VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

ASSIGNMENT 1
COMPUTER NETWORKS - CO3094

IMPLEMENT HTTP SERVER AND CHAT APPLICATION

MAJOR: COMPUTER SCIENCE

THESIS COMMITTEE: COMPUTER SCIENCE - CC02

SUPERVISOR(s): PhD. NGUYỄN PHƯƠNG DUY

——————o0o——————

STUDENT: HOÀNG XUÂN BÁCH - 2352082
NGUYỄN VIỆT HÙNG - 2352424
TRẦN TRỌNG TÍN - 2353189
NGUYỄN LÊ ĐĂNG VINH - 2353330
NGUYỄN CÔNG VINH - 2353329

HO CHI MINH CITY, December 2025

Contents

Member list & Workload

No.	Fullname	Student ID	Tasks	Participation
1	Hoàng Xuân Bách	2352082	- Task 2: P2P Client, Web Interface	100%
2	Trần Trọng Tín	2353189	- Task 1: HTTP Core (Request, Response, Backend), Proxy - LaTeX	100%
3	Nguyễn Công Vinh	2353329	- Task 1: HttpAdapter, Cookie Session Logic	100%
4	Nguyễn Lê Đăng Vinh	2353330	- Task 2: Chat Tracker, REST API	100%
5	Nguyễn Việt Hùng	2352424	- Task 2: P2P Client, Web Interface - LaTeX	100%

1 Abstract

This report presents the design and implementation of a custom HTTP server framework and a Hybrid Chat Application. The system is built entirely using Python's socket library, demonstrating a deep understanding of TCP/IP networking, HTTP protocol handling, multi-threaded processing, and Peer-to-Peer (P2P) architecture. The project consists of a Reverse Proxy, a Backend Server handling Cookie-based authentication, and a Chat System that combines a centralized Tracker (RESTful API) with direct P2P messaging.

2 Introduction

2.1 Objectives

- To understand and implement the Client-Server and Peer-to-Peer (P2P) network models.
- To practice TCP network programming using Sockets.
- To understand the HTTP protocol, specifically header parsing, MIME types, and Cookie-based session management.
- To design a custom application layer protocol for a hybrid chat system.

2.1.1 Technologies Used

- **Language:** Python 3.x
- **Libraries:** socket, threading, json, argparse, urllib.
- **Frontend:** HTML5, CSS, JavaScript (Fetch API).
- **Architecture:** RESTful API, TCP Socket Streaming.

3 SYSTEM DESIGN

3.1 Overall Architecture

The system follows a multi-tier architecture:

1. **Proxy Layer:** Acts as the entry point, routing requests to appropriate backends based on Host headers and load-balancing policies.
2. **Backend Layer:** The core HTTP server that parses requests, handles logic (like login), and generates responses.
3. **Application Layer (WeApRous):** A custom framework used to build the Chat Tracker, managing peer registration and channel logic.

3.1.1 Server Process

The server lifecycle involves:

1. **Initialization:** Binding to an IP and Port.
2. **Listening:** Waiting for incoming TCP connections.
3. **Threading:**Spawning a new thread for every client connection to ensure concurrency.
4. **Processing:** Parsing raw bytes into Request objects, executing logic via Adapters or Route Handlers, and sending back formatted Response bytes.

3.1.2 Communication Flow

- **HTTP Flow:** Client Browser $\rightarrow$ Proxy $\rightarrow$ Backend $\rightarrow$ HttpAdapter $\rightarrow$ Response.
- **Chat Flow:**
  - *Discovery:* Client $\rightarrow$ Tracker (HTTP/REST) to get Peer List.

- *Messaging:* Client $\rightarrow$ Peer (TCP Socket) for direct chat.

4 IMPLEMENTATION MODULE ANALYSIS

4.1 HTTP Server with Cookie Session

This module implements a basic HTTP server capable of serving static files and handling authentication.

4.1.1 Request *(daemon/request.py)*

This class is responsible for parsing raw incoming HTTP packets.

- **Functionality:** It extracts the Request Line (Method, Path, Version) and parses Headers into a dictionary.
- **Cookie Parsing:** It splits the Cookie header string to store session data in self.cookies.
- **Routing:** It prepares the request hook if the path matches a registered route in the WeApRous framework.

4.1.2 Response *(daemon/response.py)*

This class constructs the HTTP response packet.

