



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 _____ Романов С. К.

Руководитель курсового проекта _____ Яковидис Н. О.

2024 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

_____ Романов Семен Константинович

(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 + poll для отдачи контента с диска

Оформление курсовой работы:

Расчетно-пояснительная записка на 20-30 листах формата А4.

Дата выдачи задания « _____ » _____ 20 ____ г.

Руководитель курсовой работы

_____ (Подпись, дата)

Яковидис Н. О.

_____ (Фамилия И. О.)

Студент ИУ7-75Б
(Группа)

_____ (Подпись, дата)

Романов С. К.

_____ (Фамилия И. О.)

РЕФЕРАТ

Расчетно-пояснительная записка 32 с., 6 рис., 1 табл., 12 ист.

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

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

КЛЮЧЕВЫЕ СЛОВА

threadpool, socket, nginx, static server

Содержание

РЕФЕРАТ	2
ВВЕДЕНИЕ	5
1 Аналитическая часть	6
1.1 Статик сервер	6
1.2 Протокол HTTP	6
1.3 Архитектура	8
1.3.1 Поточная архитектура	8
1.3.2 Процессная архитектура	9
1.3.3 Многопоточная архитектура	10
1.4 Событийная архитектура	11
1.5 Смешанный подход	14
1.5.1 Поэтапная событийно-ориентированная архитектура	14
1.5.2 Специальные библиотеки	15
2 Конструкторский раздел	18
2.1 Обработка запросов HTTP	18
2.2 Uniform Resource Locator	19
2.3 Обработка ответов HTTP	19
2.4 Безопасность	20
3 Технологический раздел	21
3.1 Листинги кода	21
3.2 Демонстрация работы	24
4 Исследовательский раздел	26
4.1 Технические характеристики	26
4.2 Работа сервера	26

4.3 Нагрузочное тестирование	28
ЗАКЛЮЧЕНИЕ	30
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	32

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 будет идти сосредоточение на изучении статических серверов, их важности и показателях распространённости, а также на приобретении навыков настройки и обслуживания такой системы.

Для достижения поставленной цели, предполагается выполнение следующих задач:

1. провести формализацию задачи и определить необходимый функционал;
2. исследовать предметную область веб серверов;
3. спроектировать приложение;
4. реализовать приложение;
5. протестировать приложение на предмет корректности;

1 Аналитическая часть

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 + poll.

Должны быть учтены минимальные требования к безопасности статик-серверов. Должен быть реализован логгер. В качестве инструментов разработки должен быть использован язык C без сторонних библиотек. По результатам разработки должно быть проведено нагрузочное тестирование.

1.1 Статик сервер

Статический сервер – это программа или сервис, обрабатывающий запросы на получение статических файлов (например, HTML, CSS, JavaScript, изображения). 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

1.2 Протокол HTTP

Рассмотрение современных веб-технологий невозможно без глубокого понимания протокола HTTP (Hypertext Transfer Protocol). Этот стандарт представляет собой основной механизм взаимодействия между веб-клиентами и серверами, обеспечивая передачу данных в формате гипертекста. HTTP сыграл ключевую роль в развитии интернета и является основой для передачи ресурсов,

таких как HTML-страницы, изображения, стили и другие [1].

Протокол HTTP оперирует по принципу запрос-ответ, где клиент посылает запрос на сервер для получения определенного ресурса, и сервер возвращает соответствующий ответ. Этот обмен сообщениями строится вокруг простых принципов и стандартов, что позволяет веб-клиентам и серверам эффективно обмениваться информацией.

Существует несколько версий протокола HTTP, каждая из которых вносит улучшения и нововведения. В контексте данной работы выбран HTTP 1.1, поскольку он предоставляет возможность многократного использования соединений (connection reuse). Эта характеристика позволяет снизить задержки при обмене данными, поскольку после завершения одного запроса соединение может быть переиспользовано для последующих запросов, сэкономив трафик и улучшив производительность. В работе более детально рассмотрены механизмы переиспользования соединений в рамках HTTP 1.1 и их влияние на эффективность веб-взаимодействия.

