Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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:

«Разработка статического сервера»

Студент ИУ7-75Б (Группа) _____ (Подпись, дата) М. Е. Зевахин (И. О. Фамилия)

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

2025 г.

РЕФЕРАТ

Расчетно-пояснительная записка 36 с., 11 рис., 5 табл., 6 ист., 1 прил.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью курсовой работы является разработка HTTP-сервера для отдачи статического содержимого, построенного на архитектуре пула потоков с мультиплексированием сетевых соединений с использованием системного вызова `poll()`.

Задачи:

— Предусмотреть поддержку запросов GET и HEAD, поддержку статусов 200, 403, 404.

— Предусмотреть возможность ответа сервера на неподдерживаемые запросы статусом 405.

— Обеспечить корректную передачу файлов размером до 128 Мбайт.

— Реализовать мультиплексирование – каждый процесс или поток должен отдавать данные по нескольким сетевым соединениям.

— Сервер по умолчанию должен возвращать HTML-страницу на выбранную тему с CSS-стилем.

— Реализовать запись информации о событиях в журнал (лог).

— Учесть минимальные требования к безопасности серверов статического содержимого.

1 Аналитическая часть

В данном разделе рассматриваются ключевые понятия, протоколы и механизмы операционной системы, лежащие в основе разработанного HTTP-сервера.

1.1 Протокол HTTP

HyperText Transfer Protocol (HTTP) – это прикладной протокол передачи данных, лежащий в основе обмена информацией в сети. Он использует модель «клиент-сервер», где клиент отправляет запрос, а сервер возвращает ответ [1].

HTTP версий 1.0 и 1.1, поддерживаемые в данной работе, работают поверх транспортного протокола TCP. Каждое соединение устанавливается отдельно (в HTTP/1.0) или может переиспользоваться (в HTTP/1.1). Запрос состоит из:

1) Стартовой строки (например, `GET /index.html HTTP/1.1`);
2) Заголовков (headers) в формате «ключ: значение»;
3) Пустой строки `\r\n\r\n`;
4) Необязательного тела (body).

Методы `GET` и `HEAD` используются для получения ресурса. Отличие заключается в том, что при `HEAD` сервер возвращает только заголовки ответа без тела. Это позволяет клиенту проверить метаданные, например, размер или дату изменения без передачи всего содержимого.

Сервер должен формировать HTTP-ответы с кодами состояния:

— 200 (*OK*) – успешная обработка запроса;

— 403 (*Forbidden*) – доступ к ресурсу запрещён;

— 404 (*Not Found*) – запрашиваемый ресурс не найден;

— 405 (*Method Not Allowed*) – использован неподдерживаемый HTTP-метод.

1.2 Мультиплексирование

Мультиплексирование – это способность одного потока или процесса одновременно управлять множеством сетевых соединений. Для реализации мультиплексирования в Unix-подобных системах используются системные вызовы:

— *select()*, *poll()* – классические механизмы опроса множества файловых дескрипторов;

— *epoll()* (Linux) или *kqueue()* (BSD, macOS) – масштабируемые альтернативы.

В данной работе выбран вызов `poll()`. Он позволяет потоку ожидать готовности любого из набора сокетов к чтению или записи, не блокируя выполнение программы и не требуя отдельного потока на каждое соединение [2].

1.3 Архитектура «пул потоков»

Пул потоков (thread pool) – это шаблон проектирования, при котором создаётся фиксированное количество рабочих потоков при старте программы. Входящие задачи (сетевые соединения) помещаются в очередь и распределяются между свободными потоками.

Преимущества:

— Снижение накладных расходов на создание/уничтожение потоков;

— Контролируемое потребление памяти и CPU;

— Возможность балансировки нагрузки.

В разработанном сервере пул потоков комбинируется с мультиплексированием: каждый рабочий поток сам управляет множеством сокетов через *poll()*.

1.4 Механизмы межпоточного взаимодействия

Для передачи новых соединений от главного потока (слушающего *accept()*) к рабочим потокам используется механизм *анонимных каналов* (anonymous pipes).

Pipe – это односторонний канал связи между процессами или потоками, реализованный через два файловых дескриптора [3]:

— *fd[1]* – дескриптор для записи;

— *fd[0]* – дескриптор для чтения.

В данной реализации каждый рабочий поток имеет свой pipe. Главный поток записывает в него структуру с новым сокетом, IP и портом клиента. Рабочий поток добавляет этот сокет в свой набор для *poll()*, тем самым принимая соединение на обслуживание. Каналы также используются для сигнализации о завершении работы.

1.5 Обеспечение безопасности

При разработке сервера статического содержимого критически важна защита от атак типа *Path Traversal* (обход каталогов), при которых злоумышленник пытается получить доступ к файлам вне корневой директории (например, *GET /../../../../etc/passwd*).

Для предотвращения таких атак реализована функция *is_path_safe()*, которая:

— Нормализует путь с помощью *realpath()*, разрешая символические ссылки и удаляя .. и .;

— Сравнивает полученный абсолютный путь с каноническим путём корневой директории;

— Разрешает доступ только если результирующий путь находится внутри корневой директории.

Также сервер ограничивает максимальный размер отдаваемого файла (128 МБ) и не позволяет отдавать содержимое каталогов (возвращается 403).

1.6 Логирование

Ведение журнала событий (логирование) необходимо для мониторинга, отладки и аудита. Сервер реализует запись в лог-файл в формате, близком к Common Log Format (CLF):

```
[время_в_UTC] [IP:порт] "МЕТОД ПУТЬ" КОД_СТАТУСА ОБЪЁМ.
```

Для обеспечения потокобезопасности при записи из нескольких рабочих потоков используется мьютекс ($pthread\_mutex\_t$), защищающий критическую секцию *fprintf()*.

2 Конструкторская часть

В данном разделе представлена архитектура HTTP-сервера, описаны ключевые компоненты и алгоритмы их взаимодействия.

2.1 Общая архитектура сервера

Сервер состоит из следующих модулей:

1) **Главный поток** – создаёт слушающий сокет, запускает пул рабочих потоков и принимает новые соединения через `accept()`.

2) **Пул рабочих потоков** – фиксированное количество потоков, каждый из которых управляет множеством клиентских соединений.

3) **Каналы (pipes)** – используются для передачи новых сокетов от главного потока к рабочим.

4) **Модуль HTTP** – парсинг запросов, формирование ответов.

5) **Модуль безопасности** – проверка пути через *is_path_safe*.

6) **Логгер** – потокобезопасная запись событий.

2.2 Жизненный цикл клиентского соединения

Каждое соединение проходит через конечный автомат с четырьмя состояниями (рис. 2.1):

1. *CONN_READING* – ожидание и чтение HTTP-запроса;
2. *CONN_SENDING_HEADER* – отправка заголовков ответа;
3. *CONN_SENDING_BODY* – пошаговая отправка тела файла через *sendfile()*;
4. *CONN_DONE* – завершение, закрытие сокета и файла.

Переходы между состояниями управляются событиями *POLLIN* и *POLLOUT*, возвращаемыми вызовом *poll()*.