- **MIME Types:** It uses mimetypes to determine the Content-Type (e.g., text/html, image/png) and sets the appropriate base directory (www/ or static/).
- **Header Construction:** It builds standard headers (Date, Content-Length) and custom headers like Set-Cookie.
- **Error Handling:** Includes methods to generate standard 404 Not Found or 500 Internal Server Error responses.

4.1.3 Response *(daemon/backend.py)*

The backend is the TCP server foundation.

- **Multi-threading:** It uses threading.Thread to handle handle_client for every server.accept(), allowing multiple users to access the server simultaneously.
- **Delegation:** It passes the connection socket to HttpAdapter for protocol-specific processing.

4.1.4 Proxy *(daemon/proxy.py)*

The reverse proxy handles load balancing and routing.

- **Configuration:** It reads config/proxy.conf to map incoming Host headers to backend IP:Port pairs.
- **Routing Policy:** It implements a round-robin strategy. If multiple backends are defined for a host, it cycles through them to distribute the load.
- **Forwarding:** It opens a socket to the target backend, forwards the raw request, receives the response, and relays it back to the client.

4.1.5 HttpAdapter *(daemon/httpadapter.py)*

This acts as the bridge between the raw socket and the application logic.

- **Task 1A (Authentication):** The handle_login method intercepts POST /login requests. It validates credentials (admin/password) and returns a Set-Cookie: auth=true header upon success.
- **Task 1B (Access Control):** It protects the /index.html route. If a request comes for the index page, it checks req.cookies.get('auth'). If the cookie is missing or invalid, it returns a 401 Unauthorized response instead of the page.

4.2 Hybrid Chat Application

The chat application combines a centralized REST API for user discovery and a decentralized P2P model for messaging.

4.2.1 Objectives

To build a "Skype-like" hybrid system where a central server manages user status (Online/Offline) and channels, while clients communicate directly to reduce server load.

4.2.2 System Structure

- **Tracker Server (start_sampleapp.py):** Uses the WeApRous framework. It maintains peer_storage (list of active peers) and channel_storage (chat rooms) in memory.
- **P2P Client (chat_client.py):** A python script that runs a listening server thread for incoming P2P connections and a main thread for user input and REST polling.
- **Web Client (www/chat.html):** A frontend interface that polls the Tracker for messages and updates the UI.

4.2.3 Workflow

1. **Registration:** When a client starts, it sends a `POST /submit-info` to the Tracker with its IP and Port.
2. **Discovery:** Clients call `GET /get-list` to see who is online.
3. **Channel Management:** Clients can call `/create-list` or `/join-list` to organize into groups.

4.2.4 Communication Mechanism

- **RESTful APIs:** Implemented in `start_sampleapp.py`:
  - `/submit-info`: Register peer.
  - `/get-list`: Retrieve active peers and channels.

  - `/send-message`: Queues messages on the server (for Web polling) or handles broadcasting logic.

- **P2P Logic:** Implemented in `chat_client.py`:
- It maintains a list of `soutgoing_connections` (sockets connected to other peers).
- **Broadcast:** When the user types a message, the client loops through all connected sockets and sends the data directly, bypassing the server.

4.3 Integration

The entire system is integrated via the `WeApRous` framework.

1. Start Backend: python start_backend.py listens on port 9000 to serve static files (HTML/CSS/Images).
2. Start Proxy: python start_proxy.py listens on port 8080 and routes requests to the backend or the chat tracker.
3. Start Chat Tracker: python start_sampleapp.py listens on port 8000/8001 to handle Chat APIs.
4. Start Clients:
   - **Web:** Access http://localhost:8080/login.html, authenticate, and move to chat.html.
   - **CLI:** Run python chat_client.py –username Alice –peer-port 9101.

5 CONCLUSION

This assignment successfully demonstrated the implementation of a functional HTTP Server and Hybrid Chat Application from scratch.

- **Task 1:** The server correctly handles cookie-based authentication, protecting resources from unauthorized access. The proxy effectively balances load using round-robin.
- **Task 2:** The chat application supports both direct P2P communication (via the Python client) and server-mediated communication (via the Web interface), fulfilling the requirements of a hybrid network architecture.

The project highlights the complexity of network programming, requiring careful management of sockets, threads, and protocol parsing.