Пояснювальна записка
до дипломного проекту
на тему: «Сервер з низькорівневим кешуванням статичних файлів»

Київ – 2019 рік

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 4

ВСТУП 5

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ 7

1.1. Огляд та аналіз існуючих рішень 7

1.1.1 Nginx 7

1.1.2 Apache HTTP Server 10

1.1.3 Мережа доставки контенту 12

1.2 Вискорівневе та низькорівневе кешування 13

1.3 Опис системних викликів для роботи з файлами 16

1.4 Сумісність системного sendfile та протоколів шифрування............ 19

1.5 Опис механізму кешування сторінок операційною системою......... 20

ВИСНОВКИ ДО РОЗДІЛУ 1 24

2 АНАЛІЗ ТА ВИБІР ТЕХНІЧНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ СЕРВЕРУ СТАТИКИ. ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ 26

2.1 Огляд моделей конкурентності і паралелізму.................................. 27

2.1.1 Мультипотоковість 27

2.1.2 Конкурентна модель 29

2.1.3 Асинхронний підхід 31

2.2 Вибір мови програмування 33

2.2.1 Rust.. 34

2.2.2 Go 35

ВИСНОВКИ ДО РОЗДІЛУ 2 37

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 38

3.1. Опис задачі 38

3.2 Опис моделі паралельної обробки 39

3.3 Опис структури програми 40

3.4 Опис реалізації кешування 43

3.5 Опис конфігурації 46

3.6 Інструкція для користувача 48

3.6.1 Вимоги до програмного оточення 48

3.6.2 Компіляція в бінарний файл 49

3.6.3 Конфігурація проекту 50

3.6.4 Запуск серверу 52

ВИСНОВКИ ДО РОЗДІЛУ 3 53

4 НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ 54

4.1 Тестування без шифрування 54

4.2 Тестування з шифруванням 56

ВИСНОВКИ ДО РОЗДІЛУ 4 58

ВИСНОВКИ 59

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ 60

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CDN	Content Delivery Network – мережа доставки контенту
HTTP	Hyper Text Transfer Protocol – протокол передачі даних прикладного рівня
JSON	JavaScript Object Notation – формат серіалізації
SSL	Secure Sockets Layer – криптографічний протокол
TCP	Transmission Control Protocol – протокол транспортного рівня
TLS	Transport Layer Security – криптографічний протокол
URL	Uniform Resource Locator – унікально адреса певного ресурсу
YAML	Yet Another Markup Language – формат серіалізації

ВСТУП

Сьогодні веб-сайти перетворилися з простих статичних документів, які складаються з тексту і меню, в динамічні, функціональні додатки. І разом з цим збільшилася роль статичних файлів і, звісно ж, їх розмір. Виникає проблема з навантаженням, що призводить до погіршення якості обслуговування. Якщо говорити про мінімалістичну трьохрівневу архітектуру для невеликого веб додатку на основі одного веб серверу та бази даних, то це не матиме великого значення. Проте, якщо важлива масштабованість додатку, то спосіб збереження і обслуговування статичного контенту сильно впливає на досягнення необходної швидкості роботи та рівень економічних затрат.

В сучасних масштабованих системах для обслуговування статичного контенту застосовуються спеціалізовані рішення. Вони мають забезпечувати не тільки високий рівень продуктивності, а також гнучку конфігурацію, паралельну обробку великої кількості запитів, безпеку, можливість версіювання файлів.

Об'єктом дослідження є фактори, які найбільше впливають на швидкість отримання статичного контенту користувачем від спеціалізованого сервера. По-перше, пропускна здатність та затримка каналу. По-друге, можливості операційної системи сервера. По-третє, програмна реалізація сервера. На перший фактор розробники не завжди здатні впливати. Модифікувати роботу мережевого стеку операційної системи є занадто складним завданням. Тоді залишається лише третій фактор – це оптимізувати роботу самого сервера статичних файлів. Проте можливості для оптимізації також часто є обмеженими архітектурою сервера, і єдиними рішенням в такому випадку є горизонтальне масштабування, що призводить до значних економічних витрат.

Темою даної роботи є дослідження можливостей спеціалізованих серверів, створення власної реалізації на основі отриманих результатів з використанням сучасної мови програмування системного рівня – Rust. Основними вимогами до реалізації є висока пропускна здатність, низьке споживання ресурсів та можливість гнучкої конфігурації сервера.

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

Для обслуговування статичного контенту застосовують різні підходи. В веб-додатках малого масштабу цим часто займається сервер, який також обробляє запити до основного серверу. Це найпростіший підхід, який вимагає мінімальних економічних затрат, проте продуктивність обробки як статичного, так і динамічного зменшується.

В масштабованих системах оптимальним рішенням є обслуговувати статичний контент окремо від програмного веб сервера додатку, інакше це значно обмежує можливість горизонтального масштабування програми. По-перше, це призведе до того, що сервер буде займатися введенням та виведенням замість того, щоб виконувати свою основну роботу – формувати динамічний контент, особливо враховуючи те, що віддача статичного контенту в більшості випадків вимагає значно більше часу, ніж динамічного. Причиною цього є розмір файлів.

Існують спеціалізовані сервери, які здатні віддавати статичні файли досить швидко – Nginx, Apache Web Server, Lighthttpd. Дані реалізації здатні витримувати велику кількість з'єднань і використовують всі можливості кешування протоку HTTP [1].

1.1. Огляд та аналіз існуючих рішень

1.1.1 Nginx

Nginx – високопродуктивний HTTP-сервер і балансувач навантаження, а також проксі-сервер IMAP / POP3. Зарекомендував себе як один із найшвидших та стабільних серверів. Має низьке споживання ресурсів та високу здатність до масштабування.

Висока здатність при обслуговуванні статичного контенту досягається за допомогою асинхронного підходу. Nginx завжди має головний процес, який застосовує конфігурацію та приймає запити, а потім передає дочірнім процесам, які і обслуговують їх. Таким чином, чим більше процесів – тим більша здатність до обробки запитів. Крім того Nginx має декілька специфічних процесів – кешуючий процес та інвалідатор закешованих даних. Перший відповідає за перевірку збережених даних на диску та відповідність мета-інформації в базі даних сервера. Другий зайємається перевіркою актуальності кешованих даних та їх інвалідацією.

Стандартна конфігурація даного сервера не є оптимальною для обробки великою кількості запитів. Розробниками була включена можливість використовувати системні виклики, один із них – sendfile [2]. Якщо ця можливість вимкнена, то Nginx самостійно зчитує файли з диску, що призводить до великої затримки. Системний виклик sendfile – копіює дані з одного файлового дескриптору напряму в інший, а в нашому випадку – в сокет [3]. Оскільки копіювання здійснюється всередині ядра операційної системи, sendfile є ефективнішим за комбінацію інших системних викликів – read та write, які Nginx використовує в стандартній конфігурації. Якщо з'єднання встановлене за допомогою TLS/SSL протоколу, sendfile втрачає свою ефективність.

