Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Операционные системы и системное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему
МНОГОПОТОЧНЫЙ FTP-СЕРВЕР

БГУИР КП 1–40 02 01 01 212 ПЗ

Студент: И. В. Зинович

Руководитель: Д. Н. Басак

МИНСК 2024

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: ФКСиС. Кафедра: ЭВМ.
Специальность: 40 02 01 «Вычислительные машины, системы и сети».

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Б.В.Никульшин
«____» _____ 2024 г.

ЗАДАНИЕ
по курсовому проекту студента
Зиновича Ивана Вячеславовича

1 Тема проекта: «Многопоточный FTP-Сервер».

2 Срок сдачи студентом законченного проекта: 20 мая 2024 г.

3 Исходные данные к проекту:
3.1 Язык программирования: C++.
3.2 Среда разработки: CLion.
3.3 Операционная система: Linux (macOS).

4 Содержание пояснительной записки (перечень подлежащих разработке вопросов):
Введение 1. Обзор литературы. 2. Системное проектирование. 3. Функциональное проектирование. 4. Разработка программных модулей. 5. Программа и методика испытаний. 6. Руководство пользователя. Заключение. Список использованных источников. Приложения.

5 Перечень графического материала (с точным указанием обязательных чертежей):
5.1 Многопоточный FTP-Сервер.
Схема структурная.
5.2 Многопоточный FTP-Сервер.
Диаграмма классов.
5.3 Многопоточный FTP-Сервер.
Схема программы.

КАЛЕНДАРНЫЙ ПЛАН

Наименование этапов дипломного проекта	Объем этапа, %	Срок выполнения этапа	Примечания
Подбор и изучение литературы. Сравнение аналогов.	10	23.03 – 05.04	
Структурное проектирование	15	05.04 – 12.04	
Функциональное проектирование	25	12.04 – 24.04	
Разработка программных модулей	20	24.04 – 08.05	
Программа и методика испытаний	10	8.05 – 15.05	
Оформление пояснительной записки	15	20.05 – 30.05	

Дата выдачи задания: 22.02.2024

Руководитель Д. Н. Басак

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ ____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Многопоточный FTP-сервер является эффективным средством обмена информацией. В современном мире, где информация играет важную роль, FTP-серверы обеспечивают быстрый и надежный доступ к данным. FTP (File Transfer Protocol) - это протокол передачи файлов, который позволяет пользователям загружать и скачивать файлы с удаленного сервера.

В прошлом FTP-серверы работали в однопоточном режиме, что ограничивало их возможности обработки только одного запроса от клиента за раз. Это снижало производительность, особенно при большом количестве пользователей.

Многопоточный FTP-сервер решает эту проблему, позволяя одновременную обработку нескольких запросов. Он создает отдельный поток для каждого подключенного клиента. Таким образом, сервер может обслуживать несколько пользователей одновременно, не ухудшая производительность.

Многопоточный FTP-сервер обладает следующими преимуществами:

1 Увеличение производительности: Благодаря многопоточности сервер может обрабатывать больше запросов одновременно, что ускоряет обмен данными.

2 Повышение эффективности: Сервер способен обслуживать большее количество пользователей, не ухудшая производительность.

3 Улучшение отзывчивости: Пользователи получают более быстрый отклик от сервера, что обеспечивает более удобное взаимодействие.

4 Масштабируемость: Многопоточный сервер легко масштабируется для поддержки большего числа пользователей.

Многопоточный FTP-сервер может быть реализован на различных языках программирования, таких как C++, Java, Python и других. Однако использование системных функций языка C и C++ позволяет более низкоуровневый доступ к сетевым возможностям операционной системы, что способствует повышению производительности и эффективности сервера.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор FTP Спецификации

FTP (File Transfer Protocol) - это протокол передачи файлов, который позволяет пользователям загружать и скачивать файлы с удаленного сервера. Он является одним из старейших и наиболее широко используемых протоколов в Интернете.

FTP отличается от других приложений тем, что он использует два TCP соединения для передачи файла:

1 Управляющее соединение устанавливается как обычное соединение клиент-сервер. Сервер осуществляет пассивное открытие на заранее известный порт FTP (21) и ожидает запроса на соединение от клиента. Клиент осуществляет активное открытие на TCP порт 21, чтобы установить управляющее соединение. Управляющее соединение существует все время, пока клиент общается с сервером. Это соединение используется для передачи команд от клиента к серверу и для передачи откликов от сервера. Тип IP сервиса для управляющего соединения устанавливается для получения "минимальной задержки", так как команды обычно вводятся пользователем (рисунок 3.2).

2 Соединение данных открывается каждый раз, когда осуществляется передача файла между клиентом и сервером. (Оно также открывается и в другие моменты, как мы увидим позже.) Тип сервиса IP для соединения данных должен быть "максимальная пропускная способность", так как это соединение используется для передачи файлов.

Схема взаимодействия клиента и сервера представлена на рисунке 1.1.1.

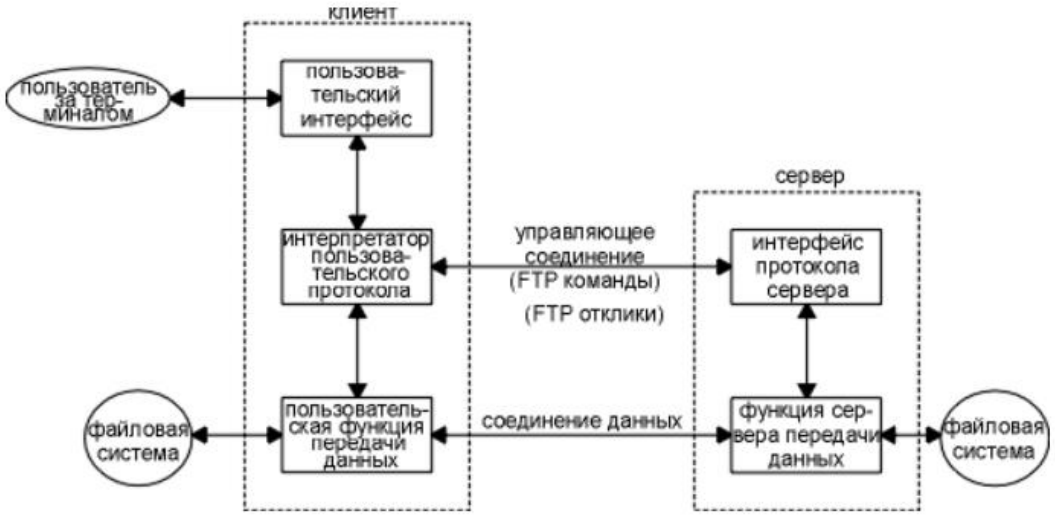

Рисунок 1.1.1 – Процессы, участвующие в передаче файлов

Из рисунка видно, что интерактивный пользователь обычно не видит команды и отклики, которые передаются по управляющему соединению. Эти детали

оставлены двум интерпретаторам протокола. Квадратик, помеченный как "пользовательский интерфейс", это именно то, что видит интерактивный пользователь (полноэкранный интерфейс, основанный на меню, командные строки и так далее). Интерфейс конвертирует ввод пользователя в FTP команды, которые отправляются по управляющему соединению. Отклики, возвращаемые сервером по управляющему соединению, конвертируются в формат, удобный для пользователя.

Команды и отклики передаются по управляющему соединению между клиентом и сервером в формате NVT ASCII.

Команды состоят из 3 или 4 байт, а именно из заглавных ASCII символов, некоторые с необязательными аргументами. Клиент может отправить серверу более чем 30 различных FTP команд. На рисунке 1.1.2 показаны некоторые наиболее широко используемые команды. Однако в данной реализации FTP-протокола по управляющему соединению отправляется команда, а по каналу данных отправляются необходимые данные для этого сервера.

Команда	Описание
ABOR	прервать предыдущую команду FTP и любую передачу данных
LIST список файлов	список файлов или директорий
PASS пароль	пароль на сервере
PORT n1,n2,n3,n4,n5,n6	IP адрес клиента (n1.n2.n3.n4) и порт (n5 x 256 + n6)
QUIT	закрыть бюджет на сервере
RETR имя файла	получить (get) файл
STOR имя файла	положить (put) файл
SYST	сервер возвращает тип системы
TYPE тип	указать тип файла: A для ASCII, I для двоичного
USER имя пользователя	имя пользователя на сервере

Рисунок 1.1.2 – Распространенные FTP команды

Отклики состоят из 3-циферных значений в формате ASCII, и необязательных сообщений, которые следуют за числами. Подобное представление откликов объясняется тем, что программному обеспечению необходимо посмотреть только цифровые значения, чтобы понять, что ответил процесс, а дополнительную строку может прочитать человек. Поэтому пользователю достаточно просто прочитать сообщение (причем нет необходимости запоминать все цифровые коды откликов). Каждая из трех цифр в коде отклика имеет собственный смысл. (К примеру, протокол передачи почтовых сообщений - SMTP, использует те же соглашения для своих команд и откликов.) В данной реализации FTP-протокола сервер отправляет не только коды откликов, но и конкретную информацию об отклике. На рисунке 1.1.3 показаны значения первых и вторых цифр в коде отклика.