В рамках рассмотрения альтернативных версий протокола HTTP, особое внимание привлекает HTTP/2.0. Эта версия была представлена с целью оптимизации производительности и улучшения пользовательского опыта в сравнении с ее предшественником, HTTP/1.1. Основными изменениями, внесенными в HTTP/2.0, стали многопоточность, мультиплексирование и сжатие заголовков, что привело к уменьшению задержек и более эффективному использованию ресурсов.

Мультиплексирование позволяет одновременно передавать несколько запросов и ответов в рамках одного соединения. Это снижает задержки и улучшает пропускную способность, что особенно важно в условиях медленных сетей или при использовании мобильных устройств. Сжатие заголовков также способствует более эффективной передаче данных [2]. Тем не менее, несмотря на ряд явных преимуществ, выбор HTTP/2.0 для данного проекта не был сделан. HTTP/2.0, несмотря на свою эффективность, обладает более сложной структурой

рой и требует более глубокого понимания принципов мультимплексирования и других инноваций, что может создать дополнительные сложности в учебных целях.

Таким образом, выбор НТТР 1.1 для данного проекта обоснован стремлением к более непосредственному и понятному представлению студентам основ работы веб-сервера и протокола НТТР.

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 [3]. 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» [4], в которой используется выделенный процесс для обработки соединения. Эта модель также использовалась для первого HTTP-сервера CERN httpd [5]. 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 стратегию, называемую предварительным разветвлением. При использовании предварительного разветвления основной серверный процесс упреждающе разветвляет несколько процессов-обработчиков при запуске. Часто дескриптор сокета (потокбезопасный) 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 уведомление 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 потребуются 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 процесса 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 соединения 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 пере-

грузку.

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 повышает 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 опе-

раций 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 [6].

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 соединений 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 [7]. 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 [8] и корпоративные сервисные шины, такие как Mule ESB [9].

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 Carriccio, созданная фон Береном обеща-

ет 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 размещаются 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 [10], основанную на монадах параллелизма. Язык программирования Scala также обеспечивает управляемый событиями и многопоточный параллелизм, который можно комбинировать для серверных реализаций.

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 могут 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 Обработка запросов HTTP

Запрос, в контексте протокола HTTP, структурирован в виде последовательности строк, начиная с первой строки, содержащей информацию о методе, URL и версии HTTP. Последующие строки представляют заголовки запроса, которые служат для дополнительной спецификации параметров запроса. Кроме того, запрос может включать тело, позволяя передавать дополнительные данные для более сложных веб-запросов.