Nginx не підтримує кешування файлів в оперативній пам'яті. Оскільки одночасних запитів на отримання файлів може бути багато, то можлива проблема отримання файлового дескриптора у операційної системи для кожного запиту. Ця проблема в Nginx вирішується директивою open_file_cache [3], яка дозволяє зберігати відкриті файлові дескриптори протягом деякого часу.

Nginx при кешуванні покладається кеш сторінок операційної системи. Вважається, що найбільш необхідні файли завжди будуть в кеші.

Проте, на даний механізм краще не покладатися, що буде описано в наступних розділах. З іншого боку, воркери в Nginx – це окремі процеси. Кешувати дані у власній пам’яті означає завантажити окрему копію файлу для кожного процесу, що призведе до надмірного використання оперативної пам’яті.

Nginx має деякі недоліки в своїй асинхронній моделі [4]. Хоча головний процес і працює в повністю асинхронному режимі, але дочірні процеси – workers – блокуються при виконанні I/O операцій, наприклад, читання з диску. Таким чином, сотні прийнятих простих запитів можуть очікувати виконання лише декількох запитів, які заблокували всі доступні дочірні процеси. Дану проблему розробники Nginx вирішили за допомогою пулу потоків для кожного дочірнього процесу – таким чином, важкі блокуючі операції, які не потребують використання процесора, можуть виконуватися в окремому потоці, що дозволяє процесу паралельно обробляти інші запити. Більш того, у операційних системах з’являються нові інтерфейси для читання з диску, наприклад, AIO, підтримка якого планується в Nginx в найближчих версіях.

Оскільки Nginx має пул процесів, кількість яких конфігурується, він має здатність до масштабування на багатоядерних системах.

Асинхронність в повному обсязі проявляється при обслуговуванні декількох HTTP запитів в одному постійному TCP-з'єднанні [5]. Один воркер може обслуговувати тисячі з’єднань одночасно. Такий результат досягається за допомогою механізму «швидкого циклу», в якому перевіряються і обробляються події, наприклад, черговий HTTP запит. Розділення обов’язків між головним процесом і воркерами дозволяє останнім обробляти лише конкретні події і не займатися встановленням з’єднання. Кожне з’єднання, яке обробляє воркер, поміщається до «циклу подій», в якому всі з’єднання обробляються асинхронно. За рахунок того,

що не створюється окремий процес на кожне з’єднання, Nginx добре масштабується та має низьке споживання пам’яті і CPU.

Оскільки Nginx не створює процес на кожне з'єднання, використання пам'яті є надзвичайно ефективним у більшості випадків. Nginx також не витрачає процесорний час оскільки відпадає необхідність у постійному створенні процесів та потоків. Основний алгорим роботи полягає у ініціалізації з’єднання, додавання до циклу подій і асинхронній обробці, після чого з’єднання вилучається з циклу За рахунок використання викликів операційної системи і детермінованої реалізації пулу потоків та процесів, Nginx досягає наднизького використання процесора навіть при великому навантаженні.

1.1.2 Apache HTTP Server

Apache HTTP Server – HTTP сервер та балансувач навантаження, розроблений Apache Software Foundation. Є найпопулярнішим на сьогодні веб сервером.

Apache має гнучку модульну систему та велику екосистему модулів. Також можна обрати один з трьох можливих підходів до обробки з’єднань [6]:

mpm_prefork – створює на кожне з’єднання окремий процес з єдиним потоком. Кожен процес обробляє лише одне з’єднання одночасно. До тих пір, поки кількість з’єднань менше кількості логічних ядер процесора, цей підхід працює дуже швидко. Споживає значний обсяг оперативної пам’яті, важко масштабується.

mpm_worker – на початку роботи сервер створює деяку кількість процесів, яка в процесі не змінюється. Кожен процес має декілька потоків, що дає змогу обрати декілька запитів паралельно. Оскільки потоки значно

ефективніші за процеси, то mpm_worker краще масштабується за mpm_prefork.

mpm_event – підхід схожий на mpm_worker, але оптимізований для роботи з постійними TCP з’єднаннями. На кожне з’єднання створюється окремий потік, який постійно прослуховує TCP з’єднання.

Для обслуговування статичного контенту перший варіант не підходить, оскільки читання з диску повністю заблокує процес. При другому та третьому варіантах блокування процеса відсутнє. Проте, підхід Nginx краще, оскільки в ньому потоки використовуються лише для довготривалих блокуючих операціях, а для паралельної обробки – асинхронність.

В Apache HTTP Server присутні декілька важливих оптимізацій для обслуговування статичних файлів: Це кешування файлових дескрипторів, кешування «ключ-значення» для оптимізації роботи протоколу SSL [7].

Кешування файлових дескрипторів відрізняється від реалізації в Nginx. На початку роботи сервер зберігає всі необхідні дескриптори, уникаючи створення нового дескриптора на кожен запит. Ця оптимізація актуальна для повільних файлових систем .

Кешування файлів в оперативній пам’яті. На початку роботи сервер завантажує вмісте файлу в пам’ять. Це дозволяє дуже швидко віддавати файли, але сервер не контролює доступну пам’ять і це може призвести до неможливості обслуговувати запити. Головний недолік, що кожен дочірній процес копіює контент файлу, що дуже збільшує обсяг необхідної пам’яті для такого кешування.

Оптимізації з кешуванням виконуються лише при старті і кеш ніколи не інвалідується. Якщо вміст файлу змінився, то кешовані дані або дескриптори будуть неактуальними.

Apache має модуль для кешування спеціальних об’єктів для протоколу SSL [8]. Зокрема, кеш «ключ-значення» може

використовуватися для перевірки деталей аутентифікації, сеансів SSL і забезпечення кешування деяких частин SSL.

1.1.3 Мережа доставки контенту

CDN (Content Delivery Network) – мережа серверів зі спеціалізованим програмним забезпеченням, створена для швидкого обслуговування статичних файлів. Основна перевага таких мереж – це наявність розташованих серверів в різних регіонах, що дозволяє знизити затримку між запитом та відповіддю.

Хоча економія не є пріоритетною задачею такої системи, проте вона теж важлива. Передати файли на сервер з іншого континенту один раз і роздавати статику звідти є набагато дешевшим і оптимальнішим варіантом, чим завантажувати ці дані кожен раз іншого континенту. Дуже актуальним це є для відеохостингів, у яких суттєвий трафік [9].