Третья цифра дает дополнительное объяснение сообщению об ошибке. Ниже приведены некоторые типичные отклики с возможными объясняющими строками:

– 125 соединение данных уже открыто, начало передачи;

– 200 команда исполнена;

– 214 сообщение о помощи (для пользователя);
– 331 имя пользователя принято, требуется пароль;
– 425 невозможно открыть соединение данных;
– 452 ошибка записи файла;
– 500 синтаксическая ошибка (неизвестная команда);
– 501 синтаксическая ошибка (неверные аргументы);
– 502 нереализованный тип MODE.

Отклик	Описание
1yz	Положительный предварительный отклик. Действие началось, однако необходимо дождаться еще одного отклика перед отправкой следующей команды.
2yz	Положительный отклик о завершении. Может быть отправлена новая команда.
3yz	Положительный промежуточный отклик. Команда принята, однако необходимо отправить еще одну команду.
4yz	Временный отрицательный отклик о завершении. Требуемое действие не произошло, однако ошибка временная, поэтому команду необходимо повторить позже.
5yz	Постоянный отрицательный отклик о завершении. Команда не была воспринята и повторять ее не стоит.
x0z	Синтаксическая ошибка.
x1z	Информация.
x2z	Соединения. Отклики имеют отношение либо к управляющему, либо к соединению данных.
x3z	Аутентификация и бюджет. Отклик имеет отношение к логированию или командам, связанным с бюджетом.
x4z	Не определено.
x5z	Состояние файловой системы.

Рисунок 1.1.3 – Значения первой и второй цифр в 3-циферном коде отклика

1.2 Обзор программирования сокетов Linux

1.2.1 Понятие сокета

Socket API был впервые реализован в операционной системе Berkley UNIX. Сейчас этот программный интерфейс доступен практически в любой модификации Unix, в том числе в Linux. Хотя все реализации чем-то отличаются друг от друга, основной набор функций в них совпадает. Изначально сокеты использовались в программах на C/C++, но в настоящее время средства для работы с ними предоставляют многие языки (Perl, Java и др.).

Сокеты представляют собой мощный и гибкий механизм, позволяющий организовать взаимодействие между процессами как на одном компьютере, так и в локальной сети или через Интернет. Это открывает широкие возможности для создания распределенных приложений различной сложности.

Более того, сокеты позволяют взаимодействовать с программами, работающими под управлением других операционных систем. Например, Windows Sockets, спроектированный на основе socket API, обеспечивает совместимость с Unix-системами.

Сокет (socket) это конечная точка сетевых коммуникаций. Он является чем-то вроде "портала", через которое можно отправлять байты во внешний мир. Приложение просто пишет данные в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.

В программе сокет идентифицируется дескриптором - это просто переменная типа int. Программа получает дескриптор от операционной системы при создании сокета, а затем передаёт его сервисам socket API для указания сокета, над которым необходимо выполнить то или иное действие.

1.2.2 Атрибуты сокета

С каждым сокет связываются три атрибута: домен, тип и протокол. Эти атрибуты задаются при создании сокета и остаются неизменными на протяжении всего времени его существования. Для создания сокета используется функция `socket()`, имеющая следующий прототип.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Чаще других используются домены Unix и Internet, задаваемые константами AF_UNIX и AF_INET соответственно (префикс AF означает "address family" – "семейство адресов"). При задании `AF_UNIX` для передачи данных используется файловая система ввода/вывода Unix. В этом случае сокеты используются для межпроцессного взаимодействия на одном компьютере и не годятся для работы по сети. Константа `AF_INET` соответствует Internet-домену. Сокеты, размещённые в этом домене, могут использоваться для работы в любой IP-сети. Существуют и другие домены (`AF_IPX` для протоколов Novell, `AF_INET6` для новой модификации протокола IP - IPv6 и т. д.).

Тип сокета определяет способ передачи данных по сети. Чаще других применяются:

1 `SOCK_STREAM`. Передача потока данных с предварительной установкой соединения. Обеспечивается надёжный канал передачи данных, при котором

фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются. Поскольку этот тип сокетов является самым распространённым, до конца раздела мы будем говорить только о нём. Остальным типам будут посвящены отдельные разделы.

2 `SOCK_DGRAM`. Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Обмен данными происходит быстрее, но является ненадёжным: сообщения могут теряться в пути, дублироваться и переупорядочиваться. Допускается передача сообщения нескольким получателям (multicasting) и широковещательная передача (broadcasting).

3 `SOCK_RAW`. Этот тип присваивается низкоуровневым (т. н. "сырым") сокетам. Их отличие от обычных сокетов состоит в том, что с их помощью программа может взять на себя формирование некоторых заголовков, добавляемых к сообщению.

Следует обратить внимание, что не все домены допускают задание произвольного типа сокета. Например, совместно с доменом Unix используется только тип `SOCK_STREAM`. С другой стороны, для Internet-домена можно задавать любой из перечисленных типов. В этом случае для реализации `SOCK_STREAM` используется протокол TCP, для реализации `SOCK_DGRAM` – протокол UDP, а тип `SOCK_RAW` используется для низкоуровневой работы с протоколами IP, ICMP и т. д.

Последний атрибут определяет протокол, используемый для передачи данных. Часто протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции `socket()` можно передать 0, что соответствует протоколу по умолчанию. Тем не менее, иногда (например, при работе с низкоуровневыми сокетами) требуется задать протокол явно. Числовые идентификаторы протоколов зависят от выбранного домена; их можно найти в документации.

1.2.3 Адреса

Перед передачей данных через сокет необходимо связать его с адресом в выбранном домене. Этот процесс называется именованием сокета. Иногда связывание происходит неявно, например, внутри функций `connect()` и `accept()`, но выполнять его необходимо во всех случаях.

Вид адреса зависит от выбранного домена. В Unix-домене адрес – это текстовая строка, представляющая имя файла, через который происходит обмен данными. В Internet-домене адрес задается комбинацией IP-адреса и 16-битного номера порта. IP-адрес определяет хост в сети, а порт - конкретный

сокет на этом хосте. Протоколы TCP и UDP используют различные пространства портов.

Для явного связывания сокета с некоторым адресом используется функция `bind()`. Её прототип имеет вид:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

В качестве первого параметра передаётся дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр, `addr`, содержит указатель на структуру с адресом, а третий – длину этой структуры. Ниже представлено, что она собой представляет.

```
struct sockaddr {
    unsigned short    sa_family;    // Семейство адресов, AF_xxx
    char              sa_data[14];  // 14 байтов для хранения
адреса
};
```

Поле `sa_family` содержит идентификатор домена, тот же, что и первый параметр функции `socket()`. В зависимости от значения этого поля по-разному интерпретируется содержимое массива `sa_data`. Разумеется, работать с этим массивом напрямую не очень удобно, поэтому можно использовать вместо `sockaddr` одну из альтернативных структур вида sockaddr_XX (XX - суффикс, обозначающий домен: "un" - Unix, "in" - Internet и т. д.). При передаче в функцию `bind()` указатель на эту структуру приводится к указателю на `sockaddr`. Ниже представлен пример структуры `sockaddr_in`.

```
struct sockaddr_in {
    short int          sin_family;  // Семейство адресов
    unsigned short int sin_port;    // Номер порта
    struct in_addr     sin_addr;    // IP-адрес
    unsigned char      sin_zero[8]; // "Дополнение" до размера
структуры sockaddr
};
```

Здесь поле `sin_family` соответствует полю `sa_family` в `sockaddr`, в `sin_port` записывается номер порта, а в `sin_addr` - IP-адрес хоста. Поле `sin_addr` само является структурой, которая имеет вид:

```
struct in_addr {
    unsigned long s_addr;
};
```

Следует обратить внимание, что структура хранит одно поле. Дело в том, что раньше `in_addr` представляла собой объединение (union), содержащее гораздо большее число полей. Сейчас, когда в ней осталось всего одно поле, она продолжает использоваться для обратной совместимости.

Существует два порядка хранения байтов в слове и двойном слове. Один из них называется порядком хоста (host byte order), другой - сетевым порядком (network byte order) хранения байтов. При указании IP-адреса и номера порта необходимо преобразовать число из порядка хоста в сетевой. Для этого используются функции `htons()` (Host TO Network Short) и `htonl()` (Host TO Network Long). Обратное преобразование выполняют функции `ntohs()` и `ntohl()`.

1.2.4 Установка соединения (сервер)

Установка соединения на стороне сервера включает четыре обязательных этапа, каждый из которых необходим для успешного соединения. Вначале создается сокет и привязывается к локальному адресу. Если компьютер имеет несколько сетевых интерфейсов с разными IP-адресами, вы можете принимать соединения только с одного из них, указав его адрес в функции `bind()`. Если вам необходимо принимать соединения через любой интерфейс, вы можете использовать константу INADDR_ANY в качестве адреса. Что касается номера порта, вы можете указать конкретный номер или 0 (в таком случае система автоматически выберет свободный порт).

