Министерство науки и высшего образования Российской Федерации
Калужский филиал
федерального государственного бюджетного
образовательного учреждения высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ***ИУК «Информатика и управление»***

КАФЕДРА ***ИУК5 «Системы обработки информации»***

ДОМАШНЯЯ РАБОТА

«Разработка сетевых приложений»

ДИСЦИПЛИНА: «Сети и телекоммуникации»

Выполнил: студент гр. ИУК4-52Б _____ (____Губин Е.В.____)
(Подпись) (Ф.И.О.)

Проверил: _____ (___Смирнов М.Е.___)
(Подпись) (Ф.И.О.)

Дата сдачи (защиты):

Результаты сдачи (защиты):

- Балльная оценка:
- Оценка:

Калуга , 2024

Цель: получение практических навыков по разработке клиент-серверных приложений, использующих для связи механизм сокетов.

Задачи:

1. Создать приложение-сервер, предназначенное для параллельной обработки запросов и работающее в ОС Windows или *NIX.
2. Создать приложение-клиент, которое будет подключаться к серверу с удаленных компьютеров. Рекомендуется использовать интерфейс сокетов и протокол TCP.
3. Разработать и реализовать протокол прикладного уровня для взаимодействия приложений.
4. Обеспечить обмен произвольными данными между клиентом и сервером.

Схема функционирования сервера:

Схема 1: Функционирование сервера

Схема функционирования клиента:

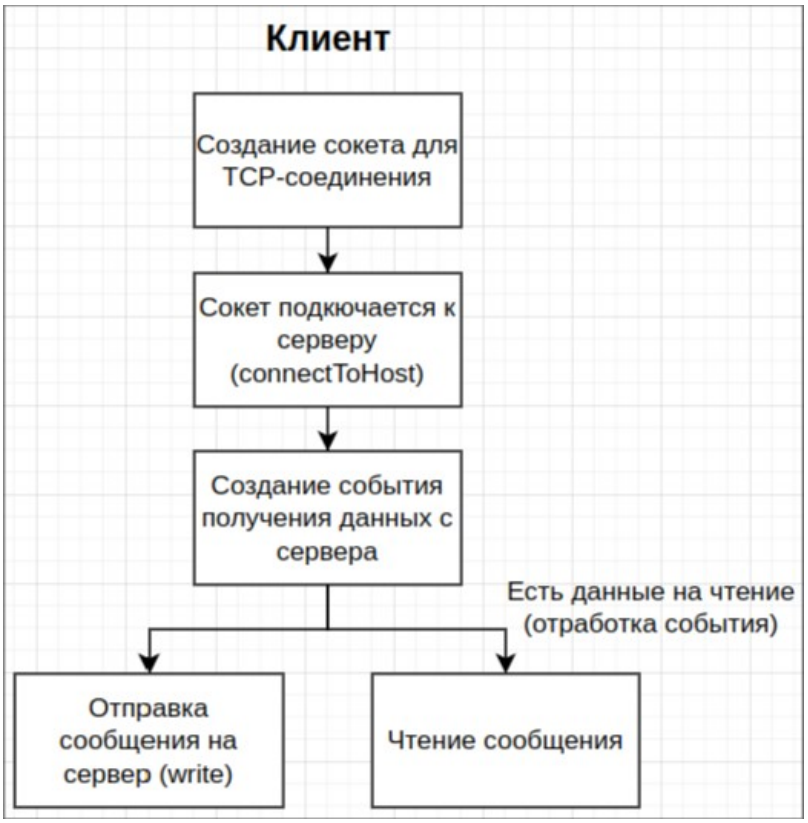

Схема 2: Функционирование клиента

Функциональность клиента:

Клиент заходит под уникальным никнеймом. После ему предоставляется возможность создать комнату или войти в уже существующую комнату. В одной комнате может находится хоть сколько пользователей. Сообщения помечаются автором этого сообщения.

Функциональность администратора:

У администратора есть возможность забанить пользователя насовсем (удалить из базы данных) и удалить комнату. После удаления комнаты пользователей выбросит в окно выбора комнаты. Если пользователь закрывает приложение (закрывает соединение), то в табличке Online users пользователь пропадает. При добавлении новой комнаты комната так же появляется в панели администратора.