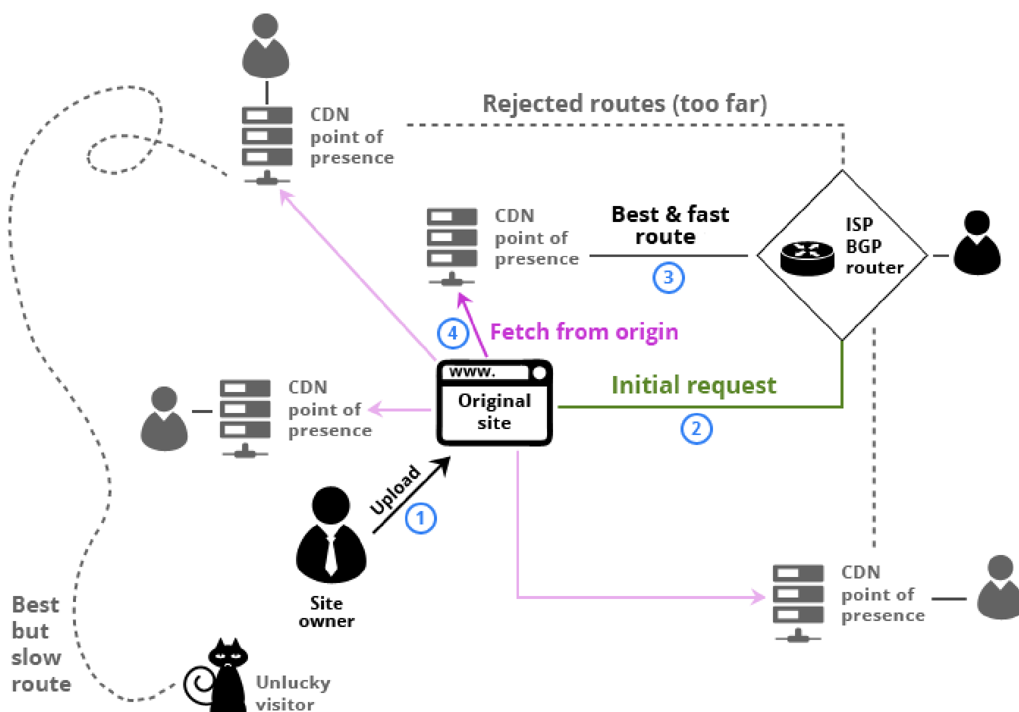

Рисунок 1.1 – Схема роботи CDN [10]

Проте CDN накладає свої обмеження. Необхідно періодично синхронізувати статичні файли і підтримувати цілісність складної системи.

Технічно мережа доставки статичних файлів не є аналогом рішення, яке розробляється в рамках цієї роботи, скоріше вона є областю для використання розробленого сервера.

1.2 Вискорівневе та низькорівневе кешування

Кешування заключається в тому, щоб зберігати вже отримані статичні дані для використання в наступних запитах. Запити, які вже знаходяться в кеш-сховищі, замість того, щоб знову звертатися до початкового серверу, отримають потрібні дані з локального кеша. Такий підхід має дві основні переваги [11]:

1. Отримання даних з локального сховища швидша за повноцінний запит на сервер на декілька порядків оскільки сховище, у випадку з HTTP кешуванням, знаходиться прямо на диску користувача.
2. Значне зниження навантаження на сервер, який обробляє запити лише тих користувачів, які ще не мають даних.

Проте, дані часто втрачають свою актуальність, тому необхідно зберігати кешовані ресурси лише до того моменту, доки вони залишаються в початковому стані.

Існують декілька типів кешування – колективного використання та приватного. В першому варіанті дані зберігаються на спеціальному кешуючому сервері, яким можуть користуватися різні клієнти. Такі сервери підтримуються провайдерами або компаніями в локальній мережі.

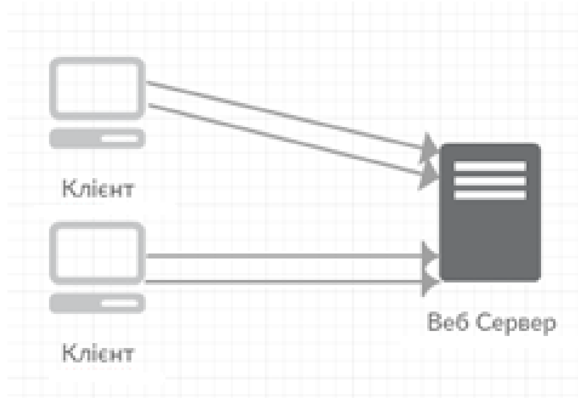

Рисунок 1.2 - Запити без кешування

На рис Х. зображено схему запитів з колективним кешем. Кеш-сервер зберігає результати лише одного запиту. Проте така схема має сенс лише в невеликих мережах.

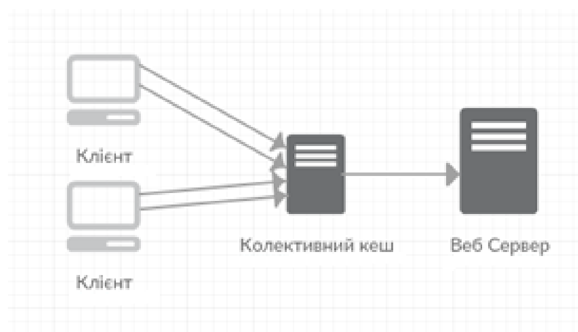

Рисунок 1.3 - Запити з колективним кешуванням

Приватний кеш призначений лише для одного користувача та зберігається на диску пристрою. Найшвидший вид кешування.

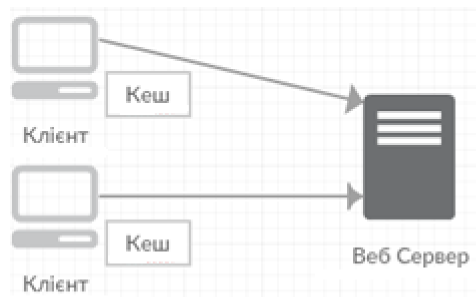

Рисунок 1.4 - Запити з приватним кешуванням

HTTP кешування працює за допомогою простих текстових заголовків. Основним заголовком, який регулює кешування є Cache-Control, що з’явився у версії HTTP/1.1. Значення no-cache забороняє кешувати ресурс. Директиви private та public відповідають за місце кешування. Public означає, що ресурс може кешуватися на колективних серверах, а private дозволяє зберігати копію відповіді лише у власному сховищі користувача, оскільки відповідь може містити дані автентифікації або статус коди, які не кешуються.

Важливою директивою є max-age, яка містить максимальний час, який ресурс може знаходитись в кеші. Від хедера Expires вона відрізняється тим, що прив’язана до моменту запиту.