На следующем шаге создаётся очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов. Всё это выполняет функция `listen()`.

```
int listen(int sockfd, int backlog);
```

Первый параметр функции – это дескриптор сокета, а второй параметр задает размер очереди запросов. Каждый раз, когда клиент пытается установить соединение с сервером, его запрос помещается в очередь, так как сервер может быть занят обработкой других запросов. Если очередь заполняется, все последующие запросы будут проигнорированы. Когда сервер готов обработать следующий запрос, он использует функцию `accept()`.

```
#include <sys/socket.h>
```

```
int accept(int sockfd, void *addr, int *addrlen);
```

Функция `accept()` создает новый сокет для общения с клиентом и возвращает его дескриптор. Параметр `sockfd` определяет слушающий сокет. После вызова функции слушающий сокет остается в режиме прослушивания и может принимать другие соединения. В структуру, на которую указывает параметр `addr`, записывается адрес клиентского сокета, который установил соединение с сервером. В переменную, адресуемую указателем `addrlen`, изначально записывается размер структуры. Функция `accept()` обновляет эту переменную, указывая фактически использованный размер. Если вас не интересует адрес клиента, вы можете передать NULL в качестве второго и третьего параметров.

Важно отметить, что новый сокет, полученный с помощью функции `accept()`, связан с тем же адресом, что и слушающий сокет. Сначала это может показаться необычным. Однако в TCP-домене адрес сокета не обязательно должен быть уникальным. Единственное требование заключается в том, чтобы соединения были уникальными, идентифицируемыми по двум адресам сокетов, между которыми происходит обмен данными.

1.2.5 Установка соединения (клиент)

На стороне клиента для установления соединения используется функция `connect()`, которая имеет следующий прототип.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int
addrlen);
```

Здесь `sockfd` представляет собой сокет, который будет использоваться для обмена данными с сервером. `serv_addr` содержит указатель на структуру с адресом сервера, а `addrlen` – длину этой структуры. Обычно нет необходимости привязывать клиентский сокет к локальному адресу заранее, так как функция `connect()` автоматически выполнит эту операцию, выбрав свободный порт. Однако, если требуется явно указать номер порта для клиентского сокета, можно использовать функцию `bind()` перед вызовом `connect()`. Это полезно в случаях, когда сервер соединяется только с клиентами, использующими определенный порт (например, серверы rlogind и rshd). В остальных случаях рекомендуется доверить выбор порта системе, что обеспечивает простоту и надежность.

1.2.6 Обмен данными

После успешного установления соединения можно приступать к обмену данными. Для этой цели используются функции `send()` и `recv()`. В операционной системе Unix также можно воспользоваться файловыми функциями `read()` и `write()` для работы с сокетами, однако они обладают ограниченными возможностями и могут быть несовместимы с другими платформами, такими как Windows. Поэтому рекомендуется избегать использования `read()` и `write()`.

Функция `send()` предназначена для отправки данных и имеет следующий прототип.

```
int send(int sockfd, const void *msg, int len, int flags);
```

Здесь `sockfd` – это дескриптор сокета, через который мы отправляем данные, `msg` - указатель на буфер с данными, `len` - длина буфера в байтах, а `flags` – набор битовых флагов, управляющих работой функции (если флаги не используются, передайте функции 0). Вот некоторые из них (полный список можно найти в документации):

1 `MSG_OOB`. Предписывает отправить данные как срочные (out of band data, OOB). Концепция срочных данных позволяет иметь два параллельных канала данных в одном соединении. Иногда это бывает удобно. Например, Telnet использует срочные данные для передачи команд типа Ctrl+C. В настоящее время использовать их не рекомендуется из-за проблем с совместимостью (существует два разных стандарта их использования, описанные в RFC793 и RFC1122). Безопаснее просто создать для срочных данных отдельное соединение.

2 `MSG_DONTROUTE`. Запрещает маршрутизацию пакетов. Нижележащие транспортные слои могут проигнорировать этот флаг.

Функция `send()` возвращает число байтов, которое на самом деле было отправлено (или -1 в случае ошибки). Это число может быть меньше указанного размера буфера. Если необходимо отправить весь буфер целиком, вам придётся написать свою функцию и вызывать в ней `send()`, пока все данные не будут отправлены.

Для чтения данных из сокета используется функция `recv()`.

```
int recv(int sockfd, void *buf, int len, int flags);
```

Функция `recv()` имеет схожий синтаксис и использование с функцией `send()`. Она также принимает дескриптор сокета, указатель на буфер и набор флагов. Флаг `MSG_OOB` используется для приема срочных данных, а флаг `MSG_PEEK` позволяет "подглядеть" данные, полученные от удаленного хоста, не удаляя их из системного буфера (это означает, что при следующем вызове

recv() вы получите те же самые данные). Полный список флагов можно найти в документации. Аналогично функции send(), функция recv() возвращает количество прочитанных байтов, которое может быть меньше размера буфера.

Вы можете легко написать свою собственную функцию recvall(), которая будет читать данные до полного заполнения буфера. Также стоит отметить особый случай, когда функция recv() возвращает 0. Это означает, что соединение было разорвано.

1.2.7 Закрытие сокета

Закончив обмен данными, необходимо закрыть сокет с помощью функции close(). Это приведёт к разрыву соединения.

```
#include <unistd.h>
int close(int fd);
```

Также можно запретить передачу данных в каком-то одном направлении, используя shutdown().

