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 ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Статик сервер»

Студент ИУ7-75Б _____ **Богаченко А. Е.**

Руководитель курсового проекта _____ **Яковидис Н. О.**

2023 г.

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)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7

_____ И. В. Рудаков

«____» _____ 20____г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине Компьютерные сети

Студент группы ИУ7-75Б

Богаченко Артём Евгеньевич
(Фамилия, имя, отчество)

Тема курсовой работы статик сервер

Направленность КП (учебный, исследовательский, практический, производственный, др.)
учебный

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения работы: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание реализовать статик сервер на архитектуре thread pool + pselect для отдачи контента с диска

Оформление курсовой работы:
Расчетно-пояснительная записка на 20-30 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
Презентация на 8-10 слайдах.

Дата выдачи задания «____» _____ 20____г.

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

Студент ИУ7-75Б _____ **Богаченко А. Е.**
(Группа) (Подпись, дата) (Фамилия И. О.)

РЕФЕРАТ

Расчетно-пояснительная записка 30 с., 7 рис., 1 табл., 8 ист.

В работе представлена реализация статик сервера с использованием архитектуры threadpool + pselect.

Изучены основные принципы работы статических серверов, проанализированы существующие архитектуры. Было проведено нагрузочное тестирование с альтернативой в виде nginx.

КЛЮЧЕВЫЕ СЛОВА

threadpool, socker, nginx, static server

Содержание

ВВЕДЕНИЕ

В настоящее время компьютерные сети играют все более важную роль в нашей повседневной жизни. Статические серверы – одна из самых распространённых и актуальных технологий в области сетевых взаимодействий. Они позволяют обеспечить эффективную и стабильную работу веб-приложений, сайтов и других сервисов, предоставляя контент пользователям через сеть. В связи с быстрым развитием интернета и все возрастающим спросом на онлайн-сервисы, понимание основ и принципов работы статических серверов является существенным для специалистов в области сетевых технологий.

В данной курсовой работе мы сосредоточимся на изучении статических серверов, их важности и показателях распространённости, а также на приобретении навыков настройки и обслуживания такой системы.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу по дисциплине «Компьютерные сети» требуется разработать статик сервер, позволяющий отдавать различный контент с диска.

Для выполнения задания требуется решить следующие задачи:

1) изучить основные принципы работы статических серверов и их роль в обеспечении эффективной отдачи контента;
2) проанализировать существующие архитектуры статических серверов и выявить их отличительные особенности;
3) изучить возможности выбранных технологий и инструментов;
4) разработать статик сервер;
5) провести исследование разработанного ПО.

По варианту задания статик сервер должен обладать следующими требованиями:

1) поддержка запросов `GET` и `HEAD` (поддержка статусов 200, 403, 404);
2) ответ на Неподдерживаемые запросы статусом 405;
3) корректное выставление content type в зависимости от типа файла (поддержка .html, .css, .js, .png, .jpg, .jpeg, .swf, .gif);
4) корректная передача файлов размером в 100мб;
5) сервер по умолчанию должен возвращать html-страницу на выбранную тему с css-стилем;
6) архитектура thread pool + pselect.

Должны быть учтены минимальные требования к безопасности статик-серверов, в том числе и проверка на выход за директорию /root/. Должен быть реализовать логгер. В качестве инструментов разработки должен быть Использован язык C без сторонних библиотек. По результатам разработки должно быть проведено нагрузочное тестирование.

Для разработки и тестирования данной работы используется портативный компьютер `Huawei Matebook X Pro` и операционная система `Linux` с дистрибутивом `Kali Linux`.

1.2 Статик сервер

Статический сервер – это программа или сервис, обрабатывающий запросы на получение статических файлов (например, HTML, CSS, JavaScript, изоб-

ражения). Он не выполняет динамическую обработку данных и предоставляет файлы клиентам без изменений. Основная задача статического сервера – быстрая и эффективная отдача контента, минимизация задержек и обеспечение безопасности

1.3 Архитектура

Традиционно существует две конкурирующие серверные архитектуры: одна основана на потоках, другая – на событиях. Со временем появились более сложные варианты, иногда сочетающие оба подхода. В течение долгого времени велся спор о том, являются ли потоки или события лучшей основой для высокопроизводительных веб-серверов [1]. Спустя более чем десятилетие этот аргумент усилился благодаря новым проблемам масштабируемости и тенденции к использованию многоядерных процессоров.

1.3.1 Поточная архитектура

Потоковый подход в основном связывает каждое входящее соединение с отдельным потоком (соответственно процессом). Таким образом, синхронная блокировка ввода-вывода является естественным способом обработки ввода-вывода. Это распространённый подход, который хорошо поддерживается многими языками программирования. Это также приводит к созданию простой модели программирования, поскольку все задачи, необходимые для обработки запросов, могут быть запрограммированы последовательно. Более того, он обеспечивает простую мысленную абстракцию, изолируя запросы и скрывая параллелизм. Реальный параллелизм достигается за счёт одновременного использования нескольких потоков/процессов.

Концептуально многопроцессные и многопоточные архитектуры разделяют одни и те же принципы: каждое новое соединение обрабатывается специальным действием.