Листинг программы сервера:

Сервер:

Client.cpp:

```
#include "Client.h"
```

```
Client::Client(int socket, std::string nickname)
{
    this→socket = socket;
    this→nickname = nickname;
}
```

Client.h:

```
#include <iostream>

struct Client
{
    int socket = 0;
    std::string nickname = "";
    Client(int, std::string);
};
```

DB.cpp:

```
#include "DB.h"

DB::DB(const char *connectionString)
{
    conn = PQconnectdb(connectionString);

    if (PQstatus(conn) ≠ CONNECTION_OK)
    {
        std::cerr << "Connection to database failed: " << PQerrorMessage(conn) <<
std::endl;
        PQfinish(conn);
        return;
    }

    std::cout << "Connected to database successfully!" << std::endl;
}

bool DB::addUser(std::string nickname)
{
    std::string query = "INSERT INTO users (nickname) VALUES ($1)";

    const char *paramValues[1] = {nickname.c_str()};
    PGresult *res = PQexecParams(conn, query.c_str(),
                                 1,
                                 nullptr,
                                 paramValues,
                                 nullptr,
                                 nullptr,
                                 0);

    if (PQresultStatus(res) ≠ PGRES_COMMAND_OK)
    {
        std::cerr << "Failed to insert user: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return false;
    }

    PQclear(res);
    return true;
}
```

```
bool DB::userExists(std::string nickname)
{
    std::string query = "SELECT 1 FROM users WHERE nickname = $1 LIMIT 1";

    const char *paramValues[1] = {nickname.c_str()};
    PGresult *res = PQexecParams(conn, query.c_str(), 1, nullptr, paramValues,
nullptr, nullptr, 0);

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        std::cerr << "Query failed: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return false;
    }

    bool exists = PQntuples(res) > 0;

    PQclear(res);
    return exists;
}

std::vector<std::string> DB::getRoomsNames()
{
    std::vector<std::string> roomsNames;
    const char *query = "SELECT name FROM rooms";

    PGresult *res = PQexec(conn, query);
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        std::cerr << "Failed to execute query: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return roomsNames;
    }

    int nRows = PQntuples(res);
    for (int i = 0; i < nRows; ++i)
    {
        roomsNames.push_back(PQgetvalue(res, i, 0));
    }

    PQclear(res);
    return roomsNames;
}

bool DB::roomExists(std::string roomName)
{
    const char *query = "SELECT 1 FROM rooms WHERE name = $1 LIMIT 1";
    const char *paramValues[1] = {roomName.c_str()};

    PGresult *res = PQexecParams(
        conn, query, 1, nullptr, paramValues, nullptr, nullptr, 0);

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        std::cerr << "Failed to execute query: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return false;
    }

    bool exists = PQntuples(res) > 0;
```

```
    PQclear(res);
    return exists;
}

bool DB::addRoom(std::string roomName)
{
    if (roomExists(roomName))
    {
        std::cerr << "Room already exists: " << roomName << std::endl;
        return false;
    }

    const char *query = "INSERT INTO rooms (name) VALUES ($1)";
    const char *paramValues[1] = {roomName.c_str()};

    PGresult *res = PQexecParams(
        conn, query, 1, nullptr, paramValues, nullptr, nullptr, 0);

    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        std::cerr << "Failed to insert room: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return false;
    }

    PQclear(res);
    return true;
}

bool DB::addMessage(std::string content, std::string senderNickname, std::string
roomName)
{
    const char* query = "INSERT INTO messages (content, sender_nickname, room_name,
date_time) VALUES ($1, $2, $3, NOW())";

    const char* values[3] = {content.c_str(), senderNickname.c_str(),
roomName.c_str()};
    int lengths[3] = {(int)content.size(), (int)senderNickname.size(),
(int)roomName.size()};
    int formats[3] = {0, 0, 0};

    PGresult* res = PQexecParams(conn, query, 3, nullptr, values, lengths, formats,
0);

    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        std::cerr << "Failed to insert message: " << PQerrorMessage(conn) <<
std::endl;
        PQclear(res);
        return false;
    }

    std::cout << "Message inserted successfully!" << std::endl;

    PQclear(res);
    return true;
}

std::vector<std::pair<std::string, std::string>> DB::getMessagesByRoom(std::string
roomName)
{
```

```
    std::vector<std::pair<std::string, std::string>> messages;

    const char* query = "SELECT sender_nickname, content FROM messages WHERE room_name
= $1 ORDER BY date_time ASC";

    const char* values[1] = {roomName.c_str()};
    int lengths[1] = {(int)roomName.size()};
    int formats[1] = {0};

    PGresult* res = PQexecParams(conn, query, 1, nullptr, values, lengths, formats,
0);

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        std::cerr << "Failed to get messages: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return messages;
    }

    int numRows = PQntuples(res);
    for (int i = 0; i < numRows; ++i)
    {
        std::string senderNickname = PQgetvalue(res, i, 0);
        std::string content = PQgetvalue(res, i, 1);
        messages.push_back({senderNickname, content});
    }

    PQclear(res);

    return messages;
}

bool DB::deleteUser(std::string nickname)
{
    std::string query = "DELETE FROM users WHERE nickname = $1";

    const char *paramValues[1] = {nickname.c_str()};
    PGresult *res = PQexecParams(conn, query.c_str(),
                                 1,
                                 nullptr,
                                 paramValues,
                                 nullptr,
                                 nullptr,
                                 0);

    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        std::cerr << "Failed to delete user: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return false;
    }

    PQclear(res);
    return true;
}

bool DB::deleteRoom(std::string roomName)
{
    std::string query = "DELETE FROM rooms WHERE name = $1";

    const char *paramValues[1] = {roomName.c_str()};
```

```
    PGresult *res = PQexecParams(conn, query.c_str(),
                                 1,
                                 nullptr,
                                 paramValues,
                                 nullptr,
                                 nullptr,
                                 0);

    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        std::cerr << "Failed to delete room: " << PQerrorMessage(conn) << std::endl;
        PQclear(res);
        return false;
    }

    PQclear(res);
    return true;
}
```