```
int shutdown(int sockfd, int how);
```

Параметр how может принимать одно из следующих значений:

– 0: запретить чтение из сокета

– 1: запретить запись в сокет

– 2: запретить и то, и другое

Хотя после вызова shutdown() с параметром how, равным 2, вы больше не сможете использовать сокет для обмена данными, вам всё равно потребуется вызвать close(), чтобы освободить связанные с ним системные ресурсы.

1.2.8 Обработка ошибок

При работе с сокетами важно учитывать возможность возникновения ошибок. Если что-то идет не так, все рассмотренные функции могут вернуть значение -1 и установить код ошибки в глобальную переменную errno. Можно проверить значение errno и принять соответствующие меры для восстановления нормальной работы программы без ее прерывания. Также возможно вывести диагностическое сообщение, используя функцию perror(), а затем завершить программу с помощью функции exit().

При обработке ошибок сокетов важно обратить внимание на различные коды ошибок, которые могут возникнуть, и принять соответствующие меры в каждом случае. Обработка ошибок является важной частью разработки надежных и стабильных программ, особенно при работе с сетевыми соединениями.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После изучения основ программирования сокетов и спецификации FTP, можно приступить к реализации FTP-сервера. В данном разделе рассмотрена функциональность, которую будет предоставлять сервер.

2.1 Обзор основных блоков

Данный многопоточный FTP-сервер можно условно разделить на три модуля:

– ядро сервера, которое позволяет обрабатывать клиентов в многопоточном режиме;

– структура с клиентом со стороны сервера

– обработчик команд клиентов, соответствуя FTP-спецификации;

– конфигурация сервера

2.1.1 Блок ядра сервера

Этот компонент отвечает за обработку входящих запросов от клиентов. Для этого создается отдельный фоновый поток, в котором происходит вечный цикл для установления соединения между сервером и клиентом, через Thread Pool, который был реализован самостоятельно. Когда установлено соединение с клиентом, создается отдельный поток для каждого клиента, чтобы обеспечить параллельную обработку запросов. Данный поток также происходит в бесконечном цикле, пока клиент не разорвет соединение с сервером.

2.1.2 Блок структуры с клиентом со стороны сервера

Данная структура на стороне сервера обеспечивает возможность обработки команд для определенного клиента, а также вывод сообщений и управление подключением и отключением клиента. Кроме того, она хранит файловые дескрипторы сокетов клиентов.

2.1.3 Блок обработчика команд клиентов, соответствуя FTP-спецификации

Данный компонент предоставляет функционал для обработки конкретных команд, отправляемых клиентом серверу в соответствии с протоколом FTP. Он позволяет осуществлять переходы по директориям,

отправлять эхо-сообщения, загружать файлы и выводить текущее содержимое директорий.

2.1.4 Блок конфигурации сервера

Эта конфигурация предназначена для обработки JSON-файла, содержащего информацию о пользователях, такую как логин, пароль, номер текущего порта и IP-адрес, по которому сервер должен функционировать. Основная функция данного компонента заключается в парсинге указанного JSON-файла.

2.2 Взаимодействие с пользователем

Взаимодействие с FTP-сервером происходит через консольное клиентское приложение, которое посылает следующие команды:

– USER: Ввод имени пользователя;

– PASS: Ввод пароля;

– LIST: Вывод списка директорий и файлов в текущем каталоге;

– CWD: Переход по директориям;

– RETR: Загрузка файла;

– QUIT: Отключение клиента от сервера.

Процесс аутентификации:

1. Клиент отправляет команду USER с именем пользователя.

2. Сервер проверяет имя пользователя в своем JSON-файле, где хранится информация о всех пользователях.

3. Если имя пользователя найдено, сервер отправляет клиенту запрос на ввод пароля.

4. Клиент отправляет команду PASS с паролем.

5. Сервер проверяет пароль, сравнивая его с паролем, хранящимся в JSON-файле для данного пользователя.

6. Если пароль верен, сервер предоставляет клиенту доступ к своим ресурсам.

2.3 Выбор языка программирования

При выборе языка программирования для написания FTP-сервера было принято решение писать C++ по следующим причинам:

1 Высокая производительность: C++ – это компилируемый язык, который отличается высокой скоростью выполнения. Это особенно важно для

FTP-сервера, который должен обрабатывать большое количество запросов от клиентов.

2 Низкоуровневый доступ: C++ предоставляет низкоуровневый доступ к системным ресурсам, таким как память и сеть. Это позволяет создавать высокопроизводительные и оптимизированные серверные приложения.

3 Гибкость и контроль: C++ - это очень гибкий язык, который позволяет программистам реализовать практически любую функциональность. Это дает полный контроль над работой сервера и позволяет создавать решения, которые идеально соответствуют моим требованиям.

4 Большое сообщество и поддержка: C++ – это один из самых популярных языков программирования в мире. Это означает, что существует огромное количество ресурсов, документации и библиотек, которые могут помочь мне в разработке сервера.

C++ является расширением языка C, которое включает в себя ряд дополнительных возможностей, таких как:

1 Классы: Классы позволяют создавать абстракции данных и реализовывать более сложные структуры программного кода.

2 Контейнеры: Контейнеры, такие как векторы и списки, предоставляют эффективные способы хранения и управления данными.

3 Современные функции для работы с файловой системой: C++ предоставляет более современные и удобные функции для работы с файловой системой, чем C.

Эти дополнительные возможности C++ делают его более подходящим языком для разработки FTP-сервера, чем C.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе пояснительной записки детально рассмотрим функционирование программного обеспечения.

Для выполнения задачи, поставленной в ходе реализации проекта, будет произведён подробный обзор архитектуры реализуемого программного обеспечения, будут рассмотрены основные классы, их строение, связи между классами и другие зависимости, данные таблиц используемых баз данных, а также используемые методы классов, поля и константы.

