Building a Scalable Chat Server with WebSocket and SQLite

Huo Yuxin

Jara Jiang Yunxin

Bob Liu Boyu

Shin Yang Bohan

Introduction

Bob Liu Boyu

Contents

1.Concepts

2.Features

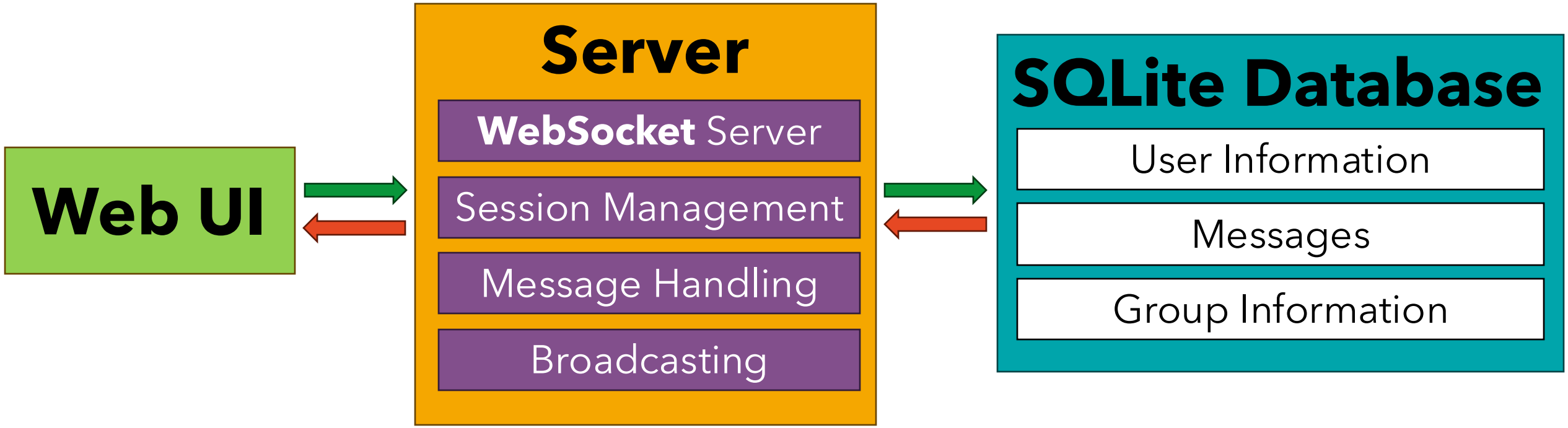
3.Implementation

4.Demo

Concepts

Bob Liu Boyu

Architecture Overview



*Passwords are hashed before being stored, and salt values are used to enhance security.

Features

Huo Yuxin

Features

- user management
- messaging functions
- group management
- Web - based Frontend

Features

User management

- Registration
- Login
- Track of online status
- Encrypt passwords

Messaging functions

- Public
- Private
- Group

Group management

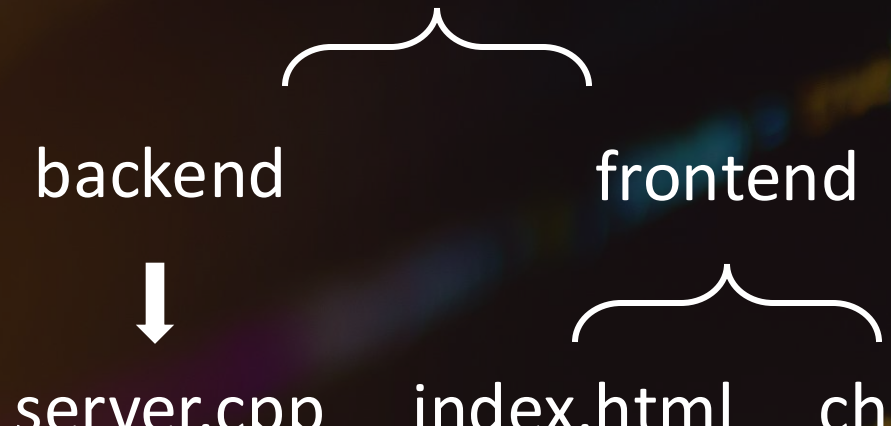
- Create groups
- Add and remove members
- View the composition
- Get historical messages

Web - based Frontend

Implementation

Jara Jiang Yunxin

Implementation



Backend

server.cpp { **Database**, **Security**, **Network Communication**

1. Design a database

- ***users***: stores user login information
- ***messages***: stores private and public chat messages
- ***groups***: stores the owner of the group
- ***group_members***: tracks group memberships
- ***group_messages***: stores group chat history

Automatically delete related members/messages when a group is removed.

