Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Реализация статического веб-сервера для отдачи контента с диска»

Студент ИУ7-73Б _____ В. Марченко
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ _____
(Подпись, дата) (И. О. Фамилия)

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка X с., X рис., 3 табл., X источн., 1 прил.
КОМПЬЮТЕРНЫЕ СЕТИ, СТАТИЧЕСКИЙ СЕРВЕР, ВЕБ-СЕРВЕР, HTTP, NGINX, НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ, APACHEBENCH

Объектом разработки является статический веб-сервер для отдачи контента с диска.

Объектом исследования является время обработки запросов статическим веб-сервером.

Цель работы: реализация статического веб-сервера для отдачи контента с диска.

В результате выполнения работы был реализован статический веб-сервер для отдачи контента с диска, а также проведено нагрузочное тестирование реализованного сервера.

В ходе проведения исследования было установлено, что сервер nginx обрабатывает запросы в среднем в 2 раза быстрее, чем реализованный статический веб-сервер.

Область применения результатов — дальнейшее развитие реализованного статического веб-сервера для поддержки отдачи контента различных MIME-типов.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей расчетно-пояснительной записке к курсовой работе применяют следующие сокращения и обозначения:

ОС	Операционная система
ПО	Программное обеспечение
API	Программный интерфейс приложения (англ. Application Programming Interface)
CSS	Каскадные таблицы стилей (англ. Cascading Style Sheets)
HTML	Язык гипертекстовой разметки (англ. HyperText Markup Language)
HTTP	Протокол передачи гипертекста (англ. HyperText Transfer Protocol)
URL	Единообразный указатель местонахождения ресурса (англ. Uniform Resource Locator)

ВВЕДЕНИЕ

Понятие «веб-сервер» может относиться как к аппаратному, так и к программному обеспечению. Или даже к обеим частям, работающим совместно. С точки зрения аппаратного обеспечения, веб-сервер — это компьютер, который хранит файлы сайта (HTML-документы, CSS-стили, JavaScript-файлы, картинки и пр.) и доставляет их на устройство конечного пользователя (веб-браузер и т. д.). Он подключен к сети Интернет и может быть доступен через доменное имя. С точки зрения программного обеспечения, веб-сервер включает в себя несколько компонентов, которые контролируют доступ веб-пользователей к размещенным на сервере файлам, как минимум — это HTTP-сервер. HTTP-сервер — это часть программного обеспечения, которая понимает URL-адреса (веб-адреса) и HTTP (протокол, который браузер использует для просмотра веб-страниц) [1].

Целью курсовой работы является реализация статического веб-сервера для отдачи контента с диска.

Задачи курсовой работы:

1) изучить понятие статического веб-сервера;
2) спроектировать архитектуру статического сервера;
3) реализовать спроектированный статический веб-сервер для отдачи контента с диска;
4) провести нагрузочное тестирование при помощи ApacheBench и сравнить реализованный сервер с сервером nginx.

1 Аналитическая часть

1.1 Требования к статическому веб-серверу

По заданию на курсовую работу к статическому веб-серверу предъявляются следующие требования.

1. Поддержка GET и HEAD запросов (поддержка статусов 200, 403, 404).
2. Ответ на неподдерживаемые запросы статусом 405.
3. Выставление Content-Type в зависимости от типа файла (поддержка .html, .css, .js, .png, .jpg, .jpeg, .swf, .gif).
4. Корректная передача файлов размером в 100 МБ.
5. Сервер по умолчанию должен возвращать HTML-страницу на выбранную тему с CSS-стилем.
6. Учесть минимальные требования к безопасности статических серверов (предусмотреть ошибку в случае, если адрес будет выходить за корневую директорию сервера).
7. Реализовать логгер.
8. Использовать язык C без сторонних библиотек.
9. Реализовать архитектуру prefork + pselect().
10. Статический веб-сервер должен работать стабильно.

1.2 Статический веб-сервер

На самом базовом уровне, когда браузеру нужен файл, размещенный на веб-сервере, браузер запрашивает его через HTTP-протокол. Когда запрос достигает нужного веб-сервера (аппаратное обеспечение), сервер HTTP (программное обеспечение) принимает запрос, находит запрашиваемый документ (если нет, то сообщает об ошибке 404) и отправляет обратно, также через HTTP. На рисунке 1.1 показана схема взаимодействия веб-сервера и браузера [1].

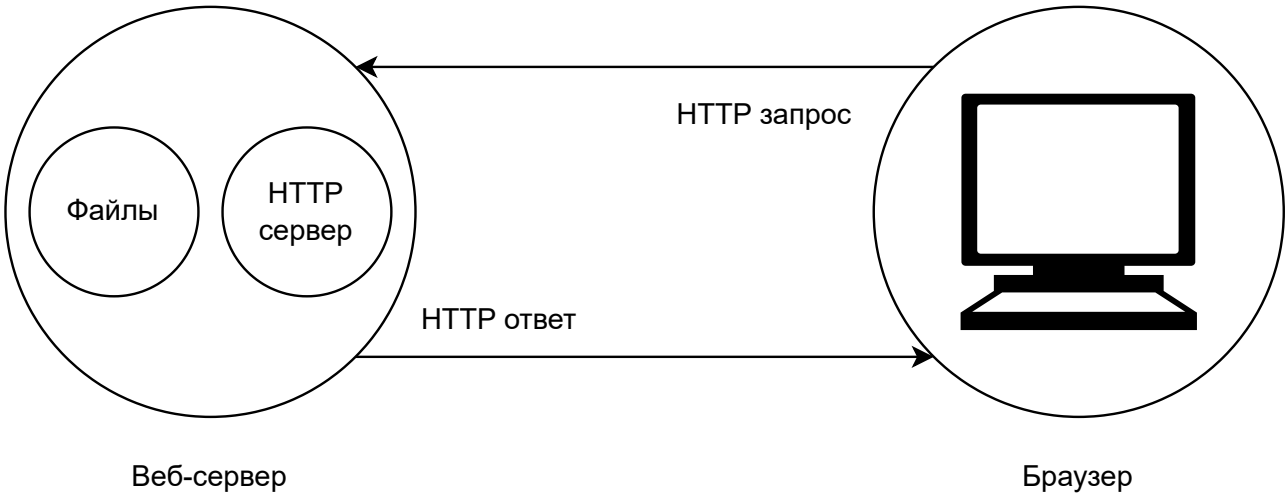

Рисунок 1.1 – Схема взаимодействия веб-сервера и браузера [1]

Статический веб-сервер, или стек, состоит из компьютера (аппаратное обеспечение) с сервером HTTP (программное обеспечение). Это называют «статикой», потому что сервер посылает размещенные файлы в браузер «как есть» [1].

