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
Фроленко К.Ю.
Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью данной лабораторной работы является практическое освоение основ построения и функционирования компьютерных сетей, стека протоколов и программных интерфейсов в *Unix*-среде. В рамках работы необходимо изучить сетевую подсистему *Unix*, разобраться в программном интерфейсе сокетов и реализовать программное решение, демонстрирующее принципы взаимодействия по сети *TCP/IP*. Лабораторная работа включает изучение иерархической модели взаимодействия открытых систем, уровневой структуры и протоколов, таких как *OSI* и *TCP/IP*, а также особенности идентификации абонентов в *IP*-сетях и функционирования транспортных протоколов *TCP* и *UDP*.

Практическая часть работы посвящена разработке сетевой утилиты, аналогичной команде *traceroute*, которая позволяет диагностировать маршрут прохождения пакетов через сеть. Программа должна отправлять *ICMP Echo*-запросы с последовательным увеличением значения *TTL* (*Time To Live*), что дает возможность определить последовательность маршрутизаторов, через которые проходит пакет, а также измерить время отклика каждого узла. В процессе работы утилита обрабатывает сообщения об истечении времени (*ICMP Time Exceeded*) от промежуточных маршрутизаторов и *ICMP Echo Reply* от целевого хоста, что позволяет корректно фиксировать адреса и, при возможности, доменные имена узлов.

Кроме того, программа должна поддерживать настройку параметров запуска через командную строку, что включает задание начального и максимального значений *TTL* и указание адреса целевого узла, а также осуществлять валидацию входных параметров для предотвращения ошибок. Для упрощения разработки, тестирования и сопровождения проекта предусмотрена автоматизация сборки с использованием *makefile*, который отвечает за компиляцию исходного кода, создание исполняемого файла и очистку временных артефактов.

Реализация данного проекта позволит не только освоить базовые принципы сетевого программирования в *Unix*-среде, но и получить практический опыт работы с сокетами, сетевыми протоколами и инструментами диагностики сетевых соединений, что является важным аспектом при разработке высокопроизводительных и масштабируемых сетевых приложений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Элементы сетевого программирования охватывают широкий спектр теоретических знаний и практических методов, необходимых для создания, поддержки и оптимизации сетевых приложений в *Unix*-среде. Базой для этих знаний служит иерархическая модель взаимодействия открытых систем, которая описывает уровневую структуру протоколов – от физического и канального уровней до сетевого, транспортного и прикладного. Модель *OSI* служит теоретическим ориентиром, тогда как модель *TCP/IP* применяется на практике для организации обмена данными в современных компьютерных сетях. Протокол *IP* отвечает за маршрутизацию пакетов между узлами, а транспортные протоколы *TCP* и *UDP* реализуют различные механизмы обеспечения надежности и скорости передачи данных, что является критически важным для построения эффективных сетевых приложений [1].

Программный интерфейс сокетов предоставляет разработчикам универсальный набор функций для создания и управления сетевыми соединениями. Основные системные вызовы, такие как *socket, bind, connect, listen, accept, send, recv, sendto* и *recvfrom*, позволяют организовать как клиентские, так и серверные приложения. Сокеты могут работать в потоковом режиме (*TCP*), который обеспечивает гарантию доставки и порядок передачи данных, или в датаграммном режиме (*UDP*), при котором обмен информацией осуществляется с минимальными задержками, но без механизмов подтверждения доставки. Такая гибкость даёт возможность адаптировать реализацию приложения к различным требованиям по надёжности и скорости обмена данными, что особенно важно для распределённых систем реального времени [2].

