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

Computer networks

Implement HTTP server and chat application

Instructor(s): Nguyễn Phương Duy

Students: Nguyễn Tiến Đạt - 2352240
Bùi Đoàn Minh Nhật - 2352855
Bùi Quốc Thái - 2353086
Trịnh Vũ Thiên Bảo - 2352112
Trịnh Lương Nhất Quân - 2353017

Member list & Workload

No.	Fullname	Student ID	Contribution
1	Nguyễn Tiến Đạt	2352240	20%
2	Bùi Đoàn Minh Nhật	2352855	20%
3	Bùi Quốc Thái	2353086	20%
4	Trịnh Vũ Thiên Bảo	2352112	20%
5	Trịnh Lương Nhất Quân	2353017	20%

Table 1: Group members and workload distribution (total 100%)

Contents

1 Introduction

1.1 Overview

This report documents our implementation of Assignment 1 for the Computer Network course. The assignment requires building an HTTP server with cookie-based session management and a hybrid chat application that combines client-server and peer-to-peer (P2P) paradigms. The system uses TCP/IP sockets for network programming and includes components such as a proxy server, backend servers, and the WeApRous framework for RESTful API handling.

The primary goals of the assignment are:

- Understand and apply client-server and P2P communication models.
- Implement socket-based networking for HTTP requests, authentication, and real-time chatting.
- Design a simple P2P protocol for message exchange and channel management.

1.2 System Architecture

The system consists of the following components:

- **Proxy Server:** Handles incoming requests and routes them to backend servers (e.g., started via `start_proxy.py` on port 8080).
- **Backend Servers:** Process HTTP requests, manage sessions, and serve static files (e.g., started via `start_backend.py` on port 9000).
- **WeApRous Webapp:** A custom RESTful framework for the chat application, handling APIs such as `/login`, `/get-list`, etc. (started via `start_sampleapp.py` on port 8000).
- **Clients:** Web browsers interacting via HTTP and WebSockets for real-time P2P chatting.

2 Implementation

2.1 HTTP Server with Cookie Session

We implemented cookie-based authentication using Python's `socket` and `threading` modules in `backend.py`. The server handles HTTP requests, parses headers, and manages sessions without relying on external web frameworks.

2.1.1 Authentication Handling

On a `POST` request to `/login`, the server:

1. Parses the request body for credentials (e.g., `username=admin&password=password`).
2. If valid, responds with:
   - `200 OK`
   - `Set-Cookie: auth=true; Path=/index.html`
   - The `index.html` file
3. If invalid, returns a `401 Unauthorized` with a custom error page.

```
if req.method == "POST" and req.path == "/login":
            print("[HttpAdapter] Handling /login")
            response = self.login_handler(req, resp)
            conn.sendall(response)
            conn.close()
            return
;
```

Listing 2.1: Authentication handling

```python
def login_handler(self, req, resp):
        """
        Handle POST /login authentication.
        """

        body = req.body.strip()
        print("[Login] raw body =", body)

        parts = body.split("&")
        data = {}
        for p in parts:
            if "=" in p:
                k, v = p.split("=", 1)
                data[k] = v

        username = data.get("username", "")
        password = data.get("password", "")

        print(f"[Login] username={username} password={password}")

        if username == "admin" and password == "password":
            resp.cookies["auth"] = "true"

            req.path = "/index.html"
            return resp.build_response(req)

        resp.status_code = 401
        resp.reason = "Unauthorized"

        body = b"401 Unauthorized"



        return self.build_error_response(resp.status_code, resp.
           reason)
```

Listing 2.2: Login handler

String parsing was used to extract form data from the request body, and validation was per-

formed using hardcoded credentials for simplicity.

2.1.2 Cookie-Based Access Control

For `GET /` requests, the server checks whether the `Cookie` header contains `auth=true`.

- If valid, the server returns protected resources such as `index.html`.
- If missing or invalid, it returns `401 Unauthorized`.

