ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

BÁO CÁO

HỌC PHẦN: LẬP TRÌNH MẠNG 2526I_INT3304_1

XÂY DỰNG CHƯƠNG TRÌNH CHAT SỬ DỤNG GIAO THỨC PUBLISH/SUBSCRIBE

Giảng viên hướng dẫn: Hồ Đắc Phương

Học kỳ 1, Năm học 2025 - 2026

I. Giới thiệu

Nhóm chúng em thiết kế chương trình **MemeYourChat**, ứng dụng chat nhóm trên nền tảng WinSock với giao thức kiểu xuất bản/đăng ký hay publish/subscribe. Ứng dụng được cài đặt 100% bằng ngôn ngữ C/C++ trên cả Windows và Linux, sử dụng Winsock cho lập trình socket, với Server đóng vai trò Broker (môi giới) và Client đóng vai trò vừa là bên công bố thông tin, vừa là bên đăng ký nhận thông tin.

Hệ thống gồm hai thành phần chính: server chịu trách nhiệm quản lý kết nối, xác thực tài khoản (login/register, kiểm tra username tồn tại, kiểm tra mật khẩu), lưu lịch sử tin nhắn và tệp đính kèm theo từng nhóm; và client cung cấp giao diện dòng lệnh cho người dùng với các lệnh như /create, /join, /pm, /file để tạo/ tham gia nhóm, trò chuyện và gửi file, v.v. Ngoài ra, mỗi nhóm chat được tổ chức thành thư mục riêng trong cây *resources/<group>/logs* và *resources/<group>/files*, giúp việc lưu trữ và truy vết dữ liệu rõ ràng hơn.

Yêu cầu	Mô tả	Lệnh	Điểm yêu cầu
1. Chat text trực tiếp cho nhau	Các user đã đăng ký trên server có thể nhắn tin trực tiếp cho nhau.	**/pm <user> <msg>**	20%
2. Chat text theo nhóm	Các user đã đăng ký trên server có thể nhắn tin lên một nhóm group/topic. Tin nhắn đó sẽ được nhận bởi tất cả các user khác là thành viên trong nhóm, và hiển thị trên lịch sử chat của nhóm.	- User đã login thành công có thể tạo nhóm với lệnh **/create <group>**. - Lệnh **/join <group>** cho user đã login thành công tham gia các nhóm có trên server. - Để rời nhóm, người dùng có thể dùng lệnh **/leave <group>** và trở về nhóm default là **global.**	20%
3. Cho phép người dùng gửi file cho nhau	Các user		20%

4. Cho phép người dùng gửi file cho một nhóm	Các user đã đăng ký trên server có thể nhắn tin lên một nhóm group/topic. Tin nhắn đó sẽ được nhận bởi tất cả các user khác là thành viên trong nhóm, và hiển thị trên lịch sử chat của nhóm.	- Lệnh **/file <path>** gửi file lên nhóm của user	20%
5. Các chức năng khác	Game X/O 8x8	Game được chia thành game_server.c host game và game_client.c cho 2 người kết nối vào 1 port và chơi X/O đấu với nhau, chia thành 2 phiên bản client linux và windows	40%
6. Môi trường chạy	Ngôn ngữ C/C++		N/A

II. Bảng chia công việc

Tên	MSV	Công việc	Chia điểm
Hoàng Minh Nghĩa	22025508	- Thiết kế khung hoạt động - Refactor code - Thêm hỗ trợ cross-OS - Code game	33%
Phạm Xuân Dương	22025518	- Thiết kế GUI client - Kết nối api client và server - Code game	33%

Phan Vũ Liêm	22025504	- Cài đặt protocol - Viết code cho server và client (terminal) - Viết báo cáo	33%

III. Các giao thức

Ứng dụng sử dụng một giao thức tầng ứng dụng tự thiết kế dựa trên TCP. Mọi dữ liệu trao đổi giữa client và server đều được đóng gói theo cấu trúc chung PacketHeader kết hợp với phần payload linh hoạt.

3.1. Cấu trúc gói tin

Mỗi gói tin có hai phần:

- **Header:** cố định, mô tả loại gói và độ dài dữ liệu.
- **Payload:** vùng dữ liệu đi kèm, độ dài thay đổi, tối đa **MAX_PAYLOAD_SIZE** byte.

Cấu trúc header được định nghĩa như sau:

```
#pragma pack(push, 1)

typedef struct {

    uint8_t type;

    uint32_t payload_size;

    char target_id[MAX_ID_LEN]; // user/ hoặc group/

    char sender_id[MAX_ID_LEN]; // user id người gửi

} PacketHeader;

#pragma pack(pop)
```

Giải thích các trường:

- **type**: loại gói tin, tương ứng với các giá trị trong enum **PacketType**.
- **payload_size**: số byte dữ liệu đi kèm sau header.
- **target_id**: đích đến của gói tin, có thể là:
- "**group/<tên_nhóm>**" đối với tin nhắn nhóm.
- "**user/<user_id>**" đối với tin nhắn riêng.
- **sender_id**: định danh người gửi, chính là **user_id** mà client dùng khi đăng nhập.

Kích thước tối đa của payload được giới hạn bởi hằng số **MAX_PAYLOAD_SIZE** là 1024. Điều này giúp tránh tràn bộ đệm, đồng thời đơn giản hóa việc xử lý trên cả client và server.