Пример запроса показан на листинге 1

Листинг 1 – HTTP запрос

```
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:122.0) Gecko/20100101
    Firefox/122.0
Accept:
    text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

2.2 Uniform Resource Locator

URL (Uniform Resource Locator) представляет собой строку, используемую для определения местоположения ресурса в интернете. Он обычно состоит из нескольких компонентов, таких как схема, хост, порт, путь, запрос, и фрагмент. Путь в URL указывает на конкретный ресурс на сервере. Это может быть директория или файл. Например, в URL «https://example.com/path/to/resource», «path/to/resource» — это путь. Строка запроса следует за путем и начинается с вопросительного знака «?». Она содержит параметры запроса в виде пар «ключ=значение», разделенных амперсандом «&». Например, в URL «https://example.com/search?q=query», «q=query» — это строка запроса. Фрагмент обозначается символом «#» и используется для указания конкретной секции внутри ресурса. Он служит для навигации внутри самого документа. Например, в URL «https://example.com/page#section», «section» — это фрагмент. Также при обработке URL особую роль играет URL-кодировка. Она применяется для передачи специальных символов в URL без их конфликта с зарезервированными символами. Она заменяет определенные символы и пробелы специальными последовательностями. Например, пробел заменяется на «%20».

Для хранения информации о запросе предлагается использовать структуру, содержащую следующие поля:

- тип метода: GET, HEAD;
- относительный путь url;
- query параметр url;
- fragment параметр url;
- список заголовков запроса;

2.3 Обработка ответов HTTP

Протокол HTTP формирует ответ, включающий в себя версию HTTP, числовое представление статуса, текстовую строку статуса, перечень заголовков и тело ответа. При этом, содержанием ответа может являться как текстовый,

так и бинарный файл. В случае изображения, которое необходимо разбить на несколько буферов для передачи клиенту, данная операция осуществляется для обеспечения эффективной передачи данных, особенно когда размер изображения превышает объем одного буфера. Для хранения информации об ответе предлагается использовать структуру, содержащую следующие поля:

1. статус;
2. относительный путь файла;
3. путь файла в запросе;
4. MIME тип ответа;
5. список заголовков ответа;

2.4 Безопасность

Особое внимание уделяется безопасности статического веб-сервера. Наибольшую угрозу представляет несанкционированный доступ к файлам, расположенным вне директории сервера. Для предотвращения такого несанкционированного доступа рекомендуется использовать механизм, известный как Chroot Jail. Этот механизм реализуется с использованием системного вызова `chroot`, который изменяет корневой каталог на указанный, таким образом, ограничивая доступ к файлам вне этой директории. Следует отметить, что для выполнения системного вызова `chroot` необходимы привилегии суперпользователя. После выполнения системного вызова происходит снижение привилегий до уровня обычного пользователя, лишённого прав для вызова `chroot`, чтобы обеспечить безопасный доступ к остальным директориям. Этот подход позволяет эффективно управлять безопасностью работы статического веб-сервера.

Вывод

Были сформулированы конструкторские требования к безопасности, асинхронности, формату запросов и ответов, а также к структурам, используемым для моделирования работы сервера.

3 Технологический раздел

3.1 Листинги кода

В листинге 2 представлена реализация запуска веб-сервера:

Листинг 2 – Запуск WEB-сервера

```
int main(int argc, char *argv[]) {
    ...
    /* defaults */
    size_t nthreads = 4;
    size_t nslots = 64;
    char *servedir = ".";

    rlim.rlim_cur = rlim.rlim_max =
        3 + nthreads + nthreads * nslots + 5 * nthreads;
    if (setrlimit(RLIMIT_NOFILE, &rlim) < 0) {
        die("setrlimit:");
    }
    insock = sock_get_ips(srv.host, srv.port);
    if (sock_set_nonblocking(insock)) {
        return 1;
    }
    /* chroot */
    if (chdir(servedir) < 0) {
        die("chdir '%s':", servedir);
    }
    if (chroot(".") < 0) {
        die("chroot:");
    }
    /* accept incoming connections */
    server_init_thread_pool(insock, nthreads, nslots, &srv);

    while (wait(&status) > 0)
        ;

    return status;
}
```

Функция обработки соединения, изображенная в листинге 3, включает в себя принятие соединения, поиск подходящего соединения для дропа, принятие заголовка, и парсинг заголовков.

Листинг 3 – Функция обработки соединения

```
struct connection *connection_accept(int insock, struct connection *connection,
                                     size_t nslots) {

    struct connection *c = NULL;
    size_t i;

    for (i = 0; i < nslots; i++) {
        if (connection[i].fd == 0) {
            c = &connection[i];
            break;
        }
    }
    if (i == nslots) {
        c = connection_get_drop_candidate(connection, nslots);
        c->res.status = 0;
        connection_log(c);
        connection_reset(c);
    }

    if ((c->fd = accept(insock, (struct sockaddr *)&c->ia,
                      &(socklen_t){sizeof(c->ia)})) < 0) {
        if (errno != EAGAIN && errno != EWOULDBLOCK) {
            warn("accept:");
        }
        return NULL;
    }

    if (sock_set_nonblocking(c->fd)) {
        return NULL;
    }
    return c;
}
```

На листинге 4 представлен поиск соединения для завершения, при новом соединении.

Листинг 4 – Функция поиска соединения

```
enum status http_prepare_header_buf(const struct response *res,
                                   struct buffer *buf) {

    char tstamp[FIELD_MAX];
    size_t i;

    memset(buf, 0, sizeof(*buf));