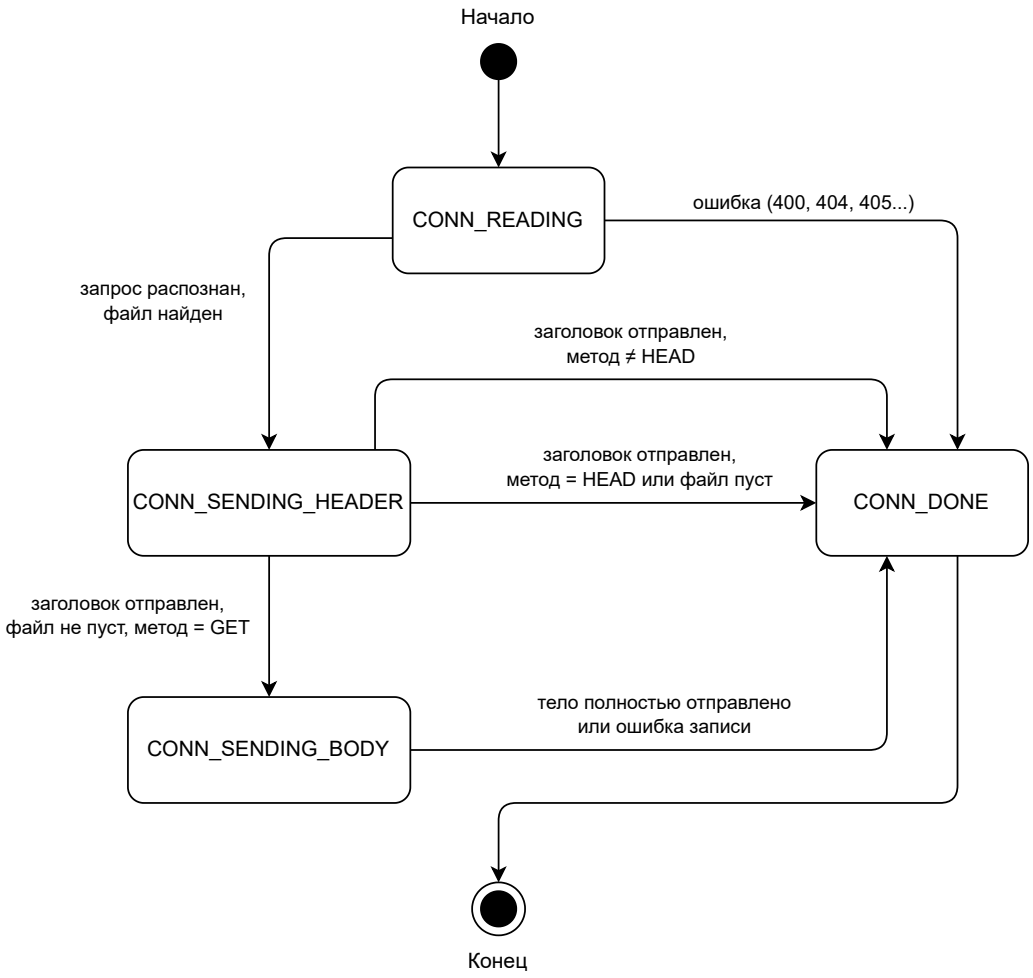

Рисунок 2.1 – Диаграмма состояний жизненного цикла клиентского соединения

2.3 Алгоритм главного потока

Главный поток выполняет следующие шаги:

1) Создать слушающий сокет.

2) Запустить пул рабочих потоков.

3) Запустить цикл, пока работа сервера не завершена:

   3.1) Если новое соединение установлено:

   — Получить IP и порт клиента;

   — Назначить соединение рабочему потоку.

   3.2) Конец цикла.

4) Остановить пул рабочих потоков.

5) Закрыть слушающий сокет.

2.4 Алгоритм рабочего потока

Рабочий поток реализует цикл обработки событий:

1) Создать массив соединений и pipe для приема новых сокетов.

2) Цикл до завершения работы потока:

   2.1) Сформировать набор дескрипторов для *poll()*.

   2.2) Вызвать *poll()*.

   2.3) Если пришли новые соединения (события на pipe) – добавить их в массив в состоянии *CONN_READING*.

   2.4) Для каждого активного сокета:

   — При *POLLIN* и состоянии *READING* – прочитать и проанализировать запрос, при успехе перейти к подготовке ответа.

   — При *POLLOUT* и состоянии *SENDING_HEADER* – отправить заголовок, при завершении перейти к отправке тела.

   — При *POLLOUT* и состоянии *SENDING_BODY* – отправить фрагмент файла.

   — При ошибках или завершении – перевести соединение состояние в *CONN_DONE*.

   2.5) Удалить все соединения в состоянии *CONN_DONE*.

3) При завершении – освободить все ресурсы.

2.5 Механизм передачи соединений через pipe

Для назначения нового соединения рабочему потоку используется *round-robin* распределение:

1. Главный поток формирует структуру *conn_msg = {fd, ip, port}*.

2. Выбирается следующий рабочий поток по индексу.

3. Структура записывается в pipe этого потока.

4. Рабочий поток получает событие *POLLIN* на своем pipe и добавляет сокет в свой набор.

Этот механизм обеспечивает потокобезопасную и неблокирующую передачу дескрипторов без использования мьютексов в основном пути.

3 Технологическая часть

В данном разделе описывается техническая реализация HTTP-сервера: выбор языка программирования, поддерживаемые типы запросов, а также ключевые особенности реализации, обеспечивающие безопасность, производительность и корректность работы.

3.1 Выбор языка программирования

Для реализации сервера был выбран язык **C** по следующим причинам:

— Прямой доступ к системным вызовам POSIX (*socket*, *poll*, *sendfile*, *open* и др.), что необходимо для точного контроля над сетевым и файловым вводом-выводом.

— Минимальные накладные расходы: отсутствие автоматической сборки мусора и runtime-библиотек позволяет достичь высокой производительности и предсказуемого поведения.

— Портативность: код компилируется на любой POSIX-совместимой системе (Linux, BSD, macOS) с помощью стандартного компилятора *gcc*.

Для сборки используется система *make*, что обеспечивает кроссплатформенность и воспроизводимость сборки.

3.2 Поддерживаемые HTTP-методы и статусы

Сервер поддерживает два метода протокола HTTP/1.1:

— *GET* – запрашивает полное содержимое ресурса (заголовки с телом).

— *HEAD* – запрашивает только метаданные (заголовки без тела).

При получении любого другого метода, сервер возвращает код состояния: 405 (*Method Not Allowed*).

Поддерживаемые коды ответа:

— 200 (*OK*) – успешная обработка запроса;

— 400 (*Bad Request*) – ошибка формата запроса (слишком длинный путь и т.п.);

— 403 (*Forbidden*) – доступ запрещён;

— 404 (*Not Found*) – файл не найден;

— 413 (*Payload Too Large*) – запрошенный файл превышает лимит в 128 МБ;

— 500 (*Internal Server Error*) – ошибка формирования заголовка.

3.3 Поддерживаемые типы файлов

Сервер автоматически определяет MIME-тип отдаваемого файла по расширению и указывает его в заголовке *Content-Type*. Поддерживаются следующие типы:

— **Веб-контент**: *.html*, *.htm*, *.css*, *.js*, *.json*;

— **Изображения**: *.png*, *.jpg*, *.jpeg*, *.gif*, *.ico*, *.webp*;

— **Видео**: *.mp4*, *.webm*, *.ogg*, *.ogv*, *.avi*, *.mov*, *.mkv*;

— **Документы**: *.txt*, *.pdf*;

— **Прочее**: все остальные файлы отдаются с типом *application/octet-stream*.