Динамический веб-сервер состоит из статического веб-сервера и дополнительного программного обеспечения, чаще всего сервера приложения и базы данных. Его называют «динамическим», потому что сервер приложений изменяет исходные файлы перед отправкой в браузер по HTTP [1].

Например, для получения итоговой страницы, которую пользователь просматривает в браузере, сервер приложений может заполнить HTML-шаблон данными из базы данных. Такие сайты, как MDN или Википедия, состоят из тысяч веб-страниц, но они не являются реальными HTML документами — лишь несколько HTML-шаблонов и гигантские базы данных. Эта структура упрощает и ускоряет сопровождение веб-приложений и доставку контента [1].

1.3 Структура HTTP-сообщений

HTTP-сообщения — это обмен данными между сервером и клиентом. Есть два типа сообщений: запросы, отправляемые клиентом, чтобы инициировать реакцию со стороны сервера, и ответы от сервера [2].

Сообщения HTTP состоят из текстовой информации в кодировке ASCII, записанной в несколько строк. В HTTP/1.1 и более ранних версиях они пересылались в качестве обычного текста. В HTTP/2 текстовое сообщение разделяется на фреймы, что позволяет выполнить оптимизацию и повысить

производительность [2].

Веб разработчики не создают текстовые сообщения HTTP самостоятельно — это делает программа, браузер, прокси или веб-сервер. Они обеспечивают создание HTTP сообщений через конфигурационные файлы (для прокси и серверов), API (для браузеров) или другие интерфейсы [2].

HTTP запросы и ответы имеют похожую структуру. Они состоят из следующих элементов [2].

1. Стартовая строка, описывающая запрос, или статус (успех или сбой). Это всегда одна строка.

2. Произвольный набор HTTP заголовков, определяющий запрос или описывающий тело сообщения.

3. Пустая строка, указывающая, что вся метаинформация отправлена.

4. Произвольное тело, содержащее пересылаемые с запросом данные (например, содержимое HTML-формы) или отправляемый в ответ документ. Наличие тела и его размер определяется стартовой строкой и заголовками HTTP.

Стартовую строку вместе с заголовками сообщения HTTP называют головой запроса, а его данные — телом [2].

1.4 Заголовки HTTP

Заголовки HTTP позволяют клиенту и серверу отправлять дополнительную информацию с HTTP запросом или ответом. В HTTP-заголовке содержится нечувствительное к регистру название, а затем после (:) непосредственно значение. Пробелы перед значением игнорируются [3].

По заданию на курсовую работу требуется выставлять Content-Type в зависимости от типа файла. Заголовок-сущность Content-Type используется для того, чтобы определить MIME-тип ресурса. В ответах сервера заголовок Content-Type сообщает клиенту, какой будет тип передаваемого контента [4].

Для следующих файлов используется такой Content-Type:

1) .html — text/html;

2) .css — text/css;

3) .js — text/javascript;

4) .png — image/png;

5) .jpg — image/jpg;

6) .jpeg — image/jpeg;

7) .jpeg — image/gif;

8) .swf — application/x-shockwave-flash.

1.5 Коды HTTP ответов и методы HTTP запросов

Код ответа (состояния) HTTP показывает, был ли успешно выполнен определенный HTTP запрос. Коды сгруппированы в 5 классов [5]:

1) информационные 100–199;

2) успешные 200–299;

3) перенаправления 300–399;

4) клиентские ошибки 400–499;

5) серверные ошибки 500–599.

Коды состояния определены в RFC 7231 [6].

Исходя из требований к курсовой работе нужно использовать следующие коды ответа:

1) 200 — OK;

2) 403 — Forbidden;

3) 404 — Not Found;

4) 405 — Method Not Allowed.

HTTP определяет множество методов запроса, которые указывают, какое желаемое действие выполнится для данного ресурса. Несмотря на то, что их названия могут быть существительными, эти методы запроса иногда

называются HTTP глаголами. Каждый реализует свою семантику, но каждая группа команд разделяет общие свойства: так, методы могут быть безопасными, идемпотентными или кешируемыми. Существует следующие HTTP глаголы [7].

1. Метод GET запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.
2. HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа.
3. POST используется для отправки сущностей к определенному ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.
4. PUT заменяет все текущие представления ресурса данными запроса.
5. DELETE удаляет указанный ресурс.
6. CONNECT устанавливает «туннель» к серверу, определенному по ресурсу.
7. OPTIONS используется для описания параметров соединения с ресурсом.
8. TRACE выполняет вызов возвращаемого тестового сообщения с ресурса.
9. PATCH используется для частичного изменения ресурса.

По заданию на курсовую работу нужно обрабатывать только GET и HEAD методы. На все остальные нужно отвечать кодом состояния 405 Method Not Allowed.

1.6 Сокеты

Сокеты — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения [8].

Каждый процесс может создать слушающий сокет (серверный сокет) и привязать его к какому-нибудь порту операционной системы. Слушающий

процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и т. д. [8]

Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (файл сокета) по заданному пути, через который смогут общаться любые локальные процессы путем чтения/записи из него. Сокеты типа INET доступны из сети и требуют выделения номера порта [8].

Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером. Все сокеты обычно ориентированы на применение датаграмм, но их точные характеристики зависят от интерфейса, обеспечиваемого протоколом [8].

Обмен между сокетами происходит по схеме, изображенной на рисунке 1.2.