Основною проблемою HTTP-кешування є інвалідація кешу – це процес перевірки актуальності наявних ресурсів і при негативному результаті завантаження нових даних. Така інвалідація доступна доступна за допомогою власних заголовків сервера (які є частиною протоколу HTTP), за допомогою спеціального заголовку E-Tag або програмної реалізації на основі GET-параметрів.

Кешування на основі HTTP є високорівневим оскільки описується за допомогою декларативних директив і працює лише для одного конкретного користувача. Але на той випадок, коли статичні файли оновилися або додалися нові, сервер має швидко обробляти велику кількість нових запитів.

Низькорівневий кеш, який є темою даного диплома, є методом оптимізації роботи сервера. Такий кеш реалізується в основному за допомогою можливостей операційної системи, а саме кешування сторінок файлової системи, кешування файлових дескрипторів, використання асинхронних обробників.

1.3 Опис системних викликів для роботи з файлами

При роботі з файлами, що є основною задачею статичного сервера, використовуються виклики операційної системі.

Системний виклик read наступну сигнатуру ssize_t read(int fd, void *buf, size_t count), де fd – номер файлового дескриптора, buf – байтовий буфер, куди буде зчитано дані, size_t – скільки байтів необхідно зчитати. Даний системний виклик намагається зчитати задану кількість байт з файлового дескриптора в буфер buf. Операція читання починається з поточного зміщення файлового дескриптора, а отже паралельне читання файлу засобами операційної системи неможливе. Після читання виклик повертає кількість зчитаних байтів. Зчитування меншої кількості байтів, ніж було задано не вважається помилкою. Таким чином, якщо файл повністю прочитаний, то буде повернено 0, а в буфер нічого не записано. Якщо при читанні сталася помилка, то системний виклик повертає значення -1 та записує опис помилки до спеціальної змінної errno [12]. За один виклик може бути зчитано не більше 2147479552 байт, що регламентується стандартом POSIX. Оскільки в операційній системі пам’ять розділяться на user space і kernel space, а системні виклики працюють лише в останньому, то необхідно зчитати дані спочатку в простір ядра, а потом звідти скопіювати в пам’ять програми.

Системний виклик write має сигнатуру ssize_t write(int fd, const void *buf, size_t count), де fd – номер файлового дескриптора, buf – байтовий буфер, звідки будуть записані дані, size_t – скільки байтів необхідно записано. Даний системний виклик намагається зчитати задану кількість байт з файлового дескриптора в буфер buf. Операція запису починається з поточного зміщення файлу по заданому дескриптору. Якщо файл відкритий в режимі O_APPEND, то перед записом зміщення встановлюється в кінець файлу. Системний виклик повертає кількість записаних байт, яка може бути меншою за count якщо, наприклад,

недостатньо місця. Якщо сталася помилка, то повертається значення -1. Помилки можуть статися при спробі паралельного запису, відсутності вільного місця або переході сокета в заблокований стан. Аналогічно до системного виклику read, дані спочатку копіюються в простір ядра, а потім в файл по дескриптору. Максимальна кількість байтів за один виклик – 2147479552 [12].

Копіювання з простору користувача в простір ядра викликає накладає обмеження на швидкість роботи. Наприклад, якщо потрібно передати файл з диску в сокет розміром 1 мегабайт, необхідно зчитувати по 552 байт в пам’ять програми через read, а потім записувати до сокету системним викликом write [13]. Операційна система насправді копіює в два рази більше даних, що призводить до вдвічі більшого часу роботи.

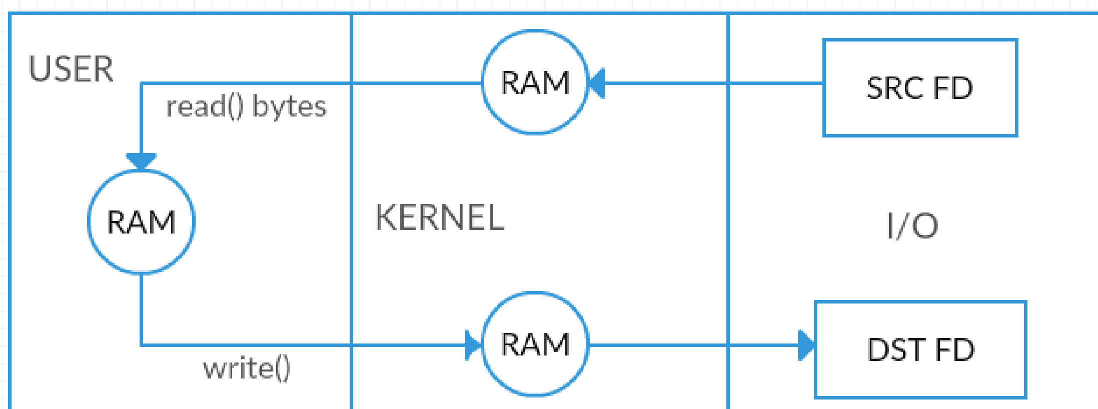

Рисунок 1.5 - Копіювання даних викликами read та write

Системний виклик sendfile був створений для вирішення проблеми подвійного копіювання. При використанні, дані не пересилаються в простів користувача, а відразу прямують до вихідного файлового дескриптора. Даний системний виклик має сигнатуру ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count), де out_fd – файловий дескриптор, куди копіюється дані, in_fd – вихідний файловий дескриптор, звідки копіюються дані, offset – поточне зміщення копіювання, count –

кількість байт даних для копіювання. Виклик sendfile пересилає вміст одного файлового дескриптора в інший. Основною перевагою виклику є пересилання всередині простору ядра, а отже дані не копіюються в простір користувача, що робить його ефективнішим за комбінацію read та write [3]. Після виконання змінна offset зберігає нове зміщення при копіюванні. Даний системний виклик не модифікує зміщення вхідного файлового дескриптора. Важливим обмеженням є необхідність того, щоб вхідний дескриптор підтримував mmap операції, що робить неможливим пересилати дані з сокету. Виклик sendfile може переслати даних менше, чим було зазначено, тому потрібно оброблювати таку ситуацію і відправляти недоотримані байти. За один виклик можливо переслати до 552 байтів. Виклик повертає кількість скопійованих байт. Якщо сталася помилка, то повертається -1.

