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.253504 Лавренова А.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

Введение3

[1 [Ц](#_agswffz6g6ei)ель работы](#_Toc178067889) 4

[2 Теоретические сведения 5](#_Toc178067890)

[3 Описание и пример выполнения программы 7](#_Toc178067891)

Заключение [9](#_Toc178067899)

[Список использованных источников 10](#_Toc178067900)

[Приложение А (обязательное) 11](#_Toc178067901)

# ВВЕДЕНИЕ

В данной лабораторной работе основное внимание уделено практическому освоению сетевого программирования на уровне сокетов в операционных системах семейства Unix. Она направлена на формирование целостного понимания того, как устроены взаимодействия между различными процессами через сетевые каналы связи, а также на закрепление теоретических основ, касающихся архитектуры и функционирования протокольных стеков TCP/IP и OSI.

Современные компьютерные сети являются неотъемлемой частью любой программной инфраструктуры. Их основой служат модели взаимодействия открытых систем, определяющие, как данные проходят через уровни протоколов и как эти уровни взаимодействуют между собой. Протоколы IP, TCP и UDP обеспечивают адресацию, доставку и надежность передачи данных, а программный интерфейс сокетов даёт разработчику универсальные инструменты для реализации прикладных сетевых решений. Эта лабораторная работа предлагает возможность применить эти знания на практике, реализуя программный сервер, принимающий подключения от клиентов, обрабатывающий поступающие сообщения и возвращающий результат обратно клиенту.

Проект, выполненный в рамках работы, представляет собой сетевую службу, в которой сервер принимает текстовые сообщения от клиентов, преобразует их в азбуку Морзе и отправляет обратно. Таким образом, продемонстрирован пример простого, но функционального клиент-серверного взаимодействия, основанного на использовании протокола TCP. Разработка включает в себя реализацию многопоточного сервера, что позволяет обрабатывать несколько клиентов параллельно, а также реализацию клиентской части, которая позволяет пользователю взаимодействовать с сервером в интерактивном режиме. Работа с сокетами, потоками, а также с преобразованием символов в код Морзе в данном случае иллюстрирует не только сетевую коммуникацию, но и общий подход к проектированию простых сетевых приложений.

В процессе выполнения лабораторной работы будут получены навыки, необходимые для создания и тестирования распределённых приложений, а также учится выстраивать структуру проекта с использованием инструментов автоматической сборки, таких как Makefile. Значимым элементом является также организация тестирования логики приложения, что обеспечивает не только корректность, но и поддержку кода в будущем. В результате проделанной работы формируется комплексное представление о принципах построения сетевых сервисов, методах их реализации и подходах к обеспечению корректного взаимодействия между компонентами распределённой системы.

1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является всестороннее освоение принципов и практик разработки сетевых приложений с использованием программного интерфейса сокетов в операционных системах семейства Unix. В рамках выполнения задания особое внимание уделяется углублённому изучению архитектуры протокольных стеков, таких как TCP/IP и OSI, а также роли транспортных протоколов TCP и UDP в обеспечении надежной и упорядоченной передачи данных между удалёнными системами.

Практическая реализация клиент-серверной модели взаимодействия позволяет не только закрепить теоретические знания, но и развить уверенные навыки проектирования многопоточных серверов, обработки подключений, организации двустороннего обмена сообщениями и реализации прикладного протокола.

Задачи, решаемые в процессе лабораторной работы, охватывают полный цикл разработки сетевого сервиса: от проектирования структуры приложения и настройки сокет-соединений до отладки, тестирования и автоматизации сборки с использованием Makefile. Благодаря реализации функционального взаимодействия между клиентом и сервером, основанного на преобразовании текстовых сообщений в азбуку Морзе, студент получает наглядное представление о прикладном уровне сетевого взаимодействия.