Это позволяет корректно отображать веб-страницы с изображениями, стилями, скриптами и мультимедиа.

3.3.1 Обработка больших файлов

Файлы размером до 128 МБ передаются по частям с использованием системного вызова *sendfile()*, который выполняет копирование данных напрямую из ядра (без копирования в пользовательское пространство – zero-copy). Отправка производится асинхронно: после каждой попытки вызывается *poll()*, что предотвращает блокировку потока.

3.3.2 Безопасность

Реализована защита от атак типа *Path Traversal*:

— Все входящие пути нормализуются через *realpath()*, что устраняет последовательности *../* и символические ссылки.

— Проверяется, что результирующий путь находится строго внутри корневой директории.

— Запросы к директориям (без *index.html*) возвращают 403 (*Forbidden*).

3.3.3 Мультиплексирование и пул потоков

Архитектура основана на комбинации:

— **Пула потоков** – фиксированное число рабочих потоков (настраивается при запуске).

— **Мультиплексирования через *poll()*** – каждый поток одновременно обслуживает до 1024 соединений.

Новые соединения распределяются между потоками по алгоритму *round-robin* через анонимные каналы (pipes), что исключает конкуренцию за мьютексы в основном пути данных.

4 Исследовательская часть

В данном разделе представлены результаты нагрузочного тестирования разработанного HTTP-сервера. Целью исследования является оценка:

— максимального количества одновременно обслуживаемых соединений;

— скорости отдачи данных по одному и множеству соединений;

— общей пропускной способности сервера.

4.1 Характеристики оборудования

Характеристики оборудования, на котором проводилось исследование приведены ниже:

— **ОС**: Ubuntu 22.04.1 LTS Release: 22.04,

— **ЦП**: AMD Ryzen 7 4800H with Radeon Graphics, 2900 МГц, ядер: 8,

— **ОЗУ**: 16 Гб, 3200 МГц,

— **ПЗУ**: INTEL SSDPEKNW512G8H 476,9Гб.

4.2 Результаты тестирования

Для генерации нагрузки использовался инструмент `wrk` версии 4.2.0, а также `curl` версии 8.5.0. Тестовые файлы:

— *index.html* – HTML-файл (5.3 КБ) для тестирования максимального количества сетевых соединений;

— *dataset.bin* – бинарный файл размером 100 МБ для тестирования отдачи данных по каждому сетевому соединению и совокупному.

— *Набор бинарных файлов*: временные бинарные файлы создаваемые, соответствующим тестирующим скриптом, имеющие разные размерности:

1) **10 Кб**;
2) **100 Кб**;
3) **1 Мб**;
4) **10 Мб**;
5) **100 Мб**.

4.2.1 Тестирование максимального количества обслуживаемых сетевых соединений

Для тестирования максимального количества обслуживаемых сетевых соединений использовалась утилита `wrk`. Сервер был запущен с помощью команды:

$$./app\ ./htdocs\ 8080\ threads, \text{ где } threads = 1, 2, 4, 8, 16.$$

Для каждого запуска с разным количеством рабочих потоков в пуле, выполнялся запуск утилиты `wrk` следующим образом:

$$wrk\ \text{-}t\mathbf{threads}\ \text{-}c\mathbf{connections}\ \text{-}d30s\ http://localhost:8080/index.html,$$

где **threads** соответствует количеству рабочих потоков в текущем запуске сервера, а **connections** $= 10, 50, 100, 200, 500, 1000$ для каждого количества рабочих потоков в пуле, кроме случая, когда **threads** $= 16$, в этом случае **connections** перебираются так: $16, 50, 100, 200, 500, 1000$.

Однако, в процессе тестирования возникла ошибка сервера: *Too many open files* – которая обозначает, что процесс превысил лимит на количество открытых файловых дескрипторов. Для решения этой проблемы, временно был вручную увеличен лимит с помощью [4]:

$$\text{ulimit -n } 4096.$$

Полученные результаты записаны в таблицах 4.1 – 4.2. На графиках 4.1 - 4.3 представлены зависимости количества обрабатываемых запросов в секунду, совокупного объема переданных данных, средней задержки сервера от числа одновременных соединений при тестовом сценарии с легковесным файлом (*index.html*) соответственно.

Таблица 4.1 – Результаты тестирования максимального количества обслуживаемых сетевых соединений (часть 1)

Количество потоков	**Количество соединений**	**Средняя задержка**, (мкс)	**Количество обработанных запросов**, ($\frac{10^3}{c}$)	**Объем отправленных данных**, (Мб)
1	10	290.28	20.51	104.47
1	50	1 450	21.09	107.41
1	100	2 820	20.72	105.52
1	200	5 320	20.17	102.66
1	500	1 364	19.43	98.80
1	1000	2 585	19.78	100.65
2	10	165.39	16.87	171.84
2	50	686.94	19.24	195.92
2	100	1 390	19.04	193.85
2	200	2 450	19.56	199.09
2	500	6 090	18.97	192.84
2	1000	1 243	18.50	187.94
4	10	123.79	8.41	171.35
4	50	461.92	14.63	297.63
4	100	743.84	16.91	344.49
4	200	1 330	17.26	351.47
4	500	3 170	17.03	346.44
4	1000	6 550	16.35	332.04

Таблица 4.2 – Результаты тестирования максимального количества обслуживаемых сетевых соединений (часть 2)

Количество потоков	**Количество соединений**	**Средняя задержка**, (мкс)	**Количество обработанных запросов**, ($\frac{10^3}{\text{с}}$)	**Объем отправленных данных**, (Мб)
8	10	72.94	4.17	169.68
8	50	457.15	8.69	353.95
8	100	663.39	10.56	430.19
8	200	1 080	12.64	514.78
8	500	2 410	12.65	514.75
8	1000	4 730	12.30	499.79
16	16	160.92	3.98	323.19
16	50	408.43	5.33	433.69
16	100	761.43	5.95	484.56
16	200	1 510	6.07	494.44
16	500	3 790	5.92	481.95
16	1000	7 370	5.65	459.24

