СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Термин «**веб-сервер**» [1] может относиться как к аппаратному, так и к программному обеспечению.

С точки зрения аппаратного обеспечения веб-сервер — это компьютер, на котором хранятся программное обеспечение веб-сервера и файлы компонентов веб-сайта (например, HTML-документы, изображения, стили CSS и файлы JavaScript). Веб-сервер подключается к Интернету и поддерживает физический обмен данными с другими устройствами, подключенными к Интернету. [1]

С точки зрения программного обеспечения веб-сервер включает в себя различные компоненты, которые контролируют доступ веб-пользователей к размещённым файлам. Одним из таких компонентов является HTTP-сервер. [1]

HTTP-сервер [1] — это программное обеспечение, которое работает с URL-адресами по протоколу HTTP, доставляя содержимое веб-сайтов на устройства конечных пользователей. Доступ к HTTP-серверу можно получить через доменные имена веб-сайтов, которые он хранит.

Статический сервер — это сервер, который предоставляет клиентам исключительно статические файлы. Это могут быть файлы HTML, CSS, JavaScript, изображения, видео, аудио и любые другие файлы, которые клиенты скачивают и используют в исходном виде, без какой-либо обработки на стороне сервера. [1]

Статические серверы существенно отличаются от динамических серверов, которые могут выполнять серверный код, чтобы формировать веб-страницы «на лету» или предоставлять другие функции, такие как обработка форм, взаимодействие с базами данных и выполнение других операций. [1]

Целью данной работы является разработка статического сервера. Для достижения поставленной цели необходимо решить следующие задачи.

1. Провести анализ предметной области и формализовать задачу.
2. Спроектировать структуру программного обеспечения.
3. Реализовать программное обеспечение, которое будет обслуживать контент, хранящийся во вторичной памяти.
4. Провести нагрузочное тестирование и сравнение с распространёнными аналогами.

1 Аналитическая часть

В данном разделе будут описаны основные теоретические аспекты, необходимые для решения поставленной задачи.

1.1. Протокол HTTP

HTTP (англ. Hypertext Transfer Protocol, гипертекстовый транспортный протокол) [2] — протокол, определяющий набор соглашений по передаче гипертекста (т.е. связанных веб-документов) между двумя компьютерами. HTTP является текстовым протоколом без сохранения состояния.

Данный протокол задаёт следующие правила взаимодействия клиента и сервера. [1]

— HTTP-запросы производятся исключительно от клиентов к серверу, который способен только отвечать на запросы.

— При запросе файла по HTTP клиент должен сформировать файловый **URL** (Uniform Resource Locator, единообразный указатель местонахождения ресурса). [3]

— Веб-сервер должен ответить на каждый HTTP-запрос, даже в случае возникновения ошибки.

HTTP-запросы и ответы имеют схожую структуру и состоят из следующих элементов. [4]

— Начальная строка, описывающая запросы.

— Необязательный набор заголовков HTTP, определяющий запрос или описывающий тело сообщения.

— Необязательное тело сообщения, содержащее данные, связанные с запросом (например, содержимое HTML-формы), или документ, связанный с ответом. Наличие тела и его размер задаются начальной строкой и заголовками HTTP.

Начальная строка и HTTP-заголовки HTTP-сообщения вместе называются **заголовком**, а его полезная нагрузка — **телом**. На рисунке 1.1 представлено описание структуры запроса и ответа HTTP версии 1.x. [4]