Работа способствует формированию фундаментального понимания внутренних механизмов сетевой коммуникации, таких как установка соединения, передача данных, синхронная и асинхронная обработка запросов. Кроме того, она развивает умения по модульной организации программного кода, применению инструментов командной строки и обеспечению воспроизводимости проекта через написание тестов, проверяющих корректность логики программы.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Linux — это свободная операционная система (точнее, семейство систем) с открытым исходным кодом. Само название Linux относится к ядру системы, которое является ее ключевым компонентом, управляющим центральным процессором (ЦП), памятью и периферийными устройствами компьютера. [1]

Сети представляют собой сложную область, однако модели OSI (Open Systems Interconnection) и TCP/IP помогают упростить понимание, организуя процессы коммуникации по уровням. В этом документе рассматриваются обе модели, подчеркиваются их различия и сходства, а также практическое применение.

Модель OSI делит сетевой процесс на семь уровней, каждый из которых отвечает за определенные функции, что способствует стандартизации коммуникации и упрощению устранения неполадок. Первый уровень, физический, обрабатывает физическое соединение между устройствами, включая такие компоненты, как кабели и беспроводные сигналы. Канальный уровень отвечает за передачу данных в одной сети, управляя кадрами и MAC-адресами. Сетевой уровень занимается маршрутизацией и IP-адресацией для связи между разными сетями. Транспортный уровень гарантирует безошибочную доставку сообщений, используя протоколы TCP и UDP. Сеансовый уровень управляет сеансами связи между приложениями, а презентационный уровень подготавливает данные для приложений, обрабатывая их шифрование и форматирование. Наконец, прикладной уровень предоставляет сетевые услуги для приложений, таких как просмотр веб-страниц и электронная почта. [2]

Модель TCP/IP, в отличие от модели OSI, состоит из четырех слоев, которые объединяют некоторые уровни OSI в более широкие категории. Прикладной слой включает в себя уровни приложения, представления и сеанса, обрабатывая высокоуровневые протоколы и обмен данными. Транспортный слой обеспечивает надежную или быструю связь, аналогично транспортному уровню OSI. Интернет-слой отвечает за IP-адресацию и маршрутизацию, соответствуя сетевому уровню OSI. Уровень доступа к сети объединяет канальный и физический уровни, занимаясь аппаратным обеспечением и низкоуровневыми протоколами.

Таким образом, модели OSI и TCP/IP служат важными инструментами для понимания сетевых процессов, их различия и сходства помогают в разработке и устранении неполадок в сетях.

Для передачи информации используются транспортные протоколы TCP и UDP. В физическом смысле эти протоколы представляют собой сетевые пакеты. Каждый сетевой пакет передаёт небольшой фрагмент информации, поэтому данные разбиваются на много пакетов.

Каждый сетевой пакет обоих протоколов TCP и UDP состоит из двух частей: заголовок и непосредственно данные.

В заголовке содержится «служебная информация» (можно сказать, что это метаданные) — порты пункта назначения и исходного узла, номер пакета в потоке, тип пакета и так далее.

Различие между протоколами TCP и UDP в том, что протокол TCP имеет механизм контроля полноты переданных данных — если какой-либо пакет был потерян или повреждён, то предусмотрен механизм для проверки этого факта и повторной отправки пакета. В протоколе UDP такого механизма нет — то есть если потерян пакет протокола UDP, то узел, который его отправил, никогда об этом не узнает, а принимающая сторона никогда не узнает, что ей был отправлен потерянный пакет UDP.

Сокеты — это фундаментальное понятие в сетевых технологиях Linux, позволяющее осуществлять связь между процессами на разных компьютерах через сеть. Проще говоря, сокет — это конечная точка двустороннего канала связи, который устанавливается между двумя процессами. [3]

Сокет состоит из комбинации IP-адреса и номера порта. IP-адрес идентифицирует удалённый компьютер, а номер порта указывает, к какой программе или службе на этом компьютере нужно подключиться.