1.3.2 Процессная архитектура

Традиционным подходом к сетевым серверам на базе UNIX является модель «процесс для каждого соединения» [2], в которой используется выделенный процесс для обработки соединения. Эта модель также использовалась для первого HTTP-сервера CERN httpd [3]. Из-за особенностей процессов они быстро изолируют разные запросы, поскольку они не используют общую память. Поскольку создание процессов является довольно тяжеловесной структурой, оно является дорогостоящей операцией, и серверы часто используют страте-

гию, называемую предварительным разветвлением. При использовании предварительного разветвления основной серверный процесс упреждающе разветвляет несколько процессов-обработчиков при запуске. Часто дескриптор сокета (потокобезопасный) используется всеми процессами, и каждый процесс блокирует новое соединение, обрабатывает его, а затем ожидает следующего соединения.

Некоторые многопроцессные серверы также измеряют нагрузку и при необходимости создают дополнительные запросы. Однако важно отметить, что тяжёлая структура процесса ограничивает максимальное количество одновременных подключений. Большой объём памяти, используемый в результате сопоставления процесса соединения, приводит к компромиссу между параллелизмом и памятью. Многопроцессная архитектура обеспечивает лишь ограниченную масштабируемость для одновременных запросов, особенно в случае длительных, частично неактивных соединений (например, запросов на уведомления с длительным опросом).

Популярный веб-сервер Apache предоставляет надёжный многопроцессорный модуль, основанный на предварительном разветвлении процессов, предварительном разветвлении Apache-MPM. Это по-прежнему многопроцессорный модуль по умолчанию для UNIX-установок Apache. Многопоточные архитектуры

1.3.3 Многопоточная архитектура

Когда стали доступны библиотеки потоков, появились новые серверные архитектуры, которые заменили тяжёлые процессы более лёгкими потоками. По сути, они используют модель «поток на соединение». Хотя многопоточный подход основан на тех же принципах, он имеет несколько важных отличий. Прежде всего, несколько потоков используют одно и то же адресное пространство и, следовательно, используют общие глобальные переменные и состояние. Это позволяет реализовать общие функции для всех обработчиков запросов, такие как общий кеш для кэшируемых ответов внутри веб-сервера. Очевидно, что тогда потребуется правильная синхронизация и координация. Ещё одним отличием более лёгких структур потоков является меньший объем памяти. По сравнению с полноценным объёмом памяти всего процесса, поток потребляет лишь ограниченную память (т. е. стек потоков). Кроме того, потоки требуют меньше ресурсов для создания/завершения. Мы уже видели, что размеры про-

цесса представляют собой серьёзную проблему в случае высокого уровня параллелизма. Потоки, как правило, являются более эффективной заменой при сопоставлении соединений с действиями.

На практике общепринятой архитектурой является размещение одного потока-диспетчера (иногда называемого потоком-получателем) перед пулом потоков для обработки соединений, как показано на рисунке. Пулы потоков – это распространённый способ ограничения максимального количества потоков внутри сервера. Диспетчер блокирует сокет для новых соединений. После установки соединение передаётся в очередь входящих соединений. Потоки из пула потоков принимают соединения из очереди, выполняют запросы и ждут новых соединений в очереди. Если очередь также ограничена, максимальное количество ожидающих соединений может быть ограничено. Дополнительные подключения будут отклонены. Хотя эта стратегия ограничивает параллелизм, она обеспечивает более предсказуемые задержки и предотвращает полную перегрузку.

Apache-MPM – это многопроцессорный модуль для веб-сервера Apache, который объединяет процессы и потоки. Модуль порождает несколько процессов, и каждый процесс, в свою очередь, управляет собственным пулом потоков.

Многопоточные серверы, использующие модель «поток на соединение», легко реализовать и следуют простой стратегии. Синхронные блокирующие операции ввода-вывода могут использоваться как естественный способ выражения доступа к вводу-выводу. Операционная система перекрывает несколько потоков посредством упреждающего планирования. В большинстве случаев, по крайней мере, блокирующая операция ввода-вывода запускает планирование и вызывает переключение контекста, позволяя продолжить работу следующему потоку. Это надёжная модель для достойного параллелизма, а также подходящая, когда необходимо выполнить разумное количество операций, связанных с ЦП. Кроме того, можно использовать несколько ядер ЦП напрямую, поскольку потоки и процессы планируются для всех доступных ядер.

При большой нагрузке многопоточный веб-сервер потребляет большие объёмы памяти (из-за одного стека потоков для каждого соединения), а постоянное переключение контекста приводит к значительным потерям процессорного времени. Косвенным наказанием за это является повышенная вероятность промахов в кэше ЦП. Уменьшение абсолютного количества потоков повыша-

ет производительность каждого потока, но ограничивает общую масштабируемость с точки зрения максимального количества одновременных подключений.

1.4 Событийная архитектура

В качестве альтернативы синхронному блокированию ввода-вывода в серверных архитектурах также распространён подход, управляемый событиями. Из-за семантики асинхронных/неблокирующих вызовов необходимы другие модели, кроме ранее описанной модели «поток на соединение». Распространённой моделью является сопоставление одного потока с несколькими соединениями. Затем поток обрабатывает все события, происходящие в результате операций ввода-вывода этих соединений и запросов. Новые события ставятся в очередь, и поток выполняет так называемый цикл обработки событий: удаляет события из очереди, обрабатывает событие, затем принимает следующее событие или ожидает отправки новых событий. Таким образом, работа, выполняемая потоком, очень похожа на работу планировщика, объединяющего несколько соединений в один поток выполнения.