Особое внимание уделяется использованию *RAW*-сокетов, которые предоставляют разработчикам возможность работы с пакетами на самом низком уровне. *RAW*-сокеты позволяют формировать и анализировать пакеты с произвольной структурой, что необходимо для реализации специализированных утилит диагностики, таких как traceroute или сниффер. В утилите traceroute управление полем *TTL* (*Time To Live*) в заголовке *IP*-пакета используется для определения маршрута прохождения данных через сеть. Каждый промежуточный маршрутизатор уменьшает значение *TTL*, и когда оно достигает нуля, генерируется *ICMP*-сообщение об истечении времени, позволяющее определить *IP*-адрес узла, обработавшего пакет. Такой механизм не только позволяет измерять задержки на каждом участке маршрута, но и даёт возможность визуализировать структуру сети, выявляя проблемные участки или узкие места в маршрутизации [3].

В современных системах масштабируемость сетевых приложений достигается за счёт использования неблокирующих сокетов и механизмов асинхронного ввода-вывода. Технологии мультиплексирования, такие как *select, poll*, а также современные реализации типа *epoll*, позволяют эффективно обрабатывать большое количество одновременных соединений в рамках одного процесса. Это особенно актуально для серверных приложений, где

требуется оперативно реагировать на запросы множества клиентов, минимизируя накладные расходы на переключение контекста и обеспечивая высокую пропускную способность системы. Такие методы способствуют динамическому распределению вычислительных ресурсов и обеспечивают стабильную работу в условиях высокой нагрузки.

Безопасность сетевых соединений становится всё более значимой в условиях современного информационного обмена. Для защиты передаваемых данных применяются технологии шифрования, а также реализуются механизмы аутентификации и контроля доступа. Надёжное шифрование, использование *SSL/TLS* и других протоколов безопасности позволяют предотвратить несанкционированный доступ и обеспечить целостность данных, передаваемых через сеть. Дополнительно, современные системы мониторинга и анализа сетевого трафика помогают обнаруживать аномалии, выявлять атаки и оперативно реагировать на угрозы, что является неотъемлемой частью построения защищённых сетевых приложений.

Автоматизация сборки проектов также играет важную роль в разработке комплексных систем. Использование системы *make* и написание *makefile*-файлов позволяют упорядочить процесс компиляции, управления зависимостями между исходными файлами и очистки временных артефактов. Такой подход не только ускоряет разработку, но и способствует поддержанию единого стиля кода, снижая вероятность возникновения ошибок при интеграции различных модулей. Это особенно важно при регулярном обновлении программного обеспечения и его тестировании на различных Unix-системах, что обеспечивает высокую надёжность и переносимость конечного продукта.

Таким образом, глубокое понимание элементов сетевого программирования – от структурных моделей и протокольных стеков до особенностей работы сокетов, методов асинхронного ввода-вывода, обеспечения безопасности и автоматизации сборки – является фундаментальным для разработки высокопроизводительных, масштабируемых и надёжных сетевых приложений. Эти знания позволяют создавать программные решения, способные эффективно использовать ресурсы системы, оптимизировать обмен данными и удовлетворять растущие требования современных распределённых систем.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

В рамках данной лабораторной работы реализована утилита *traceroute*, предназначенная для диагностики маршрута прохождения пакетов через сеть. Программа написана на языке *C++* и использует *RAW*-сокеты для отправки *ICMP Echo*-запросов, обработки *ICMP*-сообщений и измерения задержек на каждом участке маршрута. Ниже приведено подробное описание основных функций, входящих в состав программы, а также описана логика работы *makefile* для автоматизации сборки проекта.

3.1 Функция isValidTTL

Функция *isValidTTL* отвечает за проверку корректности введённого значения *TTL*, представленного в виде строки. Функция пытается преобразовать строку в целое число и проверяет, что значение находится в допустимом диапазоне от 1 до 255. При обнаружении лишних символов или несоответствия диапазону функция возвращает *false*. Такая проверка необходима для предотвращения передачи недопустимых значений в дальнейшие этапы обработки, что обеспечивает стабильность работы утилиты.