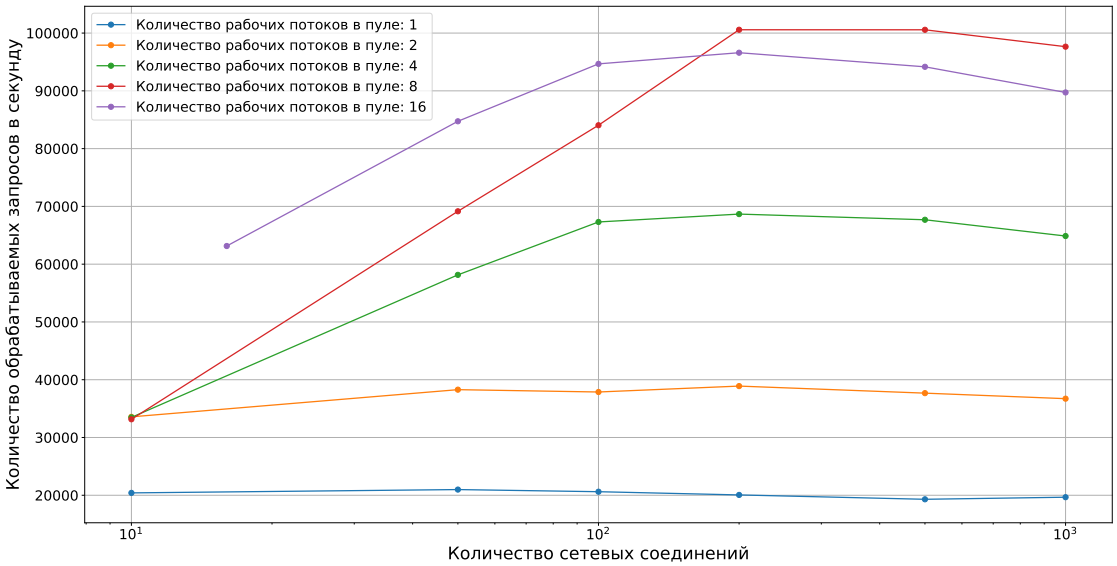

Рисунок 4.1 – Зависимость количества обрабатываемых запросов в секунду от числа одновременных соединений при отдаче index.html

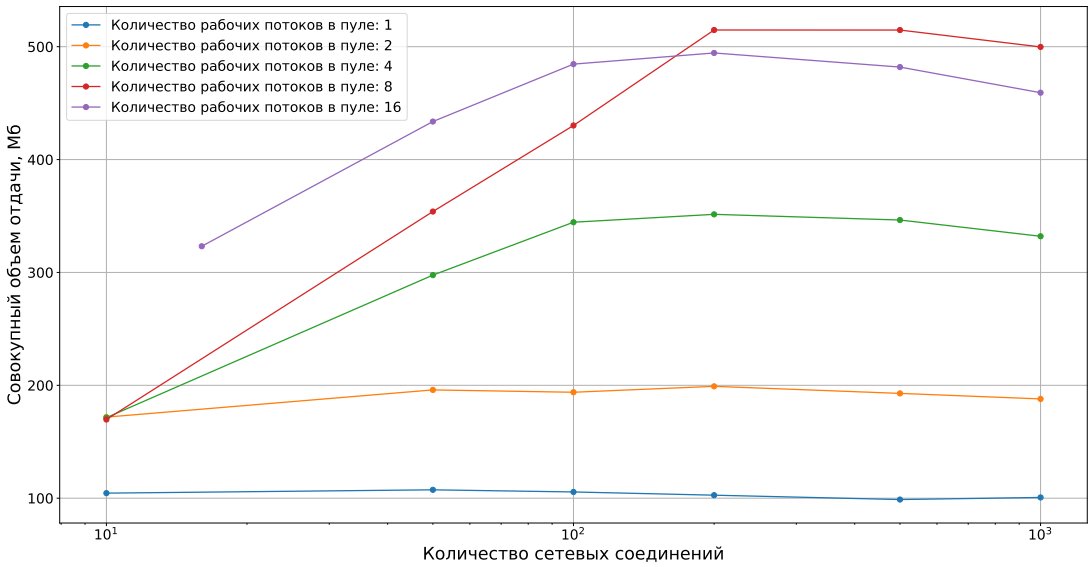

Рисунок 4.2 – Зависимость совокупного объема переданных данных от числа одновременных соединений при отдаче index.html

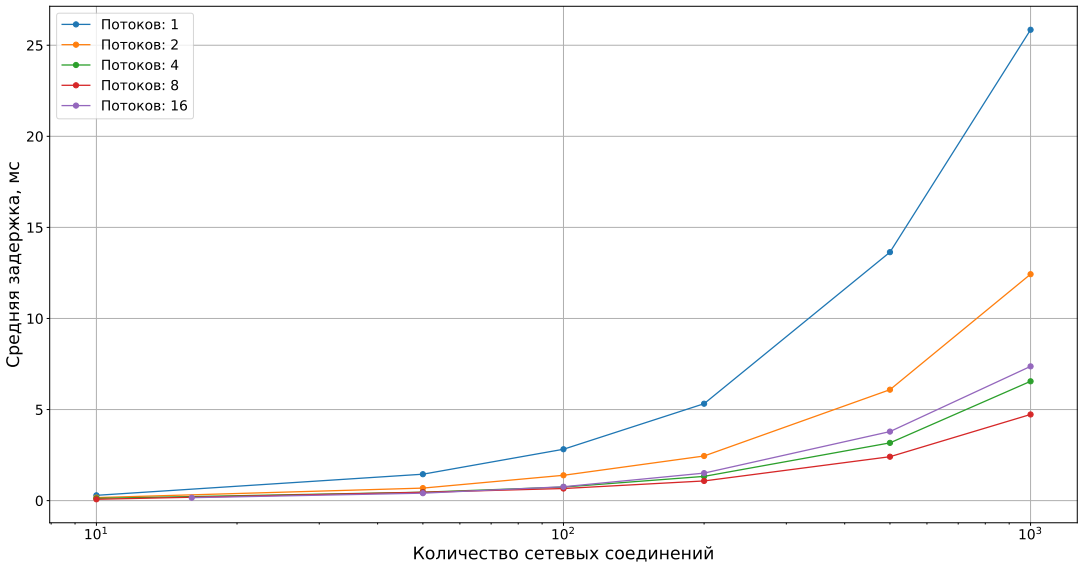

Рисунок 4.3 – Зависимость средней задержки сервера от числа одновременных соединений при отдаче index.html

4.2.2 Отдача большого файла

Для оценки передачи объемных данных сервер был запущен с оптимальным количеством потоков – 8:

./app ./htdocs 8080 8.

Для тестирования отдачи данных по одному сетевому соединению использовался файл *dataset.bin* размером 100 Мб с помощью команды:

time (curl -o /dev/null http://localhost:8080/resources/test/dataset.bin & wait),

которая была последовательно выполнена 7 раз. Как видно из таблицы 4.3, первые два замера демонстрируют сниженную пропускную способность (2400 – 2700 Мб/с), в то время как начиная с третьего запуска производительность стабилизируется на уровне 3200 – 3240 Мб/с.

Таблица 4.3 – Тестирование отдачи данных по одному сетевому соединению

№ запуска	**Время**, (мс)	**Пропускная способность**, (Мб/с)
1	56	2 444
2	52	2 676
3	46	3 236
4	54	2 547
5	46	3 221
6	46	3 237
7	46	3 227
Среднее	**49.42**	**2 941.14**

На рисунке 4.4 представлен график зависимости пропускной способности (Мб/с) сервера от номера запуска команды *curl*, а также изображена средняя пропускная способность сервера при отдаче файла объемом 100 Мб по одному сетевому соединению.

