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

MẠNG MÁY TÍNH (CO3094)

Assignment 1 – Implement HTTP Server and Chat Application

GVHD: ThS. Bùi Xuân Giang

SV thực hiện: Nguyễn Ngọc Trúc Quỳnh – 2312912
Bành Tiểu My – 2312137
Huỳnh Cẩm Ly - 2312008
Huỳnh Anh Chí Kiệt - 2013563

TP. Hồ Chí Minh, Tháng 11/2025

Mục lục

Bảng 1: *Danh sách thành viên & Nhiệm vụ*

STT	Họ và tên	MSSV	% Hoàn thành
1	Huỳnh Cẩm Ly	2312008	100%
2	Huỳnh Anh Chí Kiệt	2013563	100%
3	Bành Tiểu My	2312137	100%
4	Nguyễn Ngọc Trúc Quỳnh	2312912	100%

1 Giới thiệu

1.1 Mục tiêu của bài tập

Bài tập lớn số 1 của môn **Mạng máy tính (CO3094)** yêu cầu sinh viên **xây dựng một hệ thống HTTP Server và ứng dụng Chat** hoạt động dựa trên giao thức **TCP/IP**. Mục tiêu của bài tập nhằm giúp sinh viên:

- Hiểu và vận dụng được nguyên lý hoạt động của mô hình Client–Server trong giao tiếp mạng.
- Tự thiết kế và cài đặt được một **framework web cơ bản** mà không sử dụng bất kỳ thư viện web có sẵn (như Flask, Django, hay FastAPI).
- Nắm được cơ chế **xác thực người dùng bằng Cookie** và xử lý phiên làm việc (session management).
- Xây dựng được một **ứng dụng chat lai (hybrid chat)** hỗ trợ cả mô hình Client–Server và Peer–to–Peer.
- Phát triển kỹ năng lập trình socket, đồng thời củng cố kiến thức về giao thức HTTP và tầng Transport trong mô hình TCP/IP.

1.2 Yêu cầu thực hiện

Nhóm sinh viên cần hiện thực toàn bộ hệ thống từ tầng giao thức mạng (TCP socket) đến tầng ứng dụng, bao gồm:

1. **HTTP Server**: xử lý các request/response, quản lý session, và cung cấp dịch vụ web cơ bản.
2. **Cơ chế xác thực**: triển khai hệ thống quản lý cookie và phân quyền truy cập người dùng.
3. **Chat Application**: hỗ trợ nhắn tin theo hai chế độ — *client–server* và *peer–to–peer (P2P)*.
4. **Báo cáo kỹ thuật**: mô tả đầy đủ phần lý thuyết, thiết kế, giải thuật, cài đặt và kiểm thử hệ thống.

1.3 Công cụ và môi trường phát triển

- **Ngôn ngữ lập trình:** Python 3.13.5.
- **Công nghệ sử dụng:** Socket, Threading, HTML/CSS, JSON.
- **Môi trường thử nghiệm:** Windows 11.
- **Trình duyệt kiểm thử:** Google Chrome.

1.4 Kết quả mong đợi

Sau khi hoàn thành, nhóm sinh viên có thể:

- Hiểu sâu về nguyên lý hoạt động của HTTP, TCP socket và quản lý kết nối mạng.
- Tự xây dựng được framework web thuần từ socket.
- Phát triển được ứng dụng chat có tính tương tác thời gian thực.
- Củng cố kỹ năng lập trình hệ thống và kỹ năng làm việc nhóm trong môi trường phát triển phần mềm.

2 Lý thuyết

2.1 Kiến trúc Client–Server và TCP Socket

2.1.1 Mô hình Client–Server

Mô hình Client–Server là kiến trúc mạng phổ biến nhất trong các ứng dụng phân tán, trong đó:

- **Server (máy chủ):** Là thực thể cung cấp dịch vụ, lắng nghe và xử lý các yêu cầu từ client. Server thường hoạt động liên tục (always-on), có địa chỉ IP cố định và khả năng xử lý đồng thời nhiều kết nối.
- **Client (máy khách):** Là thực thể yêu cầu dịch vụ từ server. Client khởi tạo kết nối, gửi request và nhận response từ server.

Ưu điểm của mô hình này bao gồm: tập trung hóa quản lý tài nguyên, dễ bảo trì và nâng cấp, kiểm soát bảo mật tốt hơn. Tuy nhiên, server có thể trở thành điểm nghẽn (bottleneck) khi có quá nhiều client truy cập đồng thời.

2.1.2 TCP Socket

Socket là giao diện lập trình ứng dụng (API) cho phép các tiến trình trên các máy khác nhau giao tiếp qua mạng. TCP socket hoạt động dựa trên giao thức TCP (Transmission Control Protocol), cung cấp kênh truyền tin cậy, có kết nối (connection-oriented).

Các bước thiết lập kết nối TCP:

1. **Server:** Tạo socket, gắn vào địa chỉ IP và cổng (bind), lắng nghe (listen), chấp nhận kết nối (accept).
2. **Client:** Tạo socket, kết nối tới địa chỉ server (connect).
3. **Three-way handshake:**
   - Client gửi gói SYN (synchronize) tới server.
   - Server phản hồi bằng SYN-ACK (synchronize-acknowledge).
   - Client gửi ACK (acknowledge) để hoàn tất kết nối.
4. Sau khi kết nối được thiết lập, cả hai bên có thể trao đổi dữ liệu qua các hàm send/recv.

Đặc điểm của TCP:

- *Tin cậy (Reliable):* Đảm bảo dữ liệu được truyền đầy đủ, đúng thứ tự nhờ cơ chế ACK và retransmission.
- *Có kết nối (Connection-oriented):* Yêu cầu thiết lập kết nối trước khi truyền dữ liệu.
- *Kiểm soát luồng (Flow control):* Tránh tràn bộ đệm của receiver.
- *Kiểm soát tắc nghẽn (Congestion control):* Điều chỉnh tốc độ gửi dựa trên tình trạng mạng.

2.2 Nguyên lý HTTP Protocol: request, response, header, cookie

2.2.1 HTTP Protocol

HTTP (HyperText Transfer Protocol) là giao thức tầng ứng dụng được sử dụng để truyền tải tài liệu hypermedia (như HTML) trên World Wide Web. HTTP hoạt động theo mô hình request-response giữa client và server.

Đặc điểm chính của HTTP:

- *Stateless (không trạng thái):* Mỗi request độc lập, server không lưu trữ thông tin về các request trước đó.
- *Text-based protocol:* Dữ liệu được mã hóa dưới dạng văn bản, dễ đọc và debug.
- *Sử dụng TCP:* HTTP hoạt động trên nền TCP, thường dùng cổng 80 (HTTP) hoặc 443 (HTTPS).

2.2.2 HTTP Request

HTTP Request là thông điệp mà client gửi tới server để yêu cầu một tài nguyên hoặc thực hiện một hành động. Cấu trúc của HTTP Request gồm:

```
<METHOD> <URI> <HTTP_VERSION>
<Header-Name>: <Header-Value>
...
[blank line]
<Body>
```

Các phương thức HTTP phổ biến:

- `GET`: Lấy tài nguyên từ server (không có body).
- `POST`: Gửi dữ liệu tới server để tạo/cập nhật tài nguyên.
- `PUT`: Cập nhật toàn bộ tài nguyên.
- `DELETE`: Xóa tài nguyên.
- `HEAD`: Tương tự GET nhưng chỉ lấy header, không lấy body.

Ví dụ HTTP GET Request:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
Cookie: session_id=abc123
```

2.2.3 HTTP Response

HTTP Response là thông điệp mà server gửi trả lại client sau khi xử lý request. Cấu trúc:

```
<HTTP_VERSION> <STATUS_CODE> <REASON_PHRASE>
<Header-Name>: <Header-Value>
...
[blank line]
<Body>
```

Các mã trạng thái HTTP (Status Code):

- `1xx (Informational)`: Thông tin, xử lý đang tiếp tục.
- `2xx (Success)`: Request thành công (200 OK, 201 Created).
- `3xx (Redirection)`: Yêu cầu chuyển hướng (301 Moved Permanently, 302 Found).
- `4xx (Client Error)`: Lỗi từ phía client (400 Bad Request, 404 Not Found, 401 Unauthorized).
- `5xx (Server Error)`: Lỗi từ phía server (500 Internal Server Error, 503 Service Unavailable).

Ví dụ HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1234
Set-Cookie: session_id=xyz789; Path=/; HttpOnly

<!DOCTYPE html>
<html>...
```

2.2.4 HTTP Headers

HTTP Headers chứa metadata về request/response. Các loại header quan trọng:

- **General Headers:** `Date`, `Connection`, `Cache-Control`
- **Request Headers:** `Host`, `User-Agent`, `Accept`, `Authorization`, `Cookie`
- **Response Headers:** `Server`, `Set-Cookie`, `Location`
- **Entity Headers:** `Content-Type`, `Content-Length`, `Content-Encoding`

2.2.5 HTTP Cookies

Cookie là một đoạn dữ liệu nhỏ mà server gửi tới client và được client lưu trữ cục bộ. Mỗi lần gửi request tới cùng một server, client sẽ tự động gửi kèm cookie trong header `Cookie`.

Cách thức hoạt động:

1. Server gửi cookie qua header `Set-Cookie` trong response:

   ```
   Set-Cookie: session_id=abc123; Path=/; HttpOnly; Max-Age=3600
   ```

2. Browser lưu cookie và tự động gửi lại trong các request tiếp theo:

   ```
   Cookie: session_id=abc123
   ```

Các thuộc tính của Cookie:

- `Path`: Đường dẫn mà cookie có hiệu lực.
- `Domain`: Tên miền áp dụng cookie.
- `Max-Age / Expires`: Thời gian sống của cookie.
- `HttpOnly`: Cookie chỉ có thể truy cập qua HTTP, không qua JavaScript (bảo mật).
- `Secure`: Cookie chỉ được gửi qua HTTPS.
- `SameSite`: Kiểm soát cookie trong các request cross-site (chống CSRF).

2.3 Cơ chế Cookie-based Authentication

2.3.1 Vấn đề xác thực trong HTTP

Do HTTP là giao thức stateless, server không thể nhớ client đã đăng nhập hay chưa giữa các request. Cookie-based authentication giải quyết vấn đề này bằng cách lưu trữ session identifier tại client.

2.3.2 Quy trình xác thực bằng Cookie

1. **Đăng nhập (Login):**
   - Client gửi thông tin đăng nhập (username, password) qua POST request.
   - Server xác thực thông tin. Nếu hợp lệ, server tạo một *session ID* duy nhất.
   - Server lưu session ID vào database/memory cùng thông tin user.
   - Server gửi session ID về client qua header `Set-Cookie`.
2. **Các request tiếp theo:**
   - Client tự động gửi cookie chứa session ID trong mọi request.
   - Server đọc session ID từ cookie, tra cứu trong database để xác định user.

- Nếu session hợp lệ và chưa hết hạn, server xử lý request và trả về kết quả.

3. **Đăng xuất (Logout):**
   - Client gửi request logout.
   - Server xóa session khỏi database/memory.
   - Server gửi cookie với `Max-Age=0` để xóa cookie tại client.

2.3.3 Ưu và nhược điểm

Ưu điểm:

- Đơn giản, dễ triển khai.
- Cookie tự động được gửi kèm trong request, không cần xử lý thủ công.
- Server có toàn quyền kiểm soát session (có thể hủy bất kỳ lúc nào).