3.2 Функция printHelp

Функция *printHelp* выводит подробное справочное сообщение с описанием параметров командной строки, необходимых для запуска программы. При активации опции *--help* или при некорректном вводе программа выводит информацию о том, как задаются начальное значение *TTL* (опция *-f*), максимальное значение *TTL* (опция *-m*) и адрес целевого узла. Это помогает пользователю правильно настроить запуск утилиты и избежать ошибок ввода.

3.3 Функция createSocket

Функция *createSocket* создаёт *RAW*-сокет с использованием системного вызова *socket* и протокола *ICMP*. Поскольку *RAW*-сокеты позволяют формировать пакеты с произвольной структурой, их использование требует повышенных привилегий (например, запуска от имени суперпользователя). В случае неудачного создания сокета функция завершает выполнение программы, предотвращая дальнейшие ошибки в процессе работы.

3.4 Функция parseCommandLine

Функция *parseCommandLine* отвечает за разбор аргументов командной строки с использованием библиотеки *getopt_long*. Она обрабатывает опции *-f* и *-m* для задания начального и максимального значения *TTL*, а также проверяет

наличие обязательного параметра – адреса целевого узла. При неверном вводе или отсутствии необходимых параметров функция выводит сообщение об ошибке и справку по использованию, что способствует корректной настройке запуска утилиты.

3.5 Функция resolveHostname

Функция *resolveHostname* реализует разрешение доменного имени целевого узла в его *IP*-адрес. Для этого используется вызов *gethostbyname*, результат которого копируется в соответствующую структуру. В случае ошибки (например, если имя хоста некорректно или возникли проблемы с сетью) функция выводит сообщение об ошибке и возвращает *false*, что позволяет пользователю своевременно обнаружить и устранить проблему.

3.6 Функция checksum

Функция *checksum* вычисляет контрольную сумму для *ICMP*-пакета. Это необходимо для обеспечения целостности данных, передаваемых в пакете, поскольку получающая сторона проверяет контрольную сумму для обнаружения возможных ошибок передачи. Алгоритм расчёта включает суммирование содержимого пакета с последующим выполнением операций побитового сдвига и инвертирования суммы, что гарантирует корректное вычисление контрольной суммы.

3.7 Функция currentTimeMicroseconds

Функция *currentTimeMicroseconds* использует системный вызов *gettimeofday* для получения текущего времени с точностью до микросекунд. Это время применяется для измерения интервалов между отправкой *ICMP*-запроса и получением ответа, что позволяет определить задержку на каждом сегменте маршрута. Точное измерение времени является важным аспектом диагностики, так как помогает оценить производительность и стабильность сетевого соединения.

3.8 Функция traceRoute

Функция *traceRoute* представляет собой ядро утилиты и реализует логику пошаговой диагностики маршрута. В цикле, начиная с заданного начального значения *TTL* и до достижения максимального предела, функция отправляет три *ICMP Echo*-запроса для каждого значения *TTL*. Сначала устанавливается значение *TTL* для сокета через *setsockopt*, затем формируется *ICMP*-пакет, для которого вычисляется контрольная сумма. Отправка пакета осуществляется с помощью *sendto*, после чего функция ожидает ответа в течение заданного таймаута с использованием *select*. При получении ответа анализируется *IP*-адрес отправителя, выполняется попытка разрешения

доменного имени, а также производится измерение задержки. Если *IP*-адрес полученного ответа совпадает с адресом целевого узла, трассировка завершается. Такой пошаговый подход позволяет не только определить последовательность промежуточных маршрутизаторов, но и оценить время отклика на каждом этапе маршрута.

3.9 Функция main

Основная функция *main* объединяет все этапы работы утилиты *traceroute*. Сначала производится разбор командной строки с помощью *parseCommandLine*, после чего выполняется разрешение доменного имени целевого узла через *resolveHostname*. Затем создаётся *RAW*-сокет функцией *createSocket* и настраивается структура адреса, необходимая для передачи пакетов. После вывода сообщения о начале трассировки вызывается функция *traceRoute*, которая осуществляет отправку *ICMP*-пакетов и обработку ответов. По завершении работы сокет закрывается, и программа корректно завершается. Такая структура *main* обеспечивает последовательное выполнение всех необходимых операций для диагностики сетевого маршрута.