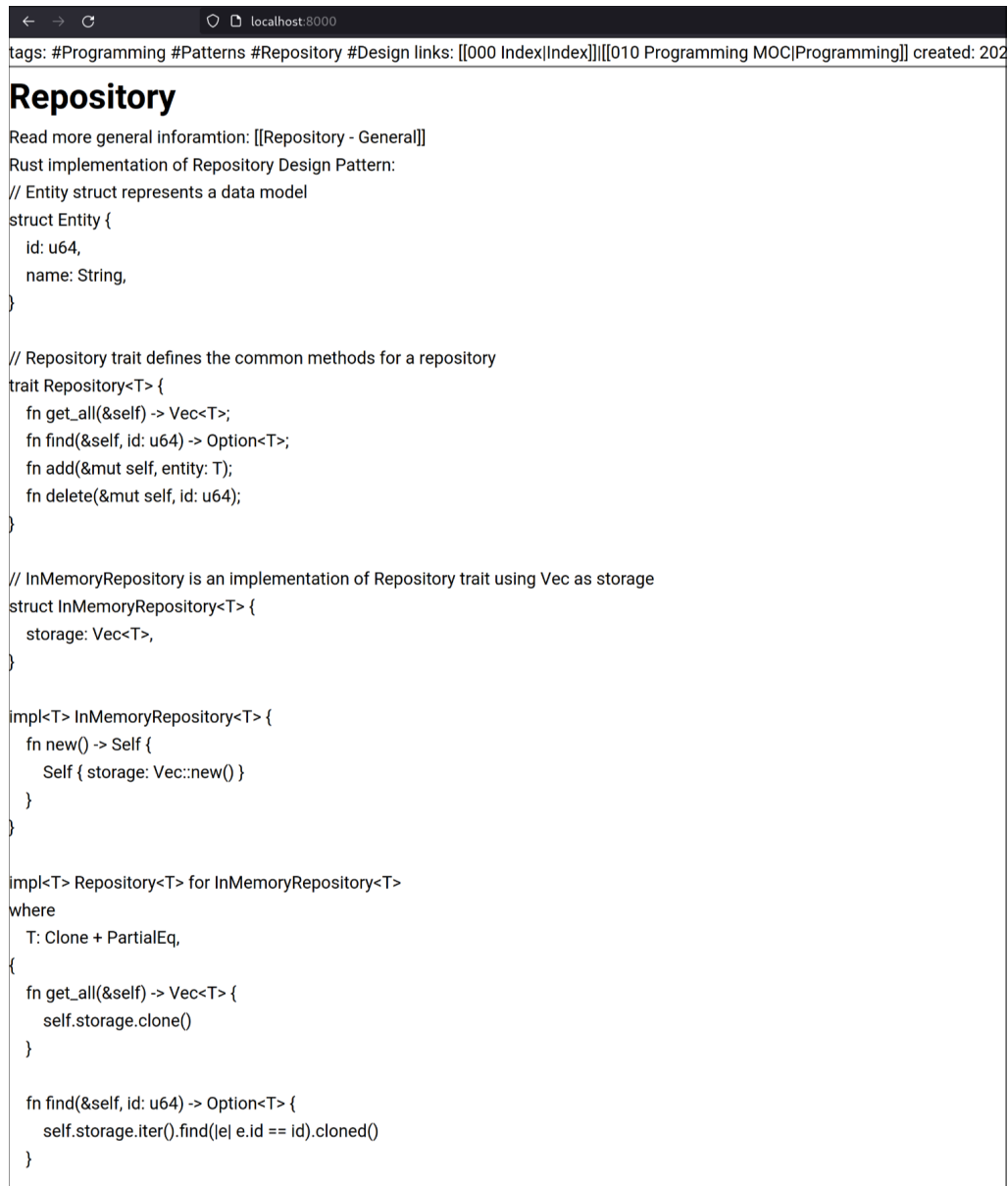
    if (timestamp(tstamp, sizeof(tstamp), time(NULL))) {
        memset(buf, 0, sizeof(*buf));
        return S_INTERNAL_SERVER_ERROR;
    }
    if (buffer_appendf(buf,
                      "HTTP/1.1 %d %s\r\n"
                      "Date: %s\r\n"
                      "Connection: close\r\n",
                      res->status, status_str[res->status], tstamp)) {
        memset(buf, 0, sizeof(*buf));
        return S_INTERNAL_SERVER_ERROR;
    }
    for (i = 0; i < NUM_RES_FIELDS; i++) {
        if (res->field[i][0] != '\0' &&
            buffer_appendf(buf, "%s: %s\r\n", res_field_str[i], res->field[i])) {
            memset(buf, 0, sizeof(*buf));
            return S_INTERNAL_SERVER_ERROR;
        }
    }