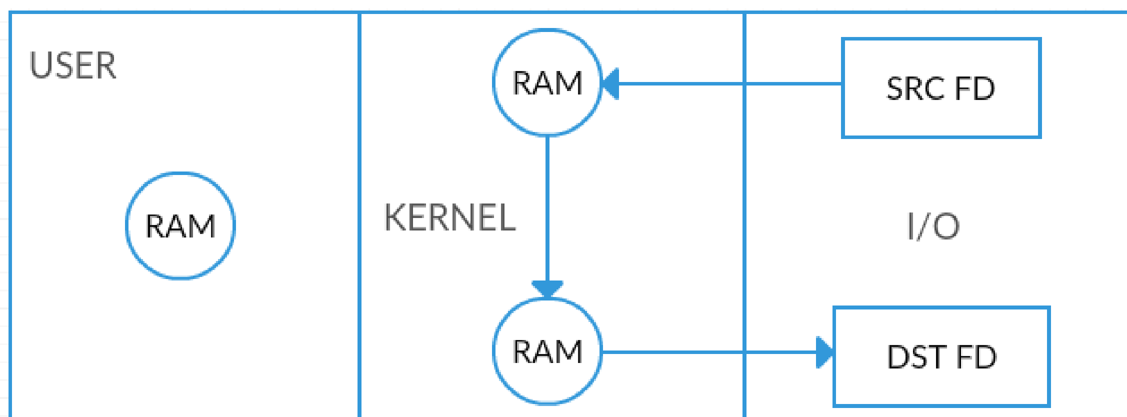

Рисунок 1.6 - Пересилання за допомогою sendfile

На рисунку X можна побачити, що дані зчитуються в простір ядра і звідти відразу відправляються в вихідний файловий дескриптор.

Були проведені проведені порівняння двох підходів при пересиланні файлів. Перший тест на пересилання одного файлу розміром 1 гігабайт, другий – пересилання 100 файлів розміром 10 мегабайт:

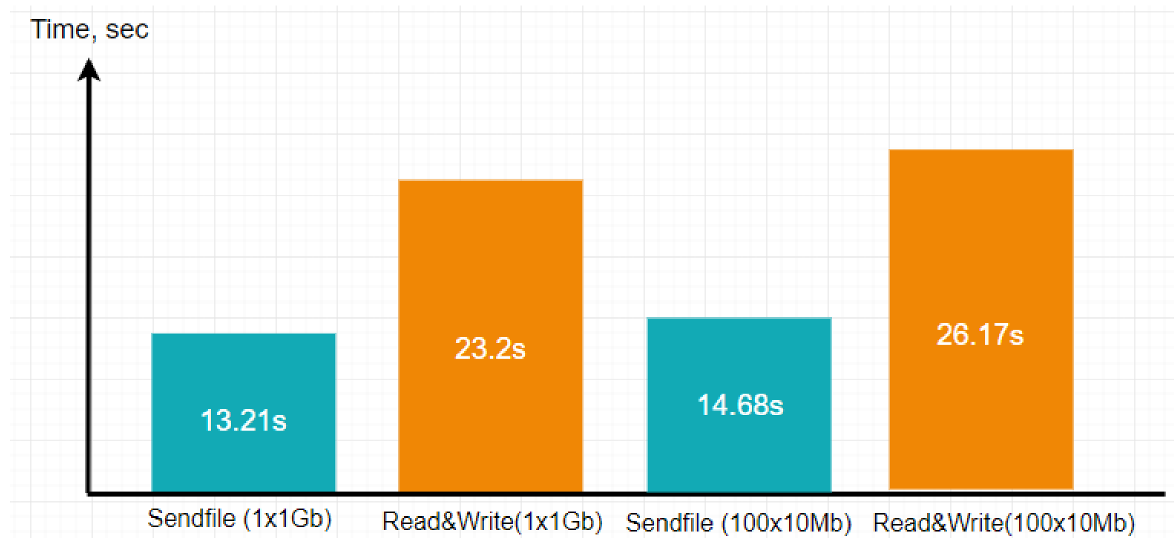

Рисунок 1.7 - Результати тестів

Результати тестів продуктивності показують очікуваний приріст майже в два рази.

Роботу Nginx при читанні з диску можливо прискорити за рахунок використання системного виклику readahead [14], який дозволяє зменшити кількість звертань до диску [15].

1.4 Сумісність системного sendfile та протоколів шифрування

Протокол HTTP першої версії не включає в себе шифрування, а це означає, що при передачі будь-яких даних зловмисник, який перехопив мережеві пакети, зможе їх прочитати. Більш того, зловмисник може проксувати передачу даних між клієнтом і сервером і підміняти дані, що є критичною проблемою. Це справедливо і для статичного контенту. В такому випадку можлива ситуація, коли клієнт отримає модифікований сценарій виконання, в якому, наприклад, дані для авторизації відсилається не на відповідний сервер, а зловмиснику. Тому для статичних серверів підтримка з'єднань протоколу SSL/TLS є ключовою вимогою, що унеможливлює атаки man-in-middle.

Операційна система Linux не підтримує реалізацію SSL/TLS протоколу на системному рівню. Це компенсується програмними реалізаціями, наприклад, OpenSSL, а це означає, що шифрування відбувається в просторі користувача. OpenSSL вимагає відкритий файловий дескриптор для передачі. Системний виклик sendfile повністю ігнорує простів користувача і працює у просторі ядра. Тому sendfile і TLS не сумісні, а оскільки від шифрування відмовитися не є можливим, sendfile втрачає свою актуальність.

1.5 Опис механізму кешування сторінок операційною системою

При роботі з файлами в операційних системах виникає декілька проблем. Основна – на декілька порядків менша швидкість роботи жорстокого диска порівняно з оперативною пам’яттю. Можна згадати про твердотільні диски, які зараз набирають популярність, проте є дуже дорогими і мають значно менший ресурс використання – 3 роки проти 10 в середньому. І продуктивність, яку вони забезпечують, все одно менше за швидкість оперативної пам’яті. Системи RAID – технологія об’єднання декількох фізичних дисків в один логічний для забезпечення – прискорюють роботу лише в декілька раз, чого недостатньо. Друга – можливість використовувати одного завантаженого в пам’ять файлу багатьма процесами. Наприклад, всі Linux програми вимагають libc або ld.so. Якщо б кожен процес завантажував їх в пам’ять, це б призвело до надмірного використання оперативної пам’яті.

Дані проблеми вирішуються за допомогою кешування сторінок файлової системи в оперативній пам’яті. Взагалі, будь-які операції введення та виведення працюють через даний кеш. Наприклад, коли виконується системний виклик read, то дані спершу завантажаться саме в кеш сторінок, а потом скопіюються в пам’ять програми. При виконанні

write, весь буфер скопіюється в даний кеш, а потім уже потрапить у файл по дескриптору. Сторінковий кеш знаходиться в частині оперативної пам'яті, яка в даний момент не використовується програмами. Якщо така пам'ять відсутня, то кеш буде існувати досить короткий період часу – до наступного читання або запису. Виникає питання – куди операційна система складе дані для виклику read(), якщо вільна пам'ять відсутня? Окремі частини пам'яті будуть переміщенні на другорядне сховище. Цей механізм називається – swapping. Цей процес займає час, адже необхідно скопіювати дані на повільний диск. Тому вільна оперативна пам'ять – це такий же важливий ресурс, як і процесорний час. На дане рішення краще не покладатися, оскільки насправді це складно назвати кешуванням. Операційна система записує всі дані, які були зчитані, і не очищує їх до наступного читання або запису. Наприклад, на даний момент в кеші знаходиться 1000 порівняно малих файлів і програма їх постійно читає, і робить це швидко. Потім, операційній системі всього один раз необхідно зчитати величезний файл – вона читає, весь кеш займає лише один файл, і тепер кожне звертання до попередніх файлів вимагає читання з диску. А величезний файл був використаний лише раз.

