữa peer

Chức năng giao tiếp trực tiếp giữa các peer được hiện thực theo cơ chế trao đổi tin nhắn giữa hai peer không phụ thuộc vào bất kỳ máy chủ trung gian nào; thay vào đó, hai ứng dụng liên kết trực tiếp với nhau thông qua kết nối TCP socket.

Luồng hoạt động chi tiết như sau:

- Kết nối vào đoạn chat — `@app.route("/connect-peer", methods=["POST"])`: Người dùng muốn thiết lập kết nối vào đoạn chat với một người dùng khác.
  1. Hệ thống lưu thông tin địa chỉ (*server\_ip*, *server\_port*) của người dùng hiện tại.
  2. Sau đó, lấy thông tin địa chỉ của peer mà người dùng muốn kết nối.
  3. Tiếp theo thực hiện chuyển hướng sang trang /chat và kèm theo tham số địa chỉ của peer cần kết nối.
  4. Tại `@app.route("/chat", methods=["GET"])`, hệ thống kiểm tra *chat\_history* để lấy lại nội dung hội thoại đã lưu tương ứng.
- Gửi tin nhắn — `@app.route("/chat", methods=["POST"])`: Người dùng gửi một tin nhắn mới trong giao diện chat.
  1. Hệ thống lấy địa chỉ peer qua `headers["query"]` và nội dung tin nhắn từ `body["message"]`.
  2. Sau đó đưa tin nhắn này vào *send\_queue* dưới dạng một tuple (*peer\_ip*, *peer\_port*, *message\_to\_send*).
  3. *Sender thread* liên tục đọc tin nhắn từ hàng đợi (*send\_queue*).
  4. *Sender thread* gọi `send_message()`, tạo timestamp và đóng gói tin nhắn theo đúng định dạng chuẩn.
  5. Hàm `send_message()` mở một socket TCP tạm thời, kết nối đến địa chỉ của peer đích và chuyển tiếp gói tin.
  6. Sau khi gửi thành công hoặc xảy ra lỗi, socket được đóng và cập nhật *chat\_history* cho peer tương ứng.
- Nhận tin nhắn — mỗi peer duy trì một *server\_thread* thực thi hàm `start_server()` mở một socket TCP tại địa chỉ `0.0.0.0:port` và lắng nghe mọi kết nối đến. Khi có một peer khác gửi tin đến:
  1. Đầu tiên, server gọi `accept()` và tạo một handler thread mới để xử lý kết nối đó.
  2. Tiếp theo đọc thông điệp từ socket, giải mã chuỗi tin nhắn theo đúng định dạng chuẩn và phân tách ba thành phần: sender, timestamp và nội dung.
  3. Hệ thống hiển thị thông tin tin nhắn lên giao diện console và cập nhật nội dung vào *chat\_history*.
  4. Và khi xử lý xong, kết nối socket được đóng lại.

#### 4.1.2 Cơ chế broadcast

Bên cạnh chức năng giao tiếp 1-1 giữa hai peer, hệ thống còn hỗ trợ cơ chế gửi tin nhắn theo mô hình broadcast, tức là một peer có thể gửi một thông điệp đồng thời đến toàn bộ các peer khác trong mạng. Broadcast là hành động gửi một gói tin duy nhất từ peer nguồn đến tập tất cả các peer được lưu trong danh sách *peer\_list*.

Luồng hoạt động như sau:

- Xác định thông tin địa chỉ — `@app.route("/broadcast0", methods=["POST"])`: Người dùng muốn gửi tin nhắn đến mọi người khác:
  1. Hệ thống lưu thông tin địa chỉ (*server\_ip*, *server\_port*) của người dùng hiện tại.
  2. Sau đó, lấy danh sách thông tin địa chỉ của các peer trong *peer-list*.

3. Tiếp theo hệ thống xác thực người dùng và thực hiện chuyển hướng sang trang /broadcast để hiển thị giao diện tương ứng.
- Gửi tin nhắn — **@app.route("/broadcast", methods=["POST"])**: Người dùng gửi một tin nhắn broadcast từ giao diện:
    1. Đầu tiên, hệ thống lấy nội dung tin nhắn và tự động thêm tiền tố "[Broadcast]" để đánh dấu đây là dạng tin broadcast.
    2. Hệ thống duyệt qua toàn bộ **peer\_list**, và với mỗi peer sẽ đưa vào **send\_queue** một yêu cầu gửi tin: (peer\_ip, peer\_port, broadcast\_message).
    3. Tiếp theo queue này sẽ được xử lý bởi sender thread.
  - Nhận tin nhắn broadcast: Vì mỗi peer duy trì một **server\_thread** mở một socket để lắng nghe mọi tin được gửi đến nên phần này hoạt động giống như P2P 1-1.

## 4.2 Channel management

Tính năng Channel Chat được thiết kế để cho phép nhiều người dùng tham gia vào một phòng chat chung, song song với tính năng chat 1-1 hiện có. Mỗi channel được xác định duy nhất bằng tên (channel name) và được bảo vệ bằng mật khẩu để kiểm soát truy cập.