3.2. Các loại gói tin (PacketType)

Enum **PacketType** định nghĩa toàn bộ các loại thông điệp mà hệ thống sử dụng:

```
typedef enum {
    LTM_LOGIN = 1,
    LTM_REGISTER,
    LTM_JOIN_GRP,
    LTM_LEAVE_GRP,
    LTM_MESSAGE,
    LTM_HISTORY,
    LTM_FILE_META,
    LTM_FILE_CHUNK,
    LTM_DOWNLOAD,
    LTM_ERROR,
    LTM_GROUP_CMD,
```

```
    LTM_AUTH_REQ,
    LTM_AUTH_RESP
} PacketType;
```

3.2.1. LTM_LOGIN

Dùng cho quá trình làm việc với tài khoản:

- Client gửi lên:
    - **sender_id**: user ID mà người dùng nhập.
    - **target_id**: để trống hoặc không sử dụng.
    - **payload**:
        - "**CHECK**" để hỏi server xem user có tồn tại hay không.
        - "**LOGIN|<password>**" để thực hiện đăng nhập.
- Server trả về:
    - Nếu đang xử lý mode "**CHECK**":
    - Gửi **LTM_ERROR** với payload "**USER_EXISTS**" hoặc "**USER_NOT_EXISTS**".
    - Nếu đang xử lý mode "**LOGIN**":
    - Nếu sai mật khẩu hoặc vi phạm độ dài, server gửi **LTM_ERROR** với các mã như "**INVALID_PASSWORD**", "**BAD_PASSWORD_LENGTH**" hoặc "**USER_NOT_EXISTS**.
    - Nếu đăng nhập thành công, server gửi gói **LTM_LOGIN** (ACK) với:
        - **type** = LTM_LOGIN
        - **sender_id** = "SERVER"
        - **target_id** = <username>
        - **payload_size** = 0

Sau khi login thành công, server tự động subscribe client vào topic "**user/<username>**" và nhóm mặc định "**group/global**", đồng thời gửi lại lịch sử nhóm global bằng các gói **LTM_HISTORY**.

3.2.2. LTM_REGISTER

Dùng để đăng ký tài khoản mới khi user chưa tồn tại:

- Client gửi lên:
    - **sender_id**: user ID muốn đăng ký.
    - **payload**: chứa mật khẩu ở dạng thuần văn bản (server sẽ xử lý và lưu dưới dạng hash trong database).
- Server phản hồi:
    - Nếu user đã tồn tại, server gửi **LTM_ERROR** với payload "**USER_ALREADY_EXISTS**".
    - Nếu mật khẩu không đạt yêu cầu độ dài, gửi "**BAD_PASSWORD_LENGTH**".
    - Nếu tạo tài khoản thành công thì gửi **LTM_ERROR** với payload "**REGISTER_OK**".
    - Đóng socket và yêu cầu client đăng nhập lại bằng tài khoản vừa tạo.

3.2.3. LTM_JOIN_GRP

Dùng để tham gia một nhóm chat đã tồn tại.

- Client gửi lên:
    - **target_id**: "group/<tên_nhóm>".
    - **sender_id**: user ID của người dùng.
    - **payload**: rỗng.
- Server xử lý:
    - Nếu nhóm tồn tại, server sẽ thêm socket vào danh sách subscriber của topic tương ứng. Sau đó, server gửi lịch sử tin nhắn gần đây của nhóm bằng nhiều gói **LTM_HISTORY**.
    - Nếu nhóm không tồn tại, server có thể trả về gói **LTM_ERROR** với một mã lỗi (ví dụ "**GROUP_NOT_FOUND**") và không subscribe client vào nhóm đó.

3.2.4. LTM_LEAVE_GRP

Dùng khi client rời khỏi một nhóm. Cơ chế hoạt động:

- Client gửi lên:
    - **target_id**: "group/<tên_nhóm>".
- Server xử lý:
    - Server loại socket đó khỏi danh sách subscriber của topic tương ứng.

3.2.5. LTM_MESSAGE

Dùng cho tin nhắn dạng văn bản. Cơ chế hoạt động:

- Client gửi lên:
    - Tin nhắn nhóm:
        - **target_id** = "group/<tên_nhóm>", ví dụ "group/global".
    - Tin nhắn riêng:
        - **target_id** = "user/<user_id>".
        - **sender_id**: user ID của người gửi.
        - **payload**: nội dung văn bản của tin nhắn.
- Server xử lý:
    - Ghi log lịch sử vào file theo nhóm hoặc theo topic.
    - Gửi lại gói LTM_MESSAGE và payload tương ứng cho tất cả subscriber của topic đó, trừ chính socket gửi ban đầu.

3.2.6. LTM_HISTORY

Dùng để phát lại lịch sử tin nhắn cho client khi vừa tham gia nhóm hoặc khi login.

- Server gửi xuống:
    - **sender_id**: "HISTORY".
    - **target_id**: topic cần phát lại, ví dụ "group/global".
    - **payload**: một dòng lịch sử đã được server lưu trong file, có dạng là:
        - `Timestamp|Topic|Sender|Kind|Content`

Client chỉ việc in ra dạng [HISTORY] <nội dung> để người dùng xem lại các tin mới gần đây.

3.2.7. LTM_FILE_META

Dùng khi gửi file. Cơ chế hoạt động:

- Client gửi lên:
    - **target_id**: topic nhận file, ví dụ "group/global" hoặc "user/bob".
    - **sender_id**: user ID.
    - **payload**: chuỗi "filename|size", ví dụ "meme.png|12345", trong đó:
    - **filename**: tên file gốc.
    - **size**: kích thước file theo byte.
- Server xử lý:
    - Tạo thư mục tương ứng với group theo cây resources/<group>/files.
    - Tạo file đích trong thư mục đó, ví dụ resources/global/files/<timestamp>_meme.png.
    - Ghi lại thông tin file vào lịch sử bằng **log_history** với loại "FILE".
    - Phát gói **LTM_FILE_META** cho toàn bộ subscriber của topic.