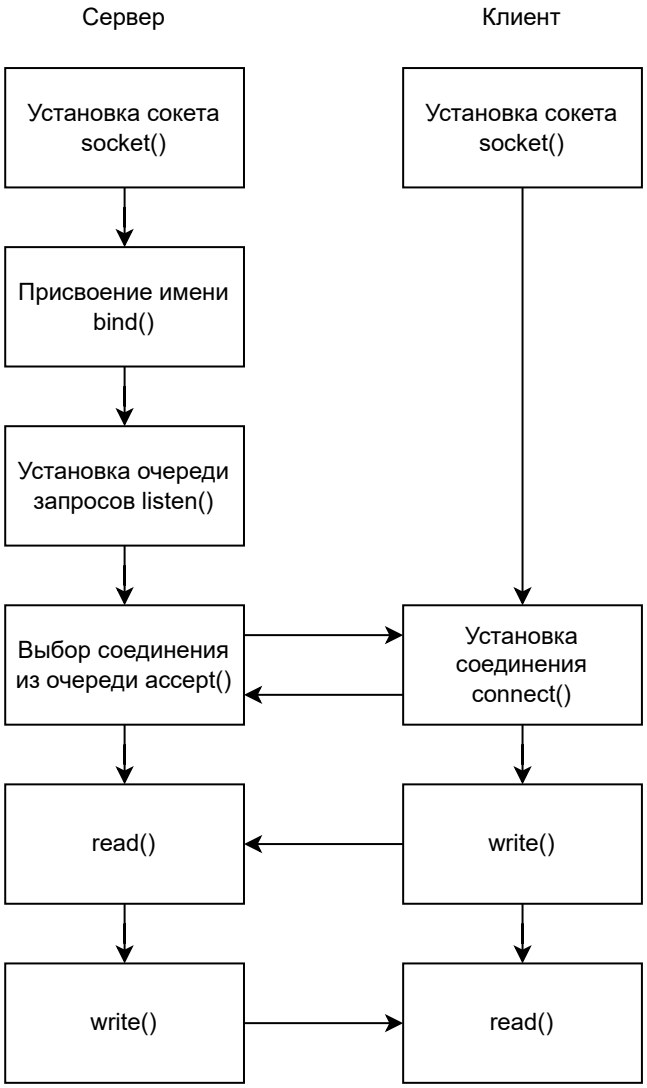

Рисунок 1.2 – Схема обмена между сокетами [8]

1.7 Диаграмма вариантов использования

На рисунке 1.3 показана диаграмма вариантов использования для пользователей статического веб-сервера.

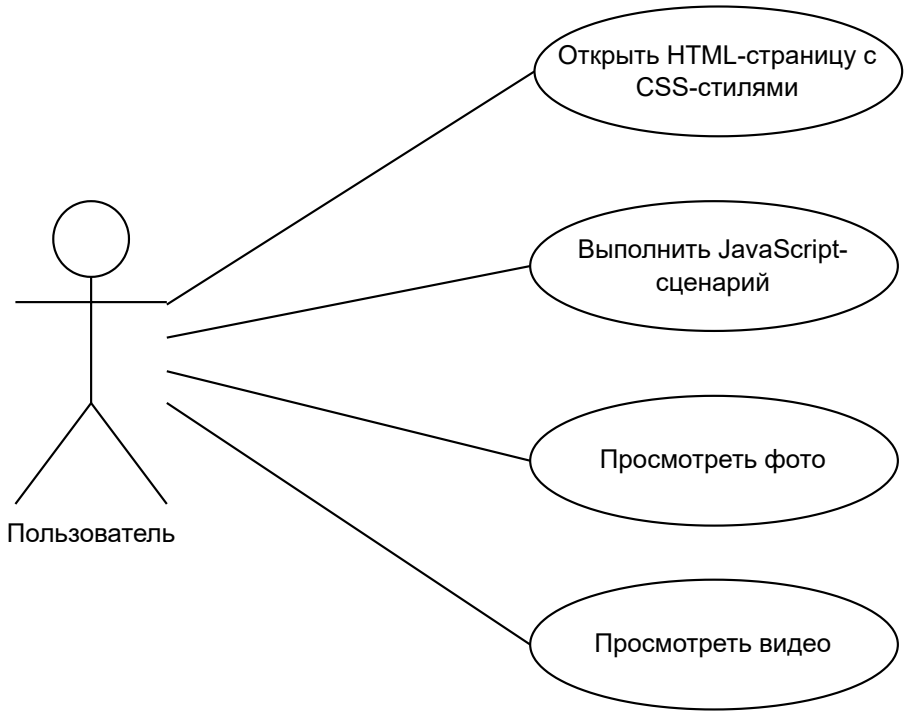

Рисунок 1.3 – Взаимодействие пользователя со статическим веб-сервером

Вывод из аналитической части

В ходе выполнения аналитической части курсовой работы был проведен анализ предметной области — изучено понятие статического веб-сервера, сформулированы требования к серверу и описаны пользовательские сценарии в виде диаграммы вариантов использования.

2 Конструкторская часть

2.1 Описание инициализации сервера

На рисунке 2.1 показана схема инициализации сервера.

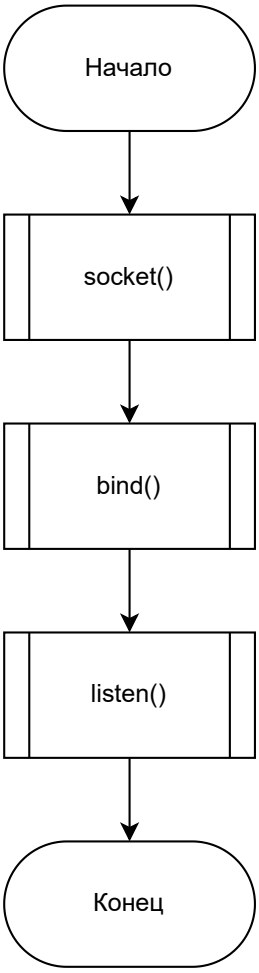

Рисунок 2.1 – Схема инициализации сервера

2.2 Описание проверки готовности клиентских сокетов

На рисунке 2.2 показана схема проверки готовности клиентских сокетов.