Luồng hoạt động chi tiết như sau:

- Tạo Channel mới - **@app.route("/create-channel", methods=["POST"])**: Người dùng khởi tạo một phòng chat mới.
  1. Đầu tiên, User cung cấp channel\_name và password qua form.
  2. Tiếp theo Server kiểm tra channel\_name có tồn tại trong hệ thống chưa. Nếu đã tồn tại thì báo lỗi cho người dùng.
  3. Nếu chưa thì tuần tự Server thực hiện hash(password) để lưu trữ an toàn, sau đó tạo một mục mới trong cấu trúc dữ liệu channels và cuối cùng thêm địa chỉ của người dùng vào danh sách thành viên đầu tiên.
- Xem danh sách và Tham gia Channel - **@app.route("/channel", methods=["GET"])** và **@app.route("/join-channel", methods=["POST"])**: Cho phép người dùng thấy các channel có sẵn và tham gia chúng. Với xem luồng danh sách thì khi truy cập channel, server hiển thị 2 danh sách bao gồm joined channels - Các channel mà người dùng đã là thành viên; và available channels - Các channel khác mà người dùng có thể tham gia. Còn đối với luồng tham gia:
  1. User chọn một channel từ "Available channels" và nhập password.
  2. Server lấy password người dùng nhập, thực hiện hash và so sánh với stored\_password\_hash của channel đó.
  3. Nếu hash(password) == stored\_password\_hash, thêm địa chỉ của user (user\_address) vào address\_list của channel kèm thông báo tham gia thành công. Nếu sai, báo lỗi mật khẩu không chính xác.
- Kết nối vào Channel Chat (Handoff to P2P) - **@app.route("/connect-channel", methods=["POST"])**: Khởi tạo phiên chat P2P sau khi người dùng đã được xác thực là thành viên.
  1. User nhấp vào "Connect" trên một channel đã tham gia.

2. Web Server lấy `address_list` (danh sách địa chỉ IP/Port) của tất cả thành viên hiện tại trong channel đó.
  3. Server thực hiện một POST redirect (chuyển hướng POST) sang ứng dụng P2P Client (`start_p2p.py`), mang theo `address_list` này.
  4. P2P Client nhận được danh sách, lưu vào `channel_members[channel_name]` và thực hiện broadcast một thông báo "join" đến tất cả các thành viên trong danh sách.
- Chat trong Channel: Luồng gửi và nhận tin nhắn P2P. Đây là luồng gửi tin nhắn:
    1. User gõ tin nhắn trong giao diện (UI) của channel.
    2. P2P App (của người gửi) xác định channel hiện tại (`channel_name`).
    3. App lấy danh sách thành viên từ `channel_members[channel_name]`.
    4. App thực hiện broadcast (gửi đồng thời) tin nhắn đến tất cả địa chỉ trong danh sách đó.

Còn đây là luồng nhận tin nhắn:

1. P2P Client (của người nhận) lắng nghe và nhận được một tin nhắn.
2. Client kiểm tra định dạng tin nhắn.
3. Client phân tích (parse) tin nhắn để lấy `sender_address`, `timestamp`, `channel_name`, và `message`.
4. Client xác định đây là tin nhắn của `channel_name` và hiển thị nội dung `message` lên UI của channel tương ứng.

Để cụ thể hơn về định dạng tin nhắn được nhắc ở trên thì cần phân tích sâu hơn ở đặc điểm chính là để phân biệt tin nhắn. Để phân biệt tin nhắn 1-1 và tin nhắn channel, một định dạng prefix đặc biệt được sử dụng ở trong này. Client P2P sẽ dựa vào prefix này để xử lý tin nhắn đúng cách.

Định dạng này được định nghĩa như sau: `[sender_address] [timestamp] [Channel]: channel_name message`. Tiền tố `[Channel]channel_name` là dấu hiệu nhận biết (flag) cho P2P Client rằng đây là tin nhắn thuộc về một channel cụ thể.

## 5 Kết luận

Bài tập lớn 1, “Implement HTTP server and chat application”, đã được nhóm 8386 hoàn thành với các chức năng cốt lõi bám sát yêu cầu đề bài. Nhóm đã thành công trong việc xây dựng một hệ thống hoàn chỉnh từ đầu, không sử dụng các framework web có sẵn, mà chỉ dùng các thư viện cơ bản của Python như **socket** và **threading**.

BTL đã áp dụng và làm rõ các khái niệm mạng máy tính trọng tâm. Mô hình client-server được sử dụng hiệu quả cho HTTP server, xử lý xác thực và quản lý phiên người dùng qua cookie. Mô hình peer-to-peer (P2P) được triển khai thành công cho ứng dụng chat; trong đó, server trung tâm đóng vai trò tracker để quản lý danh sách peer, còn việc nhắn tin 1-1 và chat nhóm diễn ra trực tiếp giữa các peer.

Nhóm đã thành công trong việc thiết kế và phân định vai trò của các tiến trình (server, client, peer) trong một hệ thống phân tán, cũng như cách thiết kế một giao thức P2P đơn giản qua TCP/IP; quản lý kết nối đa luồng và đồng bộ trạng thái giữa các peer. Hệ thống hoàn thành đã đáp ứng được các yêu cầu về chức năng, từ xác thực cho đến chat 1-1 và chat nhóm.



## Tài liệu tham khảo

- [1] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. 8th. Boston, MA: Pearson, 2021. ISBN: 978-0136681557.
- [2] PEP Editors. *Python Enhancement Proposals (PEPs) — PEP Index*. <https://peps.python.org/>. Accessed: 2025-10-12.