Concurrency is handled using `threading`, where each client connection spawns a new thread. Error handling covers malformed requests (`400 Bad Request`) and unsupported methods (`405 Method Not Allowed`).

```
if req.method == "GET" and req.path == "/index.html":
            auth_cookie = req.cookies.get("auth", "")

            if auth_cookie != "true":
                # Return 401 immediately
                print(f"[HttpAdapter] Access denied - auth cookie: '{
                    auth_cookie}'")
                response = self.build_error_response(401, "
                    Unauthorized")
                conn.sendall(response)
                conn.close()
                return
```

Listing 2.3: Cookie-Based Access Control

2.2 Hybrid Chat Application

The chat application is built using the WeApRous framework, supporting RESTful APIs for client-server interactions and TCP sockets for P2P messaging. It is designed to resemble a simplified Skype-like system, with channels, message broadcasting, and notification features.

2.2.1 Core Functional Requirements

Initialization Phase (Client-Server Paradigm)

- **Peer Registration:** Clients send a `PUT` request to `/submit_info` containing their IP and port. The server stores this in a dictionary.
- **Tracker Update:** Peers submit their status via `POST /submit-info`.
- **Peer Discovery:** Clients fetch the list of active peers via `GET /get-list`.
- **Connection Setup:** Using the returned list, clients initiate direct P2P connections via `POST /connect-peer`.

Peer Chatting Phase (P2P Paradigm)

- **Broadcast Connections:** Peers send messages to all connected peers using TCP sockets, triggered through `/broadcast-peer`.
- **Direct Peer Communication:** Messages are exchanged directly between peers using a custom JSON-based protocol that includes fields for timestamp, username, and message text.

Concurrency is achieved by using separate threads to handle multiple incoming P2P connections.

2.2.2 Runtime Configuration for Testing

To validate our implementation under realistic multi-peer conditions, we executed the following runtime setup:

- The WeApRous application server was started using:

```
python3 start_server.py 8001
```

  running on port `8001`.

- Two chat peers were launched on separate terminals:

```
python3 start_peer.py 9101
python3 start_peer.py 9102
```

Each peer used its designated port (`9101` and `9102`) for incoming P2P connections.

- Both peers submitted their network information to the server using `PUT /submit-info` right after they start, using the method register_with_tracker().

```
def register_with_tracker(self):
        """
        Register this peer with the tracker server.

        :return: bool - True if registration successful
        """
        try:
            # Create HTTP POST request
            body = json.dumps({
                "peer_id": self.peer_id,
                "ip": self.peer_ip,
                "port": self.peer_port,
                "channels": self.channels
            })

            request = (
                "POST /submit-info HTTP/1.1\r\n"
                "Host: {}:{}\r\n"
                "Content-Type: text/plain\r\n"
                "Content-Length: {}\r\n"
                "\r\n"
                "{}"
            ).format(self.tracker_ip, self.tracker_port, len(body),
               body)

            # Send to tracker
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock.connect((self.tracker_ip, self.tracker_port))
            sock.sendall(request.encode('utf-8'))

            # Receive response
            response = sock.recv(4096).decode('utf-8')
```

```
            sock.close()

            print("[Peer] Registered with tracker")
            print("[Peer] Response: {}".format(response[:200]))
            return True

        except Exception as e:
            print("[Peer] Failed to register with tracker: {}".format
                (e))
            return False
```

```
[ChatApp] Peer registration request received
[ChatApp] Registering peer: peer_id=e2d03124, ip=192.168.1.3, port=9101
[ChatApp] Added new peer: 0
[ChatApp] Peer registered: 0 (total peers: 1)
```

Figure 2.1: Register peer on port 9002

```
[ChatApp] Peer registration request received
[ChatApp] Registering peer: peer_id=9f9954fd, ip=192.168.1.3, port=9102
[ChatApp] Added new peer: 1
[ChatApp] Peer registered: 1 (total peers: 2)
```

Figure 2.2: Register peer on port 9002

Commands that can be used by peers includes:

- list - Get peer list
  When the user types `list`, the peer immediately sends a raw HTTP POST request to the tracker's `/get-list` endpoint with an empty JSON body. The tracker responds with the full current list of registered peers (each containing peer_id, IP address, port, and subscribed channels). The peer parses the HTTP response, extracts the JSON payload after the double CRLF separator, and prints a formatted list of all active peers in the network. No direct P2P connection is involved; this is a pure client–server interaction for discovery.