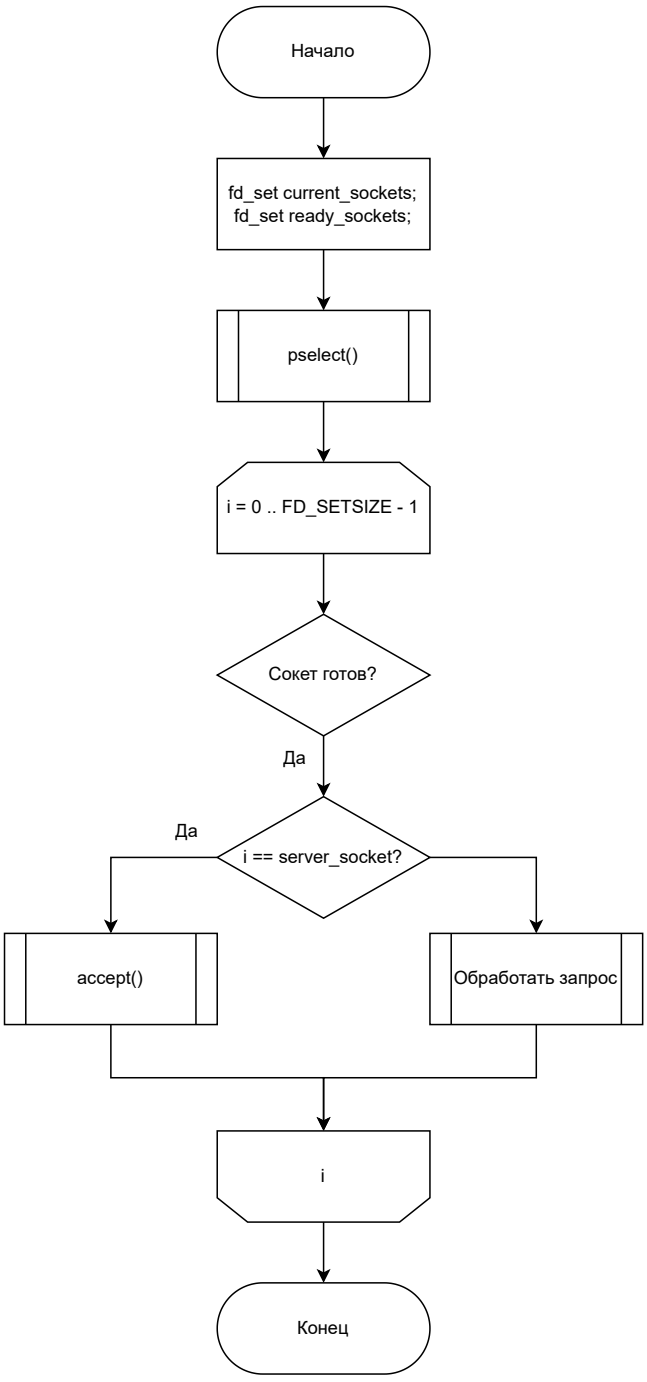

Рисунок 2.2 – Схема проверки готовности клиентских сокетов

2.3 Описание обработки HTTP запросов

На рисунке 2.3 показана схема обработки HTTP запросов.

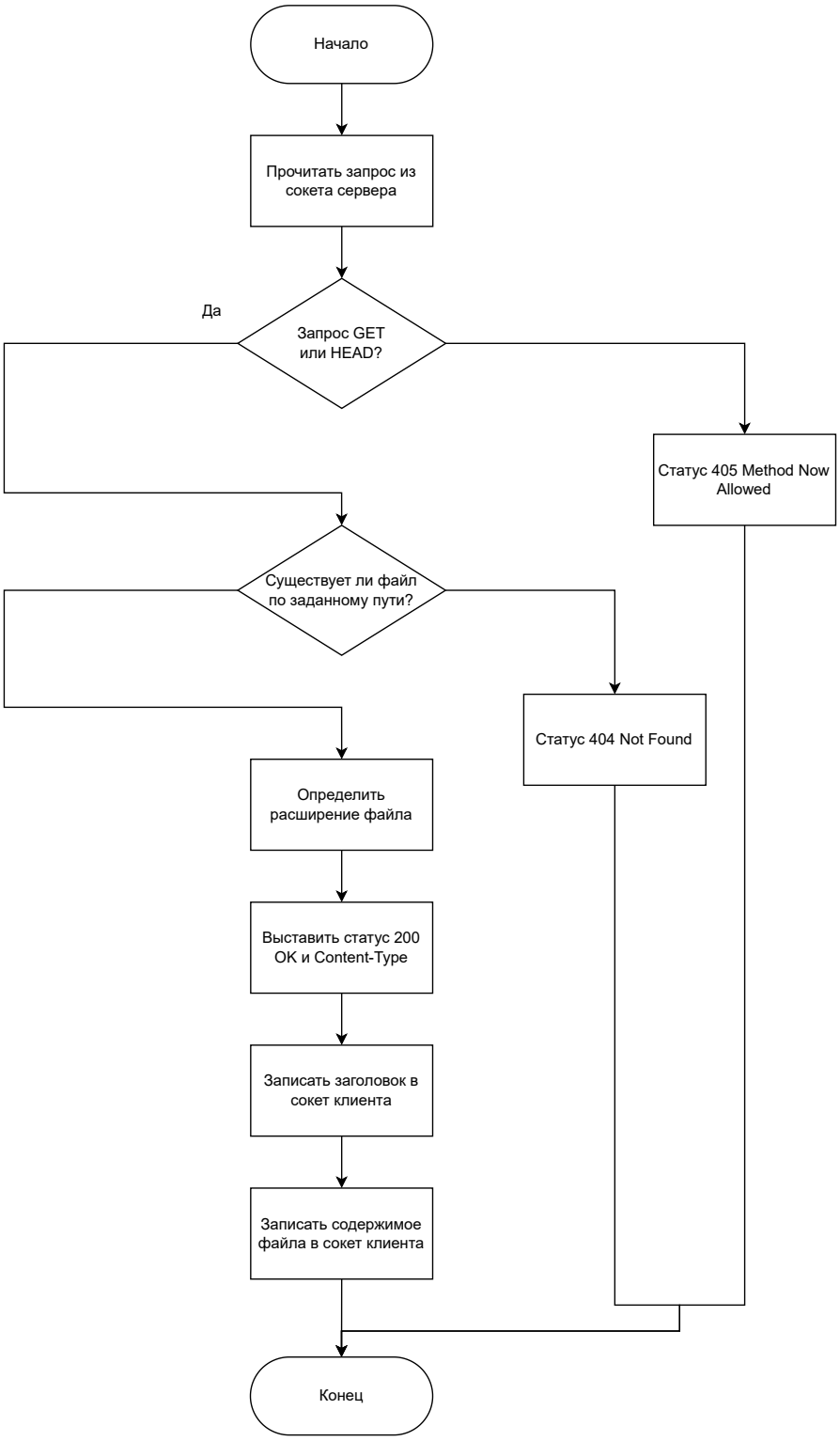

Рисунок 2.3 – Схема обработки HTTP запросов

Вывод из конструкторской части

В ходе выполнения конструкторской части куросовой работы были описаны в виде схем алгоритмов инициализация сервера, проверка готовности клиентских сокетов и процесс обработки пользовательских HTTP запросов.

3 Технологическая часть

3.1 Средства реализации

Для реализации статического веб-сервера для отдачи контента с диска использовался язык C (требование к курсовой работе). Сервер написан под операционные системы, работающие на базе ядра Linux. Для реализации архитектуры prefork + pselect() использовались системные вызовы fork() и pselect(). Для инициализации сервера использовались системные вызовы socket(), bind(), listen() и accept(). Для чтения запросов и записи ответов использовались системные вызовы recvfrom() и sendto(). Для подготовки тестовых данных (контента для отдачи с диска) были использованы HTML, CSS и JavaScript.