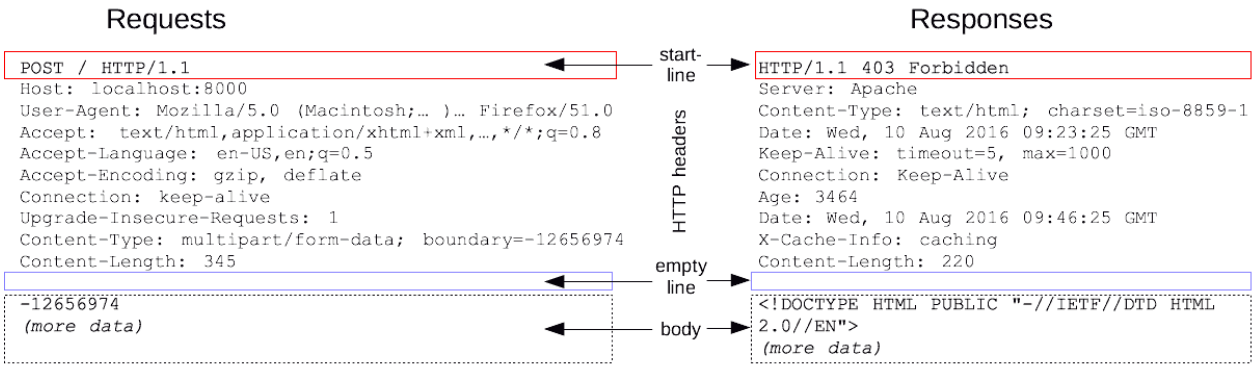

Рис. 1.1 – Структура запроса и ответа HTTP версии 1.x

1.2. Веб-серверы

Веб-серверы, как правило, работают по протоколам HTTP/HTTPS и предоставляют клиентам доступ к файлам, таким как веб-страницы. Чтобы загрузить веб-страницу, браузер отправляет запрос к веб-серверу, который выполняет поиск запрашиваемого файла в своём хранилище. Найдя файл, сервер считывает его, обрабатывает при необходимости и отсылает в браузер. [1]

Преимущества выделенных веб-серверов по сравнению с локальными хранилищами: [1]

1) бесперебойная работа;
2) постоянное подключение к интернету;
3) неизменный IP адрес;
4) обслуживание сторонним провайдером.

Веб-серверы могут работать со статическим и динамическим контентом: статический отдаётся клиенту в исходном виде без изменений, а динамический является результатом обработки данных на стороне сервера. Со статическим контентом проще работать, в то время, как динамический контент обеспечивает большую гибкость. [1]

1.3. Сокеты

Сокет — это абстракция конечной точки соединения, которая используется для обеспечения обмена данными между устройствами сети. Сокеты явля-

ются ключевым компонентом для установки и управления сетевыми соединениями.

Сокеты предоставляют интерфейс для создания конечных точек соединения в сети с использованием протоколов передачи данных, таких как TCP или UDP. При разработке веб-серверов сокеты используются для «прослушивания» входящих соединений от клиентов и передачи данных между сервером и клиентами.

Процесс создания сервера с использованием сокетов включает в себя следующие шаги.

1. Создание сокета, который будет слушать входящие соединения.
2. Привязка сокета к адресу и порту: после создания сокета необходимо привязать его к сетевому адресу и порту, на котором будет прослушиваться входящий трафик.
3. Установка сервера в состояние прослушивания входящих соединений.
4. Принятие входящего соединения.
5. Обработка запросов и передача данных: После установки соединения с клиентом сервер может принимать запросы от клиента с помощью функций чтения и записи данных через сокет.

Использование сокетов в разработке серверов позволяет эффективно управлять сетевыми соединениями и обеспечивать передачу статического контента клиентам по сети.

1.4. Мультиплексирование ввода-вывода

Мультиплексирование ввода-вывода является методом обработки нескольких операций ввода-вывода в одном потоке выполнения программы, что позволяет повысить эффективность управления множеством соединений в сетевых приложениях или приложениях с асинхронным вводом-выводом. Это позволяет уменьшить задержки и повысить производительность.

Мультиплексоры позволяют приложению ожидать ввода или вывода данных из нескольких источников, таких как сокеты, файлы или сетевые

устройства, используя один системный вызов вместо создания или управления каждым потоком или процессом отдельно.

Использование мультиплексирования ввода-вывода требует более сложной логики обработки событий, чем простое параллельное программирование, но оно может обеспечить более эффективное использование системных ресурсов и более высокую производительность.

Существуют следующие механизмы мультиплексирования сетевых соединений.

— **select** — это системный вызов, используемый в различных операционных системах для ожидания событий на нескольких файловых дескрипторах, таких как сокеты. Он позволяет одному потоку контролировать несколько соединений одновременно. Данный мультиплексор имеет ограничение на максимальное количество отслеживаемых файловых дескрипторов — 1024.