Обработка события либо требует зарегистрированного кода обработчика событий для конкретных событий, либо основана на выполнении обратного вызова, заранее связанного с событием. Различные состояния соединений, обрабатываемых потоком, организованы в соответствующие структуры данных — либо явно с использованием конечных автоматов, либо неявно через продолжения или закрытия обратных вызовов. В результате поток управления приложением, использующим событийно-ориентированный стиль, каким-то образом инвертируется. Вместо последовательных операций программа, управляемая событиями, использует каскад асинхронных вызовов и обратных вызовов, которые выполняются при возникновении событий. Это понятие часто делает поток управления менее очевидным и усложняет отладку.

Использование серверных архитектур, управляемых событиями, исторически зависело от наличия асинхронных/неблокирующих операций ввода-вывода на уровне ОС и подходящих высокопроизводительных интерфейсов уведомления о событиях, таких как epoll и kqueue. Более ранние реализации серверов, основанных на событиях, таких как веб-сервер Flash [4].

Наличие одного потока, выполняющего цикл событий и ожидающего уведомлений о вводе-выводе, оказывает иное влияние на масштабируемость, чем подход на основе потоков, описанный ранее. Отсутствие связывания соедине-

ний и потоков существенно уменьшает количество потоков сервера — в крайнем случае, вплоть до одного потока цикла событий плюс некоторых потоков ядра ОС для ввода-вывода. Тем самым мы избавляемся от накладных расходов на чрезмерное переключение контекста и не нуждаемся в стеке потоков для каждого соединения. Это уменьшает объем памяти под нагрузкой и тратит меньше времени процессора на переключение контекста. В идеале ЦП становится единственным очевидным узким местом сетевого приложения, управляемого событиями. Пока не будет заархивировано полное насыщение ресурсов, цикл событий масштабируется с увеличением пропускной способности. Как только нагрузка превышает максимальное насыщение, очередь событий начинает накапливаться, поскольку поток обработки событий не может соответствовать. В этом случае подход, управляемый событиями, по-прежнему обеспечивает высокую пропускную способность, но задержки запросов увеличиваются линейно из-за перегрузки. Это может быть приемлемо для временных пиков нагрузки, но постоянная перегрузка снижает производительность и делает службу непригодной для использования. Одной из контрмер является более ресурсосберегающее планирование и разделение обработки событий, как мы вскоре увидим при анализе поэтапного подхода.

На данный момент придерживаются событийно-ориентированных архитектур и согласовываем их с многоядерными архитектурами. Хотя модель на основе потоков охватывает как параллелизм на основе ввода-вывода, так и на основе ЦП, первоначальная архитектура, основанная на событиях, касается исключительно параллелизма ввода-вывода. Для использования нескольких процессоров или ядер серверы, управляемые событиями, должны быть дополнительно адаптированы.

Очевидный подход – создание нескольких отдельных серверных процессов на одной машине. Это часто называют подходом N-копирования для использования N экземпляров на хосте с N процессорами/ядрами. В нашем случае на машине будет запускаться несколько экземпляров веб-сервера и регистрироваться все экземпляры на балансировщиках нагрузки. Менее изолированная альтернатива использует общий сокет сервера между всеми экземплярами, что требует некоторой координации. Например, реализация этого подхода доступна для node.js с использованием модуля кластера, который разветвляет несколько экземпляров приложения и использует один серверный сокет.

Веб-серверы в архитектурной модели имеют специфическую особенность – они не имеют состояния и не имеют общего доступа. Уже использование внутреннего кэша для динамических запросов требует внесения некоторых изменений в архитектуру сервера. На данный момент более простая модель параллелизма, состоящая из однопоточного сервера и семантики последовательного выполнения обратных вызовов, может быть принята как часть архитектуры. Именно эта простая модель выполнения делает однопоточные приложения привлекательными для разработчиков, поскольку усилия по координации и синхронизации уменьшаются, а код приложения (то есть обратные вызовы) гарантированно не будет выполняться одновременно.

1.5 Смешанный подход

Потребность в масштабируемых архитектурах и недостатки обеих общих моделей привели к появлению альтернативных архитектур и библиотек, включающих в себя функции обеих моделей.

1.5.1 Поэтапная событийно-ориентированная архитектура

Формирующая архитектура, объединяющая потоки и события для масштабируемых серверов, была разработана Уэлшем, так называемый SEDA [5]. В качестве базовой концепции логика сервера разделена на ряд чётко определённых этапов, соединённых очередями. В процессе обработки запросы передаются от этапа к этапу. Каждый этап поддерживается потоком или пулом потоков, который можно настроить динамически.

Такое разделение благоприятствует модульности, поскольку конвейер этапов можно легко изменить и расширить. Ещё одной очень важной особенностью конструкции SEDA является осведомлённость о ресурсах и явный контроль нагрузки. Размер элементов в очереди на этап и рабочая нагрузка пула потоков на этап дают чёткое представление об общем коэффициенте загрузки. В случае перегрузки сервер может настроить параметры планирования или размеры пула потоков. Другие адаптивные стратегии включают динамическую реконфигурацию конвейера или намеренное завершение запроса. Когда управление ресурсами, самоанализ нагрузки и адаптивность отделены от логики приложения этапа, разрабатывать хорошо обусловленные сервисы становится проще. С точки зрения параллелизма SEDA представляет собой гибридный подход между многопоточностью «поток на соединение» и параллелизмом на основе событий. Наличие элементов удаления из очереди и обработки потока (или

пула потоков) напоминает подход, управляемый событиями. Использование нескольких этапов с независимыми потоками эффективно задействует несколько процессоров или ядер и способствует созданию многопоточной среды. С точки зрения разработчика, реализация кода-обработчика для определённого этапа также напоминает более традиционное программирование потоков.