3.2 Реализация инициализации сервера

В листинге 3.1 показана реализация инициализации сервера. Сначала выполняется системный вызов socket(), и создается конечная точка соединения. Затем заполняются поля структуры struct sockaddr_in. Далее с помощью bind() к сокету server_socket привязывается локальный адрес server_address. После этого с использованием системного вызова listen() выражается готовность сервера принимать входящие соединения.

Листинг 3.1 – Реализация инициализации сервера

```
int init_server(logger_t logger, int *server_socket,
                struct sockaddr_in *server_address)
{
    if ((*server_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        log_msg(logger, "Error: server socket can't be
           created.\n");
        return -1;
    }
    int optval = 1;
    setsockopt(*server_socket, SOL_SOCKET, SO_REUSEPORT, &optval,
               sizeof(optval));
    server_address->sin_family = AF_INET;
    server_address->sin_addr.s_addr = INADDR_ANY;
    server_address->sin_port = htons(PORT);
    if (bind(*server_socket, (struct sockaddr *)server_address,
```

```
        sizeof(*server_address)) == -1)
    {
        log_msg(logger, "Error: can't bind server socket.\n");
        return -2;
    }
    if (listen(*server_socket, MAX_CONNECTIONS) == -1)
    {
        log_msg(logger, "Error: server can't listen.\n");
        return -3;
    }
    return 0;
}
```

3.3 Реализация выставления Content-Type

В листинге 3.2 показана реализация выставления Content-Type в зависимости от типа файла.

Листинг 3.2 – Реализация выставления Content-Type

```
if (strcmp(extension, "html") == 0)
{
    snprintf(response_buffer, 1025, response_header,
                "200 OK", "text", "html");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket,
                    client_address,
                    client_address_len);
}
else if (strcmp(extension, "js") == 0)
{
    snprintf(response_buffer, 1025, response_header,
                "200 OK", "text", "javascript");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket, client_address,
           client_address_len);
}
```

```
else if (strcmp(extension, "css") == 0)
{
    snprintf(response_buffer, 1025, response_header,
                "200 OK", "text", "css");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket, client_address,
           client_address_len);
}
else if (strcmp(extension, "png") == 0 || strcmp(extension,
   "jpg") == 0
        || strcmp(extension, "jpg") == 0 || strcmp(extension,
           "jpeg") == 0
        || strcmp(extension, "gif") == 0)
{
    snprintf(response_buffer, 1025, response_header,
                "200 OK", "image", extension);
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket, client_address,
           client_address_len);
}
else if (strcmp(extension, "swf") == 0)
{
    snprintf(response_buffer, 1025, response_header,
                "200 OK", "application",
                "x-shockwave-flash\r\nContent-Disposition:
                   inline;");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket, client_address,
           client_address_len);
}
```

3.4 Реализация записи содержимого файла в сокет клиента

В листинге 3.3 показана реализация записи содержимого файла в сокет клиента.

Листинг 3.3 – Реализация записи содержимого файла в сокет клиента

```
void send_data(const char *const filename, const int fd,
               const struct sockaddr *to, socklen_t tolen)
{
    char buffer[SIZE + 1] = "";
    FILE *f = fopen(filename, "rb");
    ssize_t rlen = 0, wlen = 0;
    while ((wlen != -1) && ((rlen = fread(buffer, sizeof(char),
       SIZE, f)) > 0))
        wlen = sendto(fd, buffer, rlen, 0, to, tolen);
    fclose(f);
}
```

3.5 Реализация логгера

В листинге 3.4 показана реализация функций для использования логгера.

Листинг 3.4 – Реализация логгера

```
typedef FILE* logger_t;

logger_t init_logger(const char* const filename)
{
    return fopen(filename, "a+");
}

void log_msg(const logger_t logger, const char* const msg)
{
    char template[32 + 1] = "Datetime: %s";
    char header[128 + 1] = "";
    time_t t;
    time(&t);
    snprintf(header, 128, template, ctime(&t));
    fwrite(header, sizeof(char), strlen(header), logger);
    size_t len = strlen(msg);
    fwrite(msg, sizeof(char), len, logger);
}
```

```
void exit_logger(const logger_t logger)
{
    if (logger != NULL)
        fclose(logger);
}
```

Вывод из технологической части

В ходе выполнения технологической части курсовой работы были выбраны средства реализации программного обеспечения и написан исходный код статического веб-сервера для отдачи контента с диска.

4 Исследовательская часть

4.1 Технические характеристики устройства

Технические характеристики устройства, на котором было проведено нагрузочное тестирование:

1) операционная система Ubuntu 22.04.3 LTS;
2) оперативная память 4 ГБ;
3) процессор Intel® Core™ i7-4790K @ 4.00 ГГц;
4) версия ядра Linux 6.2.0.

Нагрузочное тестирование проводилось с использованием утилиты ApacheBench — однопоточной программы для командной строки, используемой для измерения производительности HTTP веб-серверов. Тестированое проводилось для сконфигурированного сервера nginx и реализованного статического веб-сервера.

4.2 Нагрузочное тестирование

В таблицах 4.1–4.3 приведены результаты нагрузочного тестирования для сервера nginx и реализованного статического веб-сервера с 1-м дочерним процессом и 10-ю дочерними процессами.

Таблица 4.1 – Результаты нагрузочного тестирования для сервера nginx

Кол-во запросов	2000	4000	6000	8000	10000
Общее время тестирования, с	0.35	1.017	1.592	2.488	2.878
Среднее кол-во запросов в секунду	5716.25	3933.3	3769.14	3214.81	3474.07
Среднее время выполнения одного запроса, мс	0.175	0.254	0.265	0.311	0.288
Наибольшее время выполнения запроса, мс	14	19	15	22	29

Таблица 4.2 – Результаты нагрузочного тестирования для реализованного сервера с 1-м дочерним процессом

Кол-во запросов	2000	4000	6000	8000	10000
Общее время тестирования, с	0.952	1.914	2.769	3.438	4.673
Среднее кол-во запросов в секунду	2100.37	2089.91	2166.89	2327.05	2139.94
Среднее время выполнения одного запроса, мс	0.476	0.478	0.461	0.43	0.467
Наибольшее время выполнения запроса, мс	23	22	20	26	33

Таблица 4.3 – Результаты нагрузочного тестирования для реализованного сервера с 10-ю дочерними процессами

Кол-во запросов	2000	4000	6000	8000	10000
Общее время тестирования, с	1.107	1.644	2.333	2.908	3.426
Среднее кол-во запросов в секунду	1806.29	2432.94	2571.27	2750.87	2918.62
Среднее время выполнения одного запроса, мс	0.554	0.411	0.389	0.364	0.343
Наибольшее время выполнения запроса, мс	24	22	28	25	19

На рисунке 4.1 показана зависимость общего времени нагрузочного тестирования в секундах от количества запросов.