3.2.8. LTM_FILE_CHUNK

Dùng cho các đoạn dữ liệu nhị phân của file. Cơ chế hoạt động:

- Client gửi nhiều gói:
    - **type** = LTM_FILE_CHUNK.
    - **payload_size**: số byte dữ liệu thật sự trong chunk.
    - **payload**: dữ liệu nhị phân đọc từ file.
- Server xử lý:
    - Tìm **FileUpload** tương ứng với socket đó.
    - Ghi **payload** xuống file đã mở.

- Phát lại gói **LTM_FILE_CHUNK** cho các subscriber của topic, cho phép các client khác tải về.
- Khi đủ **expected_size**, server đóng file và giải phóng cấu trúc **FileUpload**.

3.2.9. LTM_DOWNLOAD

Được dành cho chức năng tải file cụ thể từ server. Trong phiên bản hiện tại, gói này có thể chưa được sử dụng hoàn chỉnh hoặc để dự phòng mở rộng.

Tùy theo cài đặt sau này, client có thể gửi yêu cầu LTM_DOWNLOAD với thông tin file cần tải, server sẽ phản hồi bằng các gói LTM_FILE_META và LTM_FILE_CHUNK tương ứng.

3.2.10. LTM_ERROR

Dùng cho mọi loại báo lỗi và mã trạng thái đơn giản giữa client và server. Cơ chế hoạt động

- Server gửi:
  - **sender_id** = "SERVER".
  - **payload**: là một chuỗi mã lỗi, ví dụ:
    - "USER_EXISTS", "USER_NOT_EXISTS".
    - "INVALID_PASSWORD", "BAD_PASSWORD_LENGTH".
    - "USER_ALREADY_EXISTS", "REGISTER_OK", "REGISTER_FAILED".
  - Các mã khác liên quan đến group hoặc file.
- Client nhận:
  - In thông báo lỗi tương ứng ra màn hình.
  - Trong một số trường hợp, đóng socket và cho người dùng chọn thử lại hoặc thoát.

3.2.11. LTM_GROUP_CMD

Dùng cho các lệnh quản lý nhóm trên server, hiện được dùng để xử lý:

`/create <name>`: tạo nhóm mới nếu chưa tồn tại, đồng thời server trả về danh sách nhóm.

`/leave <name>`: rời nhóm và trở về nhóm global

Cơ chế hoạt động:

- Client gửi lên
    - **target_id** = "group/<name>". *Ví dụ: group/global*
    - Trong đó payload có:
        - "**CREATE**" đối với lệnh /group.
        - "**REMOVE**" đối với lệnh /removegroup.
- Server xử lý:
    - Với "**CREATE**": kiểm tra topic, tạo nếu chưa có, cập nhật cấu trúc g_topics, tạo cây thư mục resources/<group>/logs và resources/<group>/files, rồi trả về danh sách các group cho client (thường bằng các gói LTM_MESSAGE hoặc một dạng thông báo hệ thống).
    - Với "**REMOVE**": kiểm tra điều kiện xóa, loại bỏ topic khỏi g_topics, có thể xóa thư mục tài nguyên tương ứng và gửi lại danh sách group cập nhật.

3.2.12. LTM_AUTH_REQ và LTM_AUTH_RESP

Hai loại gói này được thiết kế để phục vụ các kịch bản xác thực phức tạp hơn, ví dụ gửi câu hỏi từ server và nhận lựa chọn login/register từ client. Trong phiên bản hiện tại, phần lớn logic xác thực đã được gói trong LTM_LOGIN và LTM_REGISTER, vì vậy LTM_AUTH_REQ và LTM_AUTH_RESP có thể đang ở trạng thái dự phòng để phát triển sau.

IV. Các thư viện

Để đơn giản hóa mã nguồn ở phía client và server, hệ thống xây dựng một số thư viện hỗ trợ dùng chung. Các thư viện này đóng gói các thao tác mức thấp với socket và cơ sở dữ liệu, giúp mã xử lý nghiệp vụ trong server.c và client.c rõ ràng và dễ bảo trì hơn.

4.1. Thư viện xử lý mạng net_utils

Thư viện net_utils cung cấp hai hàm tiện ích cho việc gửi và nhận dữ liệu qua TCP socket trên nền WinSock2:

- File cài đặt: **net_utils.c**

- File khai báo: **net_utils.h**
- Thư viện sử dụng: **<WinSock2.h>**

4.1.1. Hàm send_all

```
int send_all(SOCKET sock, const void *buf, int len);
```

Chức năng:

- Đảm bảo gửi hết chính xác len byte dữ liệu qua socket TCP.
- Vòng lặp nội bộ gọi **send()** nhiều lần nếu mỗi lần chỉ gửi được một phần dữ liệu.
- Nếu xảy ra lỗi hoặc kết nối bị đóng, hàm trả về -1.
- Nếu gửi thành công toàn bộ, hàm trả về 0.

Ý nghĩa:

- Trên TCP, một lời gọi **send()** không đảm bảo toàn bộ dữ liệu được gửi trong một lần.
- Việc đóng gói logic gửi nhiều lần vào send_all giúp các phần khác của chương trình chỉ cần gọi một hàm duy nhất và không phải lặp lại đoạn mã xử lý lỗi và đếm số byte đã gửi.