    if (buffer_appendf(buf, "\r\n")) {
        memset(buf, 0, sizeof(*buf));
        return S_INTERNAL_SERVER_ERROR;
    }

    return 0;
}
```


3.2 Демонстрация работы

Для демонстрации корректности работы программы был написан простой веб-сайт, состоящий из трех файлов: index.html, styles.css, README.md. На рисунке 1 показан веб-сайт открытый в браузере Firefox.



The screenshot shows a web browser window with the address bar displaying 'localhost:8000'. The page content includes a title 'Repository', a link to 'Read more general information: [[Repository - General]]', and a description 'Rust implementation of Repository Design Pattern:'. Below this, there is a Rust code snippet that defines an Entity struct, a Repository trait, and an InMemoryRepository implementation. The code is as follows:

```
tags: #Programming #Patterns #Repository #Design links: [[000 Index|Index]] [[010 Programming MOC|Programming]] created: 202

Repository

Read more general information: [[Repository - General]]
Rust implementation of Repository Design Pattern:
// Entity struct represents a data model
struct Entity {
    id: u64,
    name: String,
}

// Repository trait defines the common methods for a repository
trait Repository<T> {
    fn get_all(&self) -> Vec<T>;
    fn find(&self, id: u64) -> Option<T>;
    fn add(&mut self, entity: T);
    fn delete(&mut self, id: u64);
}

// InMemoryRepository is an implementation of Repository trait using Vec as storage
struct InMemoryRepository<T> {
    storage: Vec<T>,
}

impl<T> InMemoryRepository<T> {
    fn new() -> Self {
        Self { storage: Vec::new() }
    }
}

impl<T> Repository<T> for InMemoryRepository<T>
where
    T: Clone + PartialEq,
{
    fn get_all(&self) -> Vec<T> {
        self.storage.clone()
    }

    fn find(&self, id: u64) -> Option<T> {
        self.storage.iter().find(|e| e.id == id).cloned()
    }
}
```

Рисунок 1 – Демонстрация работы

Для демонстрации работы сервера на больших файлах был использован образ установки ArchLinux, весящий 586 Мб и команда wget.

```
archeoss at ~ [ master ] ++(8) (180) [ v1.77.0-nightly ] [ 11s ]
└─> wget http://localhost:8000/archlinux-2018.11.01-x86_64.iso
--2023-12-26 14:18:12-- http://localhost:8000/archlinux-2018.11.01-x86_64.iso
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:8000... failed: Connection refused.
Connecting to localhost (localhost)|127.0.0.1|:8000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 614465536 (586M) [application/x-iso9660-image]
Saving to: 'archlinux-2018.11.01-x86_64.iso'