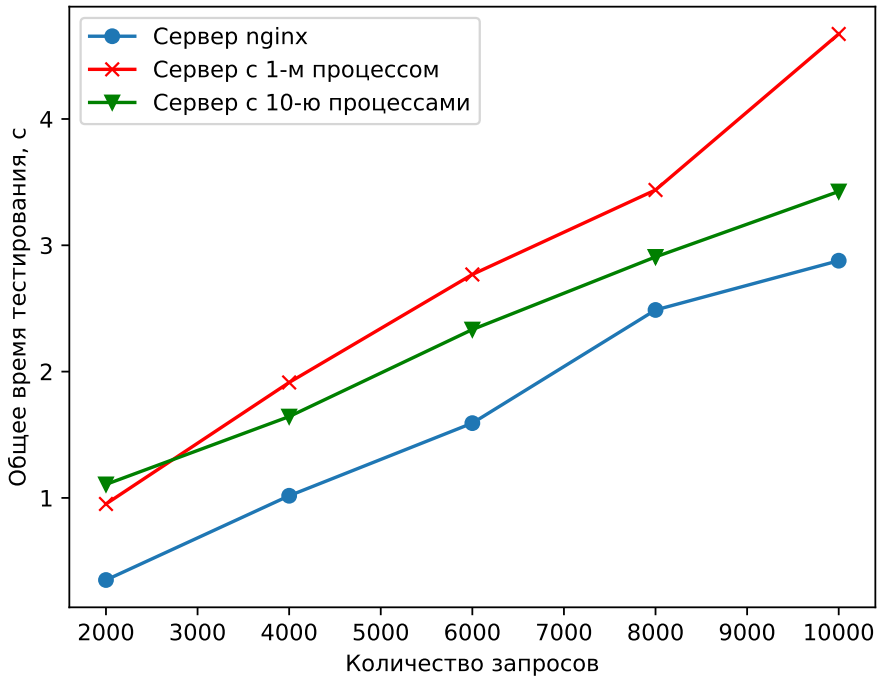

Рисунок 4.1 – Зависимость общего времени нагрузочного тестирования от количества запросов

На рисунке 4.2 показана зависимость среднего времени выполнения запроса в миллисекундах от количества запросов.

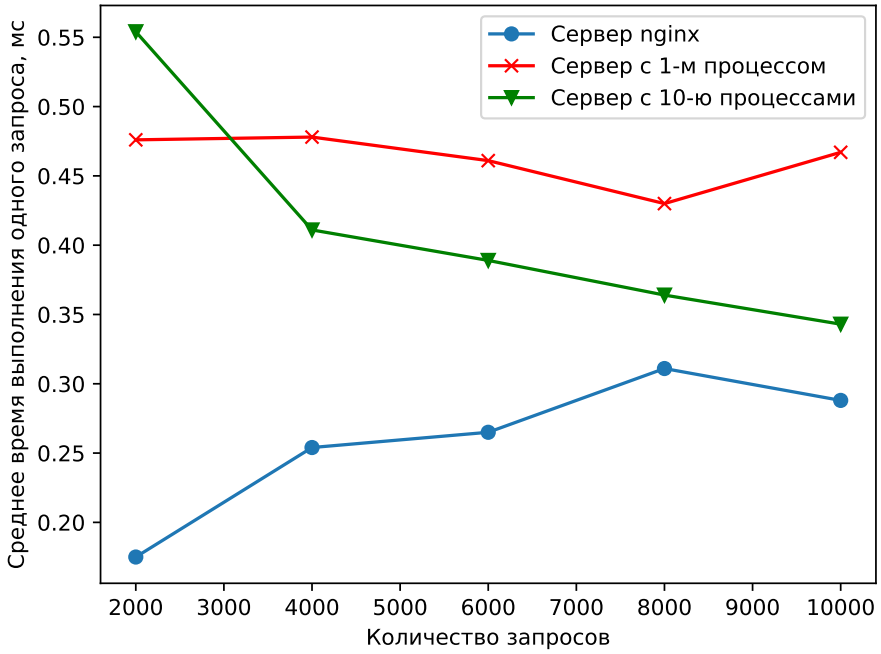

Рисунок 4.2 – Зависимость среднего времени выполнения запроса в миллисекундах от количества запросов

Вывод из исследовательской части

В ходе выполнения исследовательской части было проведено нагрузочное тестирование сервера nginx и реализованного сервера с 1-м дочерним процессом и 10-ю дочерними процессами. Согласно полученным данным, быстрее всего работает сервер nginx. Он обрабатывает запросы в среднем в 2 раза быстрее, чем реализованный статический сервер. Однако за счет увеличения количества дочерних процессов можно увеличить скорость обработки запросов реализованным веб-сервером.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы был реализован статический веб-сервера для отдачи контента с диска.

Были выполнены следующие задачи:

1) изучено понятие статического веб-сервера;
2) спроектирована архитектура статического сервера;
3) реализован спроектированный статический веб-сервер для отдачи контента с диска;
4) проведено сравнение результатов нагрузочного тестирования (при помощи ApacheBench) реализованного сервера с сервером nginx.

В ходе проведения исследования было выявлено, что сервер nginx обрабатывает запросы в среднем в 2 раза быстрее, чем реализованный статический веб-сервер с 1-м дочерним процессом. При 10 дочерних процессах статический веб-сервер обрабатывает запросы на 16% медленнее, чем nginx (для 10 тысяч запросов).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *MDNWebDocs*. Что такое веб-сервер. — 2023. — (Дата обращения: 10.11.2023). https://developer.mozilla.org/ru/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server.
2. *MDNWebDocs*. Сообщения HTTP. — 2023. — (Дата обращения: 10.11.2023). https://developer.mozilla.org/ru/docs/Web/HTTP/Messages.
3. *MDNWebDocs*. Заголовки HTTP. — 2023. — (Дата обращения: 10.11.2023). https://developer.mozilla.org/ru/docs/Web/HTTP/Headers.
4. *MDNWebDocs*. Content-Type. — 2023. — (Дата обращения: 10.11.2023). https://developer.mozilla.org/ru/docs/Web/HTTP/Headers/Content-Type.
5. *MDNWebDocs*. Коды ответа HTTP. — 2023. — (Дата обращения: 10.11.2023). https://developer.mozilla.org/ru/docs/Web/HTTP/Status.
6. *Force I. E. T.* Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. — 2014. — (Дата обращения: 10.11.2023). https://datatracker.ietf.org/doc/html/rfc7231.
7. *MDNWebDocs*. Методы HTTP запроса. — 2023. — (Дата обращения: 10.11.2023). https://developer.mozilla.org/ru/docs/Web/HTTP/Methods.
8. *ИнФОУрФУ*. Сокеты. — 2020. — (Дата обращения: 10.11.2023). https://lecturesnet.readthedocs.io/net/low-level/ipc/socket/intro.html.