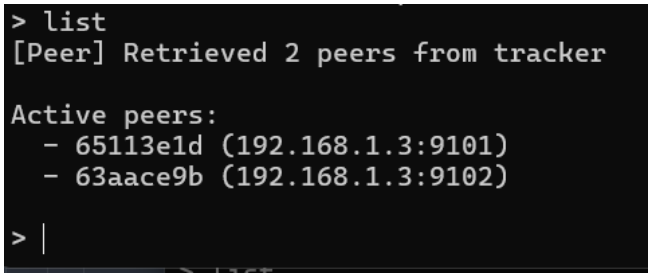

Figure 2.3: Get peer list using list

- join <channel> - Join a channel
  When the user types `join <channel>`, the peer sends a raw HTTP POST request to the tracker's `/add-list` endpoint with JSON body `{"peer_id": "<own_id>", "channel": "<channel>"}`. The tracker adds the peer to the channel's member list and returns a success response. Upon receiving it, the peer appends the channel to its local `self.channels` list and immediately calls `get_peer_list(channel=<channel>)` to obtain the current members of that channel. It then iterates through the returned peers (skipping itself and any already-connected ones) and automatically establishes a direct P2P TCP connection to each by performing the handshake sequence via `connect_peer()`. A short 0.1-second delay is added between successive connections. Finally, the peer prints how many new direct connections were successfully created, resulting in a fully meshed P2P group within the channel, ready for instant private messages and broadcasts without any further tracker interaction.

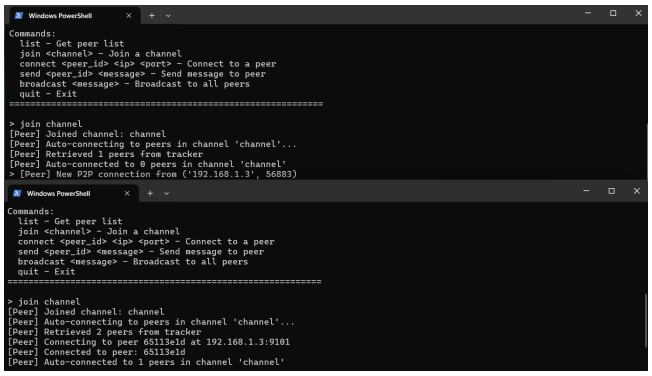

Figure 2.4: 2 peers connected by a channel

- connect <peer_id> <ip> <port> - Connect to a peer
  Upon entering `connect <peer_id> <ip> <port>`, the peer opens a direct TCP socket to the specified IP and port, bypassing the tracker entirely. It immediately sends a JSON handshake message containing its own 8-character peer ID (`{"type": "handshake", "peer_id": "..."}`). The remote peer, upon receiving this, stores the connection and replies with a `handshake_ack`. Once acknowledged, the local peer adds the socket to its connection dictionary and spawns a dedicated thread to listen for incoming messages from that peer. This establishes a persistent, direct P2P channel used for all subsequent communication with that peer.

- send <peer_id> <message> - Send message to connected peer by id
  When `send <peer_id> <message>` is issued, the peer looks up the previously established TCP socket for the given peer_id. Using this direct socket, it transmits a single JSON-encoded chat message with `type: "chat"`, the sender's ID, channel set to `"direct"`, the message text, and an ISO timestamp. The remote peer receives the data in

its listening thread, parses the JSON, and instantly displays the message in its terminal. The entire exchange is pure peer-to-peer; the tracker is not contacted at any point.

