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

ОТЧЁТ

к лабораторной работе №6

на тему

**ЭЛЕМЕНТЫ СЕТЕВОГО ПРОГРАММИРОВАНИЯ**

Выполнил: студент гр.253505 Снежко М.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc179742261)

[2 Краткие теоритические сведения 4](#_Toc179742262)

[3 Описание функций программы 5](#_Toc179742263)

[Заключение 6](#_Toc179742264)

[Список использованных источников 7](#_Toc179742265)

[Приложение А (обязательное) 8](#_Toc179742266)

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью данной лабораторной работы является изучение элементов сетевого программирования в операционных системах *Unix*, включая освоение основных механизмов работы с сокетами и организации сетевого взаимодействия между процессами. Основное внимание уделяется практическому проектированию, реализации и отладке программ, взаимодействующих через сеть *TCP/IP*. Работа направлена на создание сетевого чат-приложения, которое обеспечивает интерактивное общение между пользователями с использованием клиент-серверной архитектуры.

Программа должна поддерживать подключение нескольких клиентов к серверу и обеспечивать обмен текстовыми сообщениями между ними. Серверная часть отвечает за прием соединений, хранение сообщений и их рассылку подключенным клиентам. Клиентская часть предоставляет интерфейс для ввода сообщений пользователем и отображения получаемых сообщений от других участников чата. Реализация предполагает использование программного интерфейса сокетов и базовых сетевых функций, таких как *socket()*, *bind()*, *listen()*, *accept()*, *connect()*, *send()* и *recv()*.

Особое внимание уделяется созданию программы, которая корректно обрабатывает сетевые соединения в многопользовательской среде, обеспечивает синхронизацию доступа к общим ресурсам и минимизирует накладные расходы на сетевые операции. Для работы с несколькими клиентами одновременно сервер использует механизм многопоточности. Программа должна быть протестирована с различным количеством подключенных клиентов для проверки стабильности работы и корректности передачи сообщений. В ходе лабораторной работы также необходимо проанализировать особенности работы с сокетами при использовании протоколов *TCP* и *UDP*, выявить их преимущества и недостатки в контексте разрабатываемого приложения.

Таким образом, данная лабораторная работа позволяет изучить ключевые аспекты сетевого программирования в *Unix*, включая создание сетевых соединений, организацию клиент-серверного взаимодействия и обработку данных в распределенной среде. Полученные навыки помогут в дальнейшем разрабатывать эффективные и надежные сетевые приложения для решения задач, требующих обмена данными между удаленными системами. Практическое освоение работы с сокетами и протоколами *TCP/IP* является важной основой для понимания принципов построения современных распределенных систем и сетевых сервисов.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

*UNIX* – многопользовательская, многозадачная операционная система (ОС), разработанная для гибкости и адаптивности. Первоначально разработанная в 1970-х годах, *Unix* была одной из первых ОС, написанных на языке программирования *C*. С момента своего появления операционная система *Unix* и ее ответвления оказали глубокое влияние на компьютерную и электронную промышленность, предлагая переносимость, стабильность и совместимость в различных гетерогенных средах и типах устройств. [1]

Сетевая модель OSI имеет семь уровней, иерархически расположенных от большего к меньшему. Cамым верхним является седьмой (прикладной), а самым нижним – первый (физический). Модель *OSI* разрабатывалась еще в 1970-х годах, чтобы описать архитектуру и принципы работы сетей передачи данных. В процессе передачи данных всегда участвуют устройство-отправитель, устройство-получатель, а также сами данные, которые должны быть переданы и получены. С точки зрения рядового пользователя задача элементарна – нужно взять и отправить эти данные. Все, что происходит при отправке и приеме данных, детально описывает семиуровневая модель *OSI*. На седьмом уровне информация представляется в виде данных, на первом – в виде бит. Процесс, когда информация отправляется и переходит из данных в биты, называется инкапсуляцией. Обратный процесс, когда информация, полученная в битах на первом уровне, переходит в данные на седьмом, называется декапсуляцией. На каждом из семи уровней информация представляется в блоке данных протокола – *PDU* (*Protocol Data Unit*). [2]

Модель *TCP/IP* – это стек протоколов, которые задают правила передачи данных по Сети. Так как бал здесь правят протоколы *TCP* и *IP*, в честь них и назвали всю модель. *TCP* (*Transmission Control Protocol*) отвечает за обмен данными. Он управляет их отправкой и следит за тем, чтобы они дошли до получателя в целости. У *TCP* есть свои гарантии, что всё пройдёт успешно, – о них чуть позже. *IP* (*Internet Protocol*) отвечает за адресацию. Его задача – связывать друг с другом устройства и нарезать данные на пакеты для удобной отправки. Чтобы протокол мог быстро найти дорогу от одного компьютера к другому, придумали *IP*-адреса – уникальные идентификаторы, которые есть у каждого устройства в Сети. [3]