Purpose: achieve structured storage, the data will not be lost even if the server is restarted.

```
/*table*/
exec_sql("CREATE TABLE IF NOT EXISTS users ("
         " username TEXT PRIMARY KEY,"
         " salt     BLOB NOT NULL,"    // 16 bytes
         " hash     BLOB NOT NULL);"); // 32 bytes SHA-256

exec_sql("CREATE TABLE IF NOT EXISTS messages ("
         " id        INTEGER PRIMARY KEY AUTOINCREMENT,"
         " sender    TEXT,"
         " receiver  TEXT," // 'all' Express public
         " message   TEXT,"
         " timestamp DATETIME DEFAULT CURRENT_TIMESTAMP);");

exec_sql("CREATE TABLE IF NOT EXISTS groups ("
         " id     INTEGER PRIMARY KEY AUTOINCREMENT,"
         " name   TEXT UNIQUE,"
         " owner  TEXT);");

exec_sql("CREATE TABLE IF NOT EXISTS group_members ("
         " id        INTEGER PRIMARY KEY AUTOINCREMENT,"
         " group_id  INTEGER,"
         " username  TEXT,"
         " is_owner  INTEGER DEFAULT 0,"
         " UNIQUE(group_id,username),"
         " FOREIGN KEY(group_id) REFERENCES groups(id) ON DELETE CASCADE);");

exec_sql("CREATE TABLE IF NOT EXISTS group_messages ("
         " id        INTEGER PRIMARY KEY AUTOINCREMENT,"
         " group_id  INTEGER,"
         " sender    TEXT,"
         " message   TEXT,"
         " timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,"
         " FOREIGN KEY(group_id) REFERENCES groups(id) ON DELETE CASCADE);");
```

2. Security

when a user signs up

⬇

gen_salt() generates a unique random salt

⬇

salt plus password are hashed

⬇

ensure identical passwords from different users produce different hashes

```
std::array<unsigned char, SALT_LEN> gen_salt() {
    std::array<unsigned char, SALT_LEN> s{};
    RAND_bytes(s.data(), SALT_LEN);
    return s;
```

```
std::array<unsigned char, HASH_LEN>
hash_password(const std::array<unsigned char, SALT_LEN> &salt,
              const std::string &password) {
```

```
auto salt = gen_salt();
auto hash = hash_password(salt, p);
```

We don't store the password directly, but store the salt and hash.

⬇

ensure security

2. Security

is_owner:
checks if a user is a group owner who can add or delete users

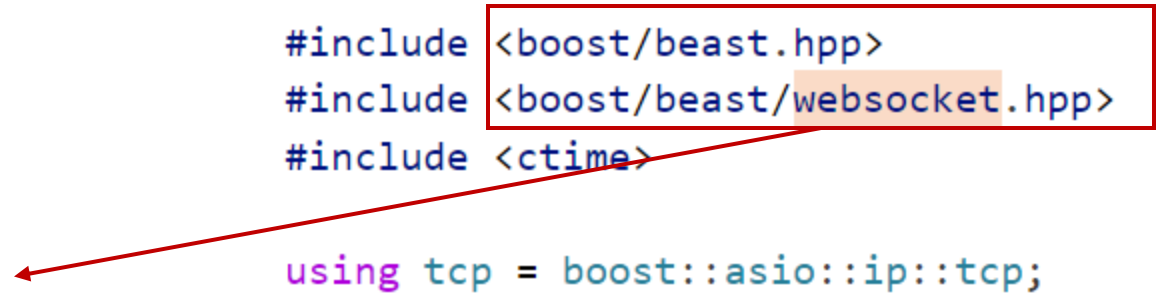
```
bool is_owner(int gid, std::string const &u) {
    sqlite3_stmt *st;
    sqlite3_prepare_v2(g_db,
                       "SELECT is_owner FROM group_members WHERE group_id=? AND username=?;",
                       -1, &st, 0);

    sqlite3_prepare_v2(g_db,
                       "INSERT OR IGNORE INTO group_members(group_id,username,is_owner)"
                       " VALUES(?,?,?);",

    sqlite3_prepare_v2(g_db,
                       "SELECT username,is_owner FROM group_members WHERE group_id=?;",
                       -1, &st, nullptr);

    members.push_back({{"username", reinterpret_cast<const char *>(sqlite3_column_text(st, 0))},
                       {"is_owner", sqlite3_column_int(st, 1) != 0}});
```

```
bool user_in_group(int gid, std::string const &u) {
    sqlite3_stmt *st;

        int gid = j.value("group_id", -1);
        if (gid < 0 || !user_in_group(gid, username_))
            return;

            return;
        if (!user_in_group(gid, username_))
            return;

        for (auto &s : g_sessions) {
            if (user_in_group(gid, s->name()))
                s->push_json(gm);
```