— **pselect** схож по принципу работы с select и имеет те же ограничения, однако реализует более продвинутую обработку сигналов.

— **epoll**, предоставляемый в ядре Linux, обеспечивает более гибкие варианты работы с событиями ввода-вывода за счёт реализации модели уведомлений. В отличие от select и pselect, epoll предоставляет более эффективный способ мониторинга множества файловых дескрипторов на предмет готовности к вводу или выводу и не имеет ограничений на обработку файловых дескрипторов.

epoll является наиболее современным и эффективным механизмом в сравнении с select и pselect и часто используется в высоконагруженных сетевых приложениях на ОС Linux.

1.5. Параллелизация обработки запросов

Для ускорения обработки запросов веб-серверы реализуют их параллельную обработку в многопроцессорной среде. Существует два подхода к параллелизации работы веб-серверов: пул потоков (thread pool) и разветвление (prefork).

Thread Pool представляет собой набор потоков, готовых к выполнению задач. Когда в систему поступает новая задача, она помещается в очередь, и один из доступных потоков в пуле забирает эту задачу на выполнение. После завершения задачи поток возвращается обратно в пул и становится доступным для выполнения новых задач. Это способствует уменьшению накладных расходов на создание и уничтожение потоков, а также позволяет управлять ресурсами более эффективно и обеспечивает параллельное выполнение задач.

Prefork — это модель параллелизации, при которой веб-сервер создаёт отдельные процессы для обработки запросов. Такой подход позволяет изолировать обработку каждого запроса и повышает надежность веб-сервера, так как сбои в одном процессе не влияют на остальные. Однако создание процессов требует больше дополнительных ресурсов, чем использование пула потоков.

Выбор между Thread Pool и Prefork зависит от конкретных требований веб-сервера, предполагаемой нагрузки и характера обрабатываемых запросов.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритма обработки клиентских запросов.

2.1. Сервер

На рисунке 2.1 представлена схема алгоритма работы сервера, обслуживающего клиентские запросы с использованием сокетов.

В целях повышения масштабируемости приложения алгоритм работы сервера не зависит от подхода к параллельной обработке запросов. Добавить параллелизацию можно с использованием паттерна проектирования «инъекция зависимостей» (dependency injection).