Недостатками SEDA являются повышенные задержки из-за обхода очередей и стадий даже при минимальной нагрузке. В более поздней ретроспективе Уэлш также раскритиковал отсутствие дифференциации границ модулей (этапов) и границ параллелизма (очередей и потоков). Такое распределение вызывает слишком много переключений контекста, когда запросы проходят через несколько этапов и очередей. Лучшее решение группирует несколько этапов вместе с общим пулом потоков. Это уменьшает переключение контекста и улучшает время отклика. Этапы с операциями ввода-вывода и сравнительно длительным временем выполнения все же можно изолировать.

Модель SEDA вдохновила на создание нескольких реализаций, включая базовую серверную структуру Apache MINA [6] и корпоративные сервисные шины, такие как Mule ESB [7].

1.5.2 Специальные библиотеки

Другие подходы были сосредоточены на недостатках потоков в целом и проблемах доступных библиотек потоков (на пользовательском уровне) в частности. Как мы скоро увидим, большинство проблем масштабируемости потоков связаны с недостатками их библиотек.

Например, библиотека потоков Capriccio, созданная фон Береном обещает масштабируемые потоки для серверов, решая основные проблемы потоков. Проблема обширных переключений контекста решается с помощью невытесняющего планирования. Потоки либо выдают результат при операциях ввода-вывода, либо при явной операции вывода. Размер стека каждого потока ограничен на основе предварительного анализа во время компиляции. Это делает ненужным упреждающее избыточное предоставление ограниченного пространства стека. Однако неограниченные циклы и использование рекурсивных вызовов делают полный расчёт размера стека априори невозможным. В качестве обходного пути в код вставляются контрольные точки, которые определяют, произойдёт ли переполнение стека, и в этом случае выделяют новые фрагменты стека. Контрольные точки вставляются во время компиляции и раз-

мещаются таким образом, чтобы в коде никогда не возникало переполнение стека между двумя контрольными точками. Кроме того, применяется планирование с учётом ресурсов, которое предотвращает перегрузку. Таким образом, дескрипторы ЦП, памяти и файлов отслеживаются и сочетаются со статическим анализом использования ресурсов потоков, планирование динамически адаптируется.

Также разработаны гибридные библиотеки, объединяющие потоки и события. Ли и Зданцевик реализовали комбинированную модель для Haskell [8], основанную на монадах параллелизма. Язык программирования Scala также обеспечивает управляемый событиями и многопоточный параллелизм, который можно комбинировать для серверных реализаций.

1.6 Выводы

Проблема масштабируемости веб-серверов характеризуется интенсивным параллелизмом HTTP-соединений. Таким образом, основной проблемой является массовый параллелизм операций ввода-вывода. Когда несколько клиентов одновременно подключаются к серверу, ресурсы сервера, такие как время процессора, память и ёмкость сокетов, должны строго планироваться и использоваться, чтобы одновременно поддерживать низкие задержки ответа и высокую пропускную способность. Поэтому мы рассмотрели различные модели операций ввода-вывода и способы представления запросов в модели программирования, поддерживающей параллелизм. Мы сосредоточились на различных серверных архитектурах, которые обеспечивают различные комбинации вышеупомянутых концепций, а именно: многопроцессные серверы, многопоточные серверы, серверы, управляемые событиями, и комбинированные подходы, такие как SEDA.

Разработка высокопроизводительных серверов, использующих потоки, события или и то, и другое, стала реальной возможностью. Однако традиционная синхронная модель блокирующего ввода-вывода страдает от снижения производительности, когда она используется как часть массового параллелизма ввода-вывода. Аналогичным образом, использование большого количества потоков ограничивается увеличением потерь производительности в результате постоянного переключения контекста и потребления памяти из-за размеров стека потоков. С другой стороны, серверные архитектуры, управляемые событиями, страдают от менее понятного стиля программирования и часто не мо-

гут напрямую воспользоваться преимуществами истинного параллелизма ЦП. Комбинированные подходы пытаются специально обойти проблемы, присущие одной из моделей, или предлагают концепции, включающие обе модели.

Теперь мы увидели, что подходы, основанные на потоках и событиях, по сути являются двойниками друг друга и уже долгое время разделяют сообщество сетевых серверов. Получение преимуществ совместного планирования и асинхронных/неблокирующих операций ввода-вывода является одним из основных желаний серверных приложений, ориентированных на ввод-вывод, однако это часто упускается из виду в более широком и смешанном споре между лагерем потоков и лагерем событий.

2 Конструкторский раздел

2.1 Архитектура сервера

Архитектура сервера изначальна определена вариантом: threadpool + pselect. На рисунке 1 представлена архитектура сервера.