В Linux используются четыре основных типа сокетов: потоковые, дейтаграммные, необработанные и секвенированные пакетные. — потоковые сокеты (SOCK\_STREAM) обеспечивают надёжную передачу потока байтов по протоколу TCP/IP и гарантируют, что пакеты будут приходить по порядку.

Сокеты дейтаграмм (SOCK\_DGRAM) обеспечивают ненадёжную передачу данных в виде пакетов по протоколу UDP.

Необработанные сокеты (SOCK\_RAW) обеспечивают доступ к базовым протоколам транспортного уровня, таким как ICMP и IGMP.

Сокеты с последовательной передачей пакетов (SOCK\_SEQPACKET) обеспечивают надёжную доставку пакетов с поддержкой последовательности, но используют SCTP (протокол управления потоковой передачей).

Сокеты — это фундаментальный компонент сетевого программирования, который обеспечивает связь между процессами в Linux. В Linux создание сокета инициируется вызовом функции socket() системного вызова.Эта функция принимает три аргумента: семейство адресов, тип сокета и протокол. Семейство адресов определяет, будет ли сокет использоваться для связи по сети с интернет-протоколом (IP) или по другому типу локального канала связи.

В данном разделе рассмотрены ключевые аспекты сетевых технологий, включая модели OSI и TCP/IP, а также механизмы передачи данных с использованием протоколов TCP и UDP. Обсуждение сокетов в Linux, их типов и функций подчеркивает важность понимания сетевых концепций для эффективного программирования. Изучение этих основ позволяет лучше ориентироваться в сетевых взаимодействиях и разрабатывать надежные приложения, способные эффективно передавать данные между процессами в различных сетевых средах.

**3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Разработанная в рамках лабораторной работы программа представляет собой сетевой сервис, реализованный по клиент-серверной архитектуре, в котором взаимодействие происходит посредством протокола TCP. Основной функциональностью сервиса является приём сообщений от клиента, их преобразование в азбуку Морзе на стороне сервера и возврат результата клиенту. Программа состоит из двух отдельных компонентов: сервера и клиента, каждый из которых реализован на языке C++ с использованием сокетов и POSIX API.

Архитектура решения основана на классической централизованной модели клиент-серверного взаимодействия. Сервер представляет собой фоновый процесс, прослушивающий заданный TCP-порт и создающий отдельный поток для обработки каждого нового подключения. Клиент, в свою очередь, инициирует соединение с сервером, отправляет ему строку текста и получает обратно результат — строку, закодированную в соответствии с азбукой Морзе.

Сервер представляет собой многопоточное приложение, которое запускается с указанием порта, на котором оно будет ожидать входящие соединения. При запуске сервер:

* создаёт сокет;
* связывает его с заданным портом;
* переводит его в режим прослушивания;
* и при поступлении запроса на соединение — принимает его.

Для каждого нового клиента создаётся отдельный поток, в котором обрабатывается обмен сообщениями. Сообщение, полученное от клиента, передаётся в функцию кодирования, где символы преобразуются в соответствующие знаки азбуки Морзе. Далее результат отправляется обратно клиенту через тот же сокет.

Функция кодирования реализована в отдельном модуле и использует таблицу соответствий символов латинского алфавита и цифр их представлению в азбуке Морзе.

Клиентское приложение подключается к серверу по IP-адресу и порту, указанным в аргументах командной строки. После установления соединения пользователь может вводить произвольные сообщения в консоль, которые будут отправляться на сервер. Сервер возвращает закодированное сообщение, и оно выводится на экран клиента.

Для наглядности функционирования программы приведён следующий пример.

Сначала необходимо запустить серверное приложение. Для этого из командной строки, находясь в корневой директории проекта, выполняется команда, изображённая на рисунке 3.1.



Рисунок 3.1 – Запуск сервера

После запуска сервер начинает прослушивать указанный порт и ожидает входящих подключений. На экране появляется сообщение об ожидании.