Таким образом, программирование на языке *C* в *Unix* предоставляет мощные средства для создания приложений, которые могут взаимодействовать с операционной системой, обрабатывать текстовые данные и управлять ресурсами. Полученные знания и навыки могут быть использованы для разработки более сложных программ, таких как утилиты для обработки текста, системные утилиты или инструменты автоматизации.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Разработанная программа представляет собой многопоточный чат-сервер с клиентами, обеспечивающий обмен сообщениями между пользователями через сетевые сокеты. Основная задача программы заключается в организации централизованного чата, где сервер принимает подключения клиентов, обрабатывает их сообщения и рассылает другим участникам чата.

Программа начинается с создания сокета и привязки его к указанному порту на серверной стороне. Сервер переходит в режим ожидания подключений, а при появлении нового клиента создает для него отдельный поток обработки. Клиентская часть устанавливает соединение с сервером, регистрирует пользователя и предоставляет интерфейс для отправки и получения сообщений. Особенностью реализации является использование JSON-формата для структурирования передаваемых сообщений. Каждое сообщение содержит поля: тип (регистрация, обычное сообщение или системное), отправитель, получатель и тело сообщения. Это позволяет гибко обрабатывать разные виды сообщений на сервере и корректно отображать их у клиентов. Результат работы программы представлен на рисунке 3.1.