user_in_group:
ensures only group members can send or view group messages

3. Network communication

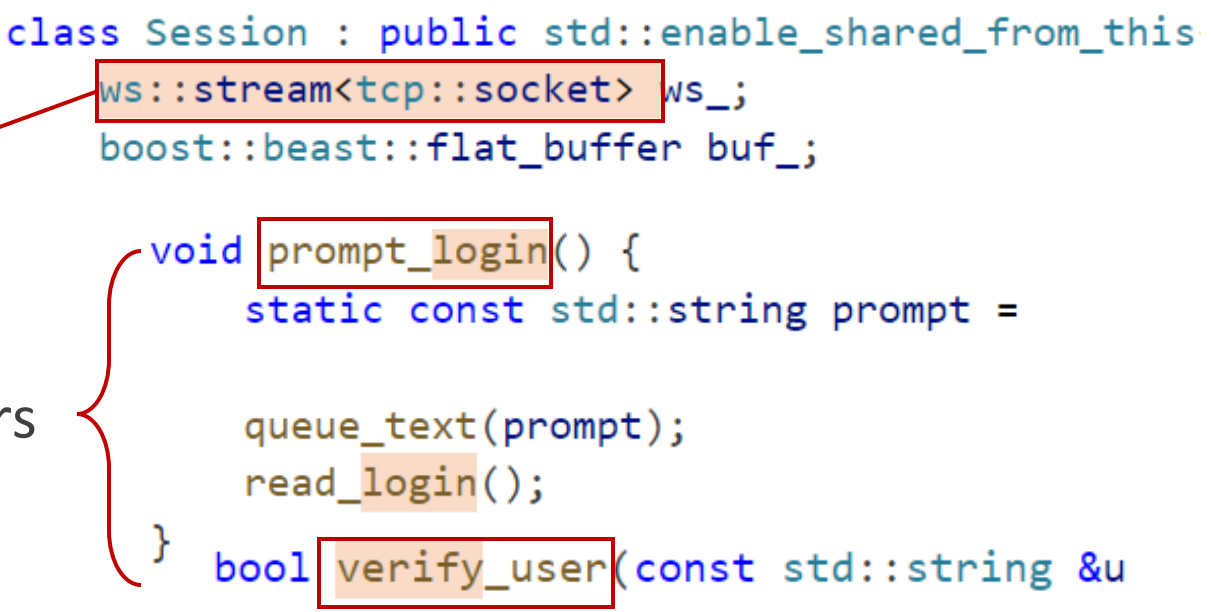
realize asynchronous communication



Core components

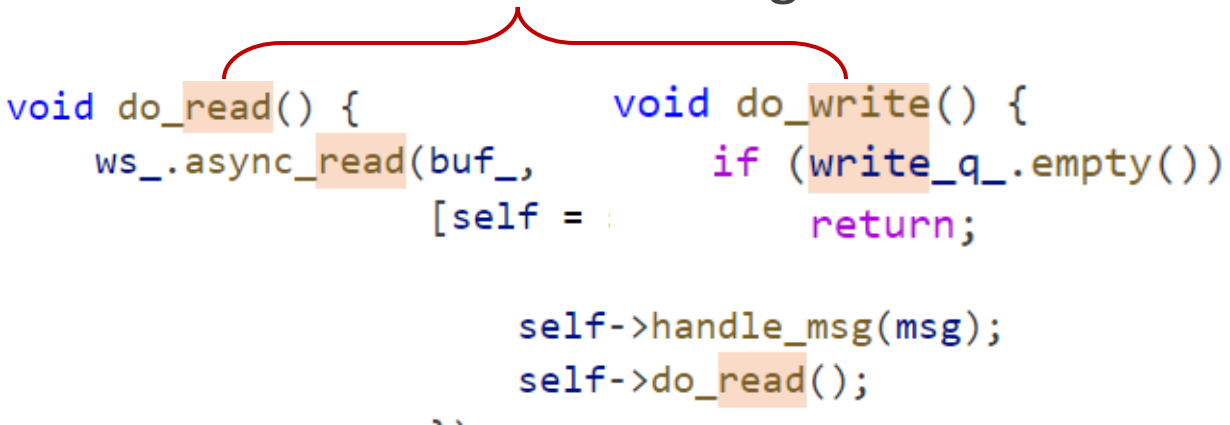
(1) Session Class: manages per-user communication

handles the WebSocket connection



authenticate users

send and receive messages



(2) **Message Handling**

handle_msg: processes different message types:

- Private messages
- Public messages
- Group messages

```
void handle_msg(std::string const &raw) {
    if (!raw.empty() && raw.front() == '{') {
        handle_json(raw);
        return;
    }
void broadcast_json(json const &j) {
    for (auto &s : g_sessions)
        s->push_json(j);
}
```

(3) **Broadcasting**

broadcast_json: sends JSON messages to all online clients (e.g., notifying user list updates)

```
void do_accept(boost::asio::io_context &ioc, tcp::acceptor &acc) {
    acc.async_accept(
        [&](boost::system::error_code ec, tcp::socket sock) {

                std::make_shared<Session>(std::move(sock))->start();
            do_accept(ioc, acc);
        });
```

(4) **Scalability**

do_accept: asynchronously accepts new connections, creating a Session for each.

handle high concurrency and avoid the overhead of thread switching and lock contention

Frontend

- *index.html*: handles the **structure and styling** of the page
- *chat.js*: contains the **core frontend logic**

```
<!-- chatroom/frontend/index.html ├ Telegram style & self bubble blue -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Chat Room</title>
    <style>
        /*  ====== Layout ====== */
        body {
            font-family: 'Segoe UI', Arial, 'Microsoft YaHei';
            margin: 0;
            padding: 0;
            display: flex;
            height: 100vh;
            background: □ #e5e5e5;
        }

        #sidebar {
            width: 260px;
            display: flex;
            flex-direction: column;
            background: #fff;
            border-right: 1px solid #ddd;
        }
```

index.html

```
/* ---------------- DOM & State ---------------- */
const ws = new WebSocket('ws://localhost:9002');
const userList = document.getElementById('user-list');
const groupList = document.getElementById('group-list');
const messageContainer = document.getElementById('message-container');
const input = document.getElementById('input');
const sendButton = document.getElementById('send-button');
const connectionStatus = document.getElementById('connection-status');
const chatTarget = document.getElementById('chat-target');
const sidebarTabs = document.querySelectorAll('.sidebar-tab');
const createGroupBtn = document.getElementById('create-group-btn');
const createGroupModal = document.getElementById('create-group-modal');
const manageGroupModal = document.getElementById('manage-members-modal');
const closeButtons = document.querySelectorAll('.close-button, .close-modal');

let myUsername = '';       // write after logging in successfully
let membersRequestPending = false;    // group members request throttling
const avatarColors = new Map();// username -> hsl()

const currentState = {
    targetType: 'room', // room / private / group
    targetId: null,
    targetName: '大厅',
    selectedUser: null,
    selectedGroup: null,
    groups: [],
    onlineUsers: []
};
```

chat.js

Core steps (*chat.js*)

1. Connect Frontend to Backend

```
const ws = new WebSocket('ws://localhost:9002');
```

2. State Management

Use the currentState object to store all states of the current chat interface.
For example, when a user clicks on a group: currentState.targetType = ‘group’

```
const currentState = {
    targetType: 'room', // room / private / group
```

```
        const u = document.getElementById('new-member-input').value.trim();
        if (u && currentState.selectedGroup) {
            ws.send(JSON.stringify({ type: 'add_group_member', group_id: currentState
            document.getElementById('new-member-input').value = '';
```

3. User Operations and Responses

sendMessage() determines the sending logic based on currentState.targetType:

- **Public message**: Send raw text directly.
- **Private message** : Send message in format @username + message content.
- **Group message** : Send JSON object {type: 'group_message', group_id: ..., content: ...}.

```
function sendMessage() {
    const text = input.value.trim(); if (!text) return;

}
sendButton.addEventListener('click', sendMessage);
input.addEventListener('keydown', e => { if (e.key === 'Enter') sendMessage(); });
```

4. UI Rendering

determine if a message
was sent by the current user

```
function isSelfMessage(raw) {
    if (!myUsername) return false;

function appendMessage(raw, type = 'user-msg') {
    const outgoing = (type === 'self-msg' || isSelfMessage(raw));
    const row = document.createElement('div');
```

```
function getAvatarColor(user) {
    if (!avatarColors.has(user))

    avatar.textContent = sender.charAt(0).toUpperCase();
    avatar.style.background = getAvatarColor(sender);
```

generate a unique color
for the sender's avatar

1. clear the current message container
2. add a "History Messages" separator
3. iterate through messages and
call appendMessage() for each

```
function displayHistory(arr) {
    if (!arr || !arr.length) return;

            case 'groups_list': updateGroupsList(j.groups); break;
            case 'history': displayHistory(j.messages); break;
            case 'create_group_response':
```

Demo

Shin Yang Bohan


Thanks for
listening!