DB.h:

```
#include <libpq-fe.h>
#include <iostream>
#include <vector>

class DB
{
public:
    DB(const char *);
    bool addUser(std::string);
    bool userExists(std::string);
    std::vector<std::string> getRoomsNames();
    bool roomExists(std::string);
    bool addRoom(std::string);
    bool addMessage(std::string, std::string, std::string);
    std::vector<std::pair<std::string, std::string>> getMessagesByRoom(std::string);
    bool deleteUser(std::string);
    bool deleteRoom(std::string);

private:
    PGconn *conn = nullptr;
};
```

Dict.cpp:

```
#include "Dict.h"

Dict::Pair::Pair(std::string key, int value, std::string nickname)
{
    this->key = key;
    this->value.push_back(std::make_pair(value, nickname));
}

void Dict::addValue(std::string key, int value, std::string nickname)
{
    for (Pair &pair : pairs)
    {
        if (pair.key == key)
        {
            pair.value.push_back(std::make_pair(value, nickname));
```

```
            return;
        }
    }
    pairs.push_back(Pair(key, value, nickname));
}

void Dict::banUser(std::string banedUser)
{
    for (Pair &pair : pairs)
    {
        for (auto it = pair.value.begin(); it != pair.value.end(); ++it)
        {
            if (it->second == banedUser)
            {
                std::string deleteUserString = "ban_user\n";
                send(it->first, deleteUserString.c_str(), deleteUserString.size(), 0);
                pair.value.erase(it);
                break;
            }
        }
    }
}

void Dict::deleteRoom(std::string roomName)
{
    for (auto it = pairs.begin(); it != pairs.end(); ++it)
    {
        if (it->key == roomName)
        {
            std::string deleteRoomString = "delete_room\n";
            for (std::pair<int, std::string> clientNickname : it->value)
            {
                send(clientNickname.first, deleteRoomString.c_str(),
deleteRoomString.size(), 0);
            }
            pairs.erase(it);
            return;
        }
    }
}

void Dict::broadcastMessage(int client, std::string content, std::string nickname,
std::string roomName)
{
    for (Pair &pair : pairs)
    {
        if (pair.key == roomName)
        {
            std::string newMessageNewLine = "new_message_from_server\n";
            std::string contentNewLine = content + '\n';
            std::string nicknameNewLine = nickname + '\n';

            for (std::pair<int, std::string> &value : pair.value)
            {
                if (value.first != client)
                {
                    send(value.first, newMessageNewLine.c_str(),
newMessageNewLine.size(), 0);
                    send(value.first, contentNewLine.c_str(), contentNewLine.size(),
0);
```

```
                    send(value.first, nicknameNewLine.c_str(), nicknameNewLine.size(),
0);
                }
            }
            return;
        }
    }
}

std::set<std::string> Dict::getSet()
{
    std::set<std::string> result;
    for (Pair &pair : pairs)
    {
        for (std::pair<int, std::string> clientNickname : pair.value)
        {
            result.insert(clientNickname.second);
        }
    }
    return result;
}

void Dict::deleteClient(int client, std::string roomName)
{
    for (Pair &pair : pairs)
    {
        if (pair.key == roomName)
        {
            auto it = std::remove_if(pair.value.begin(), pair.value.end(),
                                     [client](const std::pair<int, std::string>
&clientNickname)
                                     {
                                         return clientNickname.first == client;
                                     });
            pair.value.erase(it, pair.value.end());
            return;
        }
    }
}
```