4.1.2. Hàm recv_all

```
int recv_all(SOCKET sock, void *buf, int len);
```

Chức năng:

- Đảm bảo đọc đủ chính xác len byte từ socket, trừ khi có lỗi hoặc đầu kia đóng kết nối.
- Vòng lặp nội bộ gọi **recv()** nhiều lần cho tới khi:
- Đã nhận đủ len byte, hoặc
- Gặp lỗi (**SOCKET_ERROR**) hoặc **recv()** trả về 0 (kết nối đóng).
- Trả về 0 nếu nhận đủ dữ liệu, -1 nếu có lỗi hoặc kết nối bị đóng.

Ý nghĩa:

- Giao thức ứng dụng của hệ thống luôn gửi header với kích thước cố định (**sizeof(PacketHeader)**) và phần **payload** với kích thước được chỉ rõ trong trường **payload_size**.
- Sử dụng **recv_all** giúp server và client đọc chính xác đủ số byte mong muốn cho mỗi phần, tránh lỗi do đọc thiếu hoặc đọc dở dang khung gói tin.

4.1.3. Vai trò trong hệ thống

Cả phía client và server đều sử dụng **send_all** và **recv_all** để thao tác với cấu trúc **PacketHeader** và **payload**.

Việc trừu tượng hóa này giúp giảm lặp mã và tăng độ tin cậy cho toàn bộ giao tiếp mạng, đặc biệt khi làm việc với nhiều client đồng thời.

4.2. Thư viện truy cập cơ sở dữ liệu sqlite

Thư viện sqlite là lớp bao bọc mỏng (wrapper) quanh thư viện SQLite gốc, nhằm cung cấp các hàm truy cập dữ liệu người dùng đơn giản và an toàn hơn.

- File cài đặt: **sqlite.c**
- File khai báo: **sqlite.h**
- Thư viện sử dụng: **sqlite3.h** (thư viện SQLite chuẩn)

Cơ sở dữ liệu được sử dụng để lưu thông tin tài khoản, gồm tên đăng nhập và mật khẩu. Hệ thống hiện sử dụng một file SQLite, ví dụ **users.db**.

4.2.1. Khởi tạo và đóng cơ sở dữ liệu

```
int db_init(const char *db_path);

void db_close(void);
```

`db_init`:

- Mở hoặc tạo file cơ sở dữ liệu tại đường dẫn db_path.

- Lưu giữ một handle toàn cục g_user_db để các hàm khác sử dụng.
- Tạo bảng users nếu chưa tồn tại, với cấu trúc:

```
CREATE TABLE IF NOT EXISTS users (
  username TEXT PRIMARY KEY,
  password TEXT NOT NULL
);
```

- Trả về 0 nếu khởi tạo thành công, ngược lại trả về -1 và in thông báo lỗi.

`db_close`:

- Đóng kết nối tới cơ sở dữ liệu nếu đang mở.
- Giải phóng handle toàn cục g_user_db.

Ý nghĩa:

- **db_init** được gọi một lần khi khởi động server. Nếu không khởi tạo được cơ sở dữ liệu, server sẽ dừng.
- **db_close** được gọi khi server tắt để bảo đảm tài nguyên được giải phóng an toàn.

4.2.2. Kiểm tra tài khoản tồn tại

```
int db_user_exists(const char *username);
```

Chức năng:

- Kiểm tra xem một tài khoản có tồn tại trong bảng users hay không.
- Thực hiện truy vấn:

```
SELECT 1 FROM users WHERE username = ? LIMIT 1;
```

- Sử dụng sqlite3_prepare_v2, sqlite3_bind_text và sqlite3_step để thực thi truy vấn với tham số an toàn.

Trả về:

- 1 nếu có bản ghi thỏa mãn.
- 0 nếu không tồn tại hoặc có lỗi.

Vai trò:

- Được sử dụng trong luồng đăng nhập/đăng ký để:
- Phân biệt giữa trường hợp đăng nhập với tài khoản có sẵn và đăng ký tài khoản mới.
- Ngăn việc đăng ký trùng tên tài khoản.

4.2.3. Xác thực tài khoản

```
int db_verify_user(const char *username, const char *password);
```

Chức năng:

- Kiểm tra cặp (username, password) có hợp lệ hay không.
- Thực hiện truy vấn:

  ```
  SELECT 1 FROM users WHERE username = ? AND password = ? LIMIT 1;
  ```

Trả về:

- 1 nếu tồn tại một bản ghi trùng khớp.
- 0 nếu thông tin đăng nhập không chính xác hoặc có lỗi.

Vai trò:

- Là bước kiểm tra chính trong quá trình LTM_LOGIN mode "LOGIN".
- Nếu hàm trả về 0, server gửi mã lỗi "INVALID_PASSWORD" và đóng kết nối, đảm bảo chỉ người dùng nhập đúng mật khẩu mới được cấp quyền tham gia chat.

Lưu ý: trong phiên bản hiện tại mật khẩu đang được lưu dưới dạng văn bản thuần. Có thể mở rộng thư viện để xử lý hash mật khẩu trước khi lưu.

4.2.4. Tạo tài khoản mới

int db_create_user(const char *username, const char *password);

Chức năng:

- Tạo bản ghi tài khoản mới trong bảng users.
- Thực hiện truy vấn:

```
INSERT INTO users(username, password) VALUES(?, ?);
```

Trả về:

- 1 nếu chèn thành công.
- 0 nếu có lỗi (ví dụ trùng khóa chính, lỗi cú pháp hoặc lỗi kết nối).

Vai trò:

- Được sử dụng trong xử lý gói **LTM_REGISTER**.
- Kết hợp với **db_user_exists** và kiểm tra độ dài mật khẩu, đảm bảo rằng:
- Không tạo trùng username.
- Mật khẩu tuân thủ các quy định về độ dài.