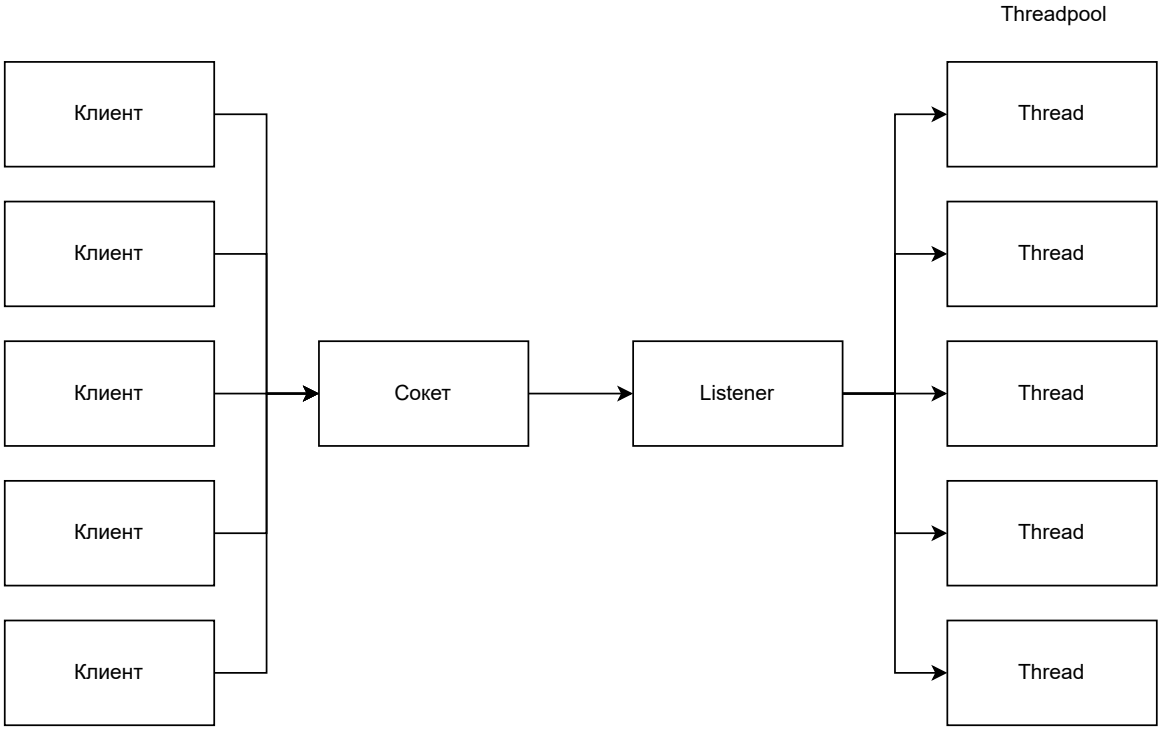

Рисунок 1 – Архитектура сервера

3 Технологический раздел

3.1 Реализация сервера

В приложении 4.2 представлен исходный код сервера.

3.2 Логгер

В приложениях 4.2 и 4.2 представлены заголовочный файл и исходный код логгера.

4 Исследовательский раздел

4.1 Работа сервера

На рисунке 2 показан пример GET запроса.

```
┌──(root💀NebuchadnezzaR)-[~/bmstu/bmstu-network-cp/src]
└─# curl -v -X GET http://localhost:8080/test.txt --output -
Note: Unnecessary use of -X or --request, GET is already inferred.
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /test.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.88.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 17
< Content-Type: text/plain; charset=us-ascii
<
request me daddy
* Connection #0 to host localhost left intact
```

Рисунок 2 – GET запрос

На рисунке 3 показан пример HEAD запроса.

```
┌──(root💀NebuchadnezzaR)-[~/bmstu/bmstu-network-cp/src]
└─# curl -v -X HEAD http://localhost:8080/public/favicon.ico
Warning: Setting custom HTTP method to HEAD with -X/--request may not work the
Warning: way you want. Consider using -I/--head instead.
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> HEAD /public/favicon.ico HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.88.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 0
< Content-Type: image/vnd.microsoft.icon; charset=binary
<
* Connection #0 to host localhost left intact
```

Рисунок 3 – HEAD запрос

На рисунке 4 показана проверка выхода за /root директорию.

```
┌──(root💀NebuchadnezzaR)-[~/bmstu/bmstu-network-cp/src]
└─# curl -v -X GET http://localhost:8080/../../..../a.txt
Note: Unnecessary use of -X or --request, GET is already inferred.
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /..../a.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.88.1
> Accept: */*
>
< HTTP/1.1 403 Forbidden
* no chunk, no close, no size. Assume close to signal end
<
* Closing connection 0
```

Рисунок 4 – Проверка выхода за /root директорию

На рисунке 5 показан пример запроса несуществующего файла.

```
┌──(root💀NebuchadnezzaR)-[~/bmstu/bmstu-network-cp/src]
└─# curl -v -X GET http://localhost:8080/not-found.txt
Note: Unnecessary use of -X or --request, GET is already inferred.
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /not-found.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.88.1
> Accept: */*
>
< HTTP/1.1 404 Not Found
* no chunk, no close, no size. Assume close to signal end
<
* Closing connection 0
```

Рисунок 5 – Пример запроса несуществующего файла

На рисунке 6 показан пример записи лога.

```
22:59:41 TRACE main.c:220: Request accepted, started thread lookup
22:59:41 TRACE main.c:66: Thread found, started request handling activity
22:59:41 DEBUG main.c:81: Method = GET Path = /oris.zip
22:59:41 DEBUG main.c:125: Full path = /root/bmstu/bmstu-network-cp/src/oris.zip
22:59:41 TRACE main.c:38: Processing file at path: /root/bmstu/bmstu-network-cp/src/oris.zip
22:59:41 DEBUG main.c:44: File size = 23546285
22:59:41 TRACE main.c:51: Sending file at path: /root/bmstu/bmstu-network-cp/src/oris.zip
22:59:41 TRACE main.c:60: File sent
```

Рисунок 6 – Пример записи лога

На рисунке 7 показана обработка большого файла.