3.10 Makefile

Для автоматизации сборки проекта используется *makefile*, который включает цели для компиляции исходных файлов и создания итогового исполняемого файла с именем *traceroute*. *Makefile* определяет переменные компилятора (например, *CXX*) и флаги компиляции (*CXXFLAGS*), включающие предупреждения, оптимизацию и поддержку стандарта *C++17*. В *makefile* также предусмотрена цель *clean*, предназначенная для удаления временных объектных файлов и артефактов сборки. Такой подход упрощает тестирование, обновление и последующую поддержку проекта, обеспечивая единообразие и надежность сборочного процесса.

Таким образом, программа *traceroute* демонстрирует практическое применение сетевого программирования в *Unix*-среде. Каждый модуль утилиты выполняет свою специализированную задачу – от проверки входных параметров и разрешения доменных имен до отправки и обработки *ICMP*-пакетов, что позволяет не только определять маршрут прохождения данных, но и измерять задержки на каждом участке сети. Реализация проекта включает тщательную обработку ошибок, валидацию данных и автоматизацию сборки, что делает утилиту надёжным и удобным инструментом для анализа сетевых соединений.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

При запуске утилиты *traceroute* пользователь указывает в командной строке начальное значение *TTL*, максимальное значение *TTL* и адрес целевого узла (например, *google.com*). Программа начинает работу с проверки корректности введённых параметров и разрешения доменного имени целевого узла в его *IP*-адрес. После успешного разрешения выводится сообщение о начале трассировки, например: *Tracing route to 216.58.215.78*.

Утилита отправляет серию *ICMP Echo*-запросов с последовательным увеличением значения *TTL*. На каждом этапе, для каждого значения *TTL*, программа отправляет несколько запросов и ожидает ответа от промежуточного маршрутизатора или от конечного узла. Если в установленное время ответ не получен, на соответствующем этапе выводится символ *, что указывает на отсутствие отклика. При получении ответа отображается *IP*-адрес узла, а при наличии – и его доменное имя, а также время, затраченное на прохождение пакета от отправителя до получателя.

Такой подход позволяет определить последовательность узлов, через которые проходит пакет, а также оценить задержку на каждом участке маршрута. При достижении целевого узла утилита прекращает дальнейшую отправку запросов, завершая процесс трассировки.

На рисунке 4.1 приведён пример консольного вывода утилиты *traceroute* при трассировке маршрута до *google.com*. Видно, как пакеты проходят через несколько этапов – от локального маршрутизатора, через узлы сети провайдера, до конечного сервера. Каждый этап сопровождается информацией о времени отклика, что позволяет детально проанализировать характеристики сети и выявить возможные проблемы с задержками.