Так как в разработке приложения используется объектно-ориентированная парадигма программирования, все модули данного программного обеспечения представлены различными классами, логически объединёнными в зависимости от выполняемой функции. В приложении Б изображена диаграмма классов данного программного обеспечения.

Ядро сервера это фундаментальный блок разработки, который обеспечивает многопоточную обработку входящих клиентских подключений. Его основная функциональность включает:

1 Обработка входящих подключений в фоновом потоке: ядро сервера обрабатывает входящие подключения клиентов асинхронно, не блокируя основной поток выполнения.

2 Создание отдельного потока для каждого клиента: для каждого подключившегося клиента создается отдельный поток, что позволяет одновременно обслуживать множество клиентов без потери производительности.

Вспомогательные блоки:

1 Структура клиента со стороны сервера: эта структура имеет связь агрегации с ядром сервера, что означает, что ядро сервера создает экземпляры клиентов.

2 Блок обработчика команд по FTP спецификации: этот блок также имеет связь агрегации с клиентом, но не является обязательным. Клиент может иметь этот обработчик, но может существовать и без него.

3 Конфигурация сервера: конфигурация сервера содержит настройки, определяющие его поведение, поэтому был создан блок, который будет считывать информацию в JSON файла.

3.1 Блок ядра сервера

Класс `ServerCore` отвечает за настройку сокетов сервера, установку их в режим прослушивания и создание фонового потока для приема подключений. При успешном подключении клиента создается отдельный поток для его обслуживания. В этом потоке происходит обработка команд,

поступающих от клиента. Класс `ServerCore` содержит следующие поля, для отображения состояния класса:

– `int server_socket`: хранит информацию о файловом дескрипторе сокета сервера;

– `int server_port`: хранит информацию о номере порта сервера;

– `std::string local_ip_address`: хранит информацию об IP-адресе сервера;

– `ThreadPool thread_pool`: хранит пул потоков, который создает фоновый; поток для обработки входящих соединений.

Данный класс содержит следующие private методы, которые используются в public методах:

– `void create_bind_listen_sockets()`: создает сокет сервера с помощью функции `socket()`, привязывает его к компьютеру с помощью `bind()`, и входит в бесконечный цикл, который принимает входящие соединения от сервера с помощью метода `handlingAccept()` данного класса;

– `void handlingAccept()`: входит в бесконечный цикл, в котором сервер ожидает входящих подключений от клиентов. Как только клиент подключился, выводит сообщение в консоль об успешном подключении клиента (с помощью метода класса `ServerClient::connected()`) после чего создает отдельный поток, который также работает в вечном цикле и обрабатывает команды, поступающие от клиентов. Обработка команд происходит с помощью соответствующий методов класса `ServerClient`;

`ServerCore` содержит следующие публичные методы, который используются в main методе, для запуска сервера и ожидания завершения его работы:

– `void start()`: вызывает метод `create_bind_listen_sockets()`, тем самым запуская работу сервера;

– `void joinLoop()`: ожидает завершения работы всех потоков, который вызваны из пула потоков сервера, тем самым ожидая завершения работы сервера.

3.2 Блок структуры клиента со стороны сервера

Данный блок содержит класс `ServerClient` предназначен для хранения информации о подключенном клиенте и предоставления методов для вывода информации в консоль сервера и обработки команд, поступающих от клиентов, в соответствии со спецификацией FTP.

Класс `ServerClient` содержит следующие поля, отображающий свойства этого класса:

– `int command_socket`: файловый дескриптор управляющего сокета клиента;

– `int data_socket`: файловый дескриптор сокета для передачи данных клиента;

– `bool is_authorized`: булева переменная, показывающая авторизован ли пользователь или нет;

– `FTPSpecification* ftp_specification`: указатель на класс `FTPSpecification`, из которого вызывается метод `handler()`, который обрабатывает команды клиента, соответствуя FTP-спецификации.

Данный класс содержит следующие публичные методы:

– `void disconnect()`: производит отключение клиента от сервера, путем закрытия сокетов и выводит сообщение об отключение клиента в консоль сервера;

– `void connected() const`: выводит сообщение об успешном подключении клиента в консоль сервера и отправляет клиенту код ответа об успешном соединении;

– `ssize_t get_command_from_client(char buffer[]) const`: считывает команду от клиента по управляющему каналу (`command_socket`);

– `size_t get_data_from_client(char buffer[])`: считывает данные, которые посылает клиент по каналу данных(`data_socket`);

– `void handle_command(char command[]) const`: обрабатывает команды, поступающие от клиента, путем делегирования работы классу `FTPSpecification`;

– `void authorize()`: обработка авторизации клиента со стороны сервера. Изначально входит в вечный цикл, в котором пытается обработать команду USER, поступающую от клиента, а затем аналогично команду PASS;

– `void clear_socket_data(int socket_fd)`: очищает сокет от лишних данных.

3.3 Блок обработчика команд клиентов, соответствуя FTP-спецификации

В этом блоке реализуется класс `FTPSpecification`, который включает в себя один публичный метод, используемый в `ServerClient` для обработки всех команд, полученных от клиентов. Кроме того, класс содержит приватные методы, предназначенные для обработки конкретных команд, таких как LIST, RETR, CWD, ECHO и QUIT.

Данный класс содержит следующие поля, для отображения состояния экземпляра класса:

– `std::string current_dir`: содержит информацию о текущей директории, в которой находится конкретный клиент;

– `std::string based_dir`: информация о начальной директории, откуда начинает свою работу сервер;

– `static std::mutex retr_mutex`: статический мьютекс для ограничения доступа других потоков к методам, которые не являются потокобезопасными (т.е. методам, которые изменяют состояние).

Для обработки команд класс `FTPSpecification` предоставляет следующие методы:

– `void handler(char command[], int fcs, int fds)`: содержит в себе все методы, которые обрабатывают стандартные команды FTP. Выбор команды происходит с помощью конструкций `if`, `if else`. В случае несоответствия команды, клиенту отправляется сообщение по управляющему соединению, что такой команды не существует;