Người dùng chỉ được thông báo đăng ký thành công sau khi việc ghi dữ liệu vào cơ sở dữ liệu hoàn tất.

V. Các chức năng backend

5.1. Vai trò tổng thể

Tệp server.c hiện thực lớp backend của hệ thống chat theo mô hình client–server. Server chịu trách nhiệm:

- Khởi tạo các thành phần nền tảng (Winsock, cơ sở dữ liệu SQLite, dịch vụ chủ đề, dịch vụ tệp).
- Lắng nghe kết nối TCP đến từ nhiều client và tạo luồng xử lý riêng cho từng kết nối.
- Giải mã các gói tin theo giao thức nội bộ PacketHeader/PacketType và điều phối tới các mô đun con.
- Thực hiện xác thực tài khoản (đăng nhập, đăng ký) dựa trên cơ sở dữ liệu người dùng.
- Quản lý nhóm chat (topic), lịch sử tin nhắn và truyền tệp tin.
- Giải phóng tài nguyên khi client ngắt kết nối hoặc khi server dừng hoạt động.

Kiến trúc được thiết kế theo hướng tách module, trong đó server.c đóng vai trò “điều phối trung tâm”, còn các chức năng chi tiết được triển khai trong các mô đun chuyên trách: topic_svc (quản lý chủ đề/nhóm), history (lịch sử), file_svc (truyền tệp).

5.2. Vòng đời kết nối và xử lý đa luồng

Trong hàm main, server:

- Khởi tạo thư viện Winsock (WSAStartup) và cơ sở dữ liệu người dùng (**db_init("users.db")**).
- Khởi tạo các dịch vụ:
    - **topic_svc_init()** để chuẩn bị cấu trúc quản lý topic và nhóm chat.
    - **file_svc_init()** để chuẩn bị trạng thái phục vụ truyền tệp.
    - Tạo socket lắng nghe TCP trên port mặc định **DEFAULT_PORT** (910), gọi bind() và listen().
    - Sau khi khởi động, server đi vào vòng lặp chấp nhận kết nối:
        - Mỗi kết nối mới từ client được nhận thông qua accept().

    - Với mỗi socket mới, server cấp phát một cấu trúc SOCKET trên heap, sau đó tạo một luồng mới (CreateThread) chạy hàm client_thread.
    - Luồng chính chỉ chịu trách nhiệm nhận kết nối, còn toàn bộ xử lý giao tiếp với từng client được thực hiện trong hàm client_thread.

Do đó cho phép nhiều client hoạt động song song, mỗi client được xử lý trong một luồng riêng biệt.

5.3. Vòng lặp xử lý trong client_thread

Hàm **client_thread** là lõi xử lý backend cho từng client. Tại đây, server:

- Đọc header gói tin **PacketHeader** bằng **recv_all**, bảo đảm nhận đủ toàn bộ cấu trúc.
- Kiểm tra kích thước payload (**payload_size**). Nếu lớn hơn **MAX_PAYLOAD_SIZE**, luồng sẽ dừng để tránh lỗi.
- Nếu có payload, server tiếp tục gọi **recv_all** để đọc đầy đủ nội dung vào buffer cục bộ.
- Dựa vào trường type trong header, server chuyển sang xử lý cụ thể trong một khối **switch (hdr.type)**.

Tất cả các gói tin đều được xử lý theo giao thức đã định nghĩa trong **protocol.h**, đảm bảo tính thống nhất giữa client và server.

5.4. Xử lý xác thực tài khoản

Backend hỗ trợ hai thao tác chính liên quan đến tài khoản: kiểm tra và đăng nhập (**LTM_LOGIN**), đăng ký (**LTM_REGISTER**).

5.4.1. Gói LTM_LOGIN

Khi nhận gói LTM_LOGIN, server:

- Gọi hàm **parse_auth(payload, mode, pw)** để tách:
    - **mode**: chuỗi chế độ, ví dụ "CHECK" hoặc "LOGIN".
    - **pw**: mật khẩu (nếu có).

Tùy theo giá trị mode là CHECK hoặc LOGIN:

CHECK:

- Server kiểm tra sự tồn tại của người dùng bằng **db_user_exists(hdr.sender_id)**.
- Trả về gói lỗi **LTM_ERROR** với payload "**USER_EXISTS**" hoặc "**USER_NOT_EXISTS**" thông qua hàm tiện ích send_err.

LOGIN:

- Server xác thực tài khoản bằng **db_verify_user(hdr.sender_id, pw)**.
- Nếu thông tin hợp lệ:
    - Gửi lại một gói **LTM_LOGIN** (ACK) từ sender_id = "SERVER" tới target_id = <username> để báo thành công.
    - Tự động đăng ký client vào hai topic:
    - Topic riêng "user/<username>".
    - Topic nhóm mặc định "group/global".
    - Gọi **history_replay(s, "group/global")** để gửi lại phần lịch sử gần nhất của nhóm mặc định.
- Nếu thông tin không hợp lệ:
    - Gửi gói **LTM_ERROR** với thông báo "**INVALID_PASSWORD**".

Cách xử lý này tách bạch rõ ràng giữa bước kiểm tra tài khoản tồn tại và bước xác thực mật khẩu, giúp client chủ động điều khiển luồng giao diện đăng nhập/đăng ký.

5.4.2. Gói LTM_REGISTER

Khi nhận gói LTM_REGISTER, server thực hiện tạo tài khoản mới bằng **db_create_user(hdr.sender_id, payload)**, trong đó payload chứa mật khẩu.