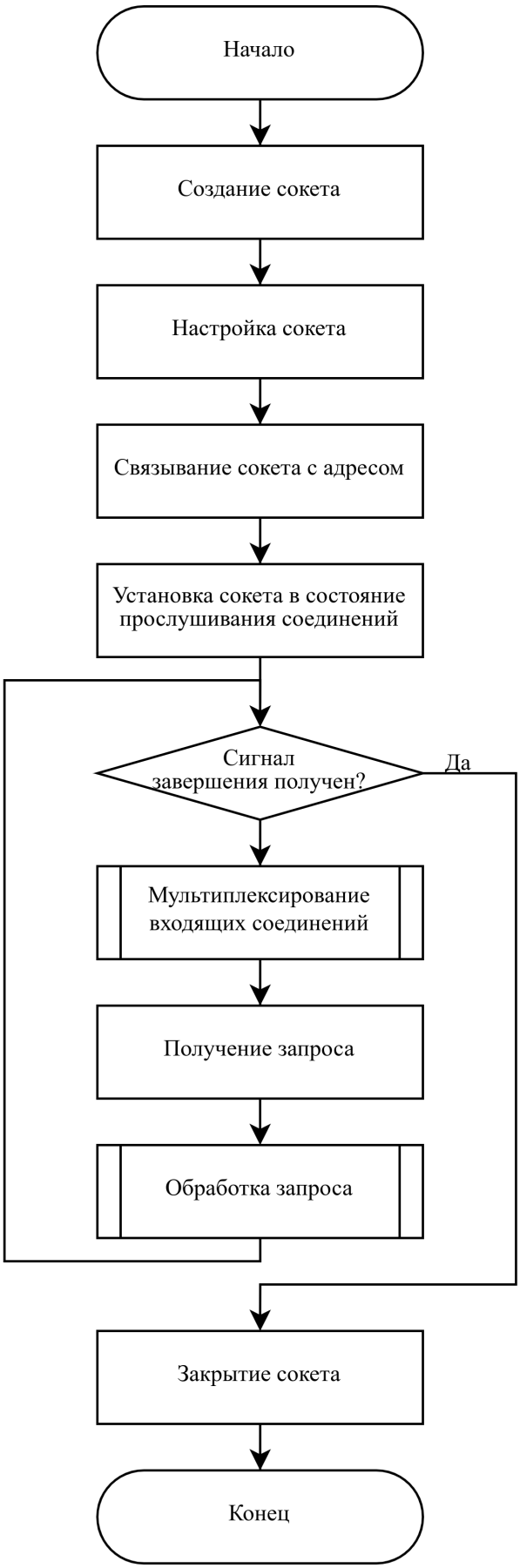

Рис. 2.1 – Алгоритм работы сервера

2.2. Пул потоков

На рисунке 2.2 представлена схема алгоритма работы пула потоков.

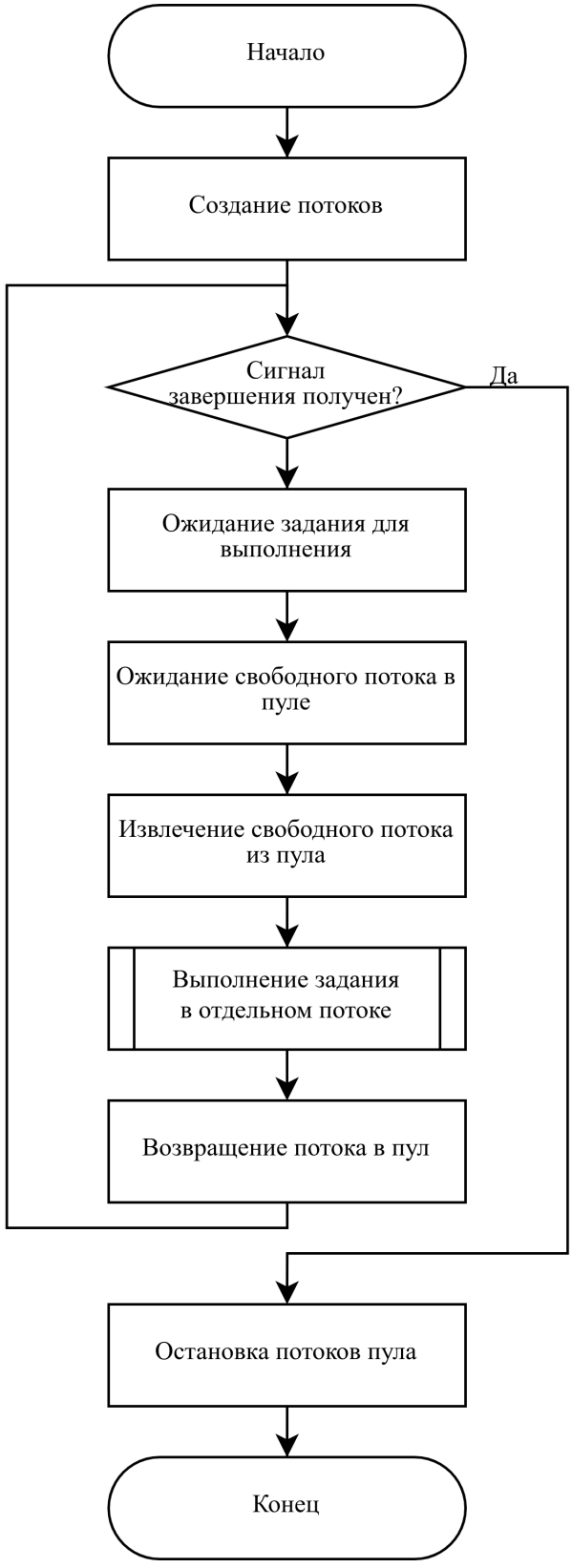

Рис. 2.2 – Алгоритм работы пула потоков

3 Технологическая часть

В данном разделе будут описаны детали реализации и развёртывания ПО, а также приведены листинги кода.