ПРИЛОЖЕНИЕ А

Исходный код программы

В листингах 4.1–4.5 показан исходный код реализованного статического веб-сервера для отдачи контента с диска.

Листинг 4.1 – Файл main.c

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#include "server.h"
#include "logger.h"

logger_t logger;
int server_socket;

void handle_sigint(int sig_num)
{
    log_msg(logger, "Info: server shut down.\n");

    close(server_socket);
    exit_logger(logger);

    exit(EXIT_SUCCESS);
}

void handle_sigpipe(int sig_num)
{
    log_msg(logger, "Warning: client socket was closed.\n");
}

void clear_buffer(char *buffer, const size_t len)
{
    for (size_t i = 0; i < len; ++i)
        buffer[i] = '\0';
```

```
}

int main(void)
{
    signal(SIGINT, handle_sigint);
    signal(SIGPIPE, handle_sigpipe);

    logger = init_logger("./log/log.txt");

    struct sockaddr_in server_address;

    if (init_server(logger, &server_socket, &server_address) !=
       0)
    {
        log_msg(logger, "Error: server can't be initialized.\n");
        return EXIT_FAILURE;
    }

    fd_set current_sockets, ready_sockets;
    FD_ZERO(&current_sockets);
    FD_SET(server_socket, &current_sockets);

    for (size_t i = 0; i < PROCCESS_NUM; ++i)
    {
        pid_t pid = fork();

        if (pid == 0)
            break;
        else if (pid < 0)
        {
            log_msg(logger, "Error: can't fork.\n");
            return EXIT_FAILURE;
        }
    }

    while (1)
    {
        ready_sockets = current_sockets;

        if (pselect(FD_SETSIZE, &ready_sockets, NULL, NULL,
           NULL, NULL) == -1)
```

```
        {
            log_msg(logger, "Error: server can't pselect.\n");
            return EXIT_FAILURE;
        }

        for (int client_socket = 0; client_socket < FD_SETSIZE;
           ++client_socket)
        {
            if (FD_ISSET(client_socket, &ready_sockets))
            {
                if (client_socket == server_socket)
                {
                    int new_socket;
                    struct sockaddr_in client_address;
                    socklen_t client_address_len =
                       sizeof(client_address);

                    if ((new_socket = accept(server_socket,
                        (struct sockaddr *)&client_address,
                           &client_address_len)) == -1)
                    {
                        log_msg(logger, "Error: server can't
                           accept.\n");
                        return EXIT_FAILURE;
                    }

                    FD_SET(new_socket, &current_sockets);
                }
                else
                {
                    struct sockaddr_in client_address;
                    socklen_t client_address_len =
                       sizeof(client_address);
                    char request_buffer[REQUEST_LEN];

                    ssize_t len = recvfrom(client_socket,
                       request_buffer, REQUEST_LEN, 0,
                        (struct sockaddr *)&client_address,
                           &client_address_len);
                    request_buffer[len] = '\0';
```

```
                    log_msg(logger, request_buffer);
                    size_t method =
                       handle_method(request_buffer);

                    handle_request(method, client_socket,
                                   (struct sockaddr
                                      *)&client_address,
                                   client_address_len,
                                      request_buffer);

                    clear_buffer(request_buffer, REQUEST_LEN);
                    FD_CLR(client_socket, &current_sockets);
                    close(client_socket);
                }
            }
        }
    }
    return EXIT_SUCCESS;
}
```

Листинг 4.2 – Файл server.h

```
#ifndef _SERVER_H_
#define _SERVER_H_

#include "logger.h"

#define PORT                8080
#define MAX_CONNECTIONS     10

#define GET_METHOD          0
#define HEAD_METHOD         1
#define NOT_ALLOWED_METHOD  2

#define PROCCESS_NUM        10

#define REQUEST_LEN         8192
#define HEADER_LEN          1024
#define HEADER_TEMPLATE_LEN 128

ssize_t parse_filename(const char *const path, char *buffer);
void send_data(const char *const filename, const int fd, const
   struct sockaddr *to,
```

```
                socklen_t tolen);
void handle_request(const size_t method, const int client_socket,
                    struct sockaddr *client_address,
                    const socklen_t client_address_len,
                    const char *request_buffer);
ssize_t handle_method(const char *const request);
int init_server(logger_t logger, int *server_socket,
                struct sockaddr_in *server_address);
size_t parse_extension(const char *const filename, char *buffer);

#endif // _SERVER_H_
```

Листинг 4.3 – Файл server.c

```
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <errno.h>

#include "logger.h"
#include "server.h"

char response_header[HEADER_TEMPLATE_LEN + 1] = "HTTP/1.1 %s\r\n"
                                                "Content-Type:
                                                   %s/%s\r\n\r\n";
char response_buffer[HEADER_LEN + 1] = "";

const char forbidden_header[] = "HTTP/1.1 403
   Forbidden\r\nContent-Type: "
                                "text/html\r\n\r\n<h1>403
                                   Forbidden</h1>";
const char not_found_header[] = "HTTP/1.1 404 Not
   Found\r\nContent-Type: "
                                "text/html\r\n\r\n<h1>404 Not
                                   Found</h1>";
const char not_allowed_header[] = "HTTP/1.1 405 Method Not
   Allowed\r\n"
                                  "Content-Type:
                                     text/html\r\n\r\n<h1>405 "
```

```
                                    "Method Not Allowed</h1>";

ssize_t parse_filename(const char *const path, char *buffer)
{
    size_t i = 0, len = 0;

    for (i = 0; path[i] != '/'; ++i);

    for (++i; path[i] != '\n' && path[i] != ' ' && i <
       strlen(path); ++i)
    {
        if (path[i - 1] == '/' && path[i] == '/')
            return -1;

        buffer[len++] = path[i];
    }

    buffer[len] = '\0';

    return len;
}

size_t parse_extension(const char *const filename, char *buffer)
{
    size_t i = 0, len = 0;

    for (i = strlen(filename) - 1; filename[i] != '.' && i != 0;
       --i, ++len);

    if (i > 0)
    {
        memcpy(buffer, filename + i + 1, len);
        buffer[len] = '\0';
    }

    return len;
}

void send_data(const char *const filename, const int fd, const
   struct sockaddr *to,
                socklen_t tolen)
```