Trường hợp đăng ký thành công, server gửi gói LTM_ERROR với thông báo "**REGISTER_OK**" cho client. Trong trường hợp thất bại, server gửi thông báo "**REGISTER_FAILED**".

Sau khi xử lý xong yêu cầu đăng ký, server đóng socket của client, buộc người dùng đăng nhập lại với tài khoản vừa tạo, đảm bảo luồng trạng thái phía server luôn bắt đầu từ một phiên đăng nhập rõ ràng.

5.5. Quản lý nhóm chat và chủ đề

Việc quản lý topic và nhóm chat được đóng gói trong các hàm thuộc mô đun topic_svc. Server tương tác với mô đun này thông qua các gói

5.5.1. Tham gia và rời nhóm

LTM_JOIN_GRP:

Server kiểm tra sự tồn tại của nhóm bằng **topic_exists(hdr.target_id)**.

Nếu nhóm tồn tại:

- Gọi **topic_subscribe(s, hdr.target_id)** để đăng ký socket hiện tại vào nhóm.
- Gọi **history_replay(s, hdr.target_id)** để gửi lại lịch sử tin nhắn của nhóm cho client.

Nếu nhóm không tồn tại:

- Gửi gói LTM_ERROR với nội dung "**GROUP_NOT_FOUND. Use '/group create <name>' first**.".

LTM_LEAVE_GRP:

Server gọi **topic_unsubscribe(s, hdr.target_id)** để xóa socket khỏi danh sách subscriber của nhóm tương ứng.

5.5.2. Lệnh nhóm tổng quát LTM_GROUP_CMD

Gói **LTM_GROUP_CMD** được dùng cho các lệnh quản trị nhóm, ví dụ:

CREATE:

- Nếu **topic_exists(hdr.target_id)** trả về true, server gửi "**GROUP_ALREADY_EXISTS**".
- Ngược lại, server gọi **topic_create(hdr.target_id)** để tạo nhóm mới.

- Sau khi tạo, server gửi một gói LTM_MESSAGE với nội dung dạng "**Group '<target>' created successfully.**" từ sender_id = "SERVER" tới chính người yêu cầu, nhằm phản hồi trực tiếp hành động.

LIST:

- Server gọi **topic_get_list(list_buf, sizeof(list_buf))** để lấy danh sách các nhóm hiện có dạng chuỗi.
- Sau đó gửi danh sách này cho client dưới dạng gói **LTM_MESSAGE** với sender_id = "SERVER" và target_id = <username>.

Nhờ đó, client có thể tạo, liệt kê và làm việc với các nhóm một cách linh hoạt.

5.6. Xử lý tin nhắn và lịch sử

Hai mô đun topic_svc và history phối hợp để hiện thực quá trình gửi nhận tin nhắn:

- Với gói **LTM_MESSAGE:**
    - Server ghi log lịch sử thông qua history_log(hdr.target_id, hdr.sender_id, "MSG", payload). Thông tin lịch sử được lưu theo từng topic để phục vụ việc xem lại sau.
    - Sau đó, server gọi topic_route_msg(s, &hdr, payload) để chuyển tiếp gói tin đến tất cả subscriber của topic hdr.target_id, trừ socket gửi.
- Với chức năng xem lại lịch sử:
    - **history_replay(s, topic)** đọc file lịch sử của topic tương ứng, chọn một số dòng gần nhất, và gửi lại cho client bằng các gói **LTM_HISTORY**.
    - Chức năng này được gọi ngay sau khi:
        - Người dùng đăng nhập thành công vào nhóm mặc định "group/global".
        - Người dùng tham gia một nhóm mới thông qua LTM_JOIN_GRP.

Cách thiết kế này bảo đảm mỗi thành viên mới tham gia nhóm có bối cảnh hội thoại tối thiểu, tăng tính liên tục cho cuộc trò chuyện.

5.7. Xử lý truyền tệp tin

Server hỗ trợ truyền tệp qua hai loại gói:

LTM_FILE_META:

- Được xử lý bởi file_handle_meta(s, &hdr, payload).
- Payload chứa thông tin mô tả tệp (tên tệp, kích thước).
- Hàm xử lý sẽ:
    - Tạo file lưu trên server.
    - Ghi nhận trạng thái upload (socket, kích thước dự kiến, số byte đã nhận, đường dẫn lưu).
    - Đăng ký trạng thái này trong mô đun **file_svc** để các gói LTM_FILE_CHUNK tiếp theo có thể được ghi đúng nơi.

LTM_FILE_CHUNK:

- Được xử lý bởi **file_handle_chunk(s, &hdr, payload)**.
- Mỗi gói chứa một đoạn dữ liệu nhị phân của tệp.
- Hàm xử lý:
    - Ghi đoạn dữ liệu vào file tương ứng theo trạng thái upload đang giữ.
    - Cập nhật số byte đã nhận.
    - Khi đủ kích thước, đóng file và giải phóng trạng thái upload.
    - Đồng thời có thể chuyển tiếp dữ liệu tới các client khác trong cùng topic (tùy cài đặt của **file_svc**).

Khi client ngắt kết nối, một hàm có tên **file_cancel_uploads(s)** được gọi để hủy các upload còn dang dở và giải phóng tài nguyên liên quan.

5.8. Dọn dẹp tài nguyên

Khi vòng lặp xử lý trong **client_thread** kết thúc (do lỗi mạng, client ngắt kết nối hoặc server dừng):

- Server gọi **file_cancel_uploads(s)** để hủy bỏ và đóng mọi tệp đang được upload từ socket đó.
- Gọi **topic_remove_socket(s)** để loại bỏ socket khỏi mọi topic mà nó đã tham gia.
- Đóng socket bằng **closesocket(s)**, ghi log thông báo client đã ngắt kết nối.