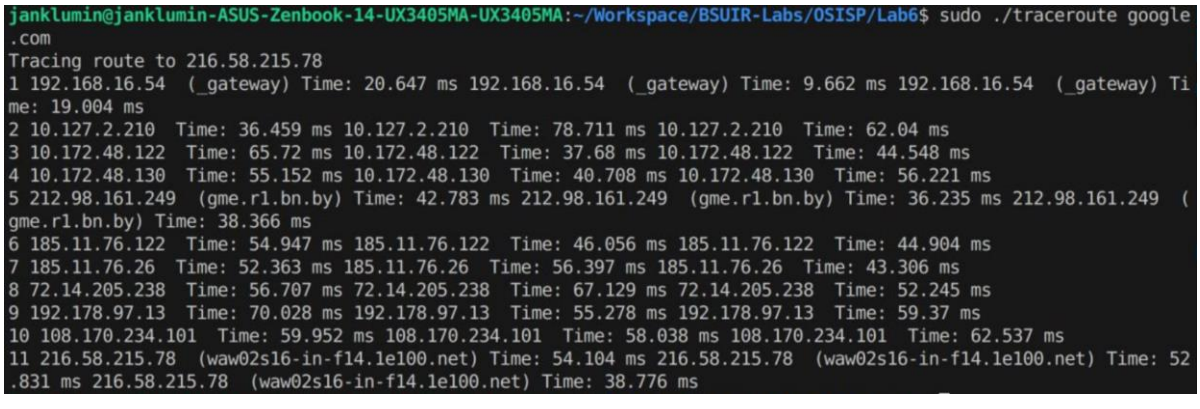

Рисунок 4.1 – Нахождение маршрута до *google.com*

Таким образом, результаты демонстрируют, что программа корректно определяет последовательность маршрутизаторов на пути к целевому узлу и точно измеряет время отклика на каждом этапе, что является важным инструментом для диагностики и анализа сетевых соединений.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана утилита *traceroute*, демонстрирующая практическое применение сетевого программного интерфейса сокетов в *Unix*-среде для диагностики маршрута прохождения сетевых пакетов. Реализация проекта позволила применить и закрепить знания о работе с *RAW*-сокетами, отправке и обработке *ICMP*-пакетов, а также об организации сетевого взаимодействия через протоколы *TCP/IP*.

Разработанная программа включает в себя обработку параметров командной строки, разрешение доменных имен, измерение времени отклика и последовательное увеличение значения *TTL*, что позволяет точно определить маршрут от источника до целевого узла. Особое внимание было уделено обработке ошибок, таким как проблемы с разрешением доменных имен, некорректными значениями *TTL* и сбоями при работе с сокетами, что повышает надежность и устойчивость утилиты к возможным сбоям в процессе работы.

Кроме того, модульная структура кода и автоматизация сборки с использованием *makefile* существенно упростили процесс компиляции, тестирования и дальнейшего сопровождения проекта. Такой подход способствует быстрому внесению изменений и поддержанию единого стиля разработки, что особенно важно для сетевых приложений, требующих высокой стабильности и переносимости между различными *Unix*-системами.

Полученные знания и практические навыки в области сетевого программирования могут быть успешно применены для разработки более сложных утилит диагностики и мониторинга сетевых соединений, а также для создания высокопроизводительных распределённых систем. Реализованный проект демонстрирует, что грамотное применение сетевых протоколов и функций *API* позволяет не только выявить маршруты прохождения данных, но и оценить качество и надёжность сетевых соединений. Таким образом, поставленные задачи лабораторной работы были успешно решены, а результаты экспериментов подтвердили эффективность разработанной утилиты для диагностики сетевых маршрутов и анализа задержек на каждом этапе соединения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Beej's Guide to Network Programming [Электронный ресурс]. – Режим доступа: http://beej.us/guide/bgnet/. – Дата доступа: 03.03.2025.

[2] man7.org. «socket(2)» [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man2/socket.2.html. – Дата доступа: 03.03.2025.