– `void echo_handler(int fcs, int fds)`: обработка команды ECHO от клиента. В случае отсутствия текста после ECHO, возвращает клиенту ошибку, которая говорит, что необходимы параметры после ECHO. В случае успеха возвращает текст, который был отправлен клиентом по каналу данных и сообщение об успешной операции по управляющему канал;

– `void list_handler(int fcs, int fds)`: возвращает список всех файлов и каталогов в текущей рабочей директории. Отправка списка происходит по 1024 байта за один цикл отправки. В случае ошибки отправки, клиенту возвращается код с ошибкой по управляющему соединению;

– `void cwd_handler(int fcs, int fds)`: обработка команды CWD, которая меняет текущую рабочую директорию для клиента. Процесс смены текущей директории не является потокобезопасным, поэтому используется статический мьютекс класса;

– `void retr_handler(int fcs, int fds)`: обработка команды RETR, которая позволяет передавать файлы с клиента на сервер. Это происходит путем открытия файла и считывания его по 1024 байта. После всей отправки данных клиенту, происходит сверка количества отправленных байтов с размером файла, если они не совпадают, возвращается код ошибки клиенту, иначе отправляется код успешной передачи файла клиенту.

– `void clear_socket_data(int socket_fd)`: очищает сокет от данных;

– `std::string parse_current_dir()`: возвращает список всех файлов и каталогов по текущей рабочей директории;

– `std::string FTPSpecification::get_client_info(int fcs)`: возвращает строку с информацией об IP-адресе и порте клиента;

3.4 Блок конфигурации сервера

В этом блоке содержится класс `Json_Reader`, который предоставляет статические методы для извлечения информации из файлов в формате JSON. Формат JSON широко используется для настройки приложений и передачи данных через интернет.

Данный класс предоставляет следующие методы:

– `static std::string get_json(const std::string& path)`: извлекает все содержимое JSON-файла и возвращает все данные в виде строки;

– `static std::string find_value(std::string json, const std::string& key)`: возвращает информацию в виде строки, найденную по JSON-ключу;

– `static std::vector<std::string> split_array (std::string array)`: возвращает вектор с JSON-структурами в случае, если наш JSON-файл состоит из массива нескольких других JSON-структур, тогда данная функция разбивает этот массив и записывает каждую структуру раздельно в `vector`.

Пример конфигурации сервера представлен на рисунке 3.4.1.

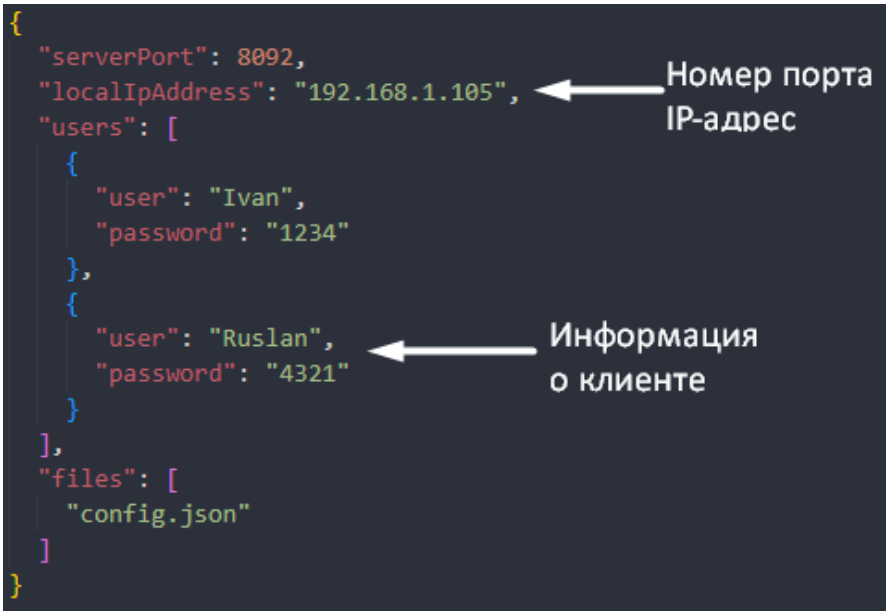

Рисунок 3.4.1 Пример конфигурационного файла

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

Этот раздел посвящен разработке программных модулей многопоточного FTP-сервера. Здесь будут рассмотрены некоторые ключевые функции, которые требовалось реализовать

4.1 Разработка схем алгоритмов

Схемы алгоритмов функций `handlingAccept()` и `parse_current_dir()` представлены в приложениях Г и Д соответственно.

4.2 Разработка алгоритмов

Здесь мы подробно рассмотрим последовательность шагов, которые наш сервер выполняет для обработки команды LIST, отправленной клиентом, а также процесс авторизации клиентов на сервере.

4.2.1 Обработка команды LIST

Класс `FTPSpecification` содержит метод `std::string list_handler(int fcs, int fds)`, который отвечает за обработку команды LIST от клиента к серверу. Данный метод реализует следующий алгоритм обработки:

Шаг 1. Инициализация строковой переменной `result`, которая будет хранить список всех файлов и каталогов текущей директории.

Шаг 2. Вызываем метод `parse_current_dir()`, который возвращает строку со всеми файлами и каталогами в текущей директории. Полученное значение помещаем в `result`

Шаг 3. Проверяем значение `result`. Если `result` равен значению `INTERNAL_SERVER_ERROR`, то сообщаем клиенту об ошибке методом `send()` и завершаем работу нашей функции, иначе продолжаем.

Шаг 4. Инициализируем переменные `bytes_sent` и `data`, которые хранят информацию об отправленных байтах и ранее полученный список в виде константного указателя на массив символ соответственно, значениями 0 и `result`.

Шаг 5. Инициализируем цикл 1.

Шаг 6. Инициализируем переменную `bytes_to_send` наименьшим значением, выбранным из двух значений: 1024 и длина строки `result` – `bytes_sent`. Для этого используем функцию `std::min()`.

Шаг 7. Инициализируем переменную `sent` значением, возвращаемой функцией `send()`, которая в свою очередь возвращает количество отправленных байт по сокету или -1, в случае ошибки. Здесь же и происходит отправка данных клиенту по каналу данных размером не более 1024 байтов.

Шаг 8. Если переменная `sent` равна -1, отправляем клиенту сообщение об ошибке по управляющему каналу функцией `send()` и завершаем выполнение функции, иначе продолжаем.

Шаг 9. К переменной `bytes_sent` прибавляем значение `sent`, то есть отслеживаем количество отправленных байтов.

Шаг 10. Если `bytes_sent` меньше длины строковой переменной `result`, переходим к шагу 6, иначе переходим к шагу 11.

Шаг 11. Завершение цикла 1.

Шаг 12. Отправка по управляющему каналу клиенту сообщение об успешной передаче данных клиенту.

Шаг 13. Вывод в консоль сервера информацию, какой из клиентов отправил команду LIST.

Шаг 14. Конец.

4.2.2 Процесс авторизации клиента на сервере.

Алгоритм авторизации клиента на сервере описывается в методе `authorize()` класса `ServerClient`, который используется в методах класса `ServerCore`. Этот метод обрабатывает последовательность команд USER и PASS, передаваемых клиентом, и пока клиент не прошел авторизацию, ему запрещено вызывать другие команды. Давайте рассмотрим алгоритм данного метода:

Шаг 1. Инициализируем переменные:

– `buffer`: временное хранилище для данных;

– `valread`: количество символов, считанных или записанных в сокет;

– `is_login`: булева переменная для отслеживания ввода имени пользователя клиентом;

– `is_password`: булева переменная для отслеживания ввода пароля клиентом;

– `json`: строковая переменная, содержащая в себе JSON с конфигурационного файла;

– `login_name`: имя авторизированного пользователя.

Шаг 2. Инициализируем цикл 1.

Шаг 3. Производим очистку временного хранилища данных `buffer`.

Шаг 4. Вызываем функцию `get_command_from_client()`, которая определена в классе `ServerClient`, тем самым мы считываем команду, которую отправил клиент и помещаем во временное хранилище данных. Количество считанных символов помещаем в переменную `valread`.