Dict.h:

```
#include <vector>
#include <iostream>
#include <netinet/in.h>
#include <set>
#include <algorithm>

class Dict
{
public:
    void addValue(std::string, int, std::string);
    void broadcastMessage(int, std::string, std::string, std::string);
    std::set<std::string> getSet();
    void deleteClient(int, std::string);
    void banUser(std::string);
    void deleteRoom(std::string);
private:
    struct Pair
    {
```

```
        Pair(std::string, int, std::string);
        std::string key = "";
        std::vector<std::pair<int, std::string>> value;
    };

    std::vector<Pair> pairs;
};
```

Server.cpp:

```
#include "Server.h"

Server::Server(int PORT)
{
    server = socket(AF_INET, SOCK_STREAM, 0);
    if (server == -1)
    {
        perror("Socket creation failed: ");
        return;
    }

    socketSettings.sin_family = AF_INET;
    socketSettings.sin_addr.s_addr = INADDR_ANY;
    socketSettings.sin_port = htons(PORT);

    this→PORT = PORT;

    if (bind(server, reinterpret_cast<sockaddr *>(&socketSettings),
sizeof(socketSettings)) == -1)
    {
        perror("Bind failed: ");
        return;
    }

    if (listen(server, 5) == -1)
    {
        perror("Listen failed");
        return;
    }

    db = new DB("host=localhost port=5432 dbname=rooms user=postgres password=123");
}

void Server::startListening()
{
    std::cout << "Server listening on port = " << PORT << "." << std::endl;

    while (true)
    {
        sockaddr_in clientAddr{};
        socklen_t clientLen = sizeof(clientAddr);

        int client = accept(server, reinterpret_cast<sockaddr *>(&clientAddr),
&clientLen);
        if (client == -1)
        {
            perror("Client accept failed: ");
            continue;
        }
```

```
        std::thread(&Server::handleConnections, this, client).detach();
    }
}

void Server::handleConnections(int client)
{
    char buffer[BUFFER_SIZE];

    int bytesRead = recv(client, buffer, sizeof(buffer) - 1, 0);
    if (bytesRead == -1)
    {
        std::cerr << "Failed to receive data or connection closed." << std::endl;
        close(client);
        return;
    }

    if (bytesRead == 0)
    {
        std::cerr << "Connection closed by client." << std::endl;
        close(client);
        return;
    }

    buffer[bytesRead] = '\0';
    std::string data(buffer);

    std::string action = readRowFromChannel(data);

    if (action == "new_user")
    {
        std::string nickname = readRowFromChannel(data);
        if (db->userExists(nickname))
        {
            std::string error = "User already exists\n";
            send(client, error.c_str(), error.size(), 0);
            return;
        }
        if (!db->addUser(nickname))
        {
            std::string error = "Error with adding user\n";
            send(client, error.c_str(), error.size(), 0);
            return;
        }
        std::string ok = "New user was successfully added\n";
        send(client, ok.c_str(), ok.size(), 0);
    }
    else if (action == "get_rooms")
    {
        std::string nickname = readRowFromChannel(data);
        std::thread(&Server::handleRoomsList, this, client, nickname).detach();
    }
    else if (action == "get_messages")
    {
        std::string roomName = readRowFromChannel(data);
        std::string nickname = readRowFromChannel(data);
        std::thread(&Server::handleRoom, this, client, roomName, nickname).detach();
    }
    else if (action == "get_info_for_admin")
    {
        std::thread(&Server::handleAdmin, this, client).detach();
    }
```

```
}

void Server::handleAdmin(int admin)
{
    if (this->admin != nullptr)
    {
        std::cerr << "Admin already on server";
        return;
    }

    this->admin = new int(admin);

    std::set<std::string> clients;

    {
        std::lock_guard<std::mutex> lock(clientsMutex);

        for (Client &clientInRoomList : clientsInRoomsList)
        {
            clients.insert(clientInRoomList.nickname);
        }

        std::set<std::string> clientsRoomsSet = clientsRooms.getSet();

        clients.insert(clientsRoomsSet.begin(), clientsRoomsSet.end());

        std::string onlineUsersResult = "";

        for (auto &client : clients)
        {
            onlineUsersResult += "online_users\n" + client + '\n';
        }

        send(admin, onlineUsersResult.c_str(), onlineUsersResult.size(), 0);

        std::vector<std::string> roomsNames = db->getRoomsNames();

        std::string roomsNamesResult = "";
        for (std::string &roomName : roomsNames)
        {
            roomsNamesResult += "existing_rooms\n" + roomName + '\n';
        }

        send(admin, roomsNamesResult.c_str(), roomsNamesResult.size(), 0);
    }

    while (true)
    {
        char buffer[BUFFER_SIZE];

        int bytesRead = recv(admin, buffer, sizeof(buffer) - 1, 0);
        if (bytesRead == 0)
        {
            std::cerr << "Connection closed by client." << std::endl;
            close(admin);
            return;
        }

        buffer[bytesRead] = '\0';
        std::string data(buffer);
```

```
        std::string action = readRowFromChannel(data);

        if (action == "ban_user")
        {
            std::string nicknameBaned = readRowFromChannel(data);
            db->deleteUser(nicknameBaned);

            for (auto it = clientsInRoomsList.begin(); it != clientsInRoomsList.end();
++it)
            {
                if (it->nickname == nicknameBaned)
                {
                    std::string deleteUserString = "ban_user\n";
                    send(it->socket, deleteUserString.c_str(),
deleteUserString.size(), 0);
                    clientsInRoomsList.erase(it);
                    break;
                }
            }

            clientsRooms.banUser(nicknameBaned);