[3] RFC 793 – Transmission Control Protocol [Электронный ресурс]. – Режим доступа: https://tools.ietf.org/html/rfc793. – Дата доступа: 03.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

```
#ifndef TRACEROUTE_FUNCIONS_H
#define TRACEROUTE_FUNCIONS_H

#include <iostream>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/time.h>
#include <stdexcept>
#include <getopt.h>

#define PACKET_SIZE 4096
#define MAX_HOPS 30
#define TIMEOUT_SEC 1

bool isValidTTL(const std::string &ttl_str);
int createSocket(int protocol);
void printHelp(const char *programName);
void traceRoute(int fd, const struct sockaddr_in &addr, int first_ttl, int
max_ttl);
bool parseCommandLine(int argc, char *argv[], int &first_ttl, int &max_ttl,
std::string &destination_host);
bool resolveHostname(const std::string &hostname, struct in_addr &address);
unsigned short checksum(void *b, int len);
long long currentTimeMicroseconds();

#endif
#include <netinet/tcp.h>
#include "funcions.h"

bool isValidTTL(const std::string& ttl_str) {
    try {
        size_t pos = 0;
        int ttl_value = std::stoi(ttl_str, &pos);
        if (pos != ttl_str.size()) {
            return false;
        }
        return (ttl_value >= 1 && ttl_value <= 255);
    } catch (const std::invalid_argument& e) {
        return false;
    }
}


void printHelp(const char *programName) {
    std::cerr << "Usage: sudo " << programName << " [-f first_ttl] [-m max_ttl]
<destination_host>\n";
    std::cerr << "Options:\n";
    std::cerr << "  -f, --first-ttl=VALUE  Start from the first_ttl hop (instead
from 1)\n";
    std::cerr << "  -m, --max-ttl=VALUE    Set the max number of hops (max TTL
to be reached). Default is 30\n";
    std::cerr << "  -h, --help             Read this help and exit\n";
```

```
}

int createSocket(int protocol) {
    int sock;
    if ((sock = socket(AF_INET, SOCK_RAW, protocol)) < 0) {
        exit(EXIT_FAILURE);
    }
    return sock;
}

bool parseCommandLine(int argc, char *argv[], int& first_ttl, int& max_ttl,
std::string& destination_host) {
    struct option long_options[] = {
            {"first-ttl", required_argument, nullptr, 'f'},
            {"max-ttl", required_argument, nullptr, 'm'},
            {"help", no_argument, nullptr, 'h'},
            {nullptr, 0, nullptr, 0}
    };

    int opt;
    while ((opt = getopt_long(argc, argv, "f:m:h", long_options, nullptr)) !=
-1) {
        switch (opt) {
            case 'f':
                if (!isValidTTL(optarg)) {
                    std::cerr << "First hop value must be in the range 1-
255.\n";
                    return false;
                }
                first_ttl = std::stoi(optarg);
                break;
            case 'm':
                if (!isValidTTL(optarg)) {
                    std::cerr << "Max TTL value must be in the range 1-255.\n";
                    return false;
                }
                max_ttl = std::stoi(optarg);
                break;
            case 'h':
                printHelp(argv[0]);
                return false;
            case '?':
                std::cerr    <<    "Unknown    option    character    '"    <<
static_cast<char>(optopt) << "'.\n";
                printHelp(argv[0]);
                return false;
            default:
                abort();
        }
    }

    if (optind >= argc) {
        std::cerr << "Missing destination_host.\n";
        printHelp(argv[0]);
        return false;
    }

    destination_host = argv[optind];

    if (max_ttl < first_ttl) {
        std::cerr << "Max TTL value must be greater than or equal to first TTL
value.\n";
        return false;
    }
```

```
    return true;
}

bool resolveHostname(const std::string& hostname, struct in_addr& address) {
    struct hostent *host = gethostbyname(hostname.c_str());
    if (host == nullptr) {
        if (h_errno == TRY_AGAIN) {
            std::cerr << "Temporary failure in name resolution. Please try again
later.\n";
        } else {
            std::cerr << "Error resolving destination host.\n";
        }
        return false;
    }
    std::memcpy(&address, host->h_addr_list[0], sizeof(struct in_addr));
    return true;
}

unsigned short checksum(void *b, int len) {