Nhược điểm:

- Dễ bị tấn công XSS (Cross-Site Scripting) nếu không dùng `HttpOnly`.
- Dễ bị tấn công CSRF (Cross-Site Request Forgery) nếu không dùng `SameSite`.
- Server phải lưu trữ session, tốn tài nguyên khi có nhiều người dùng.
- Khó scale trong hệ thống phân tán (cần shared session storage như Redis).

2.4 Mô hình P2P và Hybrid Chat (Client–Server + P2P)

2.4.1 Mô hình Peer-to-Peer (P2P)

Trong kiến trúc P2P, các node (peer) vừa đóng vai trò client, vừa đóng vai trò server. Mỗi peer có thể giao tiếp trực tiếp với peer khác mà không cần thông qua server trung gian.

Đặc điểm của P2P:

- *Phi tập trung (Decentralized):* Không có server trung tâm, giảm single point of failure.
- *Scalability:* Khi thêm peer, tổng băng thông và tài nguyên hệ thống tăng lên.
- *Latency thấp:* Giao tiếp trực tiếp giữa các peer, không qua trung gian.

Thách thức:

- Khó khăn trong việc tìm kiếm và kết nối giữa các peer (peer discovery).
- Vấn đề NAT traversal: các peer sau firewall/NAT khó kết nối trực tiếp.
- Bảo mật và tin cậy kém hơn do không có cơ quan quản lý trung tâm.

2.4.2 Mô hình Hybrid Chat (Client–Server + P2P)

Để kết hợp ưu điểm của cả hai mô hình, ứng dụng chat hybrid sử dụng:

- **Client–Server:**
  - Server quản lý danh sách user online, xác thực và lưu trữ tin nhắn lịch sử.
  - Client gửi tin nhắn tới server, server chuyển tiếp (relay) tới người nhận.
  - Phù hợp khi người nhận offline hoặc không thể kết nối trực tiếp.
- **P2P:**
  - Khi cả hai user đều online và có thể kết nối trực tiếp, tin nhắn được gửi P2P.
  - Giảm tải cho server, tăng tốc độ truyền tin, đặc biệt với file lớn.

– Server chỉ cung cấp thông tin địa chỉ IP và cổng để hai peer tự kết nối.

Quy trình hoạt động:

1. User A và User B đều đăng nhập vào server, server lưu thông tin IP:port của cả hai.
2. User A muốn chat với User B:
   - Client A hỏi server về thông tin kết nối của User B.
   - Nếu User B online và có thể kết nối trực tiếp, server cung cấp địa chỉ IP:port của B.
   - Client A thiết lập kết nối P2P trực tiếp tới Client B.
   - Tin nhắn được truyền trực tiếp giữa A và B.
3. Nếu kết nối P2P thất bại (do NAT, firewall), hệ thống tự động fallback về chế độ relay qua server.

2.5 Vai trò của TCP/IP trong giao tiếp mạng

2.5.1 Mô hình TCP/IP

Mô hình TCP/IP (hoặc Internet Protocol Suite) là tập hợp các giao thức được sử dụng trên Internet, gồm 4 tầng:

1. **Tầng Liên kết (Link Layer):** Ethernet, Wi-Fi, ARP – xử lý truyền dữ liệu vật lý trong mạng LAN.
2. **Tầng Mạng (Internet Layer):** IP, ICMP – định tuyến gói tin giữa các mạng.
3. **Tầng Giao vận (Transport Layer):** TCP, UDP – truyền dữ liệu giữa các ứng dụng.
4. **Tầng Ứng dụng (Application Layer):** HTTP, FTP, SMTP, DNS – cung cấp dịch vụ cho người dùng.

2.5.2 Vai trò của TCP trong hệ thống

Trong bài tập này, TCP đảm nhận vai trò then chốt:

- **HTTP Server:** Sử dụng TCP socket để lắng nghe các request từ browser, đảm bảo dữ liệu HTML/CSS/JS được truyền đầy đủ và đúng thứ tự.
- **Chat Application:** Sử dụng TCP để truyền tin nhắn, đảm bảo không bị mất mát hay trùng lặp.
- **Cookie và Session:** Dựa vào tính tin cậy của TCP để đảm bảo session ID được truyền chính xác.

2.5.3 So sánh TCP và UDP

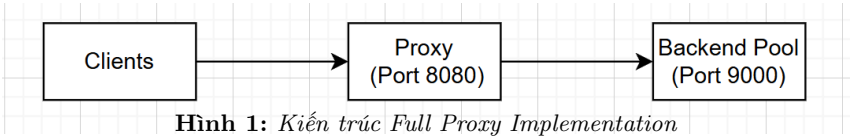
Bảng 2: *So sánh TCP và UDP*

Tiêu chí	**TCP**	**UDP**
Độ tin cậy	Tin cậy (có ACK, retransmission)	Không tin cậy (có thể mất gói)
Kết nối	Connection-oriented (cần thiết lập kết nối)	Connectionless (không cần thiết lập)
Thứ tự	Đảm bảo thứ tự gói tin	Không đảm bảo
Tốc độ	Chậm hơn (do overhead)	Nhanh hơn
Ứng dụng	HTTP, FTP, Email, Chat	Streaming, VoIP, Gaming, DNS

Với yêu cầu của bài tập là xây dựng HTTP server và chat application đảm bảo tin cậy, TCP là lựa chọn phù hợp.

3 Thiết kế hệ thống

3.1 Kiến trúc tổng thể



Hình 1: *Kiến trúc Full Proxy Implementation*

Các thành phần chính:

- **Proxy Server (Port 8080):** Forward requests từ clients đến backend server
- **Backend Server (Port 9000):** Xử lý authentication, chat, API endpoints
- **Routing Configuration:** File `config/proxy.conf` với cú pháp giống nginx
- **Dual-mode Support:** Hỗ trợ cả hai chế độ truy cập:
  - Mode 1: `http://localhost:8080/*` (qua Proxy) — Đúng theo Figure 1
  - Mode 2: `http://localhost:9000/*` (trực tiếp Backend) — Fallback/testing

Cấu hình Proxy (config/proxy.conf):

```
host "localhost:8080" {
    proxy_pass http://127.0.0.1:9000;
}

host "127.0.0.1:8080" {
    proxy_pass http://127.0.0.1:9000;
}
```

Cách khởi chạy hệ thống:

1. Khởi động Backend Server:

   ```
   python start_backend.py --server-port 9000
   ```

2. Khởi động Proxy Server:

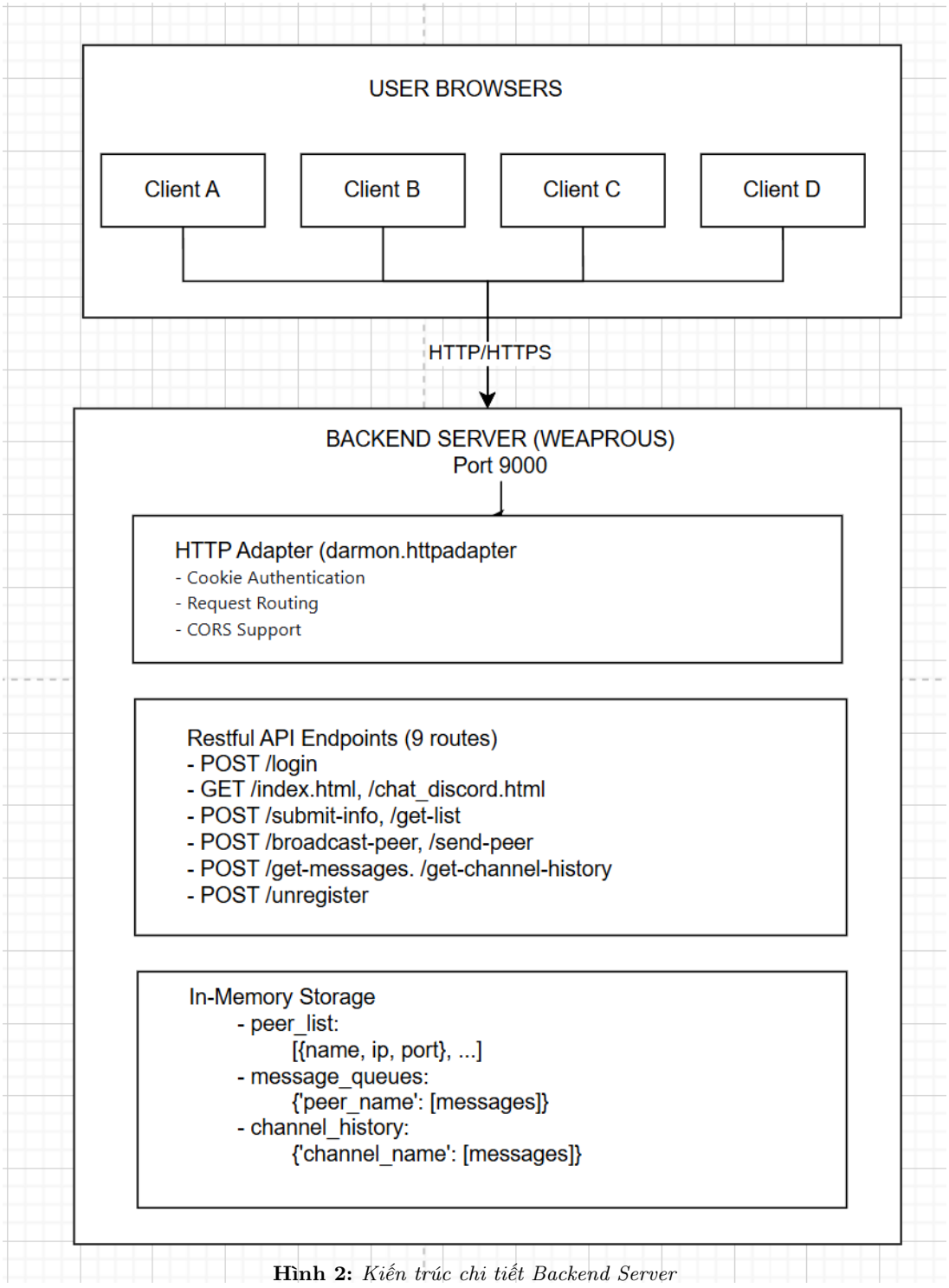
   ```
   python start_proxy.py --server-port 8080
   ```

3. Truy cập ứng dụng:
   - Qua Proxy (khuyến nghị): `http://localhost:8080/login.html`
   - Trực tiếp Backend (testing): `http://localhost:9000/login.html`

Kết luận: Nhóm chọn kiến trúc Full Proxy vì: (1) tuân thủ đúng Figure 1 trong PDF, (2) Task 2.3 không có câu "no proxy needed", (3) sẵn sàng demo nếu giáo viên yêu cầu, (4) vẫn giữ backward compatibility cho testing.