Шаг 5. Если во временном хранилище содержится слово «QUIT» или `valread` равна -1 или `valread` равна 0, то сообщаем клиенту об успешном отсоединении клиента от сервера и вызываем функцию `disconnect()`, которая определена в классе `ServerClient`, чтобы разорвать соединение между клиентом и сервером. Переходи к шагу 31. Иначе если в `buffer` содержится слово «USER», то переходим к шагу 6. Иначе переходим к шагу 15.

Шаг 6. Считываем данные, которые отправляет клиент, вызвав функцию `get_data_from_client()`, которая определена в классе `ServerClient`.

Шаг 7. Если `valread` равна 0 или -1, то отправляем клиенту сообщение о неправильном вводе имени или пароля и очищаем данные, которые хранятся по сокету данных и переходим к шагу 2.

Шаг 8. Инициализируем переменные `json_vector` и `name`, которые будут хранить информацию обо всех пользователях, которые могут подключиться к серверу, и имя пользователя соответственно.

Шаг 9. Инициализируем цикл 2.

Шаг 10. Сохраняем имя пользователя в переменную name, извлекая из i-го JSON’a с помощью функции `find_value()` класса `Json_Reader`.

Шаг 11. Если содержимое name равно содержимому `buffer`, то присваиваем переменной `login_name` значение переменной name и присваиваем булевой переменной `is_login` значение true.

Шаг 12. Если не конец `json_vector`, то переходим к шагу 10.

Шаг 13. Если значение `is_login` равно true, то отправляем серверу сообщение о том, что имя пользователя введено правильно и очищаем содержимое сокета данных, иначе отправляем сообщение о неверном имени пользователя и также очищаемся содержимое сокета данных.

Шаг 14. Если значение `is_login` равно false, то переходим к шагу 3

Шаг 15. Отправляем сообщение клиенту, что сначала необходимо пройти процесс авторизации, и очищаем содержимое сокета данных. Этот шаг происходит в том случае, если пользователь попытался ввести команды, когда он не авторизован.

Шаг 16. Инициализируем цикл 3.

Шаг 17. Очищаем содержимое `buffer`

Шаг 18. Считываем команду от клиента, вызвав функцию `get_command_from_client()` класса `ServerClient`, и помещаем количество считанных символов в переменную `valread`.

Шаг 19. Если содержимое buffer равно «QUIT» или значение `valread` равно -1 или 0, то переходим к шагу 20, иначе если содержимое `buffer` равно «PASS», то переходим к шагу 21, иначе переходим к шагу 29.

Шаг 20. Отправляем сообщение клиенту об успешном разрыве соединения между клиентом и сервером, вызываем функцию `disconnect()` класса `ServerClient`. Переходим к шагу 31.

Шаг 21. Инициализируем переменные `json_vector`, куда помещаем всех пользователей из нашего JSON файла с помощью функции `split_array()` класса `Json_Reader`, `password` и `name`.

Шаг 22. Инициализируем цикл 4.

Шаг 23. Вызываем функцию `find_value()` класса `Json_Reader` и помещаем пароль переменную `password`.

Шаг 24. Вызываем функцию `find_value()` класса `Json_Reader` и помещаем имя пользователя в переменную `name`.

Шаг 25. Если `password` равен содержимому `buffer` и если `login_name` равен содержимому `name`, то присваиваем булевой переменной `is_password` значение true.

Шаг 26. Если не конец `json_vector`, то переходим к шагу 23.

Шаг 27. Если `is_password` равен true, то отправляем сообщение клиенту, что пароль введен верно, и очищаем содержимое сокета данных, иначе отправляем сообщение, что неверный пароль, и очищаем содержимое сокета данных.

Шаг 28. Если `is_password` равен false, то переходим к шагу 17.

Шаг 29. Отправляем сообщение, что необходимо пройти процесс авторизации и очищаем содержимое сокета данных. Переходим к шагу 17.

Шаг 30. Если `is_password` и `is_login` равны true, то выводим в консоль сообщение, что авторизация произошла успешно.

Шаг 31. Конец.

4.3 Структурная схема

Структурная схема, иллюстрирующая все перечисленные блоки и связи между ними представлена в приложении Б.

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

В данном разделе представлена методика испытаний программного средства. Проведено некоторое тестирование программы и её функционала.

5.1 Тесты POSIX-совместимости

Программа прошла тестирование на различных POSIX-совместимых системах, что подтверждает ее работоспособность и совместимость с этим стандартом.

5.2 Тесты функциональности

Тестирование функциональности необходимо для устранения возможных ошибок в исходном коде, которые вызывают некорректную обработку информации, либо критические ошибки.

5.2.1 Тестирование команд

Таблица 5.2.1 – Тестирование команд

Тест	Ожидаемый результат	Полученный результат
При вводе со стороны клиента неверного имени пользователя	Сервер отправляет сообщение о неверном имени пользователя	Сервер отправляет сообщение о неверном имени пользователя
При вводе со стороны клиента неверного пароля	Сервер отправляет сообщение о неверном пароле	Сервер отправляет сообщение о неверном пароле
Попытка ввода команды без авторизации	Сервер отправляет сообщение «332: Need account for login.»	Сервер отправляет сообщение «332: Need account for login.»
Ввод правильного имени пользователя	Сервер отправляет сообщение «331: User name okay, need password.»	Сервер отправляет сообщение «331: User name okay, need password.»
Ввод правильного пароля	Сервер отправляет сообщение «230: User logged in, proceed. Logged out if appropriate»	Сервер отправляет сообщение «230: User logged in, proceed. Logged out if appropriate»

Продолжение таблицы 5.2.1

Ввод команды ECHO без аргументов	Сервер отправляет сообщение «501: Syntax error in parameters or arguments»	Сервер отправляет сообщение «501: Syntax error in parameters or arguments»
Ввод команды LIST	Сервер отправляет сообщение «226: List transfer done» и клиент выводим содержимое каталога	Сервер отправляет сообщение «226: List transfer done» и клиент выводим содержимое каталога
Ввод команды RETR с существующим файлом	Сервер отправляет сообщение «226: Successful download» и клиент загружает файл	Сервер отправляет сообщение «226: Successful download» и клиент загружает файл
Ввод команды RETR с несуществующим файлом	Сервер отправляет сообщение «550: File unavailable»	Сервер отправляет сообщение «550: File unavailable»
Ввод команды CWD с существующим каталогом	Сервер отправляет сообщение «250: Successful change» и меняет текущий рабочий каталог	Сервер отправляет сообщение «250: Successful change» и меняет текущий рабочий каталог