archlinux-2018.11.0 100%[=====>] 586.00M 194MB/s in 3.0s

2023-12-26 14:18:15 (194 MB/s) - 'archlinux-2018.11.01-x86_64.iso' saved [614465536/614465536]
```

Рисунок 2 – Демонстрация работы с большими файлами

Вывод

Программа была реализована на языке Си, проверка на корректность пройденна успешно.

4 Исследовательский раздел

4.1 Технические характеристики

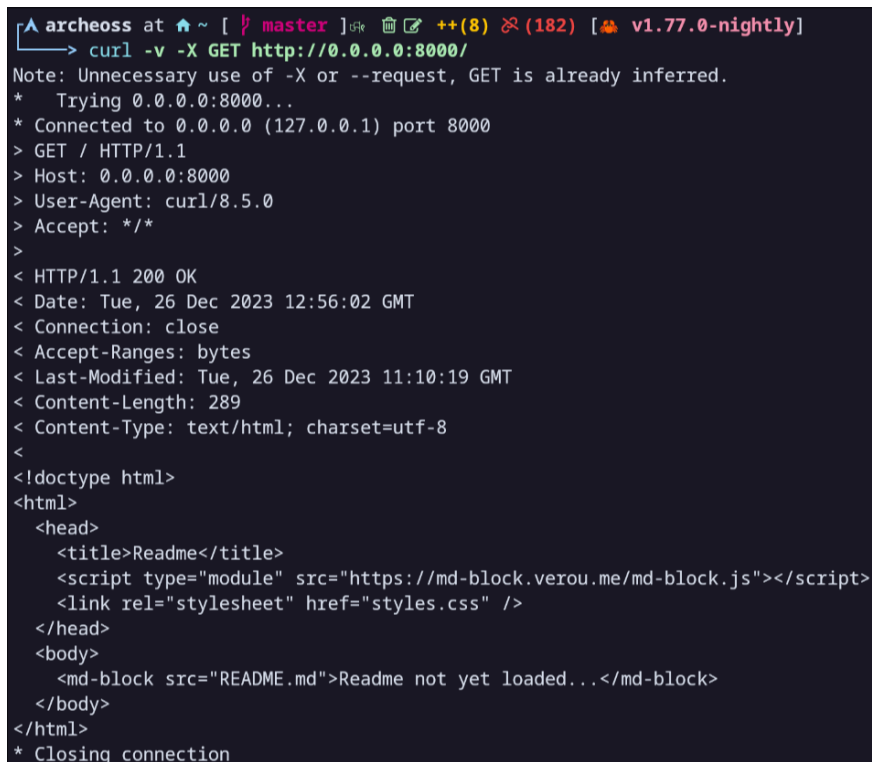
Ниже приведены технические характеристики устройства, на котором будет проведено исследование:

- Операционная система: Arch Linux [11] 64-bit;
- Количество ядра: 4 физических и 8 логических ядер;
- Оперативная память: 16 Гб, DDR4;
- Процессор: 11th Gen Intel® Core™ i5-11320H @ 3.20 ГГц [12].

Во время тестирования устройство было подключено к сети электропитания и было нагружено только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Работа сервера

На рисунке 3 показан пример GET запроса.



```
archeoss at ~ [ master ] ++(8) (182) [ v1.77.0-nightly ]
> curl -v -X GET http://0.0.0.0:8000/
Note: Unnecessary use of -X or --request, GET is already inferred.
* Trying 0.0.0.0:8000...
* Connected to 0.0.0.0 (127.0.0.1) port 8000
> GET / HTTP/1.1
> Host: 0.0.0.0:8000
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 26 Dec 2023 12:56:02 GMT
< Connection: close
< Accept-Ranges: bytes
< Last-Modified: Tue, 26 Dec 2023 11:10:19 GMT
< Content-Length: 289
< Content-Type: text/html; charset=utf-8
<
<!doctype html>
<html>
  <head>
    <title>Readme</title>
    <script type="module" src="https://md-block.verou.me/md-block.js"></script>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <md-block src="README.md">Readme not yet loaded...</md-block>
  </body>
</html>
* Closing connection
```

Рисунок 3 – Демонстрация работы: GET запрос

На рисунке 4 показан пример HEAD запроса.

```

archeoss at ~ [ master ] ++(8) (182) [ v1.77.0-nightly ]
→ curl -v -I http://0.0.0.0:8000/
* Trying 0.0.0.0:8000...
* Connected to 0.0.0.0 (127.0.0.1) port 8000
> HEAD / HTTP/1.1
> Host: 0.0.0.0:8000
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Tue, 26 Dec 2023 12:58:29 GMT
Date: Tue, 26 Dec 2023 12:58:29 GMT
< Connection: close
Connection: close
< Accept-Ranges: bytes
Accept-Ranges: bytes
< Last-Modified: Tue, 26 Dec 2023 11:10:19 GMT
Last-Modified: Tue, 26 Dec 2023 11:10:19 GMT
< Content-Length: 289
Content-Length: 289
< Content-Type: text/html; charset=utf-8
Content-Type: text/html; charset=utf-8
<
* Closing connection

```

Рисунок 4 – Демонстрация работы: HEAD запрос

На рисунке 5 показан пример запроса несуществующего файла.