3.1. Средства реализации

Для реализации статического сервера был выбран язык C в соответствии с заданием на курсовую работу. Для мультиплексирования клиентских соединений был выбран мультиплексор select, для параллелизации обработки запросов — пул потоков.

Для контроля качества кода использовался отладчик использования памяти valgrind [5].

3.2. Детали реализации

3.2.1. Сервер

В листингах 3.1 и 3.2 представлены функции, реализующие обработку клиентских запросов с использованием мультиплексора select.

Листинг 3.1 – Функция создания сервера

```
1     int server_create(server_t *server) {
2         server_t tmp_server = malloc(sizeof(struct server));
3         if (tmp_server == NULL) {
4             log_error("server_init malloc(): %s", strerror(errno));
5             return errno;
6         }
7
8         if ((tmp_server->server_socket_fd = socket(AF_INET, SOCK_STREAM, 0))
9             == 0) {
10            log_error("socket(): %s", strerror(errno));
11            free(*server);
12            return EXIT_FAILURE;
13        }
14
15        int opt = IP_PMTUDISC_WANT;
16        if (setsockopt(tmp_server->server_socket_fd, SOL_SOCKET,
17            SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
18            log_error("setsockopt(): %s", strerror(errno));
19            free(tmp_server);
20            return EXIT_FAILURE;
21        }
22
```

```
23          tmp_server->is_running = false;
24          *server = tmp_server;
25
26          return EXIT_SUCCESS;
27      }
```

Листинг 3.2 – Функция запуска сервера

```
1       int server_run(server_t server, int port, int conn_queue_len,
2           void(*handle_request)(int)) {
3           struct sockaddr_in address;
4           address.sin_family = AF_INET;
5           address.sin_addr.s_addr = INADDR_ANY;
6           address.sin_port = htons(port);
7
8           if (bind(server->server_socket_fd, (struct sockaddr*)&address,
9               sizeof(address)) == -1) {
10              log_error("bind(): %s", strerror(errno));
11              return errno;
12          }
13
14          if (listen(server->server_socket_fd, conn_queue_len) == -1) {
15              log_error("listen(): %s", strerror(errno));
16              return errno;
17          }
18
19          log_info("server started on port %d; wait for connections...",
20              port);
21
22          server->is_running = true;
23          fd_set client_fds;
24          while (server->is_running) {
25              FD_ZERO(&client_fds);
26              FD_SET(server->server_socket_fd, &client_fds);
27
28              if (select(server->server_socket_fd + 1, &client_fds, NULL,
29                  NULL, NULL) == -1) {
30                  log_error("select(): %s", strerror(errno));
31                  return errno;
32              }
33
34              int client_socket_fd = -1;
35              if (FD_ISSET(server->server_socket_fd, &client_fds)) {
36                  if ((client_socket_fd = accept(server->server_socket_fd,
37                      (struct sockaddr*)NULL, NULL)) == -1) {
38                      log_error("accept(): %s", strerror(errno));
39                      return errno;
40                  }
41
```

```
42                  handle_request(client_socket_fd);
43              }
44          }
45
46          return EXIT_SUCCESS;
47      }
```

3.2.2. Пул потоков

В листингах 3.3-3.5 представлены функции, реализующие обработку клиентских запросов с использованием мультиплексора select.

Листинг 3.3 – Функция запуска пула потоков

```
1       int thread_pool_start(thread_pool_t pool,
2           void *(*worker_thread)(void *)) {
3           for (int i = 0; i < pool->capacity; i++) {
4               if (pthread_create(&pool->threads[i], NULL, worker_thread, pool)
5                   != 0) {
6                   log_error("pthread_create(): %s", strerror(errno));
7                   return errno;
8               }
9           }
10
11          return EXIT_SUCCESS;
12      }
```