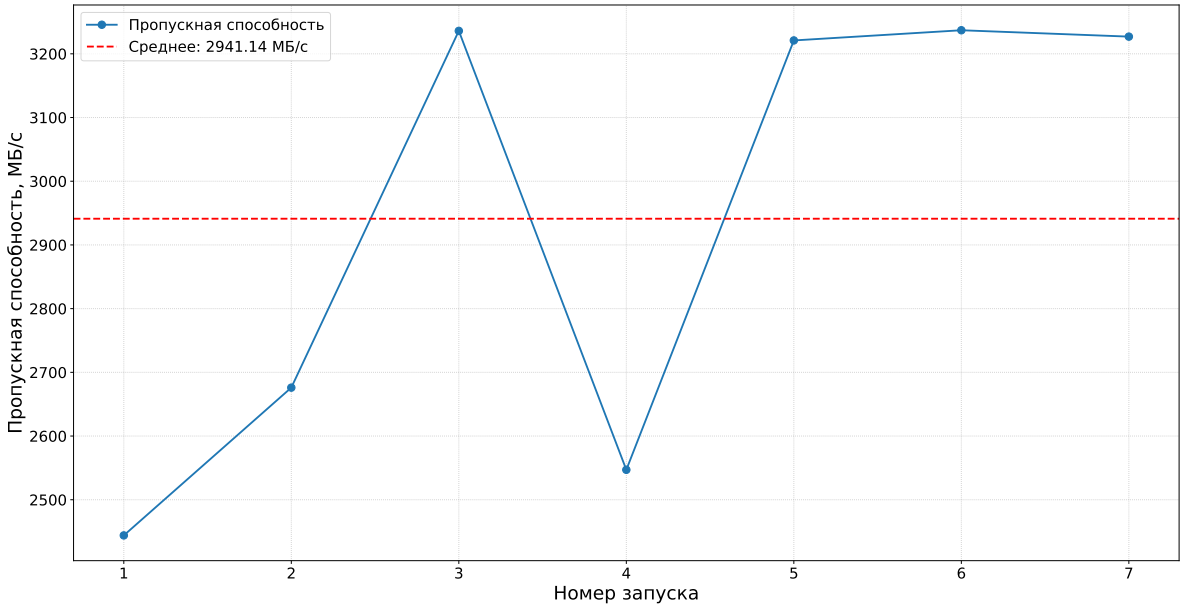

Рисунок 4.4 – Зависимость пропускной способности сервера от номера запуска команды *curl* при передаче 100 МБ по одному сетевому соединению

На рисунке 4.5 представлен график зависимости времени выполнения команды *curl* (мс) от номера ее запуска, а также изображено среднее время выполнения этой команды при отдаче файла объемом 100 Мб по одному сетевому соединению.

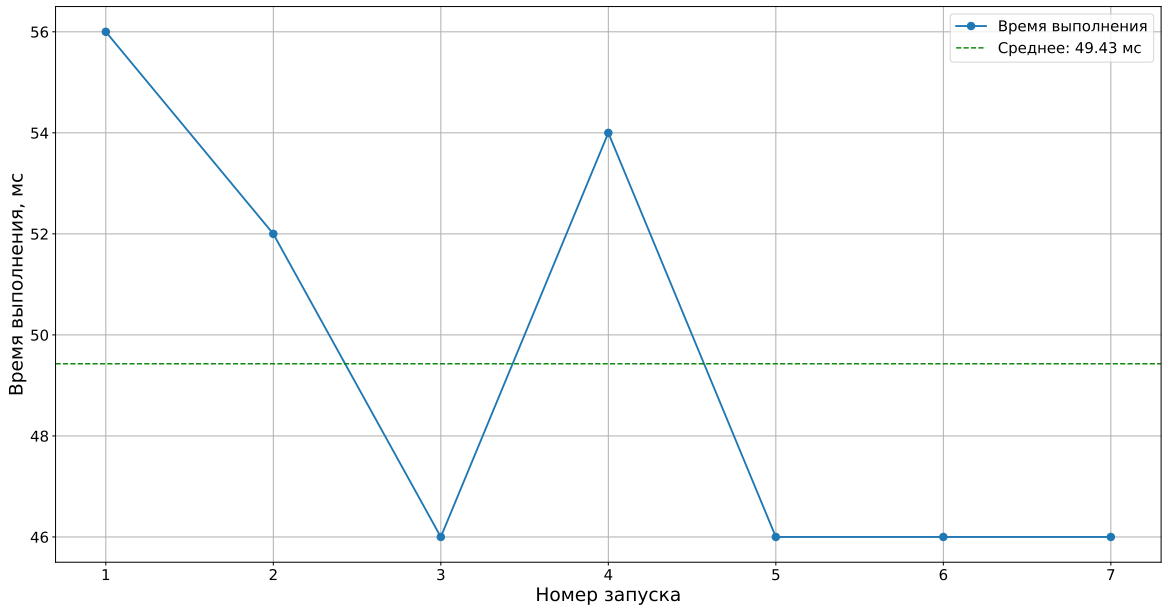

Рисунок 4.5 – Зависимость времени выполнения команды *curl* от номера ее запуска при передаче 100 МБ по одному сетевому соединению

Для тестирования отдачи данных по нескольким параллельным сетевым соединениям сервер также был запущен с оптимальным количеством рабочих поток в пуле (8), а для создания параллельных соединений использовался *bash*-скрипт 4.1 [5]. Полученные результаты представлены в таблице 4.4.

Таблица 4.4 – Тестирование отдачи данных по нескольким параллельным сетевым соединениям

Количество сетевых соединений	**Пропускная способность на соединение**, (Мб/с)	**Совокупная пропускная способность**, (Мб/с)	**Время**, (мс)
1	5 108.11	5 108.11	20
2	6 358.69	12 717.38	16
4	6 201.96	24 807.85	16
8	7 260.14	58 081.14	14
16	5 947.57	95 161.15	17
32	3 842.48	122 959.52	26
64	2 252.20	144 141.37	44
128	1 198.97	153 468.88	83
256	575.49	147 327.97	174
512	260.38	133 316.38	384
1 024	145.50	148 992.23	687
Среднее	**3559.22**	**95 098.36**	**134.63**

По полученным данным были построены графики зависимости средней скорости отдачи данных на соединение, совокупной пропускной способности сервера и времени передачи от количества сетевых соединений при отдаче файла объемом 100 Мб по нескольким параллельным сетевым соединениям соответственно (рисунки 4.6 – 4.8).