    auto *buf = reinterpret_cast<unsigned short *>(b);
    unsigned int sum;
    unsigned short result;

    for (sum = 0; len > 1; len -= 2)
        sum += *buf++;

    if (len == 1)
        sum += *(reinterpret_cast<unsigned char *>(buf));

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    result = ~sum;
    return result;
}

long long currentTimeMicroseconds() {
    struct timeval tv{};
    if (gettimeofday(&tv, nullptr) == -1) {
        perror("geotargetting");
        return -1;
    }
    return static_cast<long long>(tv.tv_sec) * 1000000LL + static_cast<long
long>(tv.tv_usec);
}

void traceRoute(int fd, const struct sockaddr_in& addr, int first_ttl, int
max_ttl)
{
    fd_set read_set;
    struct timeval timeout{};
    struct icmphdr icmp_packet{};
    char packet[PACKET_SIZE];
    int ttl;
    bool reachedDestination = false;

    for (ttl = first_ttl; ttl <= max_ttl && !reachedDestination; ++ttl) {
        std::cout << ttl << " ";
        for (int i = 0; i < 3; ++i) {
            if (setsockopt(fd, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl)) < 0) {
                perror("SocketRocket");
                return;
            }
            long long sendTime = currentTimeMicroseconds();
```

```
            if (sendTime == -1) {
                return;
            }
            std::memset(&icmp_packet, 0, sizeof(icmp_packet));
            icmp_packet.type = ICMP_ECHO;
            icmp_packet.code = 0;
            icmp_packet.un.echo.id = getpid();
            icmp_packet.un.echo.sequence = ttl;
            icmp_packet.checksum           =           checksum(&icmp_packet,
sizeof(icmp_packet));

            std::memcpy(packet, &icmp_packet, sizeof(icmp_packet));

            if      (sendto(fd,      packet,      sizeof(icmp_packet),      0,
reinterpret_cast<const struct sockaddr *>(&addr),
                        sizeof(addr)) <= 0) {
                perror("sendto");
                return;
            }

            FD_ZERO(&read_set);
            FD_SET(fd, &read_set);
            timeout.tv_sec = TIMEOUT_SEC;
            timeout.tv_usec = 0;
            if (select(fd + 1, &read_set, nullptr, nullptr, &timeout) > 0) {
                char reply[PACKET_SIZE];
                struct sockaddr_in in{};
                socklen_t addr_len = sizeof(in);
                ssize_t bytes_received = recvfrom(fd, reply, sizeof(reply), 0,
                                                  reinterpret_cast<struct
sockaddr *>(&in), &addr_len);
                if (bytes_received < 0) {
                    perror("therefrom");
                    return;
                }
                std::cout << inet_ntoa(in.sin_addr) << " ";

                struct   hostent   *host_info   =   gethostbyaddr(&in.sin_addr,
sizeof(in.sin_addr), AF_INET);
                if (host_info != nullptr) {
                    std::cout << " (" << host_info->h_name << ")";
                }

                if (in.sin_addr.s_addr == addr.sin_addr.s_addr) {
                    reachedDestination = true;
                }
            } else {
                std::cout << "* ";
            }
            long long receiveTime = currentTimeMicroseconds();
            if (receiveTime == -1) {
                return;
            }
            std::cout << " Time: " << static_cast<long double>(receiveTime -
sendTime) / 1000.0 << " ms ";
        }
        std::cout << std::endl;
    }
}
#include "funcions.h"

int main(int argc, char *argv[]) {
    int first_ttl = 1;
    int max_ttl = MAX_HOPS;
```

```
    std::string destination_host;

    struct sockaddr_in addr{};

    if (argc == 1 || (argc == 2 && std::string(argv[1]) == "--help")) {
        printHelp(argv[0]);
        return 0;
    }

    if (!parseCommandLine(argc, argv, first_ttl, max_ttl, destination_host)) {
        return 1;
    }

    struct in_addr dest_addr{};
    if (!resolveHostname(destination_host, dest_addr)) {
        return 1;
    }

    int fd;
    fd = createSocket(IPPROTO_ICMP);

    std::memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr = dest_addr;

    std::cout << "Tracing route to " << inet_ntoa(addr.sin_addr) << std::endl;

    traceRoute(fd, addr, first_ttl, max_ttl);

    close(fd);

    return 0;
}
```