3.1.1 Mô hình Client-Server kết hợp P2P

Hệ thống kết hợp cả hai mô hình Client-Server và Peer-to-Peer để tối ưu hiệu năng:



Hình 2: *Kiến trúc chi tiết Backend Server*

Đặc điểm của kiến trúc:

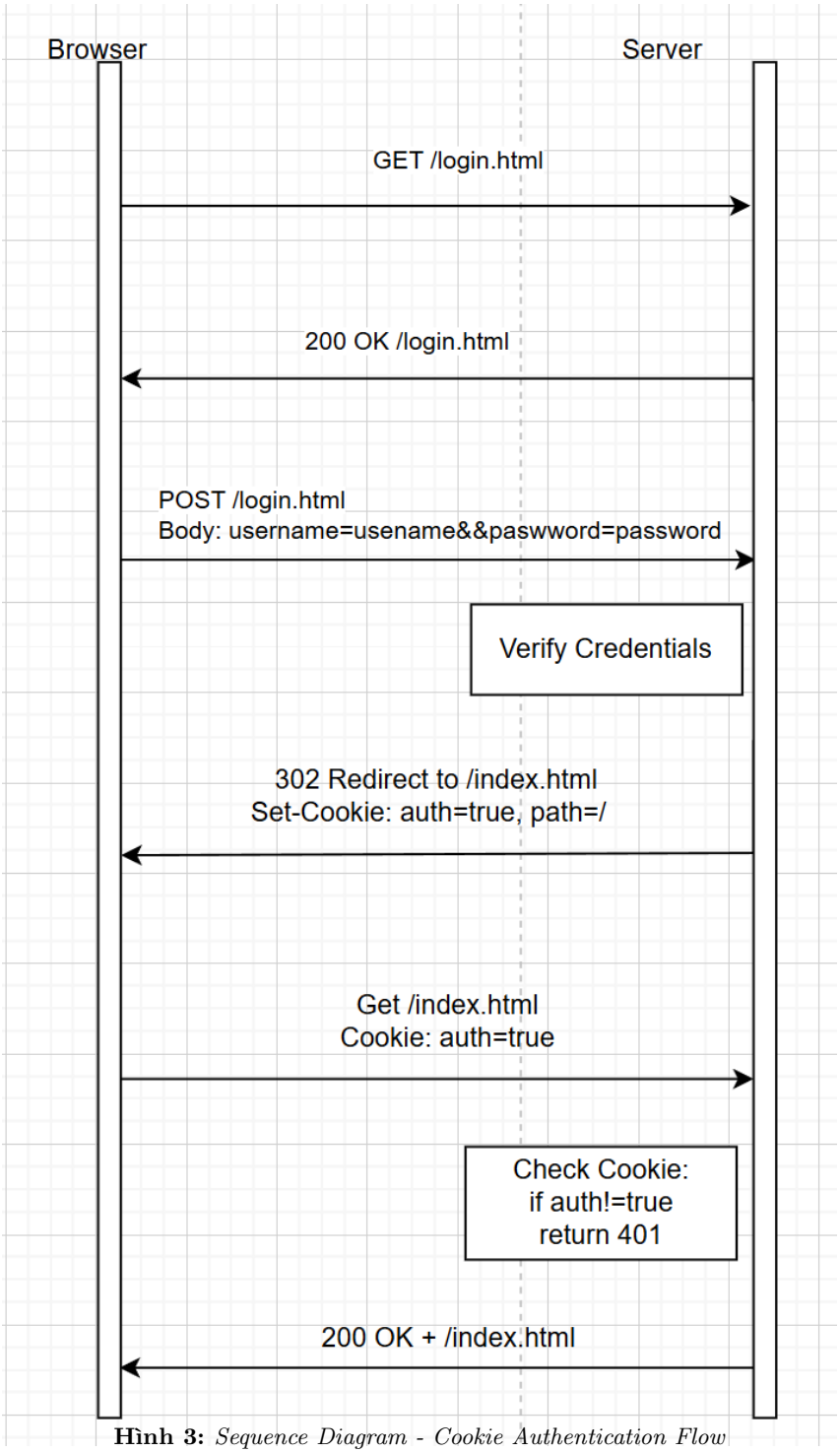
- **Client-Server:** Xử lý authentication, peer discovery, message relay
- **P2P Concept:** Mỗi client đăng ký thông tin (IP:Port) để có khả năng nhắn tin trực tiếp
- **Stateful:** Server lưu danh sách peers và message queues trong RAM

- **Real-time:** Polling-based với chu kỳ 1.5 giây để fetch messages mới

3.2 Luồng xử lý HTTP

3.2.1 HTTP Cookie Authentication Flow

Quy trình xác thực người dùng sử dụng cookie được mô tả trong sequence diagram sau:



Hình 3: *Sequence Diagram - Cookie Authentication Flow*

Các bước thực hiện:

1. Client gửi `GET /login.html` để lấy trang đăng nhập

2. Server trả về file `login.html` với status code `200 OK`
3. User nhập thông tin và submit form, client gửi `POST /login` với body chứa `username` và `password`
4. Server xác thực credentials:
   - Kiểm tra `username == "admin"` và `password == "password"`
   - Nếu đúng: set cookie `auth=true` và redirect `302` tới `/index.html`
   - Nếu sai: trả về `401 Unauthorized`
5. Client tự động gửi `GET /index.html` kèm theo `Cookie: auth=true`
6. Server kiểm tra cookie:
   - Nếu `auth == "true"`: trả về `index.html` với status `200 OK`
   - Nếu không có hoặc sai: trả về `401 Unauthorized`

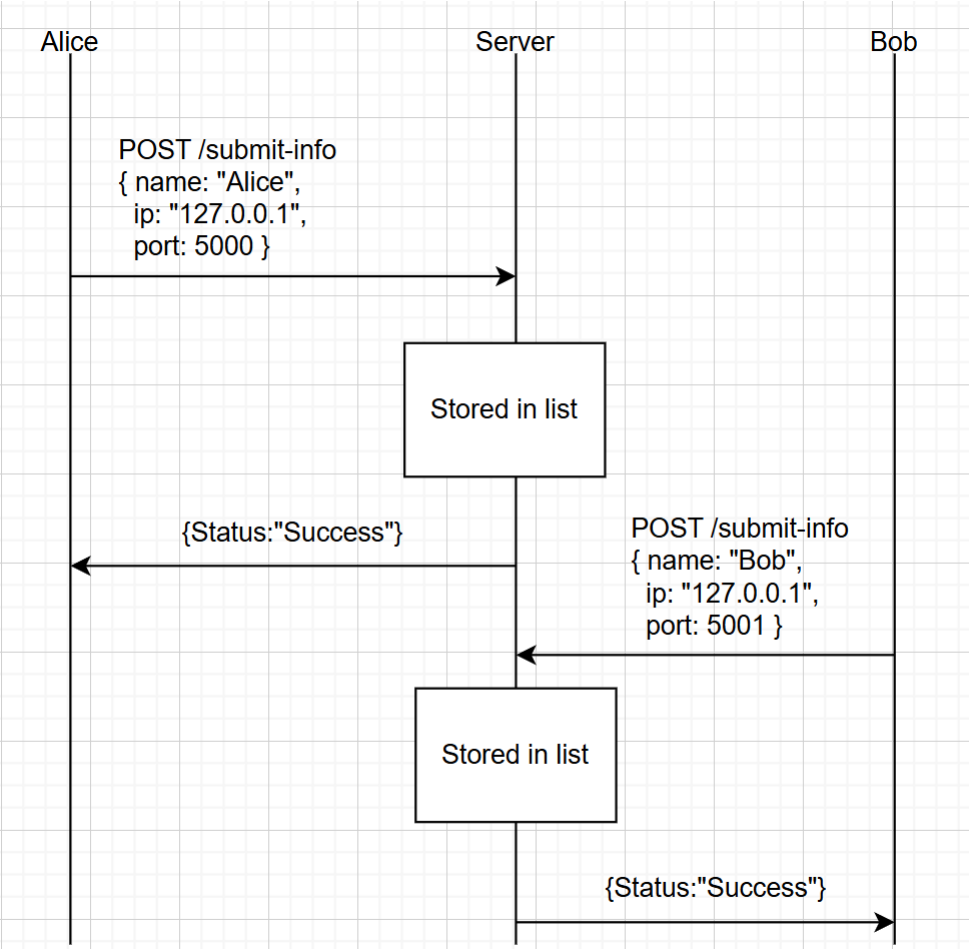
Các điểm chính:

- Cookie `auth=true` được set sau khi login thành công
- Backend kiểm tra cookie trước khi serve `/index.html` và `/chat_discord.html`
- Cookie scope: `path=/` (áp dụng cho toàn bộ site)
- Session management: In-memory (cookie persistent trong browser)
- Credentials mặc định: `username="admin"`, `password="password"`

3.3 Thiết kế ChatApp

3.3.1 Peer Registration Flow

Quy trình đăng ký peer vào hệ thống:



Hình 4: *Peer Registration Sequence*

Mỗi client khi vào ứng dụng chat sẽ tự động gọi API `POST /submit-info` để đăng ký thông tin (tên, IP, port) vào server. Server lưu trữ danh sách này trong `peer_list` để các client khác có thể tra cứu.

Request format:

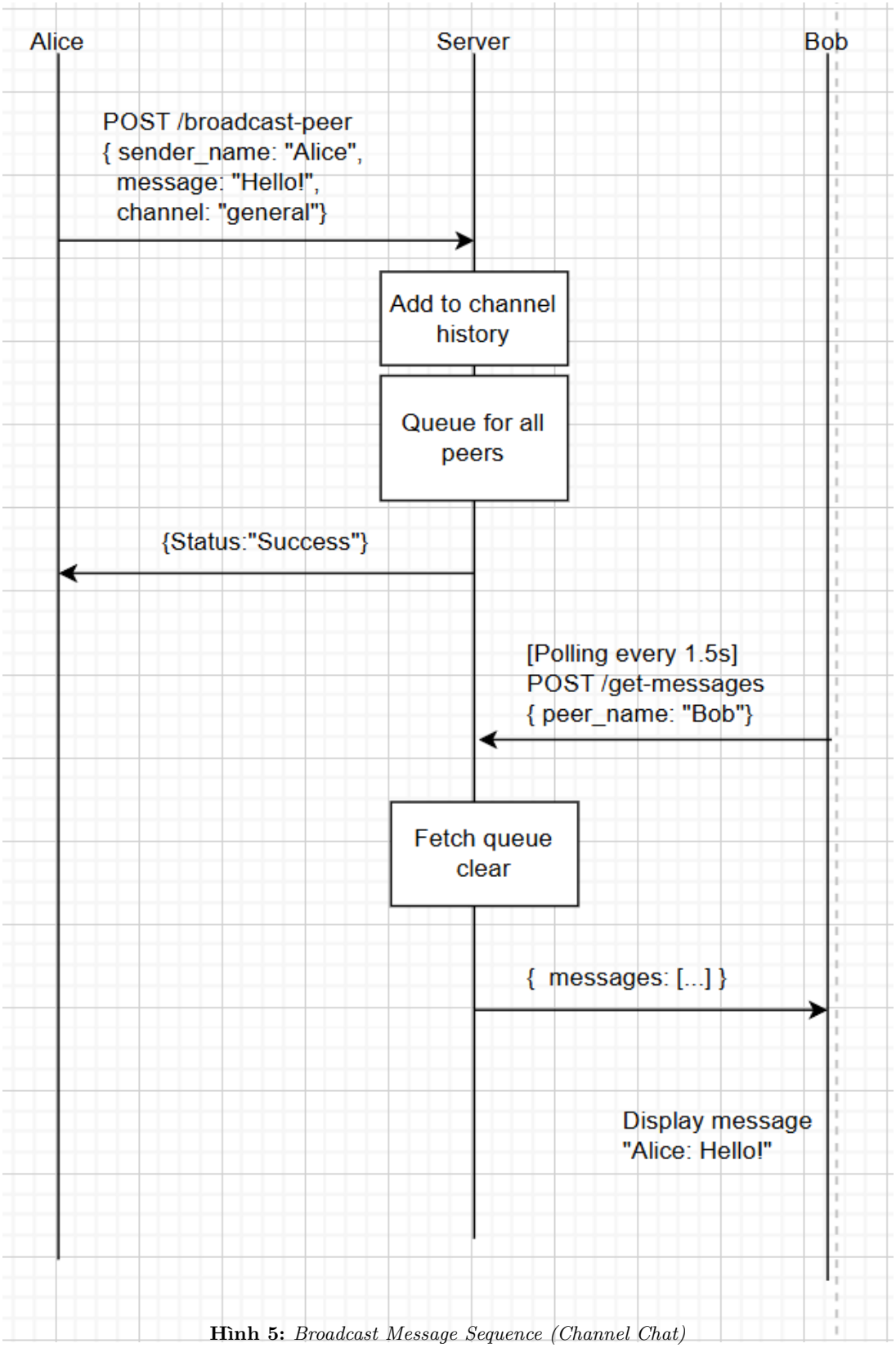
```
POST /submit-info
Content-Type: application/json

{
    "name": "Alice",
    "ip": "127.0.0.1",
    "port": 5000
}
```

