Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе № 6

на тему

**ЭЛЕМЕНТЫ СЕТЕВОГО ПРОГРАММИРОВАНИЯ**

Выполнил:

студент гр. 153503

Вергасов В.М.

Проверил:

Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Цели работы 3](#_Toc146883343)

[2 Краткие теоретические сведения 4](#_Toc146883344)

[3 Полученные результаты 5](#_Toc146883345)

[Выводы 8](#_Toc146883346)

[Список литературы 9](#_Toc146883347)

# 1 ЦЕЛИ РАБОТЫ

Практическое освоение основ построения и функционирования сетей, стеков протоколов, программных интерфейсов. Изучение сетевой подсистемы и программного интерфейса сокетов в Unix-системах. Практическое проектирование, реализация и отладка программ, взаимодействующих через сеть TCP/IP.

Создание чата, используя протокол UDP.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Практическое проектирование, реализация и отладка программ, взаимодействующих через сеть TCP/IP, включает в себя несколько ключевых аспектов. Вот теоретический обзор этих аспектов.

TCP/IP (Transmission Control Protocol/Internet Protocol) является набором протоколов, который обеспечивает связь и передачу данных в компьютерных сетях. Он состоит из нескольких протоколов на разных уровнях, включая IP (Internet Protocol), TCP (Transmission Control Protocol), UDP (User Datagram Protocol) и другие. Протокол TCP обеспечивает надежную и упорядоченную доставку данных, в то время как протокол UDP предоставляет ненадежную и неупорядоченную доставку [1].

Взаимодействие программ через сеть в TCP/IP осуществляется с помощью сокетов. Сокет представляет собой программный интерфейс, который позволяет программам создавать сетевые соединения, отправлять и принимать данные. Сокеты бывают двух типов: сокеты клиента и сокеты сервера. Клиентский сокет устанавливает соединение с сервером, а серверный сокет слушает и принимает входящие соединения от клиентов.

При проектировании программы, взаимодействующей через сеть TCP/IP, необходимо определить роли клиента и сервера, а также функциональность, которую они должны выполнять. Это включает определение протоколов обмена данными, форматов сообщений и логики обработки запросов и ответов [2].

Реализация программы включает написание кода на выбранном языке программирования, который использует сокеты для создания соединений, отправки и приема данных. Для TCP-соединений обычно используются функции, такие как socket, bind, listen, accept, connect, send и recv, для установления и управления соединениями.

# 3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ

Результат работы программы-клиента после запуска, показан на рисунке 3.1.

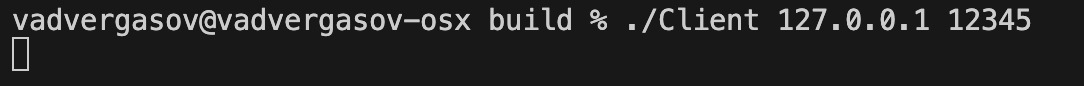


Рисунок 3.1 – Результат работы программы-клиента после запуска

Результат работы программы-сервера после запуска, показан на рисунке 3.2.

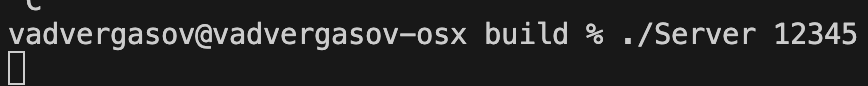


Рисунок 3.2 – Результат работы программы-сервера после запуска

Далее на клиенте можно подключиться с именем пользователя. Пример на рисунке 3.3.

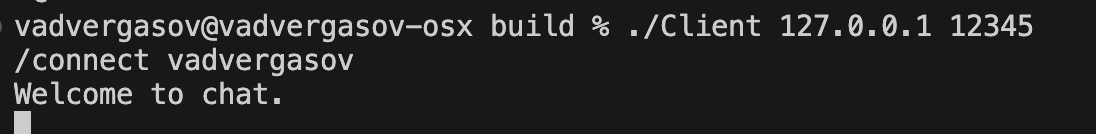


Рисунок 3.3 – Результат работы программы после подключения к серверу

После можно присоединиться к комнате и писать туда сообщения, как на рисунке 3.4.