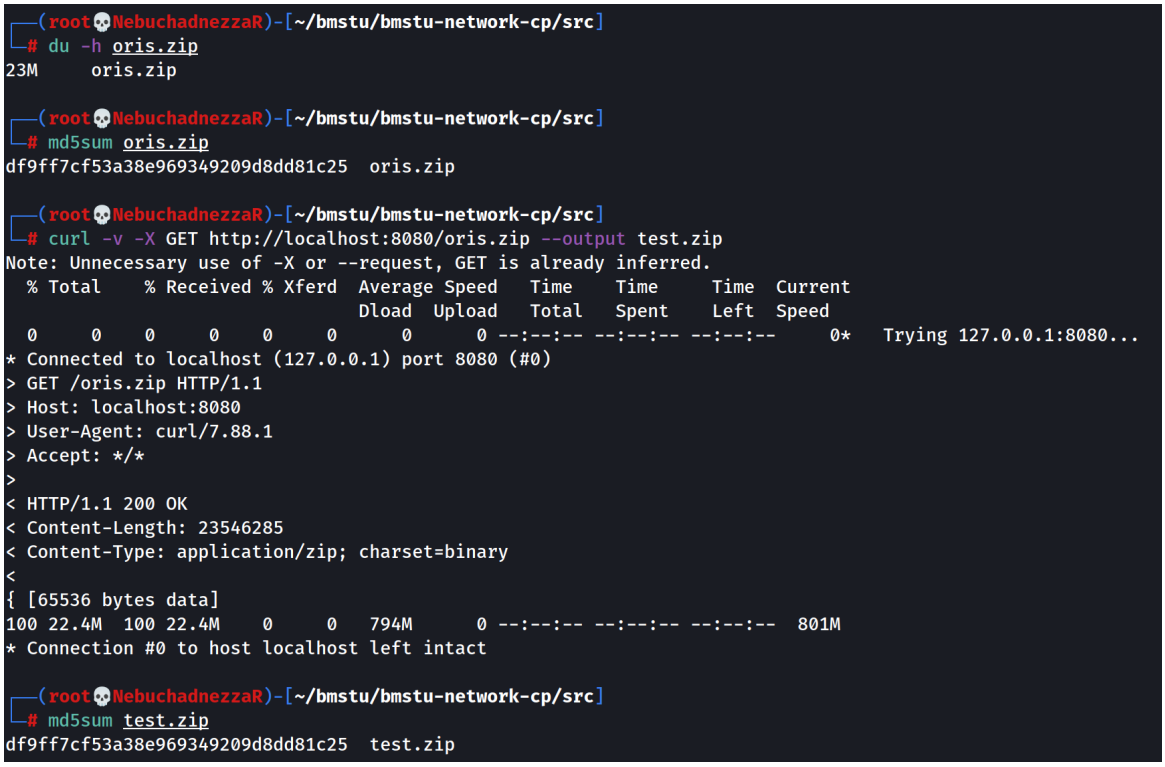

Рисунок 7 – Обработка большого файла

4.2 Нагрузочное тестирование

Нагрузочное тестирование проводилось с помощью ApacheBenchmark, 1000 запросов файла 23Мб в 8 потоков. В качестве альтернативы был настроен сервер nginx. Результаты нагрузочного тестирования представлены в таблице 1.

Таблица 1 – Результат нагрузочного тестирования

Сервер	Время	Запрос/с	Среднее время запроса	Объём передачи
nginx	7.236с	138.20	7.236мс	3.03Гб/с
TP+PS	5.611с	178.21	5.611мс	3.9Гб/с

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы был разработан статик сервер, использующий архитектуру threadpool + pselect. Самописное решение оказалось на 23% быстрее альтернативы в виде сервера nginx.

В результате работы были выполнены следующие задачи:

1) изучены основные принципы работы статических серверов;
2) проанализированы существующие архитектуры статических серверов;
3) изучены возможности выбранных технологий и инструментов;
4) разработан статический сервер;
5) проведено исследование разработанного ПО.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Li Peng, Zdancewic Steve. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives. New York, NY, USA, 2007. Т. 42, № 6. с. 189–199.

2. STEVENS W. Richard; FENNER B. R. A. M. // Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition). Addison-Wesley, 2003.

3. Daemon implementation. Режим доступа: `https://www.w3.org/Daemon/Implementation/`, свободный (дата обращения: 7 декабря 2023 г.).

4. Pai Vivek, Druschel Peter, Zwaenepoel Willy. Flash: An efficient and portable Web server. 1999. 01. С. 199–212.

5. SEDA. Режим доступа: `http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html`, свободный (дата обращения: 7 декабря 2023 г.).

6. Apache MINA [Электронный ресурс]. Режим доступа: `https://mina.apache.org/`, свободный (дата обращения: 7 декабря 2023 г.).

7. Mulesoft [Электронный ресурс]. Режим доступа: `https://www.mulesoft.com/`, свободный (дата обращения: 7 декабря 2023 г.).

8. Li Peng, Zdancewic Steve. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives // Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI ’07. New York, NY, USA: Association for Computing Machinery, 2007. с. 189–199.