```
{
    char buffer[512 + 1] = "";
    FILE *f = fopen(filename, "rb");
    ssize_t rlen = 0, wlen = 0;

    while ((wlen != -1) && ((rlen = fread(buffer, sizeof(char),
       512, f)) > 0))
    {
        wlen = sendto(fd, buffer, rlen, 0, to, tolen);
    }

    fclose(f);
}

ssize_t handle_method(const char *const request)
{
    if (request[0] == 'G' && request[1] == 'E' && request[2] ==
       'T')
    {
        return GET_METHOD;
    }
    else if (request[0] == 'H' && request[1] == 'E' &&
       request[2] == 'A'
          && request[3] == 'D')
    {
        return HEAD_METHOD;
    }

    return NOT_ALLOWED_METHOD;
}

void handle_request(const size_t method, const int client_socket,
                    struct sockaddr *client_address,
                    const socklen_t client_address_len,
                    const char *request_buffer)
{
    if (NOT_ALLOWED_METHOD == method)
    {
        sendto(client_socket, not_allowed_header,
               strlen(not_allowed_header), 0,
               client_address,
```

```
            client_address_len);
    }
    else
    {
        char extension[5];
        char filename[128];
        ssize_t f_len = parse_filename(request_buffer, filename);

        if (f_len > 0)
            parse_extension(filename, extension);

        if (f_len == -1)
        {
            sendto(client_socket, forbidden_header,
               strlen(forbidden_header), 0,
                    client_address, client_address_len);
        }
        else if (f_len == 0)
        {
            snprintf(response_buffer, 1025, response_header,
                     "200 OK", "text", "html");
            sendto(client_socket, response_buffer,
               strlen(response_buffer), 0,
                    client_address, client_address_len);
            if (GET_METHOD == method)
                send_data("home.html", client_socket,
                          client_address,
                          client_address_len);
        }
        else if (access(filename, F_OK) != 0)
        {
            sendto(client_socket, not_found_header,
               strlen(not_found_header), 0,
                    client_address, client_address_len);
        }
        else if (strcmp(extension, "html") == 0)
        {
            snprintf(response_buffer, 1025, response_header,
                     "200 OK", "text", "html");
            sendto(client_socket, response_buffer,
               strlen(response_buffer), 0,
```

```
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket,
                  client_address,
                  client_address_len);
}
else if (strcmp(extension, "js") == 0)
{
    snprintf(response_buffer, 1025, response_header,
             "200 OK", "text", "javascript");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket,
           client_address, client_address_len);
}
else if (strcmp(extension, "css") == 0)
{
    snprintf(response_buffer, 1025, response_header,
             "200 OK", "text", "css");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket,
           client_address, client_address_len);
}
else if (strcmp(extension, "ico") == 0)
{
    snprintf(response_buffer, 1025, response_header,
             "200 OK", "image", "x-icon");
    sendto(client_socket, response_buffer,
       strlen(response_buffer), 0,
            client_address, client_address_len);
    if (GET_METHOD == method)
        send_data(filename, client_socket,
           client_address, client_address_len);
}
else if (strcmp(extension, "png") == 0 ||
   strcmp(extension, "jpg") == 0
```

```
        || strcmp(extension, "jpg") == 0 ||
           strcmp(extension, "jpeg") == 0
        || strcmp(extension, "gif") == 0)
    {
        snprintf(response_buffer, 1025, response_header,
                 "200 OK", "image", extension);
        sendto(client_socket, response_buffer,
           strlen(response_buffer), 0,
               client_address, client_address_len);
        if (GET_METHOD == method)
            send_data(filename, client_socket,
              client_address, client_address_len);
    }
    else if (strcmp(extension, "swf") == 0)
    {
        snprintf(response_buffer, 1025, response_header,
                 "200 OK", "application",
                 "x-shockwave-flash\r\nContent-Disposition:
                    inline;");
        sendto(client_socket, response_buffer,
           strlen(response_buffer), 0,
               client_address, client_address_len);
        if (GET_METHOD == method)
            send_data(filename, client_socket,
              client_address, client_address_len);
    }
    else if (strcmp(extension, "mp4") == 0)
    {
        snprintf(response_buffer, 1025, response_header,
                 "200 OK", "video", "mp4");
        sendto(client_socket, response_buffer,
           strlen(response_buffer), 0,
               client_address, client_address_len);
        if (GET_METHOD == method)
            send_data(filename, client_socket,
              client_address, client_address_len);
    }
    else
    {
        sendto(client_socket, not_found_header,
           strlen(not_found_header), 0,
```

```
                    client_address, client_address_len);
        }
    }
}

int init_server(logger_t logger, int *server_socket, struct
   sockaddr_in *server_address)
{
    if ((*server_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        log_msg(logger, "Error: server socket can't be
           created.\n");
        return -1;
    }

    int optval = 1;
    setsockopt(*server_socket, SOL_SOCKET, SO_REUSEPORT,
       &optval, sizeof(optval));

    server_address->sin_family = AF_INET;
    server_address->sin_addr.s_addr = INADDR_ANY;
    server_address->sin_port = htons(PORT);

    if (bind(*server_socket, (struct sockaddr *)server_address,
        sizeof(*server_address)) == -1)
    {
        log_msg(logger, "Error: can't bind server socket.\n");
        return -2;
    }

    if (listen(*server_socket, MAX_CONNECTIONS) == -1)
    {
        log_msg(logger, "Error: server can't listen.\n");
        return -3;
    }
    return 0;
}
```

Листинг 4.4 – Файл logger.h

```
#ifndef _LOGGER_H_
#define _LOGGER_H_
```

```
typedef FILE* logger_t;

logger_t init_logger(const char* const filename);
void log_msg(const logger_t logger, const char* const msg);
void exit_logger(const logger_t logger);

#endif // _LOGGER_H_
```

Листинг 4.5 – Файл logger.c

```
#include <stdio.h>
#include <string.h>
#include <time.h>

#include "logger.h"

logger_t init_logger(const char* const filename)
{
    return fopen(filename, "a+");
}

void log_msg(const logger_t logger, const char* const msg)
{
    char template[32 + 1] = "Datetime: %s";
    char header[128 + 1] = "";

    time_t t;
    time(&t);

    snprintf(header, 128, template, ctime(&t));
    fwrite(header, sizeof(char), strlen(header), logger);

    size_t len = strlen(msg);
    fwrite(msg, sizeof(char), len, logger);
}

void exit_logger(const logger_t logger)
{
    if (logger != NULL)
        fclose(logger);
}
```