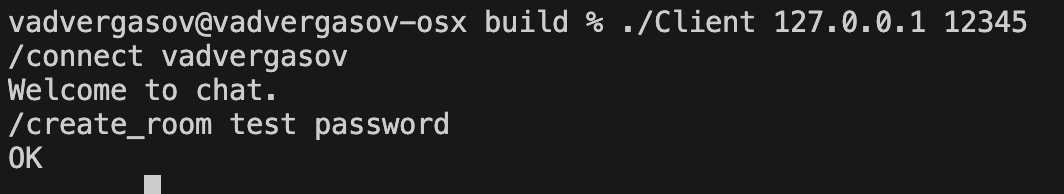


Рисунок 3.4 – Пример общения в приложении

# ВЫВОДЫ

Изучены основы построения и функционирования сетей, стеков протоколов, программных интерфейсов. Изучена сетевая подсистемы и программные интерфейсы сокетов в Unix-системах. Произведено практическое проектирование, реализация и отладка программ, взаимодействующих через сеть TCP/IP. Создан чат, симулирующий интерактивное взаимодействие пользователей, используя протокол .

# СПИСОК ЛИТЕРАТУРЫ

1. Programming reference for the Win32 API [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/ – Дата доступа: 21.09.2023.

# Приложение А

Листинг 1 **–** Код User.h:

#pragma once

#include "ChatMessage.h"

class User {

public:

User() = default;

User(std::string& username) : username\_(username) {}

User(const std::string username) : username\_(username) {}

virtual ~User() {}

virtual void deliver(const ChatMessage& message) = 0;

std::string username() const { return username\_; }

private:

const std::string username\_;

};

typedef boost::shared\_ptr<User> user\_ptr;

Листинг 2 **–** Код Session.h:

#pragma once

#include <boost/enable\_shared\_from\_this.hpp>

#include <boost/asio/io\_service.hpp>

#include <boost/asio/ip/udp.hpp>

#include "Room.h"

class Session

: public User,

public boost::enable\_shared\_from\_this<Session>

{

public:

Session(

boost::asio::ip::udp::socket& socket,

boost::asio::ip::udp::endpoint& remote\_endpoint,

std::string username

) : User(std::move(username)), socket\_(socket), remote\_endpoint\_(remote\_endpoint) {}

void deliver(const ChatMessage& message) override {

std::cout << message.text() << " " << username() << std::endl;

message\_queue\_.push\_back(message);

while(!message\_queue\_.empty()) {

send\_message(message\_queue\_.front());

message\_queue\_.pop\_front();

}

}

boost::shared\_ptr<Room> room() const {

return room\_;

}

void set\_room(const boost::shared\_ptr<Room>& room) {

room\_ = room;

}

private:

void send\_message(ChatMessage& message) {

socket\_.async\_send\_to(

boost::asio::buffer(message.data()),

remote\_endpoint\_,

boost::bind(

&Session::handle\_send,

this,

message,

boost::asio::placeholders::error,

boost::asio::placeholders::bytes\_transferred

)

);

}

void handle\_send(

ChatMessage message,

const boost::system::error\_code &error,

std::size\_t bytes\_transferred

) {

// LOG

}

MessageQueue message\_queue\_;

boost::shared\_ptr<Room> room\_;

boost::asio::ip::udp::socket& socket\_;

boost::asio::ip::udp::endpoint remote\_endpoint\_;

};

Листинг 3 **–** Код Server.cpp:

#include <boost/asio.hpp>

#include <boost/bind.hpp>

#include <boost/smart\_ptr/make\_shared.hpp>

#include <iostream>

#include <map>

#include <set>

#include <unordered\_map>

#include "Session.h"

using boost::asio::ip::udp;

class Server {

public:

Server(boost::asio::io\_service &io\_service, unsigned short port)

: io\_service\_(io\_service),

socket\_(io\_service, udp::endpoint(udp::v4(), port)) {

startReceive();

}

private:

void startReceive() {

socket\_.async\_receive\_from(

boost::asio::buffer(chat\_message\_.data()), remote\_endpoint\_,

boost::bind(&Server::handleReceive, this,

boost::asio::placeholders::error,

boost::asio::placeholders::bytes\_transferred));

}

void handleReceive(const boost::system::error\_code &error,

std::size\_t bytes\_transferred) {

if (!error || error == boost::asio::error::message\_size) {

chat\_message\_.decode();

std::cout << to\_string(chat\_message\_.type()) << std::endl;

if (!users\_.count(remote\_endpoint\_) &&

chat\_message\_.type() != MessageType::CONNECT) {

handleNonJoinedUser();

} else {

switch (chat\_message\_.type()) {

case MessageType::CONNECT:

handleConnect();

break;

case MessageType::COMMAND:

handleCommand();

break;

case MessageType::MESSAGE:

handleMessage();

break;

default:

break;

}

}

}

startReceive();

}

void handleNonJoinedUser() {

Session session(socket\_, remote\_endpoint\_, "SERVER");

session.deliver(ChatMessage("Need connect to some room."));

}

void handleConnect() {

std::vector<std::string> splitted\_message =

chat\_message\_.splitted\_text();

if (splitted\_message.size() == 2) {

std::string username = splitted\_message[1];

if (users\_.count(remote\_endpoint\_)) {

Session session(socket\_, remote\_endpoint\_, "SERVER");

session.deliver(ChatMessage("Login is already exists."));

} else {

users\_[remote\_endpoint\_] = boost::make\_shared<Session>(

socket\_, remote\_endpoint\_, username);

users\_[remote\_endpoint\_]->deliver(

ChatMessage("Welcome to chat."));

}

} else {

Session session(socket\_, remote\_endpoint\_, "SERVER");

session.deliver(ChatMessage("Invalid command form."));

}

}

void handleCommand() {

const auto session = users\_[remote\_endpoint\_];

const std::string username = session->username();

const std::string command = chat\_message\_.text();

if (command == "/users\_list") {

session->deliver(ChatMessage(users\_list(username)));

} else if (command == "/rooms\_list") {

session->deliver(ChatMessage(rooms\_list()));

} else {

std::pair<bool, std::string> result;

if (boost::algorithm::istarts\_with(command, "/create\_room")) {

result = create\_room(username);

} else if (boost::algorithm::istarts\_with(command, "/join\_room")) {

result = join\_room(username);

} else if (boost::algorithm::istarts\_with(command, "/leave\_room")) {

result = leave\_room(username);

}

session->deliver(ChatMessage(result.second));

}

}

void handleMessage() {

const auto &session = users\_[remote\_endpoint\_];

const auto &room = session->room();

const auto &username = session->username();

if (room) {

room->deliver(ChatMessage("[" + session->username() +

"]: " + chat\_message\_.text()),

username);

}

}

std::string users\_list(const std::string &username) {

std::string users =

"Users number: " + std::to\_string(users\_.size()) + "\n";

std::for\_each(begin(users\_), end(users\_),

[&](const std::pair<udp::endpoint,

boost::shared\_ptr<Session>> &pair) {

const auto &[\_, session] = pair;

const auto &username\_ = session->username();

if (username\_ != username) {

users += username\_ + '\n';

}

});

return users;

}

std::string rooms\_list() {

std::string rooms;

std::for\_each(

begin(rooms\_), end(rooms\_),

[&](const std::pair<std::string, boost::shared\_ptr<Room>> &pair) {

const auto &[room\_name, room] = pair;

if (room->is\_public()) {

rooms += room\_name + '\n';

}

});

return rooms;