        }
        if (action == "delete_room")
        {
            std::string roomDeleted = readRowFromChannel(data);
            db->deleteRoom(roomDeleted);
            clientsRooms.deleteRoom(roomDeleted);

            std::string rowToDeletedFromRooms = "delete_room\n" + roomDeleted + '\n';
            for (Client &clientInRoomsList : clientsInRoomsList)
            {
                send(clientInRoomsList.socket, rowToDeletedFromRooms.c_str(),
rowToDeletedFromRooms.size(), 0);
            }
            if (this->admin != nullptr)
            {
                std::string deleteRoomRow = "delete_room\n" + roomDeleted + '\n';
                send(admin, deleteRoomRow.c_str(), deleteRoomRow.size(), 0);
            }
        }
    }
}

std::string Server::readRowFromChannel(std::string &data)
{
    size_t pos = data.find('\n');
    if (pos == std::string::npos)
    {
        std::cerr << "Incorrect format of connect data." << std::endl;
        return "";
    }
    std::string row = data.substr(0, pos);

    data.erase(0, pos + 1);

    return row;
}

void Server::handleRoomsList(int client, std::string nickname)
{
    {
```

```
        std::lock_guard<std::mutex> lock(clientsMutex);
        clientsInRoomsList.push_back(Client(client, nickname));
    }

    std::cerr << "Добавление\t" << nickname << std::endl;

    std::vector<std::string> roomsNames = db->getRoomsNames();

    for (std::string &roomName : roomsNames)
    {
        std::string roomsSend = "rooms\n" + roomName + '\n';
        send(client, roomsSend.c_str(), roomsSend.size(), 0);
    }

    if (admin != nullptr)
    {
        std::string newUserStringToAdmin = "new_online_user\n" + nickname + '\n';
        send(*admin, newUserStringToAdmin.c_str(), newUserStringToAdmin.size(), 0);
    }

    while (true)
    {
        char buffer[BUFFER_SIZE];

        int bytesRead = recv(client, buffer, sizeof(buffer) - 1, 0);
        if (bytesRead == 0)
        {
            std::cerr << "Failed to receive data or connection closed." << std::endl;
            close(client);
            auto it = std::remove_if(clientsInRoomsList.begin(),
clientsInRoomsList.end(),
                                     [client](const Client &user)
                                     {
                                         return user.socket == client;
                                     });
            clientsInRoomsList.erase(it, clientsInRoomsList.end());
            std::cerr << "Удаление:\t" << clientsInRoomsList.size() << std::endl;

            if (admin != nullptr)
            {
                std::string deleteUserStringToAdmin = "delete_online_user\n" +
nickname + '\n';
                send(*admin, deleteUserStringToAdmin.c_str(),
deleteUserStringToAdmin.size(), 0);
            }

            return;
        }

        buffer[bytesRead] = '\0';
        std::string data(buffer);

        std::string action = readRowFromChannel(data);

        if (action == "new_room")
        {
            std::string newRoom = readRowFromChannel(data);
            if (db->roomExists(newRoom))
            {
                continue;
            }
```

```
            if (!db->addRoom(newRoom))
            {
                continue;
            }
            broadcastNewRoom(newRoom);
            if (admin != nullptr)
            {
                std::string newRoomStringToAdmin = "new_room\n" + newRoom + '\n';
                send(*admin, newRoomStringToAdmin.c_str(),
newRoomStringToAdmin.size(), 0);
            }
        }
    }
}

void Server::broadcastNewRoom(std::string room)
{
    std::lock_guard<std::mutex> lock(clientsMutex);

    for (Client &clientInRoom : clientsInRoomsList)
    {
        std::string newRoom = "new_room\n" + room + '\n';
        send(clientInRoom.socket, newRoom.c_str(), newRoom.size(), 0);
    }
}

void Server::handleRoom(int client, std::string roomName, std::string nickname)
{
    {
        std::lock_guard<std::mutex> lock(clientsMutex);
        clientsRooms.addValue(roomName, client, nickname);
    }

    std::vector<std::pair<std::string, std::string>> messages = db-
>getMessagesByRoom(roomName);

    std::string resultString = "messages\n";

    for (std::pair<std::string, std::string> &message : messages)
    {
        std::string senderNewLine = message.first + '\n';
        std::string messageNewLine = message.second + '\n';
        resultString += senderNewLine + messageNewLine;
    }

    send(client, resultString.c_str(), resultString.size(), 0);

    if (admin != nullptr)
    {
        std::string newUserStringToAdmin = "new_online_user\n" + nickname + '\n';
        send(*admin, newUserStringToAdmin.c_str(), newUserStringToAdmin.size(), 0);
    }

    while (true)
    {
        char buffer[BUFFER_SIZE];

        int bytesRead = recv(client, buffer, sizeof(buffer) - 1, 0);
        if (bytesRead == 0)
        {
            std::cerr << "Connection closed by client." << std::endl;
```

```
            close(client);
            clientsRooms.deleteClient(client, roomName);

            if (admin != nullptr)
            {
                std::string deleteUserStringToAdmin = "delete_online_user\n" +
nickname + '\n';
                send(*admin, deleteUserStringToAdmin.c_str(),
deleteUserStringToAdmin.size(), 0);
            }
            return;
        }

        buffer[bytesRead] = '\0';
        std::string data(buffer);

        std::string action = readRowFromChannel(data);

        if (action == "new_message")
        {
            std::string content = readRowFromChannel(data);
            db->addMessage(content, nickname, roomName);
            clientsRooms.broadcastMessage(client, content, nickname, roomName);
        }
    }
}
```