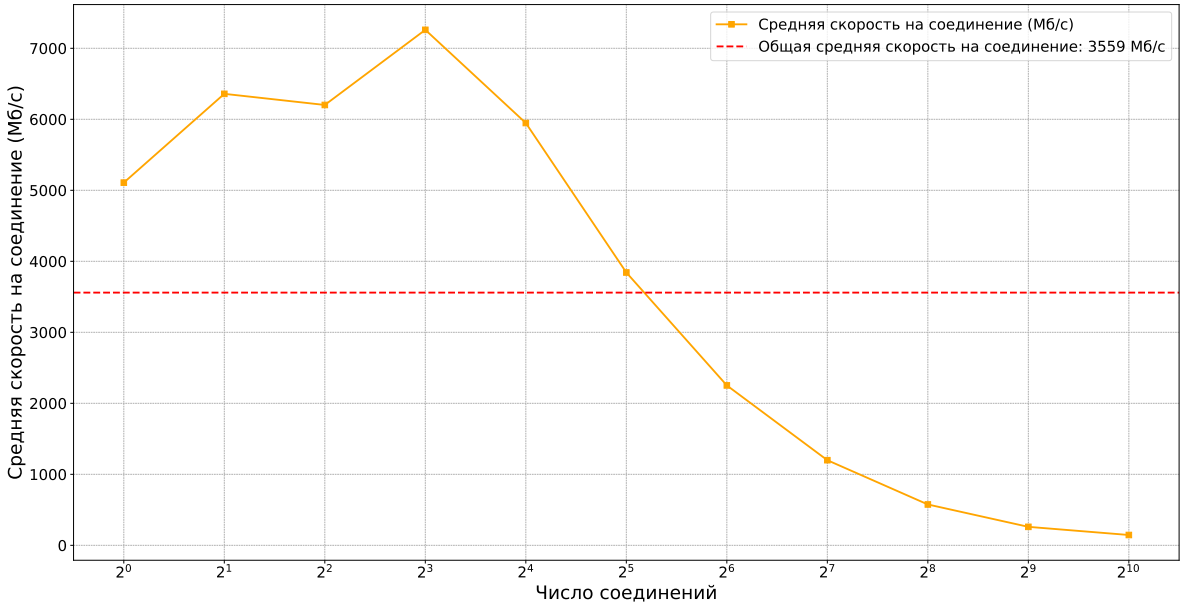

Рисунок 4.6 – Зависимость средней скорости отдачи данных на соединение от количества сетевых соединений

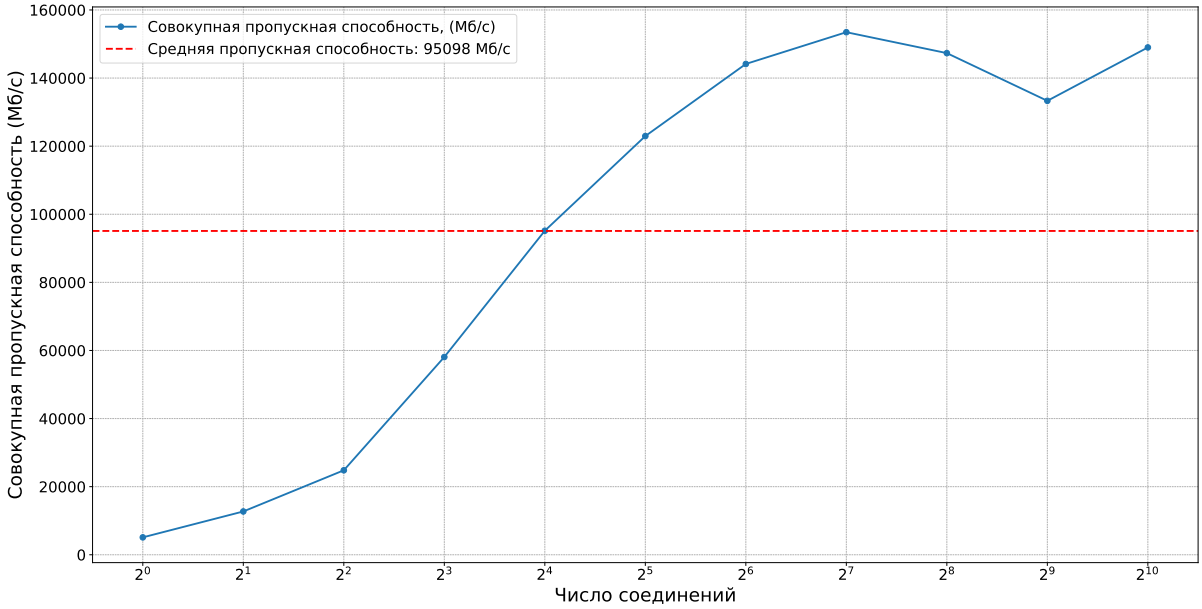

Рисунок 4.7 – Зависимость совокупной пропускной способности сервера от количества сетевых соединений

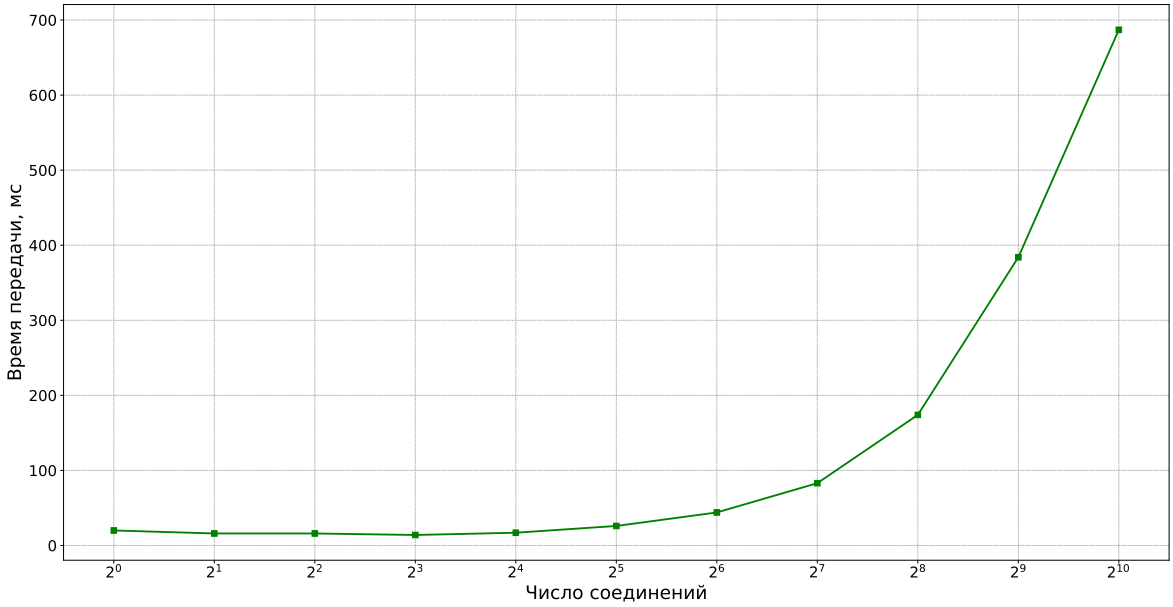

Рисунок 4.8 – Зависимость времени передачи от количества сетевых соединений

4.2.3 Исследование влияния объема данных на пропускную способность сервера

В предыдущих тестах было установлено, что оптимальное количество рабочих потоков в пуле для используемого оборудования (AMD Ryzen 7 4800H, 8 ядер) составляет 8 потоков (см. таблицы 4.1, 4.2 и графики 4.1 – 4.3).

Для оценки влияния объема передаваемых данных на производительность было проведено нагрузочное тестирование: сервер запускался с 8 рабочими потоками, а клиент последовательно запрашивал файлы различного размера (от 10 КБ до 100 МБ) по одному сетевому соединению. Каждый замер повторялся 100 раз (листинг 4.2).

Результаты измерений приведены в таблице 4.5.

Таблица 4.5 – Средние значения времени и пропускной способности при передаче файлов разного размера (1 соединение, 8 потоков)

Размер файла	**Среднее время**, (мс)	**Пропускная способность**, (Мб/с)
10 Кб	0.511	20.767
100 Кб	0.580	184.623
1 Мб	1.134	999.074
10 Мб	4.572	2 366.636
100 Мб	27.815	3 648.628

На графиках 4.9 – 4.10 изображены зависимости среднего времени передачи и средней совокупной пропускной способности сервера от размера передаваемого файла при одном сетевом соединении и 8 рабочих потоках в пуле.