Листинг 3.4 – Функция извлечения задачи из очереди

```
1       int thread_pool_take_task(thread_pool_task_t *task, thread_pool_t pool)
2       {
3           log_debug("try to take task from pool...");
4           pthread_mutex_lock(&queue_mutex);
5
6           log_debug("wait for any task pool...");
7           while (pool->size == 0) {
8               pthread_cond_wait(&queue_cond, &queue_mutex);
9           }
10
11          thread_pool_task_t t = pool->tasks[--pool->size];
12
13          log_debug("task taken from pool");
14          pthread_cond_signal(&queue_cond);
15          pthread_mutex_unlock(&queue_mutex);
16
17          *task = t;
18
19          return EXIT_SUCCESS;
```

```
20      }
```

Листинг 3.5 – Функция добавления задачи в очередь на выполнение

```
1       int thread_pool_submit(thread_pool_t pool, thread_pool_task_t task) {
2           log_debug("try to put task to pool...");
3           pthread_mutex_lock(&queue_mutex);
4
5           log_debug("wait for place in pool to put task...");
6           while (pool->size >= pool->capacity) {
7               pthread_cond_wait(&queue_cond, &queue_mutex);
8           }
9
10          pool->tasks[pool->size++] = task;
11
12          log_debug("task added to pool");
13          pthread_cond_signal(&queue_cond);
14          pthread_mutex_unlock(&queue_mutex);
15
16          return EXIT_SUCCESS;
17      }
```

3.3. Поддерживаемые запросы

Разработанный веб-сервер обрабатывает запросы GET и HEAD. В первом случае клиент получает в теле ответа запрошенный файл, во втором — только заголовки ответа Content-Type и Content-Length. Ниже перечислены поддерживаемые статусы ответов сервера.

— 200 — успешное завершение обработки запроса.

— 403 — доступ к запрошенному файлу запрещён.

— 404 — запрашиваемый файл не найден.

— 405 — неподдерживаемый HTTP-метод (POST, PUT и т.д.).

— 500 — внутренняя ошибка сервера.

— 501 — запрошен неподдерживаемый тип файла.

Разработанный сервер поддерживает следующие форматы файлов (типы контента, Content-Type):

— html (text/html);

— css (text/css);

— js (text/javascript);

— png (image/png);

— jpg (image/jpg);

— jpeg (image/jpe);

— gif (image/gif);

— svg (image/svg);

— swf (application/x-shockwave-flash);

— mp4 (application/x-shockwave-flash).

3.4. Развёртывание

Запуск разработанного приложения осуществлялся с помощью системы контейнеризации Docker [6]. Файл для сборки образа приложения представлен в листинге 3.6.

Листинг 3.6 – Dockerfile образа приложения

```
1     FROM gcc:13.2.0-bookworm AS build
2     WORKDIR /build
3     RUN --mount=target=. \
4     gcc -DLOG_USE_COLOR \
5         -std=gnu99 -Wall -Wpedantic -Wextra -Wfloat-equal \
6         -Wfloat-conversion -Wvla  \
7         -static \
8         -Iinc/app -Iinc/http -Ilib/fs -Ilib/log \
9         -O2 -o /app  \
10        src/main.c src/app/* src/http/* lib/fs/fs.c lib/log/log.c
11
12    FROM scratch
13    COPY --from=build /app /app
14    EXPOSE 8080
15    ENTRYPOINT ["/app"]
```

В листинге 3.7 представлены команды для сборки образа приложения и запуска соответствующего контейнера.

Листинг 3.7 – Команды сборки образа приложения и запуска контейнера

```
1       docker build -t static-server .
2       docker run --name static-server --volume=./media:/tmp/static -d \
3           -p 8080:8080 --cpus=8 static-server
```

4 Исследовательская часть

В данном разделе будет проведено нагрузочное тестирование разработанного сервера и сравнение его показателей производительности с NGINX.

4.1. Описание проводимых измерений

В качестве эталона для сравнения показателей производительности статического сервера был выбран NGINX [7], являющийся HTTP-сервером общего назначения. Конфигурация NGINX приведена в листинге 4.1.

Листинг 4.1 – Конфигурация NGINX

```
server {
    listen 80;
    location /tmp/static {
        alias /tmp/static;
        include /etc/nginx/mime.types;
        autoindex on;
        autoindex_exact_size off;
        autoindex_localtime on;
    }
}
```

Файл для сборки образа приложения представлен в листинге 4.2.