Nginx, на жаль, покладається саме на даний механізм. Вважається, що найбільш необхідні файли завжди будуть в кеші, але потрібно враховувати описану вище ситуацію. З іншого боку, воркери в Nginx – це окремі процеси. Кешувати файли означає завантажити окрему копію файлу для кожного процесу, що призведе до надмірного використання оперативної пам'яті.

У кешування сторінок є недолік. Оскільки даний кеш знаходиться в просторі ядра, то будь яке читання кешованого файлу обов'язково скопіює дані в пам'ять програми. При 3 читаннях одного файлу буде отримано 4 копії даних в пам'яті – одна кешована і 3 в пам'яті програми,

а це призводить до надмірного використання оперативної пам’яті. Існує рішення і для даної проблеми.

Відображення файлів в пам’ять – створення віртуального відображення за допомогою системного виклику mmap. Це ще одна можлива оптимізація читання файлів з диску, яку надає ядро операційної системи, що дозволяє відобразити файл в оперативну пам’ять. Хоча файл знаходиться в просторі ядра, програма може звертатися до нього, як до будь-якого іншої частини пам’яті. Навіть існує можливість записувати в дану частину пам’яті для того, щоб потім відобразити ці зміни у файлі на диску.

Системний виклик mmap має сигнатуру void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset), де addr – початкова адреса, де буде розміщено відображення, length – кількість байт, які необхідно записати в пам’ять по заданій адресі, port задає режим роботи з відображеним файлом, наприклад, опція PROT_READ дозволяє лише читання, а PROT_WRITE дозволяє змінювати дану пам’ять, що відобразиться на диску. Параметр fd задає файловий дескриптор, за яким необхідно зчитати дані. Цей параметр не може вказувати на файловий дескриптор сокета аналогічно до системного виклику sendfile. Параметр offset вказує поточне зміщення в файлі. Параметр addr може мати значення NULL, що дозволяє операційній системі обрати будь-яку доступну адресу пам’яті, проте таку, що дорівнює або більше за /proc/sys/vm/mmap_min_addr. Основною перевагою даного системного виклику є відсутність копіювання даних в пам'ять програми. Саме цей механізм використовується для економії оперативної пам’яті операційною систему для уникнення дублювання ключових бібліотек – libc, ld.so і т.д.

Виклик mmap мало підходить для кешування контенту на сервері статичних файлів, оскільки даний кеш є менш пріоритетним для

операційної системи, ніж пам’ять програм. Таким чином, якщо деяка програма запросить виділення пам’яті, система очистить даний кеш. В такому випадку відображення зберігається і файл буде доступний і далі по заданій адресі, проти зчитуватися буде з диску.

ВИСНОВКИ ДО РОЗДІЛУ 1

У даному виконання першого розділу дипломного проекту було розглянуті існуючі рішення, проведено аналіз їх можливостей, досліджено принцип роботи для обробки великої кількості паралельних запитів. Крім того, були розглянуті можливості операційної системи для оптимізації роботи статичного серверу – кешування сторінок пам’яті, системні виклики, описані проблеми та обмеження, які виникають при використанні розглянутих оптимізацій. На основі цього розділу можемо зробити наступні висновки:

1. При розробці серверу статичного контенту найбільшу роль відіграє модель паралельної обробки запитів, оскільки саме вона визначає можливість сервера до масштабування на багатоядерних системах. Найбільш оптимальним є рішення, за якого при старті створюється пул процесів, кількість яких відповідає кількості доступних логічних ядер системи. Це значно зменшує використання ресурсу процесора, оскільки відпадає потреба у створенні нових процесів в ході роботи програми і подальшого їх обслуговуванні. Тому модель обробки запитів, яка передбачає створення процесу на кожне з’єднання стає неактуальною. Серед існуючих моделей, асинхронна є найоптимальнішою, оскільки потребує постійної кількості процесів або потоків і при цьому дозволяє в повному обсязі використовувати процесор за рахунок відсутні блокуючих операції.
2. Алгоритм роботи розглянутих рішень базується на використанні багатьох процесів, що накладає обмеження на кешування даних в пам’яті програми. Розробники даних програмних рішень покладаються на вбудовані можливості операційної системи, проте, як було розглянуто в розділі, на них краще н6е покладатися, оскільки

неможливо передбачити поведінку системи за умови, що кешування має менший пріоритет за робочу пам’ять програм.

3. Основною проблемою кешування статичних файлів є використання кешованих даних багатьма процесами. Системний виклик mmap створений для її вирішення, але для реалізації повного потенціалу даної команди потрібно підтримувати достатньо кількість вільної оперативної пам’яті. Системний виклик sendfile працює швидко без використання оперативної пам’яті, проте перестає бути актуальним, якщо необхідно забезпечити шифрування.

2 АНАЛІЗ ТА ВИБІР ТЕХНІЧНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ СЕРВЕРУ СТАТИКИ. ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ

При розробці серверу статичного контенту найбільшу роль відіграє модель паралельної обробки запитів, оскільки саме вона визначає можливість сервера до масштабування на багатоядерних системах, оптимальність використання ресурсів процесора та пам'яті.

При розробці статичного сервера необхідно враховувати сучасні тенденції розвитку апаратного забезпечення. Протягом років розробники процесорів збільшували тактову частоту, що дозволяло однопотоковим програмам повністю розкривати потенціал процесора, проте наприкінці минулого десятиліття виробники почали надавати перевагу багатоядерним архітектурам по багатьом причинам, одною з яких є те що, стрімкий ріст кількості транзисторів, який передбачався за законом Мура, підтримувати стало неможливо через атомарну природу речовини і обмеження швидкості світла [16]. Таким чином, прямою альтернативою зростанню частоти є ріст числа ядер процесора. Для того, щоб повністю реалізувати можливості багатоядерної архітектури, програму необхідно писати відповідним чином. Проте деякі програми не можуть бути паралельними повністю, про що стверджує закон Амдала. Це правило говорить, що будь-яка програма, що містить частину коду, яку неможливо розпаралелити, обмежить загальне прискорення від паралелізму. Це обмеження описується рівнянням:

$$S = \frac{1}{p + \frac{1 - p}{n}},$$

де S – прискорення програми, порівняно з непаралельною реалізацією, p – частина, яку не неможливо розпаралелити, n – кількість процесорів [17].