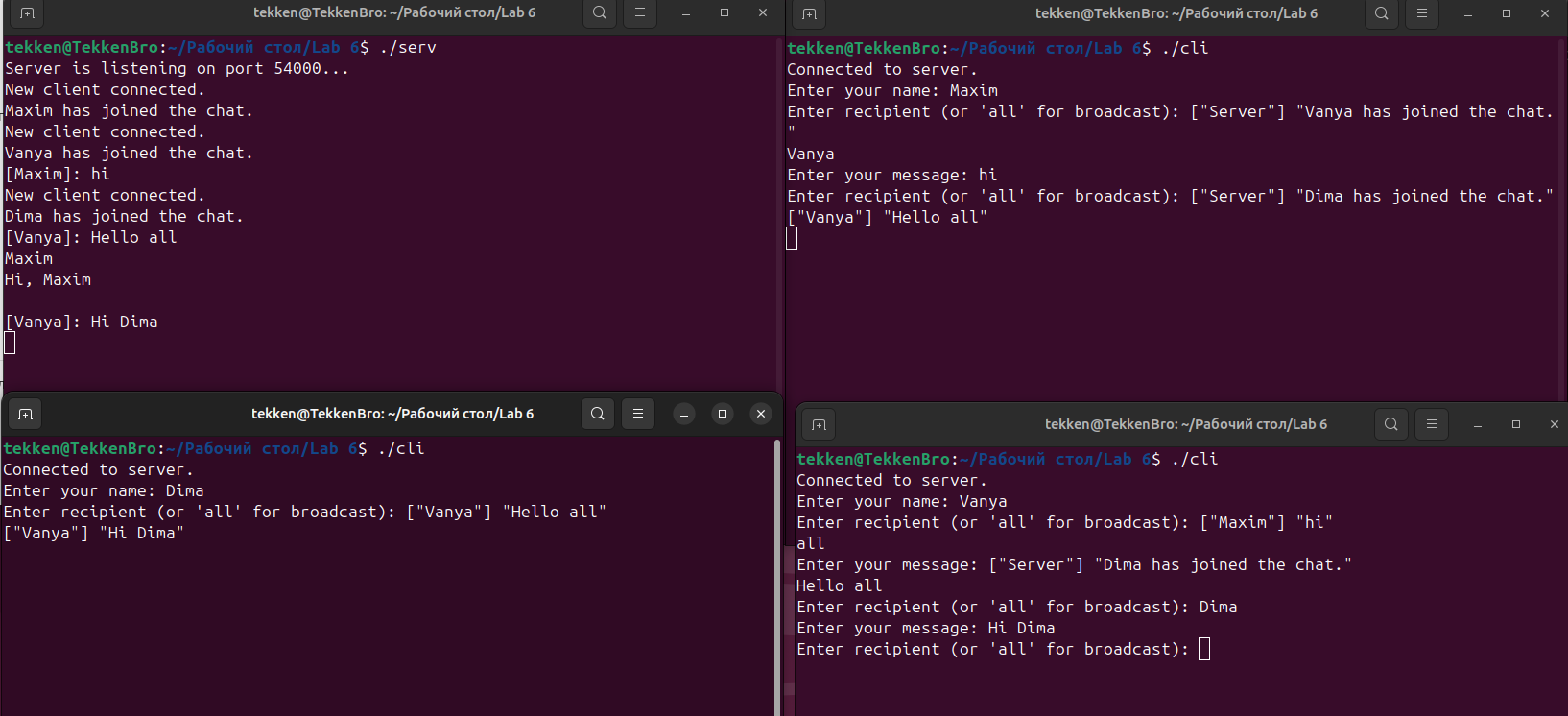


Рисунок 3.1 – Результат работы программы

Таким образом, программа иллюстрирует использование сетевых сокетов и многопоточности в *C++* для реализации клиент-серверного приложения. Она позволяет изучить работу с сокетами, организацию многопоточной обработки соединений, синхронизацию доступа к общим ресурсам, а также применение JSON для структурирования сетевых сообщений. Разработанный чат может служить основой для более сложных систем обмена сообщениями.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы была разработана клиент-серверная система чата на языке *C++*, реализующая сетевое взаимодействие между пользователями с использованием потоков и сокетов. Основной целью работы являлось изучение механизмов сетевого программирования в *Unix/Linux* системах, включая работу с сокетами, организацию многопоточной обработки соединений и синхронизацию доступа к общим ресурсам.

Разработанная система состоит из серверной части, обрабатывающей подключения клиентов, и клиентских приложений, обеспечивающих взаимодействие пользователей. В процессе работы были реализованы ключевые функции сетевого обмена: установление *TCP*-соединений, регистрация пользователей, прием и передача сообщений между клиентами, а также обработка отключений. Особенностью реализации стало использование *JSON*-формата для структурирования передаваемых сообщений, что обеспечило гибкость и расширяемость системы.

Серверная часть программы создает основной сокет для прослушивания входящих соединений и для каждого нового клиента запускает отдельный поток обработки. Это позволяет одновременно обслуживать множество пользователей. Для хранения информации о подключенных клиентах используется общая структура данных с синхронизацией доступа через мьютексы, что предотвращает возникновение состояний гонки при работе с разделяемыми ресурсами.

Клиентская часть реализует интерфейс для ввода сообщений и их отображения, работая в двух потоках: основной поток отвечает за ввод данных пользователем, а дополнительный – за прием сообщений от сервера. Такая архитектура обеспечивает плавную работу интерфейса без блокировок при ожидании входящих сообщений.

Реализация данной лабораторной работы позволила изучить не только основы сетевого программирования, но и важные аспекты разработки многопоточных приложений. Были рассмотрены вопросы эффективной синхронизации потоков, управления сетевыми соединениями и обработки ошибок в распределенной системе. Эксперименты показали, что выбранная архитектура обеспечивает стабильную работу чата при разумной нагрузке, однако при большом количестве подключений могут потребоваться дополнительные оптимизации.

Таким образом, выполнение данной работы позволило освоить ключевые технологии разработки сетевых приложений в Unix-среде, включая работу с сокетами, многопоточное программирование и синхронизацию. Полученные знания и навыки могут быть применены для создания более сложных распределенных систем и сервисов, требующих надежного сетевого взаимодействия между компонентами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Thecode "Обзор UNIX" – Электронный ресурс. Режим доступа: https://thecode.media/unix.

[2] Selectel "Модели OSI" – Электронный ресурс. Режим доступа: https://selectel.ru/blog/osi-for-beginners/.

[3] Skillbox "Что такое TCP/IP" – Электронный ресурс. Режим доступа: https://skillbox.ru/media/code/model-tcpip-chto-eto-takoe-i-kak-ona-rabotaet/.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <iostream>

#include <thread>

#include <string>

#include <nlohmann/json.hpp> // Подключаем библиотеку nlohmann/json

using json = nlohmann::json;

void receive\_messages(int socket) {

char buffer[1024];

int bytes\_received;

while (true) {

bytes\_received = recv(socket, buffer, sizeof(buffer), 0);

if (bytes\_received <= 0) {

std::cout << "Server disconnected." << std::endl;

break;

}

buffer[bytes\_received] = '\0';

json message = json::parse(buffer);

std::cout << "[" << message["from"] << "] " << message["body"] << std::endl;

}

}

int main() {

int client\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (client\_socket == -1) {

std::cerr << "Failed to create socket." << std::endl;

return 1;

}

sockaddr\_in server\_addr{};

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(54000);

inet\_pton(AF\_INET, "127.0.0.1", &server\_addr.sin\_addr);

if (connect(client\_socket, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) == -1) {

std::cerr << "Connection failed." << std::endl;

close(client\_socket);

return 1;