ПРИЛОЖЕНИЕ А

Листинг 1: Исходный код сервера

```
#define _GNU_SOURCE

#include <arpa/inet.h>
#include <fcntl.h>
#include <magic.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#include "log.h"

#define INDEX_HTML_PATH "public/index.html"
#define FAVICON_PATH "public/favicon.ico"

#define MAX_THREADS 10
#define MAX_REQUEST_SIZE 4096
#define MAX_RESPONSE_SIZE 4096
#define MAX_BUFFER_SIZE 4096

#define RESPONSE_HEADERS \
"HTTP/1.1 200 OK\r\nContent-Length: %lld\r\nContent-Type: %s\r
    \n\r\n"
#define FORBIDDEN "HTTP/1.1 403 Forbidden\r\n\r\n"
#define NOT_FOUND "HTTP/1.1 404 Not Found\r\n\r\n"
#define NOT_ALLOWED "HTTP/1.1 405 Method Not Allowed\r\n\r\n"

typedef struct {
  int client_sockfd;
  struct sockaddr_in client_addr;
} request_t;
int send_file_with_response(FILE* file, const char* path, char
    * response,
```

Листинг 2: Исходный код сервера

```
request_t* req, struct magic_set* magic) {
  log_trace("Processing file at path: %s", path);

  fseek(file, 0, SEEK_END);
  long long int file_size = ftell(file);
  fseek(file, 0, SEEK_SET);

  log_debug("File size = %lld", file_size);

  sprintf(response, RESPONSE_HEADERS,
  (long long int)strlen(response) + file_size, magic_file(magic,
      path));

  write(req->client_sockfd, response, strlen(response));

  log_trace("Sending file at path: %s", path);

  while (file_size > 0) {
    uint8_t buffer[MAX_BUFFER_SIZE] = {0};
    size_t bytes_readed = fread(buffer, 1, MAX_BUFFER_SIZE, file
       );
    file_size -= bytes_readed;
    write(req->client_sockfd, buffer, bytes_readed);
  }
  log_trace("File sent");
  return 0;
}

void request_handler(request_t* req) {
  log_trace("Thread found, started request handling activity");

  char request[MAX_REQUEST_SIZE] = {0};
  char response[MAX_RESPONSE_SIZE] = {0};
  char dir_buff[MAX_BUFFER_SIZE] = {0};

  struct magic_set* magic = magic_open(MAGIC_MIME | MAGIC_CHECK)
     ;
  magic_load(magic, NULL);

  read(req->client_sockfd, request, sizeof(request));
```

Листинг 3: Исходный код сервера

```
1  char method[10] = {0};
2  char path[256] = {0};
3  sscanf(request, "%s %s", method, path);
4
5  log_debug("Method = %s  Path = %s", method, path);
6
7  if (strcmp(method, "GET") != 0 && strcmp(method, "HEAD") != 0) {
8    sprintf(response, NOT_ALLOWED);
9
10   log_info("Method %s not allowed", method);
11
12   write(req->client_sockfd, &response, strlen(response));
13   close(req->client_sockfd);
14   free(req);
15   return;
16 }
17
18 if (strcmp(path, "/") == 0 && strcmp(method, "GET") == 0) {
19   log_info("Returning default HTML");
20
21   FILE* index_html = fopen(INDEX_HTML_PATH, "rb");
22
23   send_file_with_response(index_html, INDEX_HTML_PATH, response,
         req, magic);
24
25   fclose(index_html);
26
27   log_info("/ request succeded");
28   return;
29 }
30
31 if (strcmp(path, "/favicon.ico") == 0 && strcmp(method, "GET")
     == 0) {
32   log_info("Returning favicon");
33
34   FILE* icon = fopen(FAVICON_PATH, "rb");
35
36   send_file_with_response(icon, FAVICON_PATH, response, req,
       magic);
```

Листинг 4: Исходный код сервера

```
1
2 fclose(icon);
3
4 log_info("/favicon.ico request succeded");
5 return;
6 }
7
8 getcwd(dir_buff, MAX_BUFFER_SIZE);
9 char full_path[MAX_BUFFER_SIZE] = {0};
10 strcpy(full_path, dir_buff);
11 strcat(full_path, path);
12
13 log_debug("Full path = %s", full_path);
14
15 if (strstr(full_path, "..") != NULL) {
16 sprintf(response, FORBIDDEN);
17
18 log_info("Path forbidden: %s", full_path);
19
20 write(req->client_sockfd, &response, strlen(response));
21 close(req->client_sockfd);
22 free(req);
23 return;
24 }
25
26 FILE* file = fopen(full_path, "rb");
27 if (file == NULL) {
28 sprintf(response, NOT_FOUND);
29
30 log_info("File %s not found", full_path);
31
32 write(req->client_sockfd, &response, strlen(response));
33 close(req->client_sockfd);
34 free(req);
35 return;
36 }
37
38 if (strcmp(method, "GET") == 0) {
39 send_file_with_response(file, full_path, response, req, magic);
40 } else if (strcmp(method, "HEAD") == 0) {
```