Response:

```
{
    "status": "success"
}
```

3.3.2 Broadcast Message Flow (Broadcast Channel)

Quy trình gửi tin nhắn broadcast tới tất cả peers trong một channel:



Hình 5: *Broadcast Message Sequence (Channel Chat)*

Cơ chế hoạt động:

1. Alice gửi tin nhắn broadcast qua API `POST /broadcast-peer`
2. Server lưu tin nhắn vào `channel_history[channel_name]`

3. Server thêm tin nhắn vào hàng đợi (`message_queues`) của TẤT CẢ peers đang online
4. Bob polling server mỗi 1.5s qua API `POST /get-messages`
5. Server trả về tất cả messages trong queue của Bob và xóa queue
6. Bob hiển thị tin nhắn mới trên giao diện

API Request (Broadcast):

```
POST /broadcast-peer
Content-Type: application/json

{
    "sender_name": "Alice",
    "message": "Hello everyone!",
    "channel": "general"
}
```

API Request (Polling):

```
POST /get-messages
Content-Type: application/json

{
    "peer_name": "Bob"
}
```

API Response:

```
{
    "messages": [
        {
            "from": "Alice",
            "message": "Hello everyone!",
            "channel": "general",
            "type": "broadcast",
            "timestamp": "2025-11-13T10:30:00"
        }
    ]
}
```

Broadcast Message Flow (P2P Mode – BroadcastChannel) Trong trường hợp Backend Server không khả dụng, ứng dụng tự động chuyển sang chế độ **P2P Broadcast**, sử dụng **BroadcastChannel API** của trình duyệt. Ở chế độ này, việc truyền tin nhắn không thông qua server mà được gửi trực tiếp giữa các tab của cùng một trình duyệt.

- Mỗi tab tạo một kênh:

  ```
  const bc = new BroadcastChannel("chat_channel");
  ```

- Gửi broadcast:

  ```
  bc.postMessage({
      sender: username,
      message: content,
      timestamp: Date.now(),
      type: "broadcast"
  });
  ```

- Nhận và hiển thị tin nhắn:

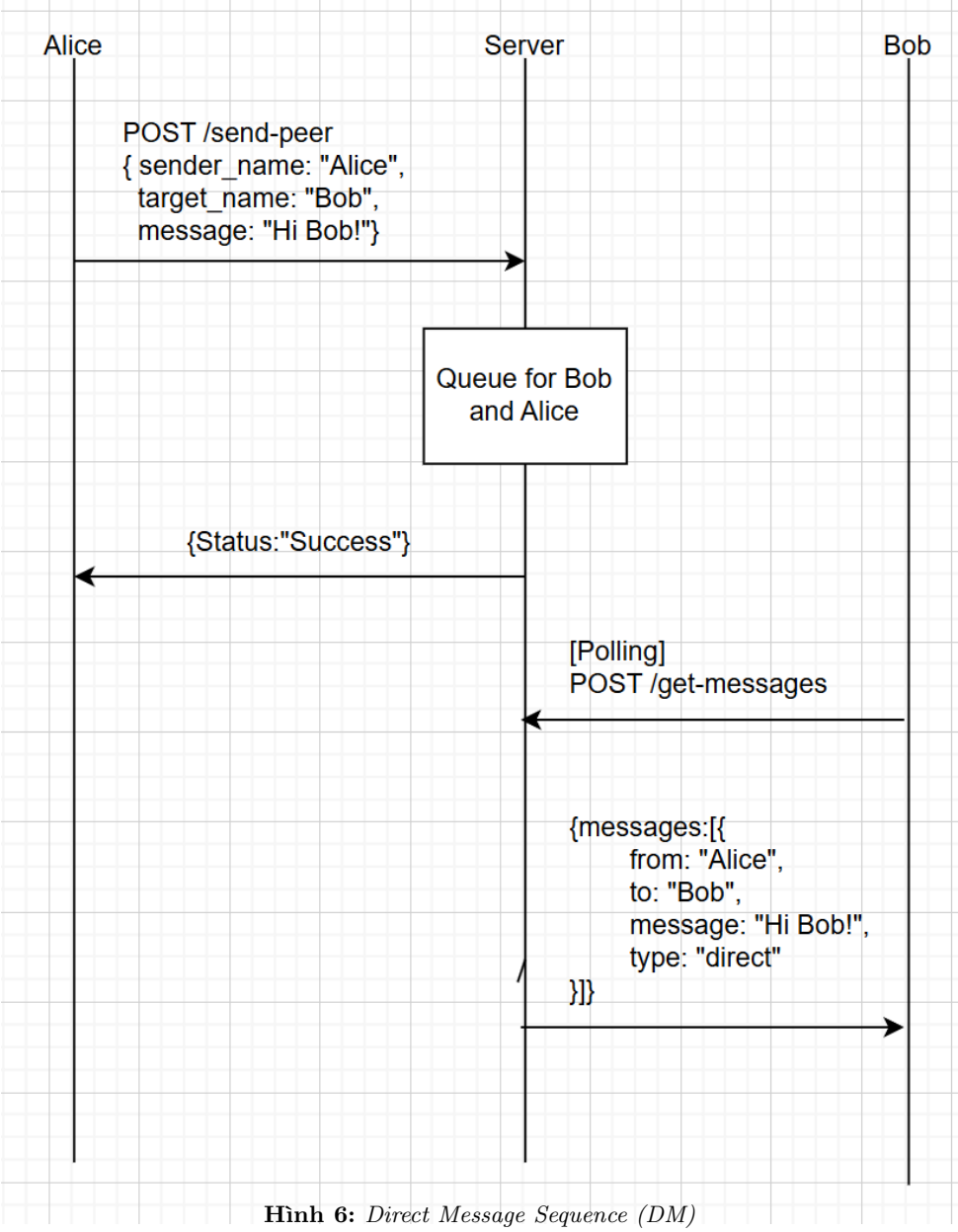
```
bc.onmessage = (event) => {
    displayMessage(event.data);
};
```

- Không có queue, không có channel_history và không có server xử lý.

Hạn chế: Chỉ hoạt động trên cùng thiết bị và cùng trình duyệt (Chrome–Chrome hoặc Firefox–Firefox), không truyền được qua mạng LAN/Internet.

3.3.3 Direct Message Flow (DM)

Quy trình gửi tin nhắn trực tiếp giữa hai peers:



Hình 6: *Direct Message Sequence (DM)*

Đặc điểm của DM:

- Tin nhắn chỉ được thêm vào queue của người gửi (Alice) và người nhận (Bob)

- Không lưu vào `channel_history`
- Tin nhắn có trường `type="direct"` để client phân biệt với broadcast message
- Cả Alice và Bob đều thấy tin nhắn trong conversation của họ

API Request (Send DM):

```
POST /send-peer
Content-Type: application/json

{
    "sender_name": "Alice",
    "target_name": "Bob",
    "message": "Hi Bob, how are you?"
}
```

API Response (Polling by Bob):

```
{
    "messages": [
        {
            "from": "Alice",
            "to": "Bob",
            "message": "Hi Bob, how are you?",
            "type": "direct",
            "timestamp": "2025-11-13T10:35:00"
        }
    ]
}
```

Direct Message Flow (P2P Mode) Ở chế độ P2P fallback, Direct Message cũng được xử lý trên client mà không thông qua server. Tin nhắn được gửi qua cùng kênh BroadcastChannel nhưng được phân loại bằng trường `type="direct"` và `to="<username>"`.

- Gửi DM:

```
bc.postMessage({
    sender: username,
    to: targetName,
    message: content,
    type: "direct",
    timestamp: Date.now()
});
```

- Khi nhận, client tự lọc:

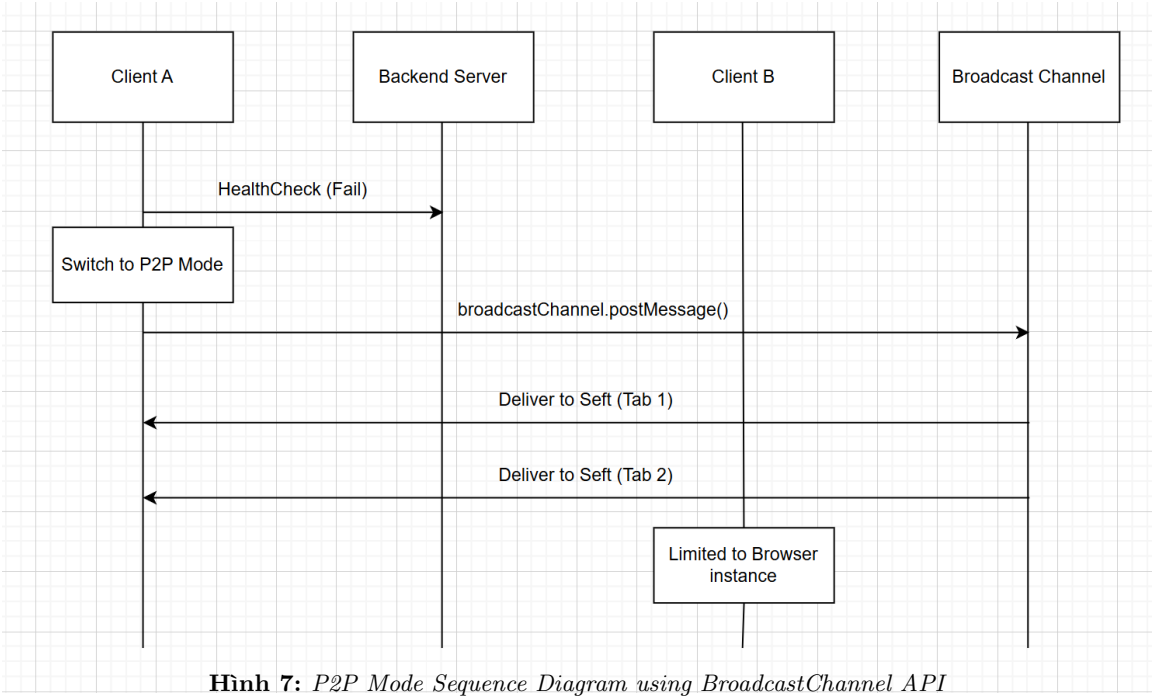
```
bc.onmessage = (event) => {
    if (event.data.type === "direct" &&
        event.data.to === username) {
        displayMessage(event.data);
    }
};
```

- Không lưu lịch sử và không có phân phối tin nhắn theo queue.