}

std::pair<bool, std::string> create\_room(const std::string &username) {

std::vector<std::string> splitted\_message =

chat\_message\_.splitted\_text();

if (splitted\_message.size() != 3) {

return {0, "Bad command"};

}

const std::string room\_name = splitted\_message[1];

const std::string room\_password = splitted\_message[2];

if (rooms\_.count(room\_name)) {

return {0, "Room with name = [" + room\_name + "] already exists."};

}

rooms\_[room\_name] = boost::make\_shared<Room>(room\_name, room\_password);

rooms\_[room\_name]->join(users\_[remote\_endpoint\_]);

users\_[remote\_endpoint\_]->set\_room(rooms\_[room\_name]);

return {1, "OK"};

}

std::pair<bool, std::string> join\_room(const std::string &username) {

std::vector<std::string> splitted\_message =

chat\_message\_.splitted\_text();

if (splitted\_message.size() != 3) {

return {0, "Bad command"};

}

const std::string room\_name = splitted\_message[1];

const std::string room\_password = splitted\_message[2];

const auto room\_iterator = rooms\_.find(room\_name);

if (room\_iterator == rooms\_.end()) {

return {0, "Wrong room name"};

}

const auto room = room\_iterator->second;

if (!room->is\_password\_valid(room\_password)) {

return {0, "Wrong password"};

}

room->join(users\_[remote\_endpoint\_]);

users\_[remote\_endpoint\_]->set\_room(room);

return {1, "OK"};

}

std::pair<bool, std::string> leave\_room(const std::string &username) {

return {};

}

ChatMessage chat\_message\_;

std::map<udp::endpoint, boost::shared\_ptr<Session>> users\_;

std::unordered\_map<std::string, boost::shared\_ptr<Room>> rooms\_;

udp::socket socket\_;

udp::endpoint remote\_endpoint\_;

boost::asio::io\_service &io\_service\_;

};

int main(int argc, char \*argv[]) {

if (argc != 2) {

std::cout << "Specify port to use!" << std::endl;

return 1;

}

try {

boost::asio::io\_service io\_service;

Server server(io\_service, std::atoi(argv[1]));

io\_service.run();

} catch (const std::exception &e) {

std::cerr << e.what() << std::endl;

}

}

Листинг 4 **–** Код Room.h:

#pragma once

#include "User.h"

class Room {

public:

Room(std::string name, std::string password)

: name\_(name), password\_(password) {}

void join(const user\_ptr& user) { users\_.insert(user); }

void leave(const user\_ptr& user) { users\_.erase(user); }

void deliver(const ChatMessage& message, const std::string& username) {

message\_queue\_.push\_back(message);

for (const auto& f : users\_) {

std::cout << f->username() << std::endl;

}

std::for\_each(users\_.begin(), users\_.end(), [&](const user\_ptr& user) {

std::cout << username << " " << user->username() << std::endl;

if (username != user->username()) {

user->deliver(message);

}

});

}

std::string name() const { return name\_; }

bool is\_public() const { return true; }

bool is\_password\_valid(const std::string& password) const {

return password == password\_;

}

private:

std::string name\_;

std::string password\_;

std::set<user\_ptr> users\_;

MessageQueue message\_queue\_;

};

Листинг 5 **–** Код Client.cpp:

#include <boost/thread.hpp>

#include <boost/array.hpp>

#include <boost/asio.hpp>

#include <iostream>

#include "ChatMessage.h"

using boost::asio::ip::udp;

class Client {

public:

Client(

boost::asio::io\_service& io\_service,

const std::string& host,

const std::string& port

) : io\_service\_(io\_service), socket\_(io\_service) {

udp::resolver resolver(io\_service\_);

udp::resolver::query query(udp::v4(), host, port);

udp::resolver::iterator iterator = resolver.resolve(query);

endpoint\_ = \*iterator;

boost::asio::connect(socket\_, iterator);

startReceive();

}

void send(const ChatMessage& msg) {

socket\_.send(boost::asio::buffer(msg.data(), msg.size()));