Листинг 4.2 – Dockerfile образа NGINX

```
FROM nginx:1.25.1
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

В листинге 4.3 представлены команды для сборки образа сервера NGINX и запуска соответствующего контейнера.

Листинг 4.3 – Команды сборки образа NGINX и запуска контейнера

```
docker build -t nginx-example .
docker run --name static-server --volume=./media:/tmp/static -d \
    -p 8080:80 --cpus=8 nginx-example
```

Ниже приведены технические характеристики устройства, на котором выполнялось тестирование.

- Операционная система: Debian 12, версия ядра 6.1.0-12-amd64.
- Объём оперативной памяти: 16 Гб.
- Процессор: Intel i5-9300H 2.4 ГГц [**intel**].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

Для замеров метрик производительности сравниваемых веб-серверов использовалась утилита Apache Benchmark (ab). [8]

4.2. Результаты измерений

Результаты нагрузочного тестирования разработанного сервера и NGINX представлены в таблицах 4.1-4.3. Для оценки производительности измерялось количество обработанных запросов в секунду (Requests Per Second, RPS).

Таблица 4.1 – RPS при обработке 10000 запросов на файл размером 2 Кб

Число клиентов	Разработанный сервер	NGINX
1	414.97	406.74
100	1878.45	1823.23
1000	1798.11	1155.56

Таблица 4.2 – RPS при обработке 1000 запросов на файл размером 468 Кб

Число клиентов	Разработанный сервер	NGINX
1	67.15	77.73
100	115.02	120.52
1000	113.96	122.20

Таблица 4.3 – RPS при обработке 10 запросов на файл размером 582 Мб

Число клиентов	Разработанный сервер	NGINX
1	0.08	0.08
5	0.10	0.10
10	0.10	0.10

4.3. Выводы

При обработке относительно лёгких запросов RPS сравниваемых серверов не отличался более чем на 2%, однако при увеличении числа конкурентных

запросов производительность резко снизилась, став на 36% ниже той, что показал разработанный сервер. При увеличении размера запрашиваемых файлов производительность NGINX оказалась выше на 4-15%. При работе с относительно большими файлами оба сервера показали примерно одинаковую производительность и пропускную способность.

Данные результаты можно объяснить тем, что разработанный сервер имеет более простую архитектуру и компактную кодовую базу, чем NGINX. Также ввиду использования мультиплексора select, имеющего ограничение на количество открытых файловых дескрипторов (не более 1024), при одновременном обслуживании более 1000 клиентов разработанный сервер склонен к потере запросов. Для устранения данной проблемы следует использовать другой мультиплексор, например epoll.

ЗАКЛЮЧЕНИЕ

В рамках курсовой работы был разработан статический сервер, который на большинстве сценариев использования оказался сравнимым по производительности с NGINX.

В ходе выполнения данной работы были решены следующие задачи.

1. Проведён анализ предметной области и формализована задача.
2. Спроектирована структура программного обеспечения.
3. Реализовано программное обеспечение, которое обслуживает контент, хранящийся во вторичной памяти.
4. Проведено нагрузочное тестирование и сравнение с распространёнными аналогами.

Таким образом, все поставленные задачи были выполнены, поставленная цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. What is a web server? [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server (дата обращения: 29.11.2023).
2. HTTP [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Glossary/HTTP (дата обращения: 29.11.2023).
3. What is a URL? [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_URL (дата обращения: 29.11.2023).
4. HTTP Messages [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages (дата обращения: 29.11.2023).
5. Valgrind Home [Электронный ресурс]. — Режим доступа, URL: https://valgrind.org/ (дата обращения: 29.11.2023).
6. Docker: Accelerated Container Application Development [Электронный ресурс]. — Режим доступа, URL: https://www.docker.com/ (дата обращения: 29.11.2023).
7. NGINX: Advanced Load Balancer, Web Server, Reverse Proxy [Электронный ресурс]. — Режим доступа, URL: https://www.nginx.com/ (дата обращения: 29.11.2023).
8. ab - Apache HTTP server benchmarking tool [Электронный ресурс]. — Режим доступа, URL: https://httpd.apache.org/docs/2.4/programs/ab.html (дата обращения: 29.11.2023).