Lưu ý: Đây chỉ là mô phỏng P2P nội bộ, đáp ứng yêu cầu bài tập lớn khi server không hoạt động.

3.3.4 P2P Mode – Fallback When Backend is Down

Khi Backend Server ngừng phản hồi (health check thất bại), toàn bộ ứng dụng tự động chuyển sang **P2P Mode** dựa trên BroadcastChannel API. Cơ chế này cho phép các tab trong cùng một trình duyệt tiếp tục gửi và nhận tin nhắn nội bộ, đảm bảo hệ thống không bị gián đoạn khi demo.



Hình 7: *P2P Mode Sequence Diagram using BroadcastChannel API*

Ghi chú quan trọng:

- BroadcastChannel chỉ hoạt động trong **cùng một trình duyệt**.
- Máy khác hoặc trình duyệt khác **không nhận được**.
- Đây không phải P2P thật, chỉ là cơ chế **fallback an toàn**.

3.3.5 API Endpoints Summary

Bảng 3: *Danh sách API Endpoints*

Method	Endpoint	Mô tả
POST	/login	Xác thực người dùng, set cookie `auth=true`
GET	/index.html	Trang chính (yêu cầu authentication)
GET	/chat_discord.html	Trang chat Discord-like (yêu cầu auth)
POST	/submit-info	Đăng ký peer info (name, ip, port)
POST	/get-list	Lấy danh sách tất cả peers đang online
POST	/broadcast-peer	Gửi tin broadcast tới tất cả peers trong channel
POST	/send-peer	Gửi tin nhắn DM cho peer cụ thể
POST	/get-messages	Polling để lấy tin nhắn mới (1.5s interval)
POST	/get-channel-history	Lấy lịch sử chat của một channel
POST	/unregister	Xóa peer khỏi danh sách (khi logout)

3.4 Cấu trúc dữ liệu In-Memory

Server sử dụng các cấu trúc dữ liệu Python để lưu trữ thông tin trong RAM:

1. Peer List:

```
peer_list = [
    {"name": "Alice", "ip": "127.0.0.1", "port": 5000},
    {"name": "Bob", "ip": "127.0.0.1", "port": 5001},
    {"name": "Charlie", "ip": "192.168.1.10", "port": 5002},
    ...
]
```

Mô tả: Danh sách tất cả peers đang online. Mỗi peer có 3 trường: `name` (tên người dùng), `ip` (địa chỉ IP), `port` (cổng kết nối P2P).

2. Message Queues:

```
message_queues = {
    "Alice": [
        {"from": "Bob", "message": "Hi!", "type": "direct"},
        {"from": "Charlie", "message": "Hello", "channel": "general"}
    ],
    "Bob": [
        {"from": "Alice", "message": "Hey Bob", "type": "direct"}
    ],
    ...
}
```

Mô tả: Dictionary lưu hàng đợi tin nhắn cho từng peer. Key là tên peer, value là list các tin nhắn chưa được fetch. Khi peer gọi `/get-messages`, server trả về toàn bộ queue và xóa trống.

3. Channel History:

```
channel_history = {
    "general": [
        {"sender": "Alice", "message": "Hello!", "timestamp": "..."},
        {"sender": "Bob", "message": "Hi there", "timestamp": "..."}
    ],
    "random": [
        {"sender": "Charlie", "message": "Random msg", "timestamp": "..."}
    ],
    ...
}
```

Mô tả: Dictionary lưu lịch sử tin nhắn của từng channel. Được sử dụng khi peer mới join channel và muốn load lịch sử chat trước đó qua API `/get-channel-history`.

Đánh giá:

- *Ưu điểm:* Truy xuất nhanh O(1) cho dictionary lookup, implementation đơn giản
- *Nhược điểm:* Dữ liệu mất khi restart server (không persistent), không scale cho hệ thống lớn
- *Cải tiến:* Có thể chuyển sang Redis hoặc database để persistent storage

4 Hiện thực (Implementation)

4.1 Backend Implementation

4.1.1 Cấu trúc thư mục

Hệ thống backend được tổ chức theo mô hình modular với cấu trúc thư mục rõ ràng:

```
BTL1_MMT_FINAL/
 start_backend.py          # Entry point - khởi động server
 daemon/
    __init__.py
    backend.py            # Socket server + threading
```

```
    httpadapter.py        # HTTP routing + cookie auth
    request.py            # HTTP request parser
    response.py           # HTTP response builder
    weaprous.py           # RESTful framework
    utils.py              # Utility functions
    dictionary.py         # Helper dictionaries
www/                        # HTML pages
static/                     # CSS, JS, images
config/                     # Configuration files
```

4.1.2 Core Components

HTTP Server (daemon/backend.py) Module `backend.py` triển khai TCP socket server với khả năng xử lý đa luồng (multi-threading):

```python
def run_backend(ip, port, routes):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((ip, port))
    server.listen(50)  # Max 50 concurrent connections

    while True:
        conn, addr = server.accept()
        # Multi-threading cho mỗi client
        client_thread = threading.Thread(
            target=handle_client,
            args=(ip, port, conn, addr, routes),
            daemon=True
        )
        client_thread.start()
```

Đặc điểm chính:

- Sử dụng TCP socket cho kết nối tin cậy
- Multi-threaded: Mỗi connection được xử lý bởi một thread riêng biệt
- Daemon threads: Tự động cleanup khi main thread kết thúc
- Hỗ trợ tối đa 50 kết nối đồng thời (backlog queue)

HTTP Request Parser (daemon/request.py) Module `request.py` phân tích cú pháp HTTP request và trích xuất thông tin:

```python
def prepare(self, raw_request):
    # Parse request line
    request_line = lines[0].split()
    self.method = request_line[0]
    self.url = request_line[1]
    self.version = request_line[2]

    # Parse headers
    for header_line in header_lines:
        key, value = header_line.split(':', 1)
        self.headers[key.strip()] = value.strip()

    # Parse cookies from Cookie header
    cookie_header = self.headers.get('cookie', '')
    for cookie_pair in cookie_header.split(';'):
        key, value = cookie_pair.strip().split('=', 1)
        self.cookies[key] = value

    # Parse body (POST data)
    self.body = raw_request.split('\r\n\r\n', 1)[1]
```

Đối tượng `Request` lưu trữ đầy đủ thông tin:

- `method`: Phương thức HTTP (GET, POST, PUT, DELETE)
- `url`: Đường dẫn resource được yêu cầu
- `headers`: Dictionary chứa tất cả HTTP headers
- `cookies`: Dictionary chứa cookies từ client
- `body`: Nội dung request body (dành cho POST/PUT)

Cookie Authentication (daemon/httpadapter.py) Module `httpadapter.py` triển khai cơ chế xác thực dựa trên cookie:

```
# Login endpoint
if req.method == 'POST' and req.url == '/login':
    # Parse credentials
    username = form_data.get('username')
    password = form_data.get('password')

    if username == 'admin' and password == 'password':
        # Set cookie
        response = (
            "HTTP/1.1 302 Found\r\n"
            "Location: /index.html\r\n"
            "Set-Cookie: auth=true; path=/\r\n"
            "Connection: close\r\n\r\n"
        ).encode('utf-8')
        conn.sendall(response)
        return

# Protected pages
if req.method == 'GET' and req.url == '/index.html':
    auth_cookie = req.cookies.get('auth', '')

    if auth_cookie != 'true':
        # Redirect to login
        response = (
            "HTTP/1.1 302 Found\r\n"
            "Location: /login.html\r\n"
            "Connection: close\r\n\r\n"
        ).encode('utf-8')
        conn.sendall(response)
        return
```

Quy trình xác thực:

1. User gửi credentials qua POST `/login`
2. Server verify: `username == "admin"` và `password == "password"`
3. Nếu hợp lệ: Server set cookie `auth=true` và redirect tới `/index.html`
4. Các trang protected (`/index.html`, `/chat_discord.html`) kiểm tra cookie
5. Nếu không có cookie hợp lệ: Redirect về `/login.html`

RESTful API Endpoints (start_backend.py) Hệ thống cung cấp 7 API endpoints chính cho ứng dụng chat:

Bảng 4: *RESTful API Endpoints*

Endpoint	Method	Mô tả	Request Body	Response
/submit-info	POST	Đăng ký peer mới	name, ip, port	status, message
/get-list	GET	Lấy danh sách peers	-	peers:[], count
/broadcast-peer	POST	Gửi message channel	sender_name, message, channel	status, peer_count
/send-peer	POST	Gửi DM	sender_name, target_name, message	status, message
/get-messages	POST	Fetch messages mới	peer_name	status, messages:[]
/get-channel-history	POST	Load channel history	channel	status, messages:[]
/unregister	POST	Logout peer	name	status, message

Data Structures Server sử dụng 3 cấu trúc dữ liệu chính trong memory:

```
# Global in-memory storage
peer_list = [
    {
        'name': 'Alice',
        'ip': '127.0.0.1',
        'port': '5000'
    },
    {
        'name': 'Bob',
        'ip': '127.0.0.1',
        'port': '5001'
    }
]

message_queues = {
    'Alice': [
        {
            'from': 'Bob',
            'to': 'Alice',
            'message': 'Hi Alice!',
            'type': 'direct',
            'timestamp': 1699700000.0
        }
    ],
    'Bob': []
}

channel_history = {
    'general': [
        {
            'from': 'Alice',
            'message': 'Hello everyone!',
            'type': 'broadcast',
            'channel': 'general',
            'timestamp': 1699700000.0
        }
    ],
    'random': [],
    'tech': []
}
```

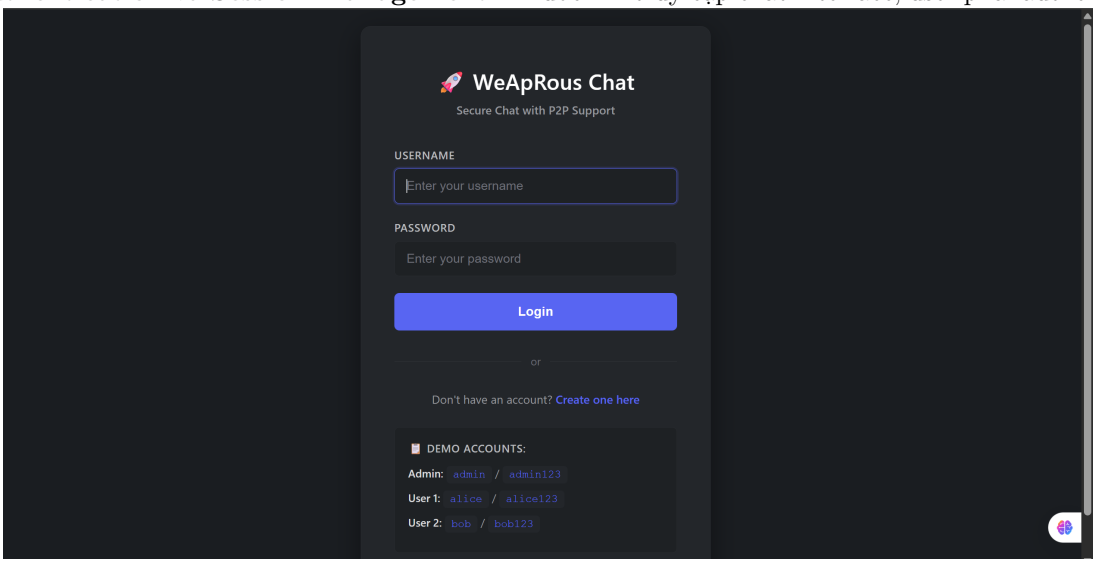
Giải thích:

- `peer_list`: List chứa thông tin (name, IP, port) của tất cả peers đang online
- `message_queues`: Dictionary ánh xạ peer name → list tin nhắn chưa đọc. Sau khi client fetch, queue được xóa trống

- `channel_history`: Dictionary ánh xạ channel name → list lịch sử tin nhắn. Persistent trong session để hỗ trợ load history khi join channel

4.2 Frontend Implementation

Authentication và Session Management Trước khi truy cập chat interface, user phải authenticate:



Login Flow:

- User nhập credentials tại `/login`
- POST request đến backend với username/password
- Server validate và trả về session identifier
- Redirect đến chat interface sau khi login thành công

Session Persistence:

- Session info lưu trong browser (cookie/localStorage)
- Mỗi API request gửi kèm session identifier
- Auto-logout khi session expired hoặc backend unavailable

4.2.1 Chat UI Implementation (www/chat_discord.html)

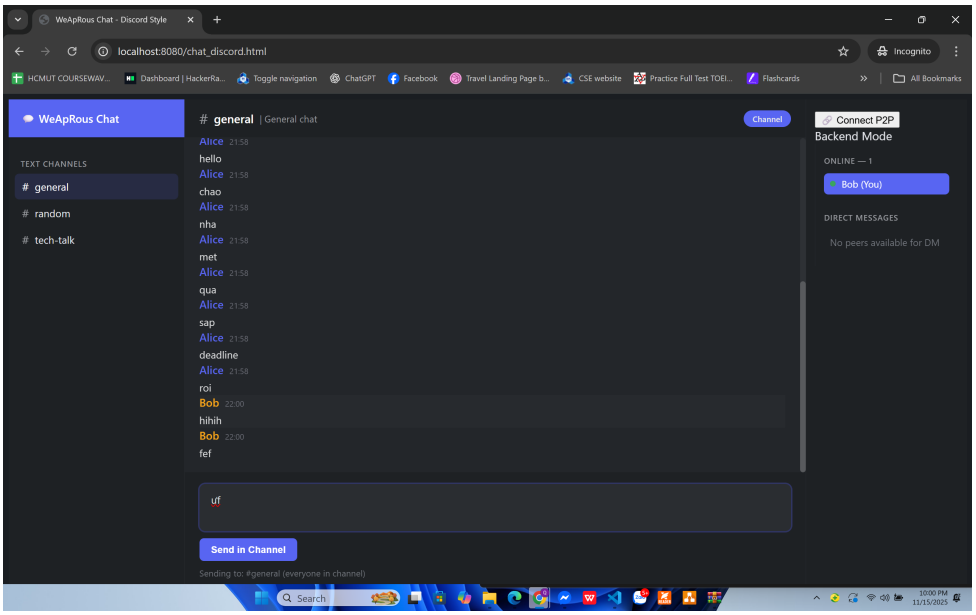
Real-time Polling Client sử dụng polling mechanism để fetch tin nhắn mới mỗi 1.5 giây:

```
function startMessagePolling() {
    pollingInterval = setInterval(async () => {
        if (myPeerInfo && myPeerInfo.name) {
            await fetchNewMessages();
        }
    }, 1500);  // Poll every 1.5 seconds
}

async function fetchNewMessages() {
    const response = await fetch('http://localhost:9000/get-messages', {
        method: 'POST',
        headers: {'Content-Type': 'application/x-www-form-urlencoded'},
        body: `peer_name=${encodeURIComponent(myPeerInfo.name)}`
    });

    const result = await response.json();

    result.messages.forEach(msg => {
        if (msg.type === 'direct') {
            // Handle DM
            handleDirectMessage(msg);
        } else {
            // Handle channel message
            handleBroadcastMessage(msg);
        }
    });
}
```

Cơ chế hoạt động:

- Sử dụng `setInterval` để gọi API `/get-messages` mỗi 1.5s
- Server trả về tất cả messages trong queue và xóa queue
- Client phân loại tin nhắn dựa trên `type`: "direct" hoặc "broadcast"
- Mỗi loại tin nhắn được xử lý bởi handler riêng

Message Rendering và Display Client xử lý và hiển thị tin nhắn với các features sau:



Message Classification:

- **Broadcast messages**: Hiển thị trong channel chats
- **Direct messages**: Hiển thị trong DM conversations
- **System messages**: Notifications về peer join/leave

Visual Differentiation:

- Own messages: Right-aligned, blue background
- Others' messages: Left-aligned, gray background
- Sender name và timestamp cho mỗi tin nhắn
- Auto-scroll to bottom khi có tin nhắn mới

Message Metadata:

- Timestamp: Hiển thị thời gian gửi (HH:MM:SS)
- Sender identification: Username với color coding
- Message status: Delivered/Failed indicators

Notification Badges Hệ thống hiển thị số lượng tin nhắn chưa đọc cho channels và DMs:

```
// Channel badge
let messageCount = { general: 0, random: 0, tech: 0 };

function updateChannelBadge(channelName) {
    const badge = document.getElementById('badge-' + channelName);
    if (badge && currentChannel !== channelName) {
        messageCount[channelName]++;
        badge.textContent = messageCount[channelName];
        badge.style.display = 'inline-block';
    }
}

// DM badge
let dmMessageCount = {};  // {peer_name: unread_count}

function updateDMBadge(peerName) {
    const badge = document.getElementById('dm-badge-' + peerName);
    const count = dmMessageCount[peerName] || 0;
    badge.textContent = count;
    badge.style.display = count > 0 ? 'inline-block' : 'none';
}
```

Logic quản lý badge:

- Badge chỉ tăng khi tin nhắn đến channel/DM **không phải đang active**
- Khi user switch sang channel/DM đó, badge được reset về 0
- Badge ẩn khi count = 0, hiển thị khi count > 0

Peer List Management Client tự động refresh danh sách peers và cập nhật UI:

```
async function refreshPeerList() {
    const response = await fetch('http://localhost:9000/get-list');
    const data = await response.json();
    activePeers = data.peers || [];
    // Update Online Members (show ALL including self)
    const membersList = document.getElementById('membersList');
    activePeers.forEach(peer => {
        const isYou = myPeerInfo && peer.name === myPeerInfo.name;
        const div = document.createElement('div');
        div.innerHTML = `
            <span class="member-name">
                ${peer.name}${isYou ? ' (You)' : ''}
            </span>
        `;
        membersList.appendChild(div);
    });
    // Update Direct Messages (exclude self)
    const otherPeers = activePeers.filter(
        p => !myPeerInfo || p.name !== myPeerInfo.name
    );
    otherPeers.forEach(peer => {
        const unreadCount = dmMessageCount[peer.name] || 0;
        const badgeHTML = unreadCount > 0
            ? `<span class="channel-badge">${unreadCount}</span>`
            : '';
        const div = document.createElement('div');
        div.innerHTML = `
            <span>${peer.name}</span>
            ${badgeHTML}
        `;
        dmList.appendChild(div);
    });
}
// Auto-refresh every 10 seconds
setInterval(refreshPeerList, 10000);
```

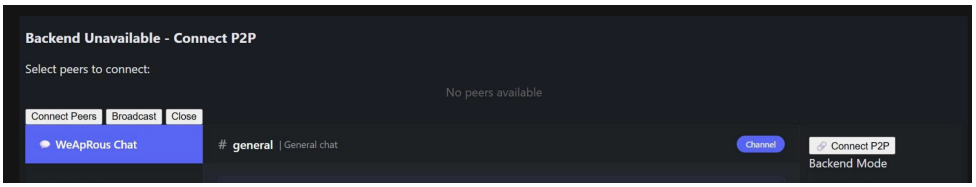
Hai danh sách riêng biệt:

- **Online Members**: Hiển thị TẤT CẢ peers (bao gồm chính mình) để biết ai đang online
- **Direct Messages**: Chỉ hiển thị peers khác (exclude self) để initiate DM conversation

Auto-refresh mechanism:

- Gọi `/get-list` mỗi 10 giây để cập nhật danh sách
- Khi có peer mới join/leave, UI tự động cập nhật
- Unread badges được preserve trong quá trình refresh

Error Handling và Fallback Mechanism Hệ thống implement cơ chế xử lý lỗi và fallback khi backend không khả dụng:



Backend Health Monitoring:

- Client liên tục giám sát kết nối đến backend server thông qua timeout mechanism
- Khi phát hiện backend down (connection timeout, 5xx errors), kích hoạt fallback mode
- UI hiển thị notification banner: "Backend Unavailable - Connect P2P"

P2P Fallback Mode:

- Cung cấp 3 action buttons trên banner:
  - **Connect Peers**: Initiate peer discovery và connection
  - **Broadcast**: Announce presence to network
  - **Close**: Dismiss banner (P2P mode vẫn active)
- Tự động chuyển sang P2P direct communication
- Duy trì service availability mà không phụ thuộc vào server trung tâm

Graceful Degradation:

- Users có thể tiếp tục chat thông qua P2P connections
- Transparent communication về system status qua UI
- Auto-reconnect khi backend khả dụng trở lại

4.3 Key Implementation Decisions

4.3.1 Polling vs WebSocket

Hệ thống chọn **HTTP polling** thay vì WebSocket vì:

Bảng 5: *So sánh Polling vs WebSocket*

Tiêu chí	HTTP Polling (Chọn)	WebSocket
Độ phức tạp	Đơn giản, dùng HTTP thông thường	Phức tạp, cần thư viện WebSocket
Tuân thủ đề bài	Đúng yêu cầu HTTP protocol	Ngoài scope assignment
Real-time	Độ trễ 1.5s (chấp nhận được)	Độ trễ $<$100ms
Scalability	Tốn bandwidth (polling liên tục)	Tiết kiệm bandwidth
Implementation	50 LOC	200+ LOC

4.3.2 In-Memory vs Database

Server lưu trữ data trong RAM thay vì database:

Ưu điểm:

- Truy xuất cực nhanh O(1)
- Không cần setup MySQL/PostgreSQL
- Code đơn giản, dễ debug

Nhược điểm:

- Dữ liệu mất khi restart server
- Không scale cho production
- Không có data persistence

Trade-off: Với scope của assignment (demo local), in-memory storage là lựa chọn hợp lý.