}

std::cout << "Connected to server." << std::endl;

std::string name;

std::cout << "Enter your name: ";

std::getline(std::cin, name);

json register\_message = {

{"type", "register"},

{"from", name},

{"to", "server"},

{"body", ""}

};

std::string register\_message\_str = register\_message.dump();

send(client\_socket, register\_message\_str.c\_str(), register\_message\_str.length(), 0);

std::thread receive\_thread(receive\_messages, client\_socket);

receive\_thread.detach();

while (true) {

std::string message\_body;

std::string to\_client;

std::cout << "Enter recipient (or 'all' for broadcast): ";

std::getline(std::cin, to\_client);

std::cout << "Enter your message: ";

std::getline(std::cin, message\_body);

if (message\_body.empty()) {

continue;

}

json message = {

{"type", "message"},

{"from", name},

{"to", to\_client},

{"body", message\_body}

};

std::string message\_str = message.dump();

send(client\_socket, message\_str.c\_str(), message\_str.length(), 0);

}

close(client\_socket);

return 0;

}

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <iostream>

#include <thread>

#include <vector>

#include <string>

#include <mutex>

#include <algorithm>

#include <nlohmann/json.hpp> // Подключаем библиотеку nlohmann/json

using json = nlohmann::json;

struct ClientInfo {

int socket;

std::string name;

};

std::vector<ClientInfo> clients;

std::mutex clients\_mutex;

ClientInfo\* find\_client\_by\_name(const std::string& name) {

std::lock\_guard<std::mutex> guard(clients\_mutex);

for (auto& client : clients) {

if (client.name == name) {

return &client;

}

}

return nullptr;

}

void handle\_client(int client\_socket) {

char buffer[1024];

int bytes\_received;

std::string client\_name;

while (true) {

bytes\_received = recv(client\_socket, buffer, sizeof(buffer), 0);

if (bytes\_received <= 0) {

std::cout << "Client disconnected." << std::endl;

break;

}

buffer[bytes\_received] = '\0';

json message = json::parse(buffer);

if (message["type"] == "register") {

client\_name = message["from"];

{

std::lock\_guard<std::mutex> guard(clients\_mutex);

clients.push\_back({client\_socket, client\_name});

}

std::cout << client\_name << " has joined the chat." << std::endl;

json join\_message = {

{"type", "system"},

{"from", "Server"},

{"to", "all"},

{"body", client\_name + " has joined the chat."}

};

std::string join\_message\_str = join\_message.dump();

{

std::lock\_guard<std::mutex> guard(clients\_mutex);

for (const auto& client : clients) {

if (client.socket != client\_socket) {

send(client.socket, join\_message\_str.c\_str(), join\_message\_str.length(), 0);

}

}

}

} else if (message["type"] == "message") {

std::string to = message["to"];

std::string from = message["from"];

std::string body = message["body"];

std::string full\_message = "[" + from + "]: " + body;

std::cout << full\_message << std::endl;

if (to == "all") {

std::lock\_guard<std::mutex> guard(clients\_mutex);

for (const auto& client : clients) {

if (client.socket != client\_socket) {

send(client.socket, buffer, bytes\_received, 0);

}

}

} else {

ClientInfo\* target\_client = find\_client\_by\_name(to);

if (target\_client) {

send(target\_client->socket, buffer, bytes\_received, 0);

} else {

std::cerr << "Client '" << to << "' not found." << std::endl;

}

}

}

}

close(client\_socket);

{

std::lock\_guard<std::mutex> guard(clients\_mutex);

clients.erase(std::remove\_if(clients.begin(), clients.end(),

[client\_socket](const ClientInfo& client) {

return client.socket == client\_socket;

}),

clients.end());

}

json leave\_message = {

{"type", "system"},

{"from", "Server"},

{"to", "all"},

{"body", client\_name + " has left the chat."}

};

std::string leave\_message\_str = leave\_message.dump();

{

std::lock\_guard<std::mutex> guard(clients\_mutex);

for (const auto& client : clients) {

send(client.socket, leave\_message\_str.c\_str(), leave\_message\_str.length(), 0);

}

}

}

int main() {

int server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket == -1) {

std::cerr << "Failed to create socket." << std::endl;

return 1;

}

sockaddr\_in server\_addr{};

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(54000);

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind(server\_socket, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) == -1) {

std::cerr << "Bind failed." << std::endl;

close(server\_socket);

return 1;

}

if (listen(server\_socket, SOMAXCONN) == -1) {

std::cerr << "Listen failed." << std::endl;

close(server\_socket);

return 1;

}

std::cout << "Server is listening on port 54000..." << std::endl;

while (true) {

int client\_socket = accept(server\_socket, nullptr, nullptr);

if (client\_socket == -1) {

std::cerr << "Accept failed." << std::endl;

continue;

}

std::cout << "New client connected." << std::endl;

std::thread client\_thread(handle\_client, client\_socket);

client\_thread.detach();

}

close(server\_socket);

return 0;

}