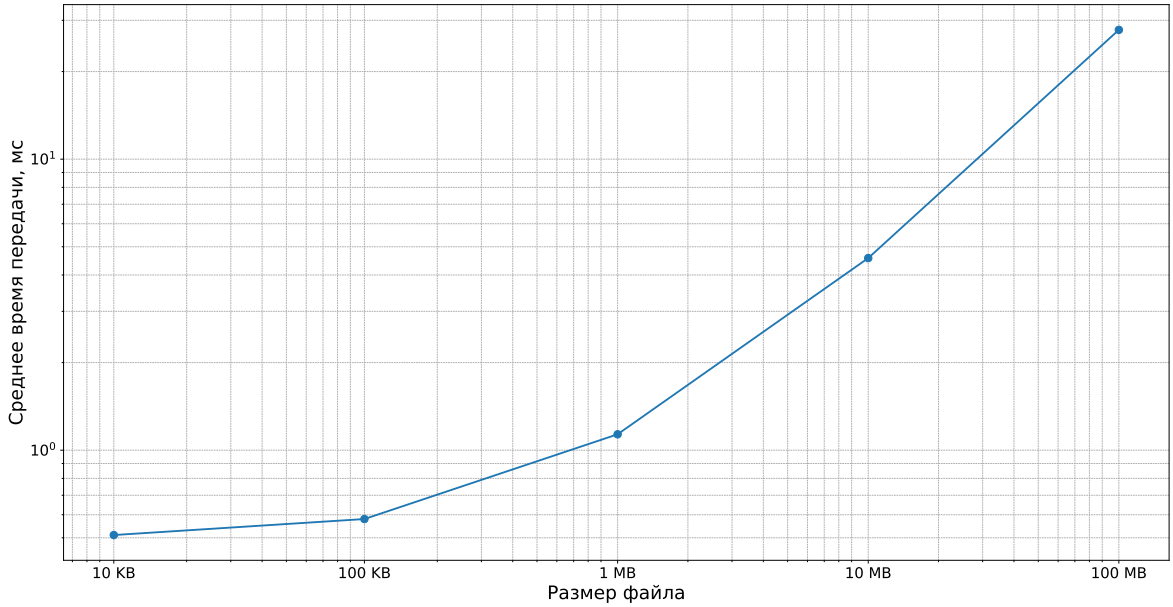

Рисунок 4.9 – Зависимость времени передачи от размера файла

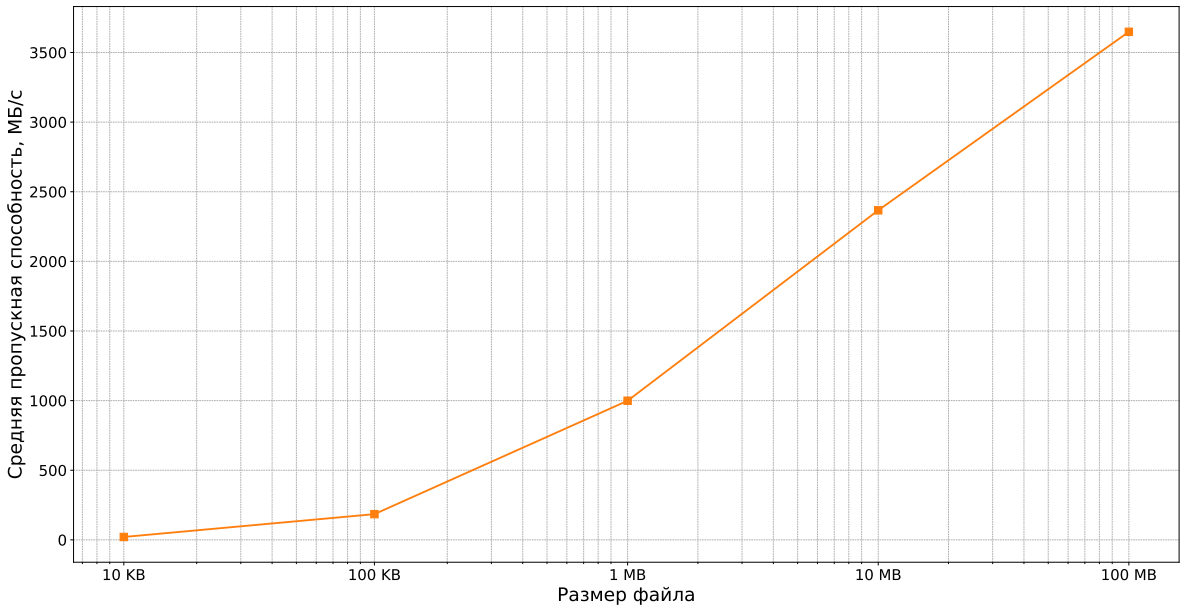

Рисунок 4.10 – Зависимость пропускной способности от размера файла

4.3 Вывод

На основе проведенного нагрузочного тестирования сервера сделаны следующие выводы:

— При увеличении количества рабочих потоков с 1 до 8 наблюдается рост совокупной пропускной способности (в Мб/с) и числа обработанных запросов в секунду.

— Однако дальнейшее увеличение числа потоков до 16 приводит к снижению производительности.

— Сервер демонстрирует максимальную эффективность при 8 потоках, что соответствует числу физических ядер ЦП.

— При передаче 100 Мб по одному соединению средняя скорость – 2941 Мб/с, а в пике – до 3237 Мб/с. Такие значения близки к пропускной способности ПЗУ (INTEL SSDPEKNW512G8: до 2000-3500 Мб/с при последовательном чтении).

— При параллельной передаче 100-Мб файла по множеству соединений, совокупная пропускная способность растет до 153 Гб/с при 128 соединениях, но начиная с 256 соединений начинает снижаться (147 Гб/с при 256, 133 Гб/с при 512).

— Результаты исследования влияния объема данных на пропускную способность сервера показывают, что время передачи файла линейно зависит от его размера, что подтверждает отсутствие нелинейных эффектов в реализации сервера. Пропускная способность стабилизируется на уровне 3650 МБ/с для файлов объёмом 100 МБ, что близко к предельной скорости локального SSD и сетевого стека ОС в конфигурации localhost.

ЗАКЛЮЧЕНИЕ

Цель, которая была поставлена в начале работы, была достигнута: разработан HTTP-сервер для отдачи статического содержимого, построенного на архитектуре пула потоков с мультиплексированием сетевых соединений с использованием системного вызова `poll()`.

Решены все поставленные задачи:

— Предусмотрена поддержка запросов GET и HEAD, поддержка статусов 200, 403, 404.

— Предусмотрена возможность ответа сервера на неподдерживаемые запросы статусом 405.

— Обеспечена корректная передача файлов размером до 128 Мбайт.

— Реализовано мультиплексирование – каждый процесс или поток должен отдавать данные по нескольким сетевым соединениям.

— Сервер по умолчанию возвращает HTML-страницу на выбранную тему с CSS-стилем.

— Реализована запись информации о событиях в журнал (лог).