Листинг 5: Исходный код сервера

```
sprintf(response, RESPONSE_HEADERS, (long long int)strlen(
    response),
magic_file(magic, full_path));
write(req->client_sockfd, &response, strlen(response));

log_info("HEAD request succeded");
}

close(req->client_sockfd);

fclose(file);
free(req);
}

int main(void) {
setbuf(stdout, NULL);

int server_sockfd;
int client_sockfd;
struct sockaddr_in server_addr;
struct sockaddr_in client_addr;
socklen_t client_addr_len;

server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (server_sockfd < 0) {
log_error("Failed to create socket");
exit(EXIT_FAILURE);
}

server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(8080);

if (bind(server_sockfd, (struct sockaddr*)&server_addr, sizeof(
    server_addr)) <
0) {
log_error("Failed to bind socket");
exit(EXIT_FAILURE);
}
```

Листинг 6: Исходный код сервера

```
1  if (listen(server_sockfd, SOMAXCONN) < 0) {
2    log_error("Failed to listen");
3    exit(EXIT_FAILURE);
4  }
5  pthread_t threads[MAX_THREADS];
6  for (int i = 0; i < MAX_THREADS; ++i) {
7    threads[i] = 0;
8  }
9  log_info("Started at 0.0.0.0:8080");
10 fd_set rset;
11 FD_ZERO(&rset);
12 while (1) {
13   FD_SET(server_sockfd, &rset);
14   pselect(server_sockfd + 1, &rset, NULL, NULL, NULL, NULL);
15   if (!FD_ISSET(server_sockfd, &rset)) continue;
16   client_addr_len = sizeof(client_addr);
17   client_sockfd = accept(server_sockfd, (struct sockaddr*)&
       client_addr, &client_addr_len);
18   if (client_sockfd < 0) {
19     log_error("Failed to accept");
20     continue;
21   }
22   request_t* req = (request_t*)malloc(sizeof(request_t));
23   if (req == NULL) {
24     log_fatal("Failed to allocate memmory for request");
25     continue;
26   }
27   req->client_sockfd = client_sockfd;
28   req->client_addr = client_addr;
29   log_trace("Request accepted, started thread lookup");
30   for (int i = 0; i < MAX_THREADS; ++i) {
31     if (threads[i] == 0 || (pthread_tryjoin_np(threads[i], NULL)
         == 0)) {
32       pthread_create(&threads[i], NULL, (void* (*)(void*))
           request_handler,
33       (void*)req);
34       break;
35     }}}
36 return EXIT_SUCCESS;
37 }
```

ПРИЛОЖЕНИЕ Б

Листинг 7: Заголовочный файл логгера

```
#ifndef LOG_H
#define LOG_H

#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <time.h>
typedef struct {
    va_list ap;
    const char *fmt;
    const char *file;
    struct tm *time;
    void *udata;
    const int line;
    const int level;
} log_event_t;
enum { LOG_TRACE, LOG_DEBUG, LOG_INFO, LOG_WARN, LOG_ERROR,
    LOG_FATAL };
#define log_trace(...) log_log(LOG_TRACE, __FILE__, __LINE__,
    __VA_ARGS__)
#define log_debug(...) log_log(LOG_DEBUG, __FILE__, __LINE__,
    __VA_ARGS__)
#define log_info(...) log_log(LOG_INFO, __FILE__, __LINE__,
    __VA_ARGS__)
#define log_warn(...) log_log(LOG_WARN, __FILE__, __LINE__,
    __VA_ARGS__)
#define log_error(...) log_log(LOG_ERROR, __FILE__, __LINE__,
    __VA_ARGS__)
#define log_fatal(...) log_log(LOG_FATAL, __FILE__, __LINE__,
    __VA_ARGS__)

void log_set_level(const int level);
void log_set_quiet(const bool enable);

void log_log(int level, const char *file, int line, const char *
    fmt, ...);
#endif
```

ПРИЛОЖЕНИЕ В

Листинг 8: Исходный код логгера

```c
#include "log.h"
static struct {
  void *udata;
  int level;
  bool quiet;
} Logger;

static const char *level_strings[] = {"TRACE", "DEBUG", "INFO",
  "WARN",  "ERROR", "FATAL"};

static const char *level_colors[] = {"\x1b[94m", "\x1b[36m", "\
   x1b[32m",
  "\x1b[33m", "\x1b[31m", "\x1b[35m"};

static void stdout_callback(log_event_t *ev) {
  char buf[16];
  buf[strftime(buf, sizeof(buf), "%H:%M:%S", ev->time)] = '\0';
  fprintf(ev->udata, "%s %s%-5s\x1b[0m \x1b[90m%s:%d:\x1b[0m ",
     buf,
  level_colors[ev->level], level_strings[ev->level], ev->file,
  ev->line);
  vfprintf(ev->udata, ev->fmt, ev->ap);
  fprintf(ev->udata, "\n");
  fflush(ev->udata);
}

void log_set_level(const int level) { Logger.level = level; }

void log_set_quiet(const bool enable) { Logger.quiet = enable; }

static void init_event(log_event_t *ev, void *udata) {
  if (!ev->time) {
    time_t t = time(NULL);
    ev->time = localtime(&t);
  }
  ev->udata = udata;
}
```

Листинг 9: Исходный код логгера

```
1 void log_log(int level, const char *file, int line, const char *
    fmt, ...) {
2   log_event_t ev = {
3     .fmt = fmt,
4     .file = file,
5     .line = line,
6     .level = level,
7   };
8
9   if (!Logger.quiet && level >= Logger.level) {
10    init_event(&ev, stderr);
11    va_start(ev.ap, fmt);
12    stdout_callback(&ev);
13    va_end(ev.ap);
14  }
15 }
```