Khi server tắt hoàn toàn (kết thúc hàm main):

- Gọi **topic_svc_cleanup()** để giải phóng tài nguyên của mô đun topic.
- Gọi **file_svc_cleanup()** để dọn dẹp các cấu trúc phục vụ truyền tệp.
- Gọi **db_close()** để đóng kết nối cơ sở dữ liệu SQLite.
- Gọi **WSACleanup()** để giải phóng tài nguyên Winsock.

Cách tổ chức xử lý dọn dẹp rõ ràng theo từng tầng đảm bảo hệ thống ổn định, tránh rò rỉ tài nguyên và dễ bảo trì khi mở rộng thêm chức năng trong tương lai.

VI. CÁC CHỨC NĂNG FRONTEND

Giao diện người dùng (Frontend) của hệ thống được xây dựng dựa trên kiến trúc hướng sự kiện, sử dụng bộ đôi thư viện **Dear ImGui** và **GLFW**. Giải pháp này cho phép xây dựng giao diện đồ họa (GUI) hiệu năng cao, tiêu tốn ít tài nguyên và đảm bảo khả năng chạy đa nền tảng (Windows/Linux) một cách nhất quán.

6.1. Kiến trúc Giao diện và Luồng xử lý

Frontend được thiết kế theo mô hình Immediate Mode GUI, trong đó toàn bộ giao diện được vẽ lại ở mỗi khung hình (frame) dựa trên trạng thái dữ liệu lưu trữ tại cấu trúc AppState.

- **Vòng lặp sự kiện** (Event Loop): Sử dụng GLFW để quản lý cửa sổ và bắt sự kiện từ thiết bị ngoại vi.
- **Quản lý trạng thái** (State Management): Một đối tượng g_State duy nhất (singleton-like) lưu trữ toàn bộ dữ liệu từ danh sách hội thoại, lịch sử tin nhắn đến trạng thái tải tệp tin.
- **Đồng bộ hóa dữ liệu**: Do việc nhận dữ liệu từ Server diễn ra trên một luồng riêng (ReceiverLoop), cơ chế std::mutex (data_mutex) được áp dụng triệt để khi truy cập vào

g_State để tránh hiện tượng tranh chấp dữ liệu (Race condition) giữa luồng UI và luồng Network.

6.2. Module Đăng nhập và Xác thực (Authentication)

Hàm **RenderLogin()** đảm nhận vai trò điều hướng người dùng trước khi tiến vào hệ thống chat chính.

- **Cấu hình kết nối**: Cho phép người dùng linh hoạt nhập địa chỉ IP Server và số hiệu cổng (Port).
- **Xử lý Logic**: Khi người dùng nhấn "Login" hoặc "Register", hệ thống sẽ gọi các hàm wrapper **Net_Connect()** để thiết lập socket, sau đó đóng gói gói tin **LTM_LOGIN** hoặc **LTM_REGISTER**.
- **Phản hồi trạng thái**: Biến login_status trong AppState được cập nhật liên tục từ luồng mạng để hiển thị các thông báo lỗi (như "Invalid Password", "User does not exist") hoặc trạng thái "Connecting..." trực tiếp trên giao diện.

6.3. Module Quản lý Hội thoại (Sidebar)

Thanh bên (Sidebar) được thiết kế để quản lý không gian làm việc của người dùng, bao gồm:

- **Tham gia nhóm** (Join Group): Cung cấp ô nhập tên nhóm và gửi gói tin LTM_JOIN_GRP tới Server.
- **Trò chuyện cá nhân** (Private Message): Cho phép tìm kiếm và khởi tạo phiên chat với người dùng khác bằng cách thêm prefix user/ vào ID hội thoại.
- **Danh sách hội thoại động**: Sử dụng cấu trúc std::map<std::string, Conversation> để liệt kê các phòng chat hiện có. Hệ thống phân biệt trực quan giữa Nhóm (màu xanh nhạt) và Cá nhân (màu xanh lá) để tối ưu trải nghiệm người dùng.

6.4. Module Hiển thị và Tương tác Tin nhắn

Vùng nội dung chính sử dụng các Widget của ImGui để hiển thị luồng dữ liệu thời gian thực:

- **Lịch sử Chat**: Hiển thị tin nhắn đi kèm dấu thời gian (Timestamp) và định danh người gửi. Hệ thống hỗ trợ tính năng Auto-scroll, tự động cuộn xuống tin nhắn mới nhất khi người dùng đang ở cuối danh sách.
- **Định dạng nội dung**: Phân biệt tin nhắn văn bản thông thường và tin nhắn chứa tệp tin. Nếu là tệp tin, UI sẽ vẽ một nút bấm đặc biệt cho phép người dùng kích hoạt lệnh tải về.

- **Gửi tin nhắn**: Hỗ trợ gửi qua nút "Send" hoặc phím tắt "Enter" thông qua cờ ImGuiInputTextFlags_EnterReturnsTrue.

6.5. Module Xử lý Truyền tải Tệp tin (File UI)

Tích hợp chặt chẽ với logic mạng để quản lý các luồng dữ liệu nhị phân:

- **Đính kèm tệp**: Sử dụng hàm wrapper **OpenFileDialog()** để gọi hộp thoại chọn file hệ thống (Windows API hoặc Terminal Input trên Linux).
- **Tiến trình tải** (Download Progress): Khi nhận được các chunk dữ liệu (LTM_FILE_CHUNK), một thanh tiến trình (ProgressBar) sẽ xuất hiện ở Sidebar, tính toán dựa trên tỉ lệ received_size / total_size để người dùng theo dõi thời gian thực.