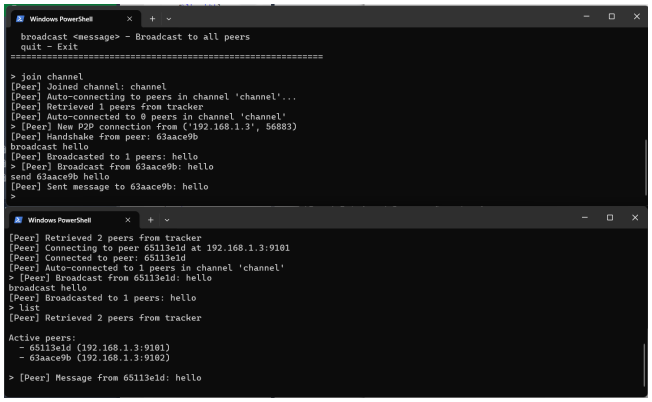

Figure 2.5: Successfully send a message to a peer

If the peer is not connected, an error message will be printed:

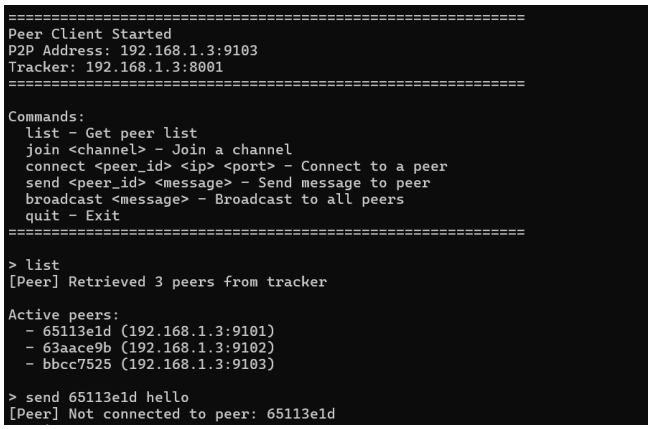

Figure 2.6: Error beacuse of no connection between two peers

- broadcast <message> - Broadcast to all peers.
  Typing `broadcast <message>` triggers the peer to perform a local fan-out: it iterates over every currently open P2P socket in its connection table and sends an identical JSON message to each one, marked with `type: "broadcast"` and `channel: "broadcast"`. Each connected peer receives the message independently through its own listening thread and prints it immediately. This mechanism ensures efficient mesh-wide broadcasting without requiring any central server or tracker involvement—the sender itself distributes the message to all its direct neighbours.
  Now we run another peer on port 9003, and connect it to channel "Channel", which was joined by the previous two peers. We broadcast a message from peer in port 9001, other can receive the broadcast

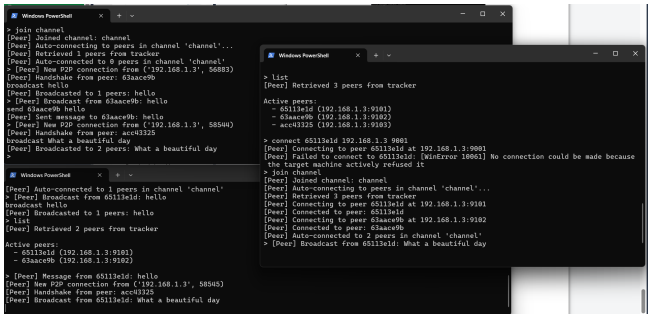

Figure 2.7: Enter Caption

- quit - Exit

  When the user types `quit` (or interrupts with Ctrl-C/D), the command loop terminates and the peer begins a graceful shutdown sequence. First, it sends a final HTTP POST request to the tracker's `/leave` endpoint containing its own peer_id. The tracker removes the peer from the global peer list and from all channel member lists. Simultaneously, the local peer performs a clean shutdown on every outgoing P2P socket (using `shutdown()` followed by `close()`), closes its own listening server socket, and sets its running flag to false. The user sees a confirmation that the tracker has been notified and the peer has left the network cleanly.