

DẠI 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, 10/2025



## Mục lục

|   |           |
|---|-----------|
| Danh sách hình ảnh                                  | 3         |
| Danh sách bảng                                      | 3         |
| Danh sách thành viên                                | 3         |
| <b>1 Mở đầu</b>                                     | <b>4</b>  |
| <b>2 HTTP server with cookie session</b>            | <b>5</b>  |
| 2.1 Implement authentication handling . . . . .     | 5         |
| 2.1.1 Routing & Validation . . . . .                | 5         |
| 2.1.2 Invalid . . . . .                             | 5         |
| 2.1.3 Valid . . . . .                               | 5         |
| 2.2 Implement cookie-based access control . . . . . | 6         |
| 2.2.1 Routing . . . . .                             | 6         |
| 2.2.2 Cookie Check . . . . .                        | 6         |
| 2.2.3 Invalid cookie . . . . .                      | 6         |
| 2.2.4 Valid cookie . . . . .                        | 6         |
| <b>3 Hybrid chat application</b>                    | <b>7</b>  |
| 3.1 Chatting between 2 peer . . . . .               | 7         |
| 3.1.1 Giao tiếp trực tiếp 1-1 giữa peer . . . . .   | 7         |
| 3.1.2 Cơ chế broadcast . . . . .                    | 8         |
| 3.2 Channel management . . . . .                    | 9         |
| <b>4 Kết luận</b>                                   | <b>11</b> |
| <b>Tài liệu tham khảo</b>                           | <b>12</b> |



## Danh sách hình ảnh

### Danh sách bảng

|   |                                   |   |
|---|-----------------------------------|---|
| 1 | Danh sách thành viên và phân công | 3 |
|---|-----------------------------------|---|



## Danh sách thành viên

nào xong sửa sau

| STT | Họ và tên          | MSSV    | Phân công      | % hoàn thành |
|-----|--------------------|---------|----------------|--------------|
| 1   | Lê Nguyễn Kim Khôi | 2311671 | - abc<br>- xyz | 100%         |
| 2   | Nguyễn Công Minh   | 2312080 | -<br>-         | 100%         |
| 3   | Bùi Ngọc Phúc      | 2312665 | -<br>-         | 100%         |
| 4   | Lê Trọng Thiện     | 2313233 | -<br>-         | 100%         |
| 5   | Phạm Trần Minh Trí | 2313622 | -<br>-         | 100%         |

Bảng 1: Danh sách thành viên và phân công



## 1 Mở đầu



## 2 HTTP server with cookie session

### 2.1 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 (diều phối), request.py (xử lý body), và response.py (tạo phản hồi).

#### 2.1.1 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.

#### 2.1.2 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.

#### 2.1.3 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.



## 2.2 Implement cookie-based access control

### 2.2.1 Routing

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

### 2.2.2 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`.

### 2.2.3 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.

### 2.2.4 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.



### 3 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): Dóng 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.

#### 3.1 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.

##### 3.1.1 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.

### 3.1.2 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.

### 3.2 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] là dấu hiệu nhận biết (flag) cho P2P Client rằng đây là tin nhắn thuộc về một channel cụ thể.



## 4 Kết luận



## 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.