6.6. Tích hợp Tiện ích Đa phương tiện và Đa nền tảng

Nhằm đáp ứng yêu cầu mở rộng của bài toán:

- **Tích hợp Game**: Một nút "X0" được bố trí cạnh ô nhập liệu, sử dụng hàm LaunchGame() để thực hiện gọi tiến trình hệ thống (System Call), khởi chạy game client được tích hợp sẵn.
- **Xử lý Font chữ**: Hệ thống nạp font **SVN-Arial.ttf** với dải ký tự **GetGlyphRangesVietnamese()**, đảm bảo hiển thị tiếng Việt hoàn hảo trên giao diện đồ họa OpenGL.
- **Trình chọn tệp đa nền tảng**: Sử dụng **_WIN32** để gọi **GetOpenFileNameA** trên Windows và giải pháp dự phòng trên Linux, đảm bảo tính linh hoạt của phần mềm.

VII. CÁC CHỨC NĂNG KHÁC – MINIGAME X/O (8x8)

Bên cạnh chức năng truyền tin và tệp tin, hệ thống còn tích hợp một minigame giải trí trực tuyến (X/O) nhằm minh chứng cho khả năng mở rộng của giao thức và khả năng xử lý đồng bộ hóa trạng thái thời gian thực giữa các Client thông qua Server trung gian.

7.1. Mô tả bài toán và Luật chơi

Minigame được xây dựng dựa trên trò chơi Tic-Tac-Toe truyền thống nhưng được cải tiến để tăng tính thử thách:

- **Kích thước bàn cờ:** 8x8 ô vuông, cho phép không gian chiến thuật rộng hơn so với bàn cờ 3x3 tiêu chuẩn.
- **Điều kiện thắng:** Người chơi cần tạo thành một hàng liên tục gồm **4 quân cờ** (X hoặc O) theo hàng ngang, hàng dọc hoặc đường chéo.
- **Cơ chế bắt cặp (Matchmaking):** Server sử dụng thuật toán **FCFS (First-Come, First-Served)**. Khi một Client kết nối, nếu chưa có ai chờ, họ sẽ vào hàng đợi; nếu đã có người chờ, Server sẽ lập tức khởi tạo một phiên đấu (GAME_SESSION) cho hai người đó.

7.2. Kiến trúc và Giao thức giao tiếp

Hệ thống sử dụng mô hình Client-Server chuyên biệt cho game, tách biệt với luồng chat chính để đảm bảo hiệu năng.

Các lệnh điều khiển (Protocol Commands):

- ASSIGN;[X/O]: Server chỉ định quân cờ cho từng người chơi.
- UPDATE;[BoardString];[Turn]: Server gửi toàn bộ trạng thái bàn cờ (chuỗi 64 ký tự) và thông tin lượt đi tiếp theo.
- MOVE;[row];[col]: Client gửi tọa độ ô muốn đánh về Server.
- GAMEOVER;[Message];[FinalBoard]: Thông báo kết quả cuối cùng và trạng thái bàn cờ lúc kết thúc.

7.3. Thiết kế phía Server (Game Server)

Server đóng vai trò là "Trọng tài" (Broker) tập trung, đảm bảo tính công bằng và toàn vẹn của trò chơi:

- **Quản lý đa luồng:** Mỗi phiên đấu giữa hai người chơi được xử lý trong một luồng riêng biệt (HandleGame), cho phép Server phục vụ nhiều cặp đấu cùng lúc mà không gây trễ.
- **Kiểm tra điều kiện thắng (Logic Win-Check):** Hàm CheckWin sử dụng thuật toán duyệt theo 4 hướng (ngang, dọc, 2 đường chéo) từ vị trí quân cờ vừa đánh để xác định chuỗi 4 quân liên tiếp.
- **Xử lý ngắt kết nối:** Nếu một trong hai người chơi thoát giữa chừng, Server sẽ gửi lệnh OPPONENT_QUIT để thông báo cho người còn lại và đóng phiên đấu an toàn, giải phóng tài nguyên socket.

7.4. Thiết kế phía Client (Game Client)

Client được phát triển với hai phiên bản tương thích tối ưu cho Windows và Linux:

- **Phiên bản Windows (GDI/Win32 API):** Sử dụng thư viện gdi32.lib để vẽ bàn cờ và các quân cờ thông qua hàm DrawBoard. Hệ thống sử dụng Windows Message (WM_LBUTTONDOWN) để bắt tọa độ chuột và ánh xạ vào ô cờ tương ứng.
- **Phiên bản Linux (GLFW/OpenGL):** Sử dụng thư viện đồ họa OpenGL để vẽ các thực thể hình học (draw_circle, draw_x). Luồng mạng (network_thread) chạy song song với luồng xử lý đồ họa để cập nhật trạng thái bàn cờ ngay khi nhận được gói tin từ Server.

7.5. Luồng hoạt động của hệ thống

1. Người chơi nhấn nút **"X0"** trên giao diện Chat, ứng dụng sẽ gọi hàm LaunchGame() để khởi chạy tiến trình Client game tương ứng với hệ điều hành.
2. Client game kết nối tới Port **55656** của Server.
3. Server bắt cặp người chơi theo thứ tự kết nối và bắt đầu gửi gói tin cập nhật trạng thái.
4. Dữ liệu bàn cờ được đồng bộ liên tục sau mỗi nước đi, đảm bảo tính nhất quán giữa hai Client.