— Учтены минимальные требования к безопасности серверов статического содержимого.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Компьютерные сети : учебное пособие / А. Н. Алексахин, С. А. Алексахина, А. В. Батищев [и др.] ; под редакцией А. М. Нечаева. — Москва : Университет «Синергия», 2023. — 312 с. — ISBN 978-5-4257-0558-7. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/156712.html (дата обращения: 18.12.2025). — Режим доступа: для авторизир. пользователей

2. Кобылянский, В. Г. Локальные компьютерные сети. Базовый курс : учебное пособие / В. Г. Кобылянский. — Новосибирск : Новосибирский государственный технический университет, 2023. — 127 с. — ISBN 978-5-7782-4894-6. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/155418.html (дата обращения: 18.12.2025). — Режим доступа: для авторизир. пользователей

3. pipe(2) – Linux manual page [Электронный ресурс]. – URL: `https://man7.org/linux/man-pages/man2/pipe.2.html` (дата обращения: 18.12.2025)

4. ulimit(3) – Linux manual page [Электронный ресурс]. – URL: `https://man7.org/linux/man-pages/man3/ulimit.3.html` (дата обращения: 18.12.2025)

5. Bash Features [Электронный ресурс]. – URL: `https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html` (дата обращения: 18.12.2025)

6. Matplotlib documentation [Электронный ресурс]. – URL: `https://matplotlib.org/stable/index.html` (дата обращения: 18.12.2025)

Приложение А

Листинг 4.1 – Скрипт для измерения совокупной пропускной способности

```
#!/bin/bash
export LC_NUMERIC=C

URL="http://localhost:8080/resources/test/dataset.bin"
SIZES=(1 2 4 8 16 32 64 128 256 512 1024)
OUTPUT_FILE="throughput_results.csv"

if [[ ! -f "$OUTPUT_FILE" ]]; then
    echo 'connections,"aver transfer","transfer",time' >
       "$OUTPUT_FILE"
fi

echo "Тестирование совокупной пропускной способности"
echo "Файл: 100 МБ, Сервер: 8 потоков"
echo

for n in "${SIZES[@]}"; do
    echo ">>> Запуск с $n соединениями..."

    start_time=$(date +%s.%N)
    for ((i=0; i<n; i++)); do
        curl -o /dev/null -s "$URL" &
    done
    wait
    end_time=$(date +%s.%N)

    total_time=$(echo "$end_time - $start_time" | bc -l)
    time_rounded=$(printf "%.0f" "$total_time")
    total_mb=$((n * 100))
    throughput=$(echo "scale=2; $total_mb / $total_time" | bc)
    avg_per_conn=$(echo "scale=2; $throughput / $n" | bc)

    echo "  Время: $(printf "%.3f" $total_time) c"
    echo "  Совокупная пропускная способность: $throughput МБ/c"
    echo "  На соединение (среднее): $avg_per_conn МБ/c"
done
```

Листинг 4.2 – Скрипт для измерения зависимости пропускной способности сервера от объема отдаваемых данных

```
#!/bin/bash

set -e

if [ "$#" -lt 4 ]; then
    echo "Использование: $0 <сервер> <htdocs> <порт> <потоки> [повторы]"
    exit 1
fi

SERVER="$1"
HTDOCS="$2"
PORT="$3"
THREADS="$4"
NUM_RUNS="${5:-5}"

RESOURCES_DIR="$HTDOCS/resources"
TEMP_DIR="$RESOURCES_DIR/temp_filesize_test"

OUTPUT_FILE="results.csv"

if [[ ! -f "$OUTPUT_FILE" ]]; then
    echo 'name,size_bytes,run,time_ms,throughput_mbs' > "$OUTPUT_FILE"
fi

cleanup() {
    echo "Выполняется очистка..."

    if [ -n "${SERVER_PID-}" ]; then
        kill "$SERVER_PID" 2>/dev/null || true
        wait "$SERVER_PID" 2>/dev/null || true
    fi

    if [ -d "$TEMP_DIR" ]; then
        rm -rf "$TEMP_DIR"
    fi
    echo "Очистка завершена."
}
```

```
38
39 trap cleanup EXIT
40
41 if command -v lsof >/dev/null 2>&1; then
42     if lsof -i :"$PORT" > /dev/null 2>&1; then
43         echo "Порт $PORT занят. Завершите другие процессы
               (например, pkill -f app)."
44         exit 1
45     fi
46 else
47     echo "Предупреждение: lsof не найден, проверка порта
           пропущена."
48 fi
49
50 mkdir -p "$TEMP_DIR"
51
52 FILES=(
53     "10KB:10"
54     "100KB:100"
55     "1MB:1024"
56     "10MB:10240"
57     "100MB:102400"
58 )
59
60 wait_for_server() {
61     echo "Ожидание запуска сервера на порту $PORT..."
62     for i in {1..30}; do
63         if curl -s --max-time 1 "http://localhost:$PORT/"
               >/dev/null; then
64             echo "Сервер готов."
65             return 0
66         fi
67         sleep 0.2
68     done
69     echo "Ошибка: сервер не запустился на порту $PORT"
70     exit 1
71 }
72
73 echo "Запуск сервера: $SERVER $HTDOCS $PORT $THREADS"
74 "$SERVER" "$HTDOCS" "$PORT" "$THREADS" &
75 SERVER_PID=$!
```

```
76
77 wait_for_server
78
79 echo "filename,size_bytes,run,time_ms,throughput_MBs"
80
81 for file in "${FILES[@]}"; do
82     name="${file%%:*}"
83     size_kb="${file##*:}"
84     filepath="$TEMP_DIR/file_$name.bin"
85     url="http://localhost:$PORT/\
86     resources/temp_filesize_test/\
87     file_$name.bin"
88     size_bytes=$((size_kb * 1024))
89
90     dd if=/dev/urandom of="$filepath" bs=1024 count="$size_kb"
          status=none
91
92     for run in $(seq 1 $NUM_RUNS); do
93         time_sec=$(curl -o /dev/null -s -w "%{time_total}"
              "$url")
94
95         time_ms=$(awk "BEGIN {printf \"%.3f\", $time_sec *
              1000}")
96         if (( $(echo "$time_sec <= 0" | bc -l) )); then
97             throughput_mbs="inf"
98         else
99             throughput_mbs=$(awk "BEGIN {printf \"%.3f\",
                  ($size_bytes / 1024 / 1024) / ($time_sec)}")
100        fi
101        echo "$name,$size_bytes,$run,$time_ms,$throughput_mbs"
102        echo "$name,$size_bytes,$run,$time_ms,$throughput_mbs"
              >> "$OUTPUT_FILE"
103    done
104 done
105
106 AVG_FILE="results_avg.csv"
107 echo 'name,size_bytes,avg_time_ms,avg_throughput_mbs' >
      "$AVG_FILE"
108
109 awk -F, '
110 NR > 1 {
```

```
111     key = $1 "," $2
112     sum_time[key] += $4
113     sum_thr[key] += $5
114     count[key]++
115 }
116 END {
117     for (k in sum_time) {
118         avg_time = sum_time[k] / count[k]
119         avg_thr = sum_thr[k] / count[k]
120         printf "%s,%.3f,%.3f\n", k, avg_time, avg_thr
121     }
122 }' "$OUTPUT_FILE" >> "$AVG_FILE"
```