Цей закон є актуальним і для статичних серверів, які оптимізовані для роботи на багатоядерних системах. Незалежно від того, який підхід буде обрано, у структурі сервера буде присутня частина, в якій буде відбуватися встановлення нового з'єднання. Саме цю частину неможливо розпаралелити.

2.1 Огляд моделей конкурентності і паралелізму

2.1.1 Мультипотоковість

Модель, в основі якої лежить ідея одночасної роботи декількох потоків, які виконують однакові задачі. Наприклад, для кожного з'єднання створюється новий потік, який зчитує файл та формує відповідь. Є найочевиднішим підходом при розробці програм для багатоядерних платформ і в той же час найскладнішим в експлуатації.

Основною перевагою потоку перед процесом є спільна пам'ять всіх потоків однієї програми. Проте мультипотоковість вимагає ретельного планування роботи потоків, оскільки в даній моделі можуть виникати гонки за даними та стани гонок.

Стан гонки (race condition) – помилка при проектуванні мультипоточної програми, при якій робота додатку залежить від того, в якому порядку відпрацьовують різні частини коду в різних потоках. Проявляється у випадкові моменти і дуже складно відтворити. Одна з найнебезпечніших помилок при паралельному програмуванні. Можливі наступні наслідки:

1. Взаємні блокування потоків
2. Витік пам'яті
3. Псування даних

Серед способів вирішення стану гонок – використання локальних для потоків копій даних, синхронізація потоків за допомогою механізму блокування, що забезпечує ексклюзивний доступ потому до частини коду або даних. Найпростіший приклад ексклюзивного блокування – бінарний семафор.

Найважливішою частиною розробки мультипотокової програми є розробка моделі додатку – поставити чіткі вимоги до функціоналу програми та декомпозиція на незалежні модулі. У кожній програмі є загальний змінюваний стан і чим більше він використовується, тим більша вірогідність того, що розробники зустрінуться з вищеописаною помилкою. Тому головною вимогою для стабільної роботи програми є декомпозиція на окремі функціональні модулі та розділення їх станів. Проте складно написати програму, яка не матиме глобального стану і все ж таки необхідно якось зробити його використання безпечним. Для цього добре підходять примітива синхронізації. Наприклад, семафор – примітив для синхронізації доступу до даних без розділення типу доступу. Наприклад, якщо один потік читає вміст синхронізованої семафором структури даних, то інший буде очікувати завершення читання першого. Аналогічно із записом.

Дана модель підходить для розробки статичного сервера з кешем, оскільки потоки мають загальну пам’ять в рамках одного процеса, що дозволяє використовувати кеш всім одразу. Крім того, основна робота кеша полягає в читанні даних із пам’яті і майже відсутні будь-які зміни, тому вірогідність виникнення гонок за даними мінімальна.

Можливі два сценарія використання даної моделі при розробці сервера:

1. На початку роботи програми створювати пул потоків і розподіляти всі з’єднання між ними. Це дозволить не витрачати час на створення нових потоків під час роботи програми, що призведе до помірного

використання ресурсу процесора. Основний недолік – при створенні, наприклад, 10 потоків сервер зможе обробляти не більше 10 запитів одночасно, всі інші з'єднання мають очікувати. Кількість потоків при старті програми має бути більшою за кількість логічних ядер системи, оскільки операція читання блокує потік, і інші потоки можуть зайняти його місце. Схожий підхід застосовують оглянуті в першому розділі рішення, проте дещо модифіковані.

2. Створення потоку для обробки кожного з'єднання. Такий підхід дозволяє обробляти максимально можливу кількість з'єднань паралельно. Головний недолік – операція створення потоку не є дешевою. Крім того, операційній системі доведеться часто перемикатися між великою кількістю потоків, що зменшує ефективність моделі в цілому.

Отже, мультипотоковість є прийнятним рішенням для реалізації статичного серверу з низькорівневим кешем за рахунок розподіленою пам'яті.

2.1.2 Конкурентна модель

В конкурентній моделі задачі виконуються одна за одною, а не паралельно. Результат виконання даних задач повністю детермінований, що означає, що при однакових вхідних даних буде завжди отримано однакові вихідні, незалежно від того, чи виконувалась програма на одному потоці, чи на багатьох. На відміну від паралельного програмування, у якому кожна одиниця паралелізму виконує різні частини однієї задачу, в конкурентному кожен конкретний потік виконує різні задачі, які не перетинаються.

Конкурента модель залежно від реалізації може масштабуватися на багатоядерних системах за рахунок створення легких потоків. Легкі

потоки – це програмна реалізація потоків, яка дає змогу розпоряджатися часом виконання кожного з них програмі, що надає змогу передбачити початок та завершення роботи всіх потоків. Це вирішує основну проблему паралельної моделі, яка повністю покладається на потоки операційної системи та їх планувальник, який практично випадковим чином передає управління від одного потоку або процесу до іншого.

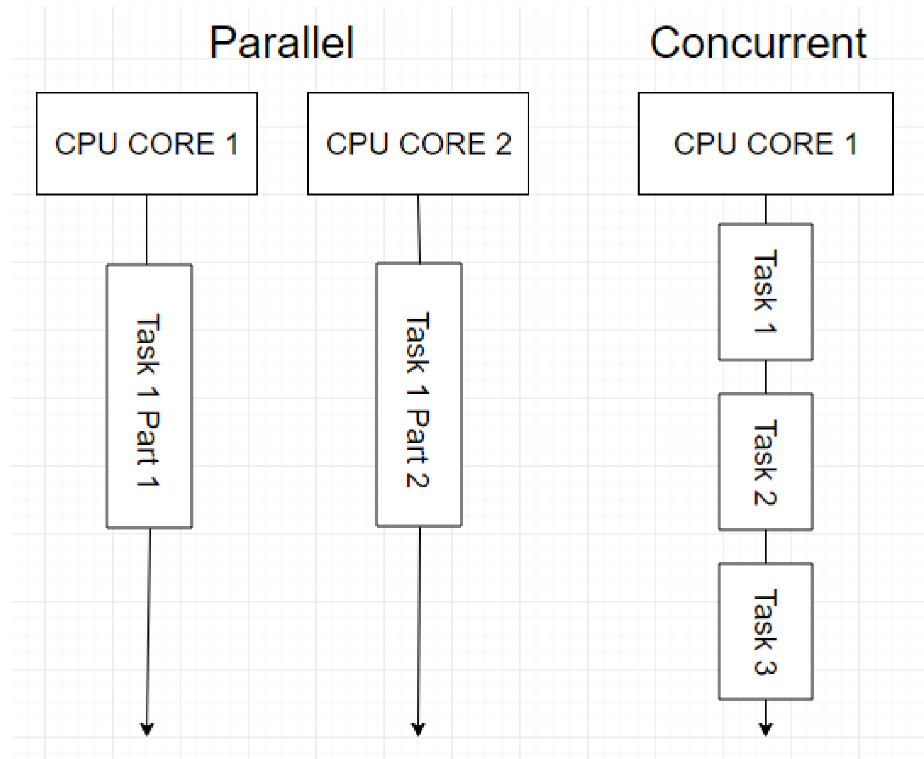

Рисунок 2.1 - Порівняння виконання задач при паралельній та конкурентній моделі

Хоча паралельна модель є більш оптимальною для обробки великого класу задач, конкурентне програмування часто обирається через свою економічність. При конкурентній обробці задач із застосуванням легких потоків блокуючі виклики не є критичними, оскільки в цей час може виконуватися інша задача.