Ввод команды CWD с несуществующим каталогом	Сервер отправляет сообщение «404: No such directory»	Сервер отправляет сообщение «404: No such directory»
Ввод команды QUIT	Происходит разрыв соединения между сервером и клиентом	Происходит разрыв соединения между сервером и клиентом

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

При первом запуске программы необходимо открыть терминал и прописать команду cd /корневой/путь/к/программе для перехода в каталог с исполняемый файлов FTPServer. В другом терминале необходимо прописать путь к исполняемому файлу клиента. Каталоги должны располагаться, как продемонстрировано на рисунке 6.1.

Рисунок 6.1 Расположение каталогов для корректной работы сервера

В clients_downloads содержатся все файлы, которые были загружены с сервера клиентом. Конфигурация сервера находится в файле config.json, которая располагается по пути server_core/resources. Если необходимо изменить расположение и размещение всех файлов, необходимо поменять в исходном коде проекта пути к нужным нам файлам, после этого скомпилировать все заново.

6.1 Авторизация на сервере.

Для авторизации на сервере необходимо ввести имя пользователя. Скриншот выполнение данной команды представлен на рисунке 6.1.1.

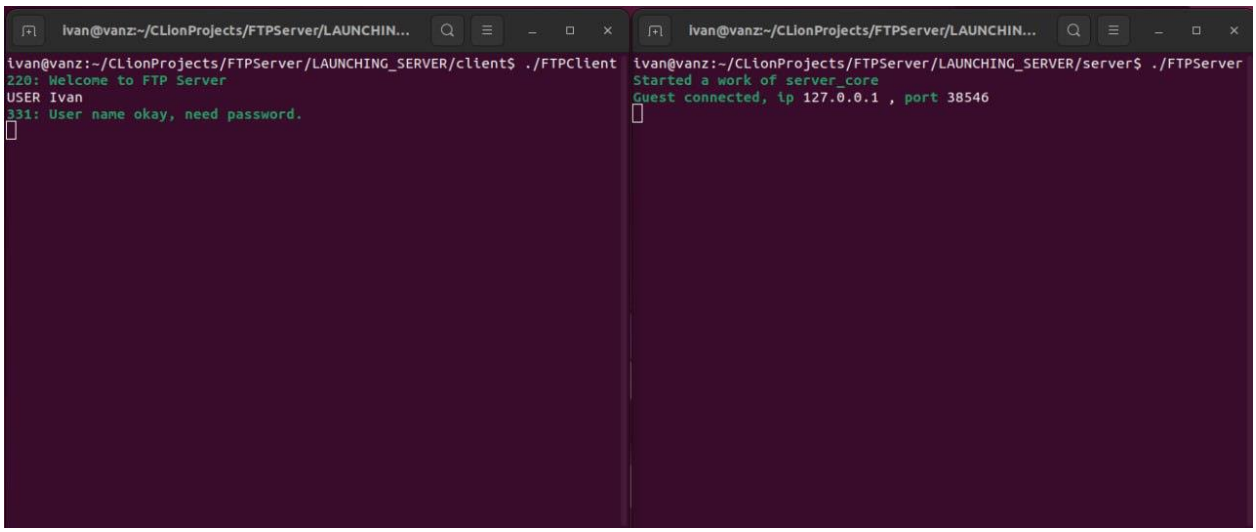

Рисунок 6.1.1 – Окно терминала с командой на ввод имени пользователя

После ввода имени пользователя, необходимо ввести пароль. Скриншот выполнения этого действия представлен на рисунке 6.1.2

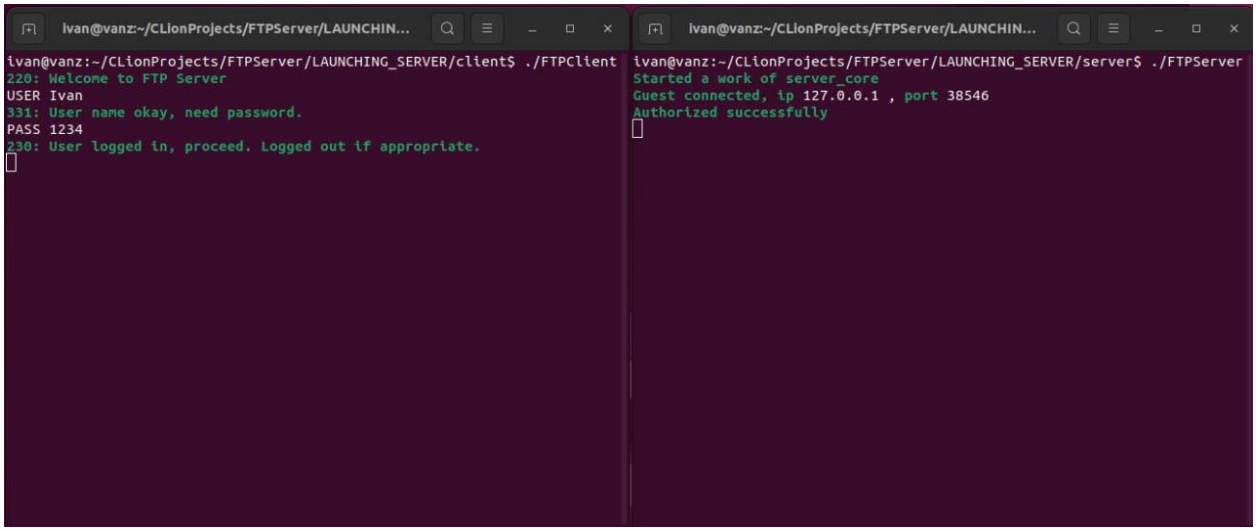

Рисунок 6.1.2 – Окно терминала с командой на ввод пароля

После выполнения авторизации пользователя на сервере, нам доступны следующие команды: ECHO, LIST, CWD, RETR, QUIT. Эти команды служат для отправки и получения сообщения с клиента на сервер и обратно, отображения файлов и папок в текущем каталоге, переход в другой каталог, загрузка файлов с сервера и разрыв соединения между серверов и клиентом соответственно.

6.2 Выполнение проверки состояния соединения с сервером

Чтобы проверить состояние соединения с сервером, нам необходимо выполнить команду ECHO «текст», которая отправит введенный текст на сервер, а сервер в ответ отправит обратно успешно ли произошла обработка этой команды и сам текст. Выполнение команды ECHO представлен на рисунке 6.2.1.

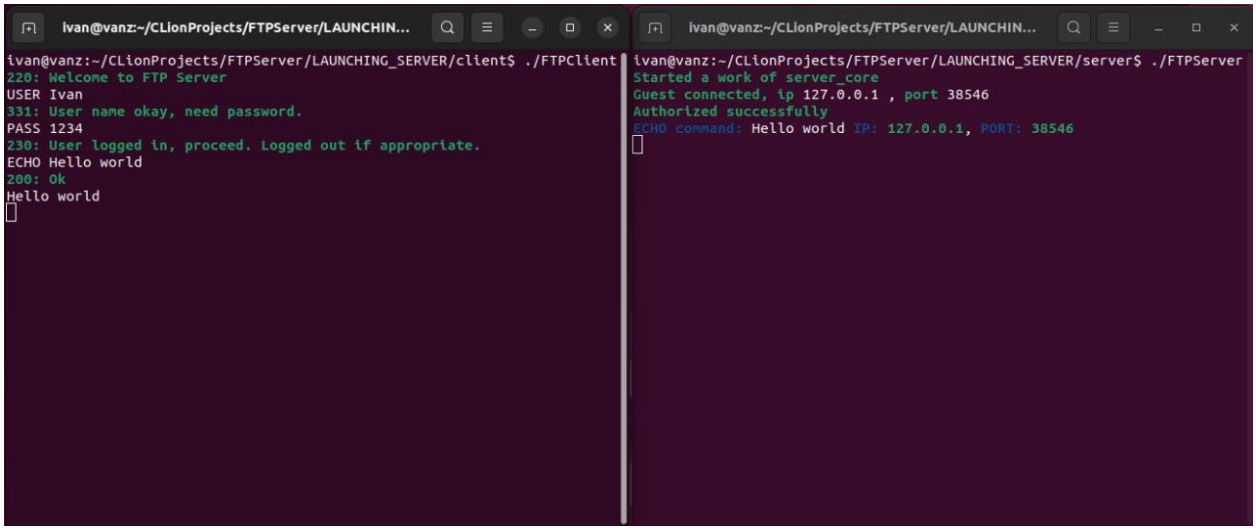

Рисунок 6.2.1 – Окно терминала с выполнением команды ECHO

Если не передать никаких аргументов после того, как написать ECHO, то сервер вернет нам сообщение об ошибке. Это представлено на рисунке 6.2.2.

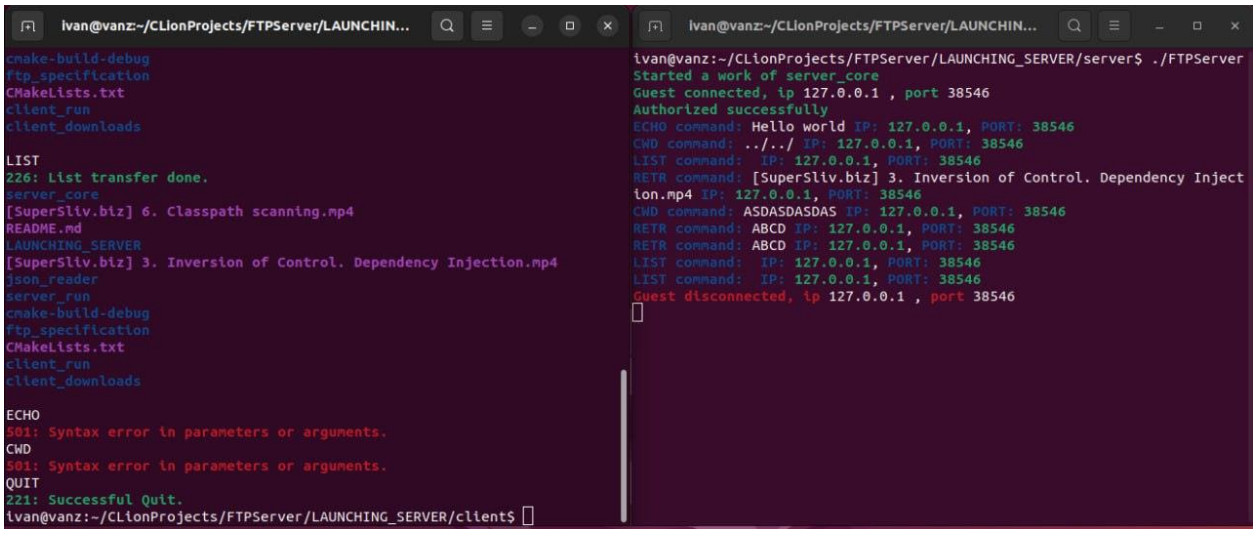

Рисунок 6.2.2 – Окно терминала с неправильным выполнением команды ECHO

6.3 Изменение текущего рабочего каталога

Команда CWD позволяет изменить текущий рабочий каталог. Чтобы изменить каталог необходимо в терминале прописать «CWD путь/к/каталогу». Пример выполнения данной команды представлен на рисунке 6.3.1

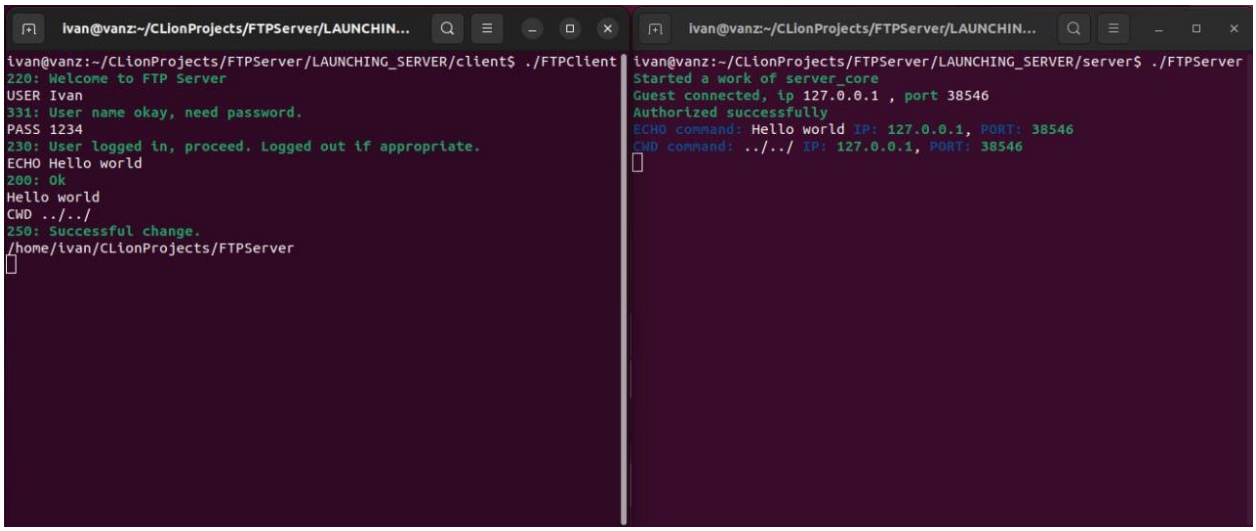

Рисунок 6.3.1 – Окно терминала с выполнением команды CWD

6.4 Получение содержимого текущего рабочего каталога

Для того, чтобы получить содержимое текущего рабочего каталога, необходимо прописать команду LIST. Пример выполнения этой команды представлен на рисунке 6.4.1.

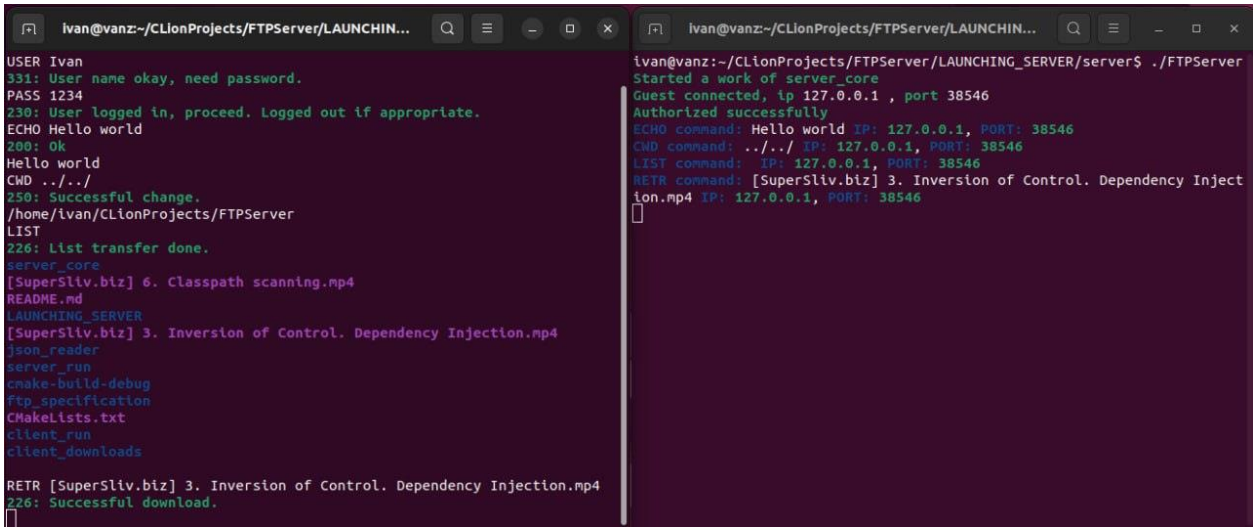

Рисунок 6.4.1 – Окно терминала с выполнением команды LIST

6.5 Загрузка файлов с сервера

Выполнение загрузки файлов с сервера производится путем ввода команды RETR. Если мы хотим загрузить файл, то нам необходимо в терминале прописать «RETR имя_файла». Все загруженные файлы будут загружаться в каталог clients_downloads, если не было переопределения его в исходном коде проекта. Пример выполнения этой команды представлен на рисунке 6.5.1.

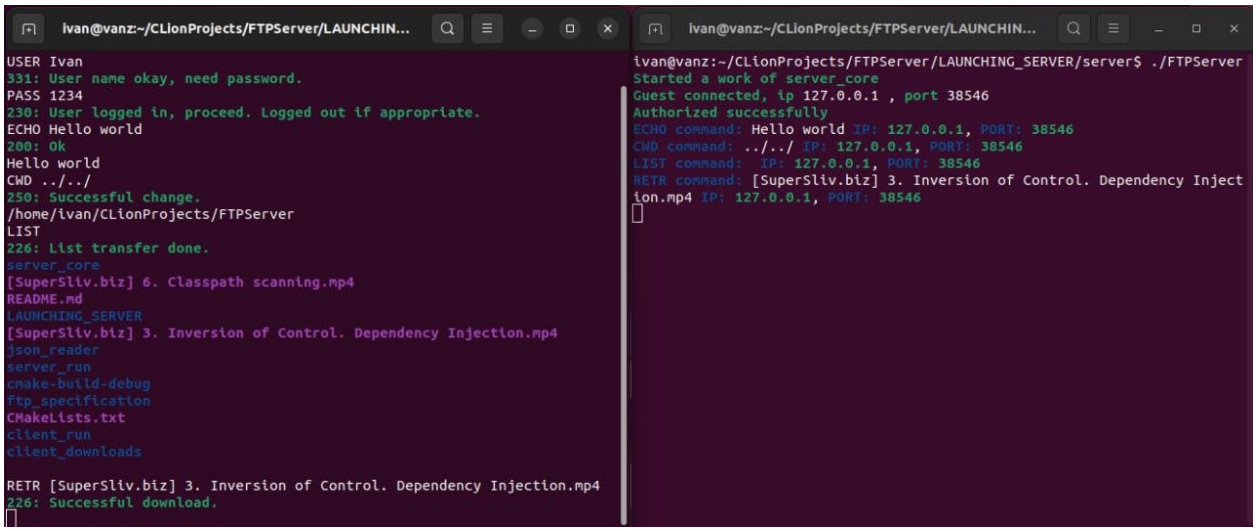

Рисунок 6.5.1 – Окно терминала с выполнением команды RETR

Этот файл был загружен в каталог client_downloads. Результат выполнения команды RETR продемонстрирован на рисунке 6.5.2.

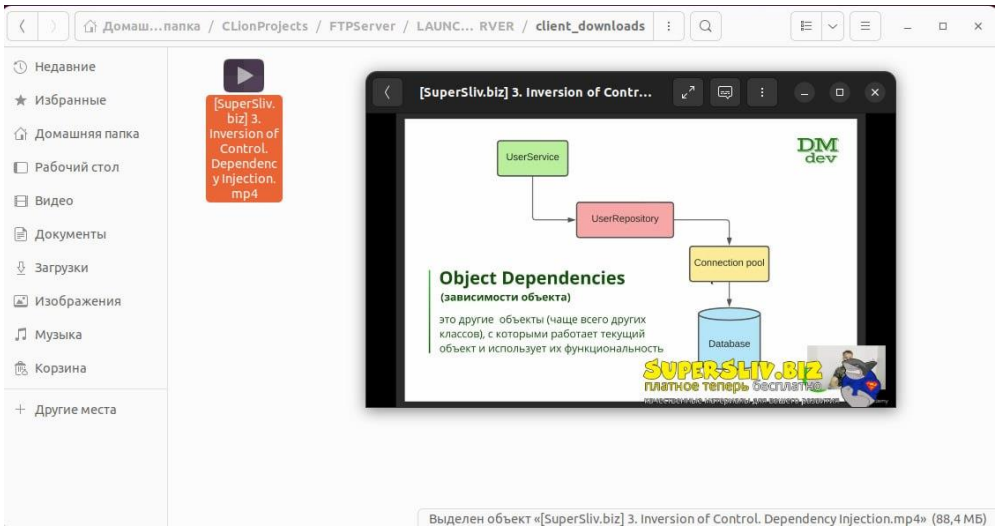

Рисунок 6.5.2 – Результат выполнения команды RETR

ЗАКЛЮЧЕНИЕ

В данной курсовой работе был разработан многопоточный FTP-сервер на языке C++ для операционной системы Linux. Сервер использует сокеты для связи с клиентами, спецификацию FTP для обработки команд и многопоточность для одновременного обслуживания нескольких клиентов.

В ходе работы были рассмотрены следующие темы:

1 Спецификация FTP: изучены основные команды FTP, структура сообщений и формат данных.

2 Сокеты: изучены основы работы с сокетами, включая создание сокетов, связывание сокетов с адресами, прослушивание сокетов и обмен данными.

3 Многопоточность: изучены основы многопоточности, включая создание потоков, синхронизацию потоков и обмен данными между потоками.

В результате работы был разработан многопоточный FTP-сервер, который поддерживает основные команды FTP, такие как LIST, RETR, CWD, CDUP, QUIT. Сервер способен одновременно обслуживать несколько клиентов и использует сокеты для связи с ними. Он также реализует спецификацию FTP для обработки команд и данных и использует многопоточность для одновременного обслуживания нескольких клиентов.

В дальнейшем можно расширить функциональность сервера, добавив поддержку следующих возможностей: шифрование данных, поддержка сторонних клиентов и дополнительных команд FTP. Также можно улучшить производительность сервера, оптимизировав код и используя более эффективные алгоритмы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] OpenNET [Электронный ресурс]. – Sockets. – Режим доступа: https://www.opennet.ru/man.shtml?topic=socket&category=2&russian=2 – Дата доступа: 22.03.2024

[2] IETF Datatracker [Электронный ресурс]. – FTP Specification RFC 959. – Режим доступа: https://datatracker.ietf.org/doc/html/rfc959 – Дата доступа: 22.03.2024

[3] DevDocs [Электронный ресурс]. – C++ documentation – Режим доступа: https://devdocs.io/cpp/ – Дата доступа: 22.03.2024

[4] Linux Kernel Docs [Электронный ресурс]. – The Linux Kernel documentation. – Режим доступа: https://docs.kernel.org – Дата доступа: 22.03.2024

ПРИЛОЖЕНИЕ А
(Обязательное)

Ведомость документов

ПРИЛОЖЕНИЕ Б
(Обязательное)

Схема структурная

ПРИЛОЖЕНИЕ В
(Обязательное)

Диаграмма классов

ПРИЛОЖЕНИЕ Г

(Обязательное)

Схема алгоритма функции handlingAccept()

ПРИЛОЖЕНИЕ Д
(Обязательное)

Схема алгоритма функции parse_current_dir()

ПРИЛОЖЕНИЕ Е
(Обязательное)

Код программы