```

archeoss at ~ [ master ] ++(8) (182) [ v1.77.0-nightly ]
→ curl -v -X GET http://0.0.0.0:8000/test.txt
Note: Unnecessary use of -X or --request, GET is already inferred.
* Trying 0.0.0.0:8000...
* Connected to 0.0.0.0 (127.0.0.1) port 8000
> GET /test.txt HTTP/1.1
> Host: 0.0.0.0:8000
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Date: Tue, 26 Dec 2023 13:01:09 GMT
< Connection: close
< Content-Type: text/html; charset=utf-8
<
<!DOCTYPE html>
<html>
  <head>
    <title>404 Not Found</title>
  </head>
  <body>
    <h1>404 Not Found</h1>
  </body>
</html>
* Closing connection

```

Рисунок 5 – Демонстрация работы: GET запрос к несуществующему файлу.

На рисунке 6 показан пример записи логгера.

```
./main: [0]: 0
2023-12-26T13:04:00Z    127.0.0.1      404    0.0.0.0 /test.txt
./main: [0]: 0
2023-12-26T13:04:09Z    127.0.0.1      200    0.0.0.0 /
./main: [0]: 0
2023-12-26T13:04:20Z    127.0.0.1      200    0.0.0.0 /
./main: [0]: 0
2023-12-26T13:04:37Z    127.0.0.1      200    0.0.0.0 /README.md
```

Рисунок 6 – Демонстрация работы: Запись логгера

4.3 Нагрузочное тестирование

Нагрузочное тестирование проводилось с помощью ApacheBenchmark, 1000 запросов файла 27Мб 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	Объем передачи
nginx	25.373с	39.41	25.373мс	1.092Гб/с
Thr.Pool и Poll	24.651с	40.57	24.651мс	1.112Гб/с

Вывод

Проведено исследование и нагрузочное тестирование разработанного приложения и nginx, в ходе которого выявлено, что сервер nginx показал схожую производительность приведенной имплементации.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы был проведен анализ предметной области, сформированы требования к приложению. Разработано приложение, проведены тесты для корректной работы программы. Также проведены стресс тесты для тестирования поведения при большой нагрузке. Таким образом, были выполнены следующие задачи:

1. проведена формализацию задачи и определен необходимый функционал;
2. исследована предметная область веб серверов;
3. приложение спроектировано;
4. приложение реализовано;
5. приложение протестировано на предмет корректности;

Все поставленные задачи были выполнены успешно.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Олифер, Н. Олифер. Компьютерные сети. Принципы, технологии, протоколы. Учебник – Питер, 2016. – 996 с.
2. Таненбаум. Компьютерные сети. 6-е изд. – Питер, 2024 – 992 с.
3. Li P., Zdancewic S. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives. — 2007.
4. M. S. W. R. F. B. R. A. // Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition). — Addison-Wesley, 2003.
5. Daemon implementation. — Режим доступа: <https://www.w3.org/Daemon/Implementation/> , свободный (дата обращения: 13 февраля 2024 г.)
6. Pai V., Druschel P., Zwaenepoel W. Flash: An efficient and portable Web server. // . — 01.1999. — С. 199—212.
7. SEDA. — Режим доступа: <http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html>, свободный (дата обращения: 13 февраля 2024 г.)
8. Apache MINA [Электронный ресурс]. — Режим доступа: <https://mina.apache.org/>, свободный (дата обращения: 13 февраля 2024 г.)
9. Mulesoft [Электронный ресурс]. — Режим доступа: <https://www.mulesoft.com/>, свободный (дата обращения: 13 февраля 2024 г.)
10. Li P., Zdancewic S. 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. — New York, NY, USA : Association for Computing Machinery, 2007. — С. 189—199. — (PLDI '07). — ISBN 9781595936332.

11. Arch Linux [Электронный ресурс]. — (дата обращения: 10.09.2023). Режим доступа: <https://archlinux.org/>.
12. Процессор Intel® Core™ i5-11320H [Электронный ресурс]. — Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html>.