Клиентское приложение запускается в отдельном терминале командой, указанной на рисунке 3.2. После успешного подключения клиенту выводится сообщение об успешном подключении. Затем пользователю предлагается ввести текст для кодирования.

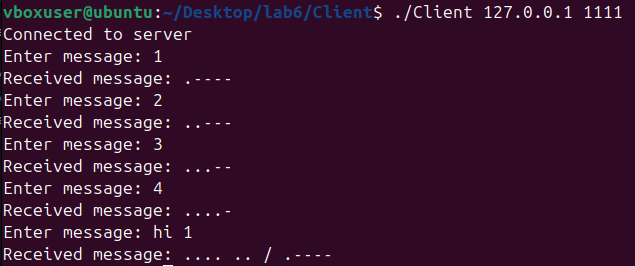


Рисунок 3.2 – Запуск клиента

После ввода строки клиент отправляет её на сервер, получает обратно строку в азбуке Морзе и выводит результат.

Благодаря реализации многопоточности на стороне сервера, к нему могут одновременно подключаться несколько клиентов. Каждый из них обрабатывается в отдельном потоке, что позволяет обеспечить параллельную работу без задержек. Это можно продемонстрировать, запустив несколько клиентов в разных окнах терминала. Все они смогут одновременно отправлять запросы и получать ответы, не мешая друг другу.

При завершении работы клиентское приложение закрывает соединение, и соответствующий поток на сервере завершается. Если все клиенты завершили работу, сервер остаётся активным и продолжает ожидать новых подключений до тех пор, пока его не остановят вручную.

Таким образом, разработанное приложение корректно реализует заданную сетевую функциональность. Сервер успешно принимает соединения, корректно обрабатывает запросы клиентов, производит преобразование текста в азбуку Морзе и возвращает результат обратно. Использование многопоточности обеспечивает возможность обслуживания нескольких клиентов параллельно, без потери производительности или блокировок.

# ЗАКЛЮЧЕНИЕ

Выполнение данной лабораторной работы позволило глубоко погрузиться в принципы функционирования сетевого взаимодействия в Unix-подобных системах и на практике освоить технологии, лежащие в основе клиент-серверной архитектуры. В процессе реализации проекта были изучены основные механизмы работы с программным интерфейсом сокетов, что дало понимание внутренней логики протоколов транспортного уровня и позволило сформировать устойчивые навыки проектирования и отладки сетевых приложений.

Разработка многопоточного сервера продемонстрировала необходимость грамотной организации параллельной обработки запросов, а также показала, насколько важно учитывать структуру данных и поведение соединений при взаимодействии с удалёнными клиентами. Создание прикладного протокола обмена данными, даже в упрощённой форме, позволило прочувствовать специфику построения сетевых сервисов и необходимость точного соблюдения логики общения между участниками сети.

Клиентская часть программы наглядно продемонстрировала основные этапы установления и использования соединения: от инициализации сокета и подключения к серверу до отправки данных и обработки ответа. В ходе работы был обеспечен интерактивный режим общения с сервером, что позволило визуально оценить эффективность и стабильность соединения, а также корректность обработки текстовых сообщений. Применение преобразования текста в азбуку Морзе стало удачным примером простой, но наглядной функции, демонстрирующей смысл сетевой обработки данных.

Дополнительно был реализован модуль автоматического тестирования функции преобразования, что позволило удостовериться в корректности логики кодирования и выявить потенциальные несоответствия в формате вывода. Это укрепило представление о важности модульного подхода к разработке и тестированию даже в рамках небольших учебных проектов.

В конечном итоге лабораторная работа сформировала не только технические навыки взаимодействия с низкоуровневыми средствами Unix-сетей, но и дала представление о базовых принципах построения распределённых систем. Полученные знания и умения являются фундаментом для более сложных задач, связанных с разработкой масштабируемых сетевых приложений, систем удалённого управления, онлайн-сервисов и других сетевых решений, востребованных в современной программной инженерии.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Linux и UNIX: программирование в shell. Руководство разработчика. Дэвид Тейнсли – [Электронный ресурс]. – Режим доступа: https://coollib. com/b/275969/read?ysclid=m6ujdy2lag999562018.