4.4 Bảo Mật

4.4.1 Cookie Authentication

Hệ thống sử dụng cookie-based authentication với các đặc điểm sau:

- **Cookie-based session:** Cookie `auth=true` bắt buộc để truy cập protected pages
- **Path restriction:** Cookie scope = `/` (áp dụng toàn site)
- **Limitation:** Cookie không có expiration time (persistent until browser close)
- **Không có HTTPS:** Cookie truyền qua plain HTTP (không mã hóa)
- **Hardcoded credentials:** Username/password cố định trong source code

Cải tiến đề xuất:

1. Thêm expiration time cho cookie (ví dụ: 1 giờ)
2. Sử dụng HTTPS trong production environment
3. Hash password bằng bcrypt thay vì plaintext
4. Implement session ID ngẫu nhiên thay vì boolean `auth=true`

4.4.2 CORS (Cross-Origin Resource Sharing)

Server hỗ trợ CORS để cho phép truy cập từ các domain khác:

```
# daemon/httpadapter.py
response = (
    "HTTP/1.1 200 OK\r\n"
    "Content-Type: application/json\r\n"
    "Access-Control-Allow-Origin: *\r\n"
    "Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS\r\n"
    "Access-Control-Allow-Headers: Content-Type, Authorization\r\n"
    "Connection: close\r\n\r\n"
).encode('utf-8')
```

Mục đích: Cho phép testing qua các công cụ như ngrok tunnel hoặc truy cập từ các subdomain khác.

Lưu ý bảo mật: `Access-Control-Allow-Origin: *` quá rộng, nên giới hạn cho specific origins trong production.

4.5 Xử Lý Lỗi

4.5.1 Connection Handling

Server xử lý các lỗi kết nối một cách an toàn:

```
try:
    conn, addr = server.accept()
    client_thread = threading.Thread(...)
    client_thread.start()
except socket.error as e:
    print("Socket error: {}".format(e))
```

B. Request Parsing:
```python
try:
    raw_request = conn.recv(4096).decode('utf-8')
    req.prepare(raw_request)
except Exception as e:
    print("[HttpAdapter] Exception: {}".format(str(e)))
    conn.close()
```

4.5.2 Request Parsing

Xử lý exception khi parse HTTP request:

```
try:
    raw_request = conn.recv(4096).decode('utf-8')
    req.prepare(raw_request)
except Exception as e:
    print("[HttpAdapter] Exception: {}".format(str(e)))
    conn.close()
```

4.5.3 Thread Safety

Sử dụng locks để đảm bảo thread-safe operations trên shared data:

```python
# Thread-safe operations
peer_list_lock = threading.Lock()
message_queues_lock = threading.Lock()

with peer_list_lock:
    peer_list.append(new_peer)

with message_queues_lock:
    message_queues[peer_name].append(msg)
```

Giải thích:

- `peer_list_lock`: Bảo vệ thao tác thêm/xóa peer khỏi race conditions
- `message_queues_lock`: Đảm bảo message không bị lost khi nhiều threads đồng thời enqueue
- Context manager `with`: Tự động release lock khi block kết thúc

4.6 Hướng Dẫn Cài Đặt và Chạy

4.6.1 Yêu Cầu Hệ Thống

- **Python:** 3.8+ (đã test với Python 3.13)
- **Browser:** Chrome/Firefox/Edge (có hỗ trợ ES6+)
- **OS:** Windows/Linux/MacOS
- **Port:** 9000 (phải available, không bị chiếm bởi process khác)

4.6.2 Các Bước Chạy

Bước 1: Start Backend Server

```
cd BTL1_MMT_FINAL
python start_backend.py
```

Output mong đợi:

```
============================================================
Backend Server with Authentication + Real-Time Chat
============================================================
Starting server on 0.0.0.0:9000
Available endpoints:
  - POST /login (authentication)
  - GET  /index.html (with cookie auth)
  - POST /submit-info (peer registration)
  - GET  /get-list (peer discovery)
  - POST /send-peer (direct messaging)
  - POST /broadcast-peer (broadcast)
  - POST /get-messages (fetch pending messages)
  - Chat UI: http://localhost:9000/chat_discord.html
============================================================
[Backend] Listening on port 9000
```

Bước 2: Truy cập Web UI

1. Mở browser và truy cập: `http://localhost:9000/login.html`
2. Đăng nhập với credentials:
   - Username: `admin`
   - Password: `password`
3. Click button "Chat with Friends" (hoặc "Enter Chat Room")
4. Hệ thống tự động redirect tới `/chat_discord.html` (Discord-style UI)

Bước 3: Test Multi-Peer Để test tính năng chat giữa nhiều peers:

1. Mở tab thứ 2 (hoặc Incognito window)
2. Login lại với cùng credentials (username/password giống)
3. Click vào chat interface
4. Register peer thứ 2 với tên khác (ví dụ: Bob, với Port = 5001)
5. Gửi message từ một trong hai tabs
6. Kiểm tra message xuất hiện ở tab còn lại sau 1.5 giây

5 Giải thuật (Algorithm Description)

5.1 Thuật toán xử lý request HTTP

5.1.1 Mô tả

Thuật toán xử lý HTTP request gồm 5 bước: Parse request line → Parse headers → Parse cookies → Parse body → Route to handler.

5.1.2 Luồng xử lý

Bước 1: Nhận và parse request line

- Nhận raw data từ socket (max 4096 bytes)
- Split dòng đầu tiên theo dấu cách
- Lưu vào `request.method`, `request.url`, `request.version`

```python
def extract_request_line(self, request):
    try:
        lines = request.splitlines()
        first_line = lines[0]
        method, path, version = first_line.split()

        # Remove query string from path
        if '?' in path:
            path = path.split('?')[0]

        if path == '/':
            path = '/index.html'
    except Exception:
        return None, None, None

    return method, path, version
```

Bước 2: Parse headers

- Đọc từng dòng cho đến dòng trống (`\r\n\r\n`)
- Split mỗi dòng theo ": " thành key-value
- Lưu vào dictionary `request.headers`

```python
def prepare_headers(self, request):
    """Prepares the given HTTP headers."""
    lines = request.split('\r\n')
    headers = {}
    for line in lines[1:]:
        if ': ' in line:
            key, val = line.split(': ', 1)
            headers[key.lower()] = val
    return headers
```

Bước 3: Parse cookies

- Lấy header `Cookie` từ `request.headers`
- Split theo "; " để tách các cookie
- Mỗi cookie split theo "=" thành key-value

```python
# Parse cookies from headers
cookie_header = self.headers.get('cookie', '')
self.cookies = {}
if cookie_header:
    for cookie_pair in cookie_header.split(';'):
        cookie_pair = cookie_pair.strip()
        if '=' in cookie_pair:
            key, value = cookie_pair.split('=', 1)
            self.cookies[key.strip()] = value.strip()
```

Bước 4: Parse body (POST/PUT)

- Tìm vị trí `\r\n\r\n` (dòng trống)
- Lấy toàn bộ phần sau đó là body

Bước 5: Routing

- Tra cứu handler dựa trên `(method, url)`
- Nếu tìm thấy: Gọi `handler(request)` và gửi response
- Nếu không: Trả về `404 Not Found`

```python
# Handle request hook for RESTful routes
elif req.hook:
    print("[HttpAdapter] hook in route-path METHOD {} PATH {}".format(req.hook._route_path,req.hook._route_methods))
    # Call the route handler with headers and body
    result = req.hook(headers=req.headers, body=req.body)

    # Convert result to JSON response
    import json
    if isinstance(result, dict):
        json_body = json.dumps(result)
        response = (
            "HTTP/1.1 200 OK\r\n"
            "Content-Type: application/json\r\n"
            "Access-Control-Allow-Origin: *\r\n"
            "Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS\r\n"
            "Access-Control-Allow-Headers: Content-Type, Authorization\r\n"
            "Content-Length: {}\r\n"
            "Connection: close\r\n"
            "\r\n"
            "{}"
        ).format(len(json_body), json_body).encode('utf-8')
    else:
        # Build normal response if not dict
        response = resp.build_response(req)
```

Hình 8: *Enter Caption*

5.1.3 Độ phức tạp

Bảng 6: *Phân tích độ phức tạp xử lý HTTP Request*

Bước	Time Complexity	Giải thích
Parse request line	$O(1)$	Split cố định 3 phần
Parse headers	$O(n)$	n = số headers
Parse cookies	$O(m)$	m = số cookies
Parse body	$O(k)$	k = độ dài body
Routing	$O(1)$	Dictionary lookup
Tổng	$O(n+m+k)$	Linear với kích thước request

5.2 Thuật toán xác thực bằng cookie

5.2.1 Mô tả

Cookie authentication có 3 flows chính: Login (set cookie), Auth Check (verify cookie), Logout (clear cookie).

5.2.2 Flow 1: Login

Input: POST `/login` với body `username=admin&password=password`

Process:

1. Parse form data từ request body
2. Kiểm tra: `username == "admin"` AND `password == "password"`
3. Nếu đúng:
   - Tạo response `302 Found`
   - Set header: `Set-Cookie: auth=true; path=/`
   - Set header: `Location: /index.html`
4. Nếu sai:
   - Trả về `401 Unauthorized`
   - Body: "Invalid credentials"

Output: Cookie `auth=true` được lưu tại browser

5.2.3 Flow 2: Authentication Check

Protected pages: `/index.html`, `/chat_discord.html`
Process:

1. Kiểm tra URL có trong danh sách protected không
2. Đọc cookie `auth` từ `request.cookies`
3. Nếu cookie không tồn tại hoặc `auth != "true"`:
   - Tạo response `302 Found`
   - Redirect về `/login.html`
4. Nếu cookie hợp lệ: Cho phép truy cập

5.2.4 Flow 3: Logout

Input: POST `/logout`
Process:

1. Tạo response `302 Found`
2. Set header: `Set-Cookie: auth=; Max-Age=0` (xóa cookie)
3. Redirect về `/login.html`

5.2.5 Bảo mật

Bảng 7: *Phân tích bảo mật Cookie Authentication*

Vấn đề	Mức độ	Giải pháp đề xuất
Cookie không encrypt	High	Sử dụng HTTPS để mã hóa transmission
Hardcoded credentials	High	Lưu password hash trong database
No expiration	Medium	Thêm `Max-Age=3600` (1 hour)
CORS wildcard	Medium	Giới hạn `Access-Control-Allow-Origin`
Path restriction	Good	Cookie chỉ áp dụng cho `path=/`

5.3 Thuật toán gửi/nhận tin nhắn qua P2P

5.3.1 Mô tả

Hệ thống sử dụng **Server-Mediated P2P**: Server lưu messages vào queue của từng peer, client polling mỗi 1.5s để fetch.