Server.h:

```
#include <netinet/in.h>
#include <iostream>
#include <thread>
#include <unistd.h>
#include "DB.h"
#include <vector>
#include <mutex>
#include "Dict.h"
#include "Client.h"
#include <set>
#include <algorithm>

class Server
{
public:
    Server(int);
    void startListening();

private:
    int server = -1;
    sockaddr_in socketSettings{};

    DB *db = nullptr;

    int PORT = -1;

    static const int BUFFER_SIZE = 1024;

    std::mutex clientsMutex;

    std::vector<Client> clientsInRoomsList;
```

```
    Dict clientsRooms = Dict();

    void handleConnections(int);
    void handleRoomsList(int, std::string);
    void broadcastNewRoom(std::string);
    void handleRoom(int, std::string, std::string);
    void handleAdmin(int);

    int *admin = nullptr;

    std::string readRowFromChannel(std::string &);
};
```

Main.cpp

```
#include "Server.h"

constexpr int PORT = 5000;

int main()
{
    Server *server = new Server(PORT);

    server→startListening();

    return 0;
}
```

Результаты работы:

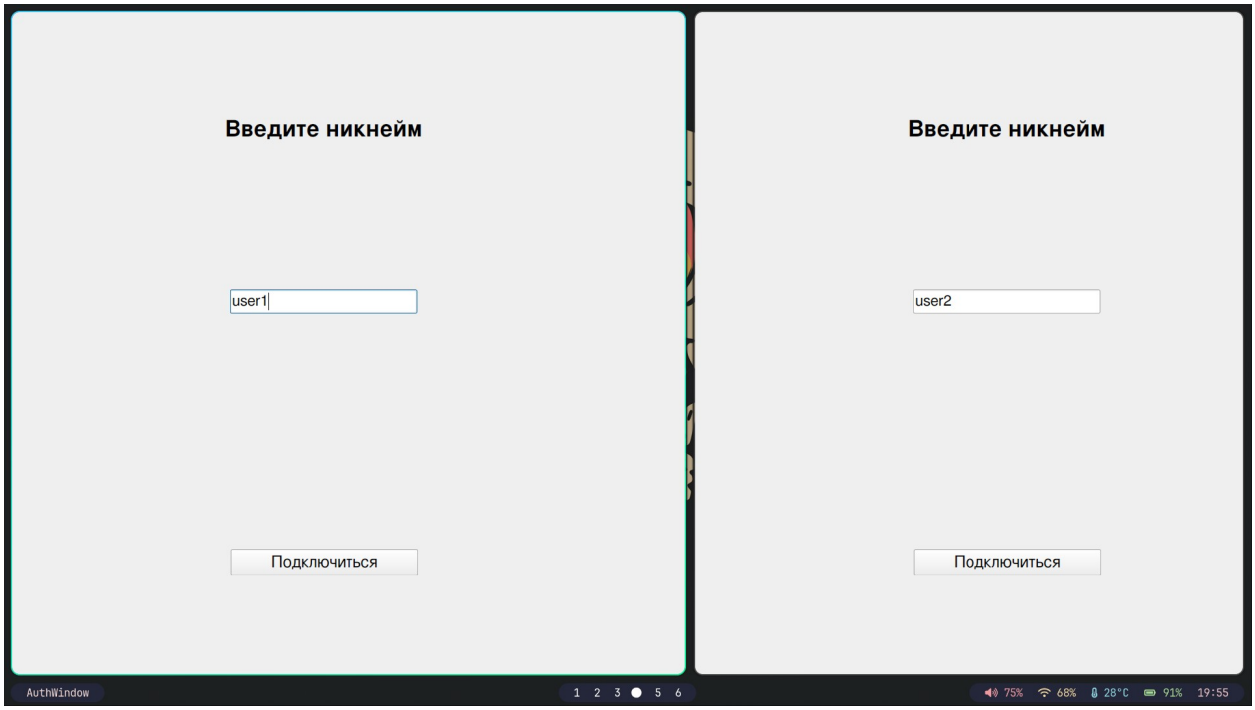

Рис. 1: Авторизация двух пользователей

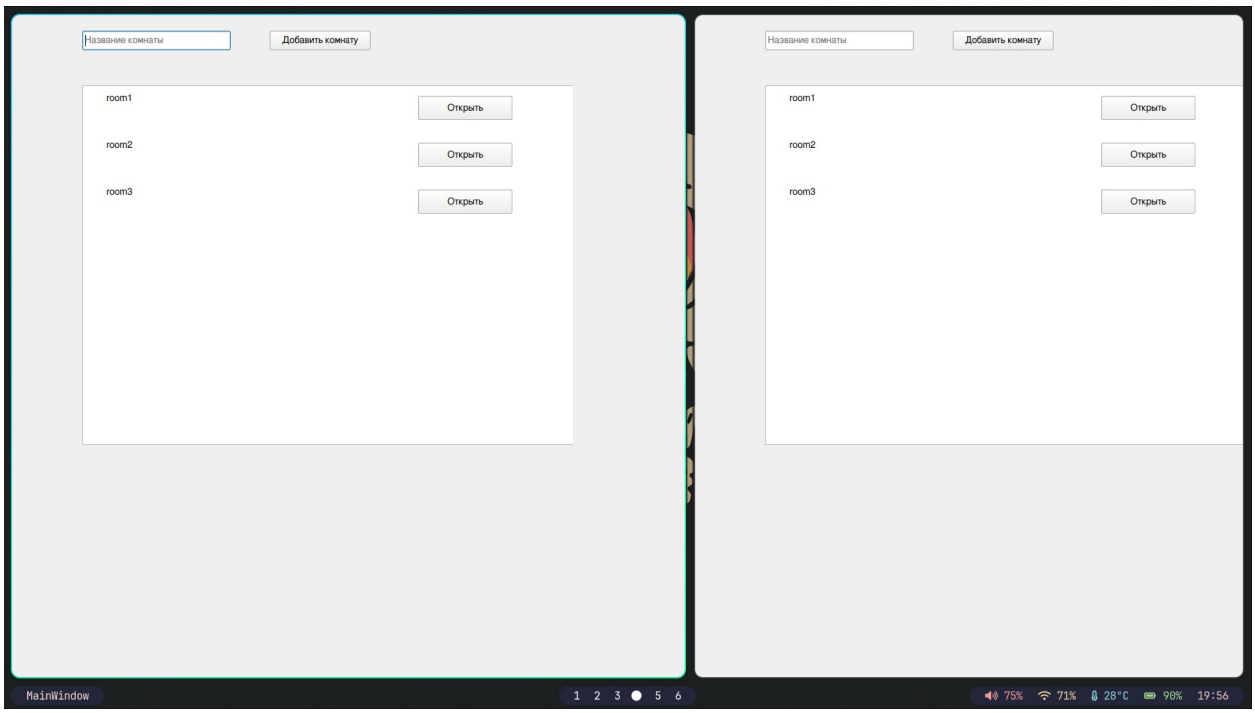

Рис. 2: Возможность выбора и создания комнаты для пользователей

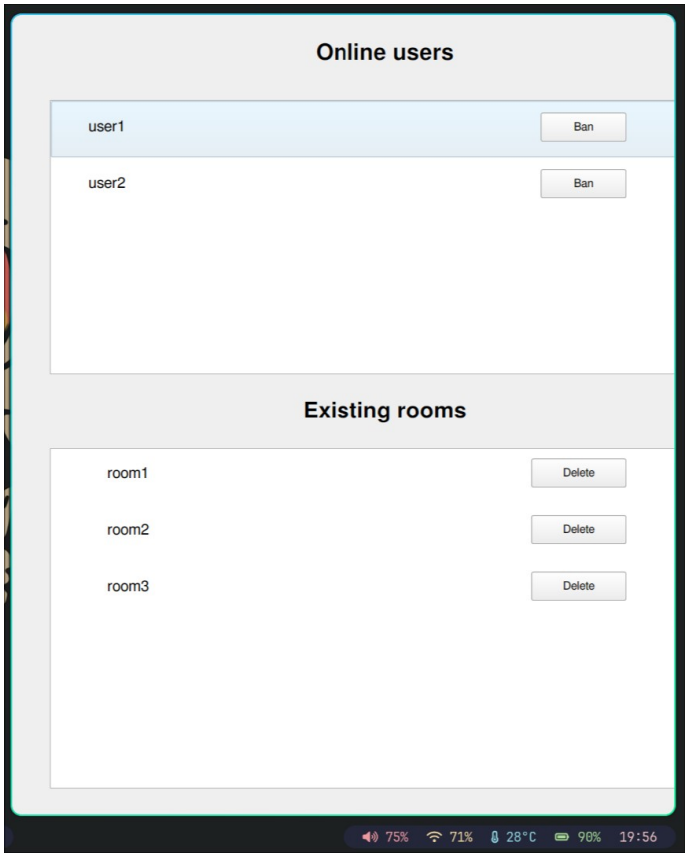

Рис. 3: Панель администратора

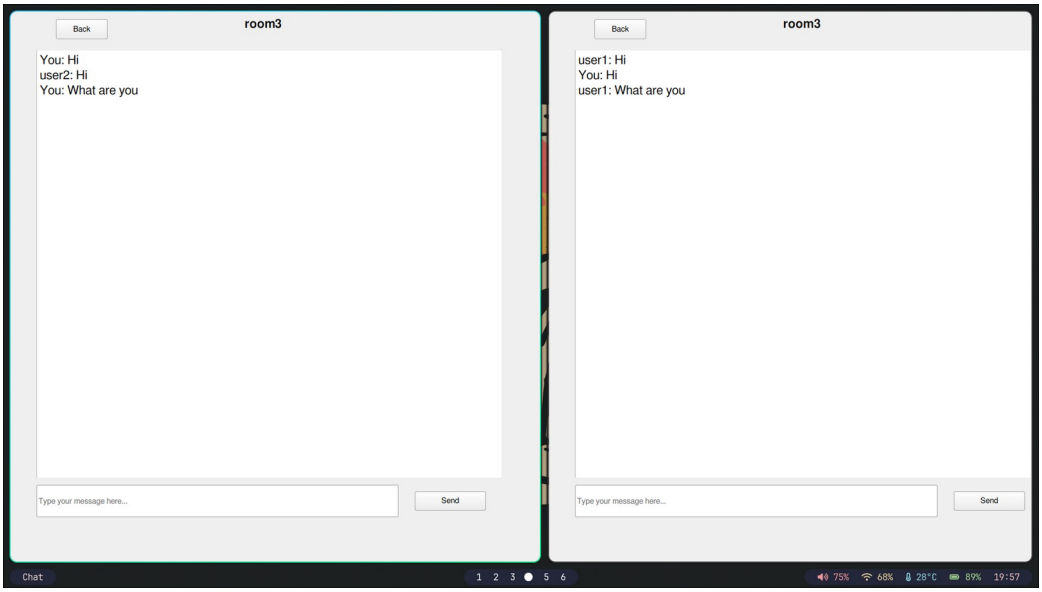

Рис. 4: Общение в комнате

Вывод: в ходе лабораторной работы был изучен и реализован на примере комнат механизм сокетов.