[2] Внутреннее устройство Linux. Дмитрий Кетов – [Электронный ресурс]. – Режим доступа: https://djvu.online/file/wkYDVsl8LdABg?ysclid=mdi of2iooo685605108.

[3] UNIX. Профессиональное программирование. У. Ричард Стивенс, Стивен А. Раго – [Электронный ресурс]. – Режим доступа: http://amplab.ru/bo oks/7.pdf.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#define MAX\_MESSAGE\_LENGTH 256

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

{

int sock;

struct sockaddr\_in server;

char message[MAX\_MESSAGE\_LENGTH];

if(argc < 3)

{

printf("Usage: %s <ip address> <port>\n", argv[0]);

return 1;

}

//Create socket

if((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

printf("Could not create socket\n");

return 1;

}

server.sin\_addr.s\_addr = inet\_addr(argv[1]);

server.sin\_family = AF\_INET;

server.sin\_port = htons(atoi(argv[2]));

//Connect to remote server

if(connect(sock, (struct sockaddr \*)&server, sizeof(server)) < 0)

{

perror("Connect failed");

return 1;

}

puts("Connected to server");

while(1)

{

printf("Enter message: ");

fgets(message, MAX\_MESSAGE\_LENGTH, stdin);

if(send(sock, message, strlen(message), 0) < 0)

{

printf("Send failed\n");

return 1;

}

if(recv(sock, message, MAX\_MESSAGE\_LENGTH, 0) <= 0)

{

printf("Answer failed\n");

return 1;

}

printf("Received message: %s\n", message);

}

close(sock);

return 0;

}

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <pthread.h>

#include "functions.h"

#define MAX\_CLIENTS 100

#define BUF\_SIZE 2000

void \*handle\_client(void \*arg) {

int socket = \*((int \*)arg);

char buf[BUF\_SIZE];

int len;

// Читаем данные из сокета. 0 - флаг, чтоб удалять данные из сокета после считывания

while((len = recv(socket, buf, BUF\_SIZE, 0)) > 0)

{

buf[len] = '\0';

std::string answ = encode(std::string(buf));

// Записываем данные в сокет

send(socket, answ.c\_str(), answ.length(), 0);

}

// Закрываем сокет и разрываем все соединения

close(socket);

return NULL;

}

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

int server\_sock, client\_sock, c;

struct sockaddr\_in server, client;

pthread\_t client\_thread;

if(argc != 2) {

printf("Usage: %s <port>\n", argv[0]);

return 1;

}

// Создаем интернетный потоковый сокет

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

if(server\_sock < 0) {

printf("Could not create socket\n");

return 1;

}

server.sin\_family = AF\_INET;

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

// Чтоб наверняка перевести в big endian

server.sin\_port = htons(atoi(argv[1]));

// Привязываем сокет к адресу (типа даем имя)

if(bind(server\_sock, (struct sockaddr \*)&server, sizeof(server)) < 0) {

printf("Bind failed\n");

return 1;

}

// Информируем, что ожидаем запросы на связь

// 1 - сколько запросов можно принять одновременно

// (! не максимальное число подключений)

listen(server\_sock, 1);

printf("Waiting for incoming connections...\n");

// При подключении клиента заносим в переменную дескриптор файла

while((client\_sock = accept(server\_sock, (struct sockaddr \*)&client, (socklen\_t\*)&c))) {

printf("Connection accepted\n");

int \*new\_sock = (int \*)malloc(sizeof(int));

\*new\_sock = client\_sock;

if (pthread\_create(&client\_thread, NULL, handle\_client, (void \*)new\_sock) < 0)

{

printf("Error on create thread\n");

return 1;

}

}

if(client\_sock < 0) {

printf("Accept failed\n");

return 1;

}

return 0;

}