}

~Client() {

socket\_.close();

}

private:

void startReceive() {

socket\_.async\_receive\_from(

boost::asio::buffer(message\_.data()),

endpoint\_,

boost::bind(

&Client::handleReceive,

this,

boost::asio::placeholders::error,

boost::asio::placeholders::bytes\_transferred

)

);

}

void handleReceive(

const boost::system::error\_code& error,

std::size\_t bytes\_transferred

) {

if(!error) {

message\_.decode();

std::cout << message\_.text() << std::endl;

}

startReceive();

}

udp::socket socket\_;

udp::endpoint endpoint\_;

boost::asio::io\_service& io\_service\_;

ChatMessage message\_;

};

int main(int argc, char\* argv[]) {

try {

if(argc != 3) {

std::cerr << "Use: <host> <port>" << std::endl;

}

boost::asio::io\_service io\_service;

Client client(io\_service, argv[1], argv[2]);

boost::thread t(boost::bind(&boost::asio::io\_service::run, &io\_service));

std::string input;

while(std::getline(std::cin, input)) {

ChatMessage msg(input);

client.send(msg);

}

client.~Client();

t.join();

} catch (const std::exception& e) {

std::cout << e.what() << std::endl;

}

}

Листинг 6 **–** Код ChatMessage.h:

#include <boost/algorithm/string.hpp>

#include <boost/array.hpp>

#include <algorithm>

#include <string>

#include <vector>

#include <deque>

enum class MessageType {

CONNECT = 0,

COMMAND = 1,

MESSAGE = 2,

UNKNOWN = 3

};

std::string to\_string(MessageType type) {

switch (type) {

case MessageType::CONNECT: return "CONNECT";

case MessageType::COMMAND: return "COMMAND";

case MessageType::MESSAGE: return "MESSAGE";

default: return "UNKNOWN";

}

}

class ChatMessage {

public:

static constexpr std::size\_t MESSAGE\_LENGTH = 1024;

ChatMessage() : data\_(), type\_(MessageType::UNKNOWN) {}

ChatMessage(const std::string& text) : data\_() {

initMessage(text);

}

boost::array<char, MESSAGE\_LENGTH>& data() {

return data\_;

}

boost::array<char, MESSAGE\_LENGTH> data() const {

return data\_;

}

MessageType type() {

return type\_;

}

std::string text() const {

if(type\_ == MessageType::UNKNOWN) {

return "UNKNOWN MESSAGE";

}

const auto position = std::find(data\_.begin() + 1, data\_.end(), 0);

return std::string(data\_.begin() + 1, position);

}

std::vector<std::string> splitted\_text(const std::string& delims = " ") const {

std::vector<std::string> words;

std::string text\_ = text();

boost::split(words, text\_, boost::is\_any\_of(delims));

return words;

}

std::size\_t size() const {

return data\_.size();

}

void decode() {

switch (data\_[0]) {

case 0:

type\_ = MessageType::CONNECT;

break;

case 1:

type\_ = MessageType::COMMAND;

break;

case 2:

type\_ = MessageType::MESSAGE;

break;

default:

type\_ = MessageType::UNKNOWN;

break;

}

}

private:

void initMessage(const std::string& text) {

if (text.empty()) {

type\_ = MessageType::UNKNOWN;

} else if(boost::algorithm::starts\_with(text, "/connect")) {

type\_ = MessageType::CONNECT;

} else if(boost::algorithm::starts\_with(text, "/")) {

type\_ = MessageType::COMMAND;

} else {

type\_ = MessageType::MESSAGE;

}

data\_[0] = static\_cast<char>(type\_);

memcpy(data\_.data() + 1, text.data(), sizeof(char) \* std::min(MESSAGE\_LENGTH - 1, text.size()));

}

MessageType type\_;

boost::array<char, MESSAGE\_LENGTH> data\_;

};

typedef std::deque<ChatMessage> MessageQueue;