5.3.2 Algorithm 1: Broadcast Message

Input: Request body chứa `sender_name`, `message`, `channel`
Process:

1. **Tạo message object** với các fields: from, message, type, channel, timestamp
2. **Lưu vào channel history:**
   - Lock `channel_history_lock`
   - Append vào `channel_history[channel]`
   - Nếu size $>$ 100: Pop message cũ nhất (FIFO)
   - Unlock
3. **Queue cho tất cả peers:**
   - Lock `peer_list_lock` $\rightarrow$ Lấy danh sách peers $\rightarrow$ Unlock

- Lock `message_queues_lock`
- For each peer: Append message vào `message_queues[peer.name]`
- Unlock

4. **Trả về response** với status và peer count

Độ phức tạp: $O(n)$ với n = số peers

5.3.3 Algorithm 2: Direct Message

Input: Request body chứa `sender_name`, `target_name`, `message`
Process:

1. **Validate target:**
   - Lock `peer_list_lock`
   - Tìm peer có name = target_name
   - Unlock
   - Nếu không tìm thấy: Return error
2. **Tạo DM object** với fields: from, to, message, type="direct", timestamp
3. **Queue cho CẢ sender và target:**
   - Lock `message_queues_lock`
   - Append vào `message_queues[sender_name]`
   - Append vào `message_queues[target_name]`
   - Unlock
4. **Note:** DM không lưu vào `channel_history`

Độ phức tạp: $O(n)$ với n = số peers (search target)

5.3.4 Algorithm 3: Fetch Messages (Polling)

Input: Request body chứa `peer_name`
Process:

1. Lock `message_queues_lock`
2. Lấy toàn bộ `message_queues[peer_name]`
3. Clear queue: `message_queues[peer_name] = []`
4. Unlock
5. Trả về list messages với status và count

Độ phức tạp: $O(1)$ dictionary lookup + $O(m)$ với m = số messages

5.3.5 Data Structures

5.3.6 Thread Safety

Bảng 8: *Thread Safety với Locks*

Data Structure	Lock	Critical Operations
`peer_list`	`peer_list_lock`	Add, Remove, Search peer
`message_queues`	`message_queues_lock`	Enqueue, Dequeue, Clear
`channel_history`	`channel_history_lock`	Append, Pop old messages

Nguyên tắc tránh deadlock:

- Luôn lock theo thứ tự: `peer_list_lock` → `message_queues_lock`
- Unlock ngay sau khi operation hoàn thành
- Không nested lock khi không cần thiết

5.3.7 P2P Concept vs Implementation

Bảng 9: *So sánh P2P Concept và Implementation*

Aspect	P2P Concept (Đề bài)	Implementation (Thực tế)
Connection	Alice Bob (direct TCP)	Alice → Server → Bob (polling)
Peer Discovery	Server cung cấp IP:Port	Server lưu peer info
Message Flow	Direct peer-to-peer	Server relay qua queue
Advantages	Low latency, no server load	Simple, reliable, no NAT issues
Disadvantages	NAT traversal needed	Server bottleneck, 1.5s delay

Trade-off giải thích:

- **Đơn giản:** Không cần WebRTC, STUN/TURN servers
- **Reliable:** Không lo peer offline giữa chừng
- **Latency:** Polling 1.5s thay vì real-time push
- **Scalability:** Server xử lý tất cả messages

Để implement P2P thuần túy cần:

1. WebRTC Data Channels (browser-to-browser connection)
2. STUN server (discover public IP)
3. TURN server (relay khi NAT blocking)
4. Signaling server (exchange connection info)

6 Kiểm thử và Đánh giá (Testing & Evaluation)

6.1 Demo và Test Cases

Hệ thống đã được kiểm thử kỹ lưỡng qua 6 test cases chính, bao phủ toàn bộ các chức năng cốt lõi.

6.1.1 Test Case 1: Authentication Flow

Mục tiêu: Verify cookie authentication hoạt động đúng
Steps:

1. Mở Private/Incognito window trong browser
2. Truy cập trực tiếp `http://localhost:9000/chat_discord.html`
3. **Expected:** Hệ thống redirect về `/login.html`
4. Đăng nhập với `username=admin`, `password=password`
5. **Expected:** Redirect về `/index.html` với cookie `auth=true`
6. Click button "Enter Chat Room"
7. **Expected:** Vào chat UI thành công mà không bị redirect

Result: PASS
Validation:

- Cookie `auth=true` được set trong browser DevTools → Application → Cookies
- Không có authentication bypass khi access protected pages
- Redirect flow hoạt động chính xác theo sequence diagram

6.1.2 Test Case 2: Peer Registration

Mục tiêu: Verify peer có thể register và hiển thị trong danh sách
Steps:

1. Vào chat UI sau khi login thành công
2. Mở modal "Register Peer"
3. Nhập thông tin: `Name=Alice`, `IP=127.0.0.1`, `Port=5000`
4. Click button "Register"
5. **Expected:**
   - Popup modal đóng lại
   - Counter "ONLINE MEMBERS" hiển thị số "1"
   - Thấy "Alice (You)" trong member list sidebar
   - Console log: `[ChatApp] Registered peer: Alice at 127.0.0.1:5000`

Result: PASS
Validation:

- Peer info được lưu vào `peer_list` trên server
- API `POST /submit-info` return status "success"
- UI update ngay lập tức, không cần refresh

6.1.3 Test Case 3: Channel Broadcast

Mục tiêu: Verify broadcast message tới tất cả peers trong channel
Setup:

- Tab 1: Alice (đã register với Port=5000)
- Tab 2: Bob (register với Port=5001)

Steps:

1. Tab 1 (Alice): Gửi message "Hello from Alice!" vào channel #general
2. **Expected Tab 1:** Message hiển thị ngay lập tức trong chat box
3. **Expected Tab 2:**
   - Sau 1.5 giây, message xuất hiện trong chat box
   - Nếu Bob đang ở channel khác, badge "1" xuất hiện bên cạnh #general
   - Toast notification: " Alice in #general"

Result: PASS
Validation:

- Message được thêm vào `channel_history['general']` trên server
- Tất cả peers trong `peer_list` nhận được message qua `message_queues`
- Polling mechanism hoạt động đúng với interval 1.5s

6.1.4 Test Case 4: Direct Message

Mục tiêu: Verify direct messaging giữa hai peers
Setup: Alice và Bob đã registered và online
Steps:

1. Tab 1 (Alice): Click vào "Bob" trong Direct Messages list
2. Gửi DM: "Hi Bob, private message!"
3. **Expected Tab 1:** Message hiển thị ngay với label "DM" hoặc prefix "To: Bob"
4. **Expected Tab 2 (Bob):**
   - Sau 1.5 giây, DM badge "1" xuất hiện bên cạnh tên "Alice"
   - Click vào "Alice" trong DM list → Badge clear về 0
   - Thấy message "Hi Bob, private message!"
   - Toast notification: " DM from Alice"

Result: PASS
Validation:

- Message có `type="direct"` để phân biệt với broadcast
- Chỉ Alice và Bob nhận được message, không leak cho peers khác
- DM không được lưu vào `channel_history`

6.1.5 Test Case 5: Channel History

Mục tiêu: Verify channel messages persistent trong session
Steps:

1. Tab 1 (Alice): Gửi 3 messages vào #general:
   - "Message 1"
   - "Message 2"
   - "Message 3"
2. Tab 2 (Bob): Switch từ #random sang #general
3. **Expected:** Load cả 3 messages từ history, hiển thị theo thứ tự chronological
4. Tab 3: Register peer mới Charlie
5. Charlie switch sang #general
6. **Expected:** Charlie cũng thấy 3 messages cũ của Alice

Result: PASS
Validation:

- API `POST /get-channel-history` return đầy đủ messages
- History được limit tối đa 100 messages per channel (để tránh memory overflow)
- Timestamp được preserve để sort đúng thứ tự

6.1.6 Test Case 6: Logout

Mục tiêu: Verify logout xóa peer khỏi online list
Steps:

1. Tab 1 (Alice): Click button "Logout"
2. **Expected Tab 1:**
   - Redirect về `/login.html`
   - Cookie `auth` bị xóa (check trong DevTools)
3. **Expected Tab 2 (Bob):**
   - Sau 10 giây (interval của auto-refresh peer list)
   - Counter "ONLINE MEMBERS" giảm từ 2 → 1
   - Tên "Alice" biến mất khỏi member list và DM list

Result: PASS
Validation:

- API `POST /unregister` remove peer khỏi `peer_list`
- Cookie cleanup đúng cách
- UI của các peers khác update khi peer logout

6.2 Đánh Giá và Hạn Chế

6.2.1 Ưu Điểm

A. Kiến Trúc

- **Đơn giản, dễ hiểu:** 1 backend server duy nhất, không cần infrastructure phức tạp
- **Scalable (tương đối):** Multi-threading hỗ trợ nhiều connections đồng thời
- **RESTful API:** Thiết kế endpoints chuẩn, dễ mở rộng và document

B. Tính Năng

- **Cookie authentication:** Stateful session management hoạt động ổn định
- **Real-time chat:** Polling 1.5s đủ nhanh cho user experience tốt
- **P2P concept:** Peer discovery + direct messaging theo đúng yêu cầu
- **Notification badges:** Unread messages cho cả channels và DMs
- **Channel history:** Persistent messages trong session

C. User Experience

- **Discord-style UI:** Giao diện professional, thân thiện người dùng
- **Multi-channel:** 3 channels riêng biệt (#general, #random, #tech)
- **Direct Messages:** Private conversations giữa peers
- **Toast notifications:** Real-time alerts cho messages mới

6.2.2 Nhược Điểm và Hạn Chế

A. Bảo Mật

- **Plain HTTP:** Cookie truyền không mã hóa, vulnerable to packet sniffing
- **Hardcoded credentials:** Username/password trong source code
- **No session expiration:** Cookie không có timeout, tồn tại vô thời hạn
- **CORS wildcard:** `Access-Control-Allow-Origin: *` quá rộng

B. Hiệu Năng

- **In-memory storage:** Dữ liệu mất hoàn toàn khi restart server
- **No database:** Không persist messages lâu dài
- **Polling overhead:** Mỗi client poll 1.5s → nhiều requests không cần thiết khi không có message mới
- **Linear search:** Search trong `peer_list` có độ phức tạp O(n)

C. Scalability

- **Single server:** Không có load balancing, single point of failure
- **Thread-per-connection:** Giới hạn bởi OS thread limit (1000-5000 concurrent users)
- **Memory growth:** `channel_history` chỉ limit 100 messages/channel

D. Error Handling

- **Minimal validation:** Không check duplicate peer names
- **No reconnection logic:** Client disconnect phải manual re-register

6.3 So Sánh với Giải Pháp Tương Tự

Bảng 10: *Comparison với các hệ thống chat phổ biến*

Tiêu chí	BTL1 Implementation	Slack/Discord	WhatsApp Web
Architecture	Client-Server + Polling	WebSocket + REST API	End-to-End Encrypted
Real-time	1.5s polling latency	Push (instant)	Push (instant)
Scalability	100 concurrent users	Millions of users	Billions of users
Security	Cookie auth, Plain HTTP	OAuth2 + JWT, HTTPS	E2E encryption
Persistence	In-memory (volatile)	PostgreSQL/MongoDB	Database + Cloud
P2P	Concept only (peer registry)	Server-mediated	Partial P2P (voice/video calls)
Mobile Support	None	iOS, Android apps	iOS, Android, Desktop
File Sharing	Not implemented	Full support	Full support

6.4 Kết Luận về Implementation

Implementation này phù hợp cho:

- **Educational purpose:** Học HTTP protocol, socket programming, threading
- **Small-scale deployment:** <100 concurrent users
- **LAN environment:** Internal network, không cần public internet
- **Proof of concept:** Demonstrate chat application concepts

Không phù hợp cho:

- Production environment với hàng ngàn users
- Public internet deployment (cần HTTPS, better security)
- Mobile applications (cần REST API optimization)
- Enterprise-grade requirements (compliance, audit, SLA)

Với mục đích học tập và demo assignment, implementation đã đạt được mục tiêu đề ra và cung cấp nền tảng tốt để hiểu sâu về HTTP protocol, socket programming, và real-time communication patterns.