Для реалізації сервера статичних файлів конкурента модель підходить по декількох причинам:

1. Можливість масштабування на багатоядерних платформах за рахунок легких потоків.
2. Економність – немає потреби у створенні системних потоків або процесів, а легкі потоки майже безкоштовні. Отримуємо виграш як у процесорному часі, так і в пам’яті.
3. Читання файлу з диску – це блокуюча операція, а легкі потоки дозволяють не блокувати цілий системний потік.
4. Загальна пам’ять для всіх легких потоків в рамках однієї програми, що є найоптимальнішим варіантом для реалізації низькорівневого кешування.

Отже, конкурента модель є оптимальнішою для реалізації статичного сервера, ніж паралельна модель [18].

2.1.3 Асинхронний підхід

В традиційному синхронному підході більшість операцій введення та виведення виконуються синхронно. В такому випадку, потік заблокується, до тих пір, поки синхронна операція не завершиться. Для того, щоб вирішити цю проблему було створено асинхронний підхід, який фактично є підвидом конкурентного.

Асинхронні виклики виконуються окремо від основного потоку програми. Програма підписується на повідомлення про завершення операції та продовжує обробляти інші запити. Коли виклик завершився, то всі, хто підписався, отримають повідомлення. Програма після отримання повідомлення виконує зворотні виклики або повертається до контексту, де був здійснений виклик та продовжує працювати. Оскільки асинхронних операцій може бути кілька, для того, щоб керувати асинхронними викликами, існує спеціальна черга, яка є частиною циклу подій.

Цикл подій – це менеджер асинхронних викликів, який перевіряє стан операцій, викликає необхідні обробники.

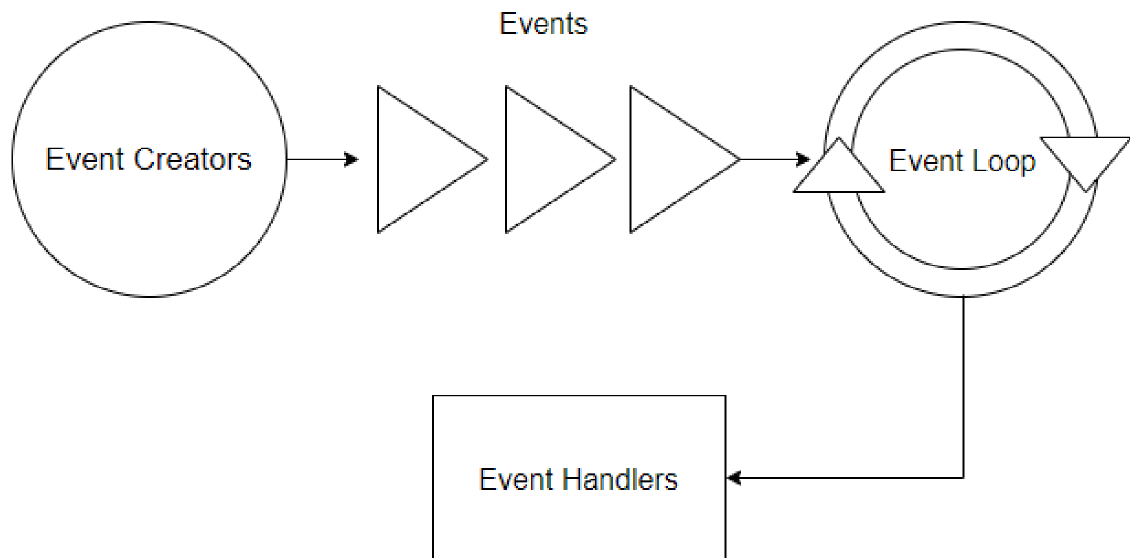

Рисунок 2.2 - Діаграма роботи циклу подій

Основним недоліком даної моделі є складність у написанні коду, оскільки для виконання обробників важливою умовою є контекст початково виклику. Для вирішення описаної проблеми існують декілька підходів:

1. Зворотний виклик – це функція, створена в контексті асинхронного виклику, а отже має доступ до змінних. Є найпростішим варіантом вирішення проблеми контексту, проте зворотній виклик може почати іншу асинхронну операцію, що також матиме свій зворотній виклик. Такий код важко підтримувати. Іншим недоліком є складна обробка помилок.
2. Фьючер – при асинхронному виклику повертається спеціальний об’єкт, що за контрактом після виконання операції, має містити результат виконання. Майже аналогічний до зворотних викликів, проте надає більше можливостей. Наприклад, можливо викликати обробник лише після завершення цілої групи

асинхронних операцій. Серед недоліків – складність у написанні, підтримці та обробці помилок.

3. Співпрограма – це аналог функції, проте має безліч точок входу, а також можливість зупиняти та продовжувати виконання. Основна перевага – асинхронний код виглядає майже як синхронний, що робить розробку та підтримку легшою. Більш того, при такому підході обробка помилок є значно простішою.

Оскільки основна робота статичного серверу – це робота з диском та мережею, то асинхронний підхід дозволить не блокувати потік при читанні, запису та запитах, а також зменшити кількість необхідних потоків в пулі. В результаті, це призводить до оптимального використання процесора та пам’яті.

2.2 Вибір мови програмування

Основні вимоги до мови програмування для розробки статичного сервера з низькорівневим кешуванням:

1. Підтримка асинхронного підходу, як найоптимальнішого при розробці даного сервера, оскільки основна робота - з диском та мережею, а асинхронний підхід дозволить не блокувати потік.
2. Хоча основний час обробки запитів – це робота з диском та мережею, мова має бути швидкою, оскільки присутні такі елементи, як формування відповіді та алгоритми стиснення.
3. Необхідно є пряма робота з пам’яттю, яка дозволить оптимальним чином реалізувати кеш.

Мови C, C++ не розглядаються, оскільки реалізацію та підтримка асинхронного коду є дуже важкою задачею і вимагає значних витрат на розробку.

Мова Java та похідні від неї мають не надають інструментів для роботи з пам’яттю.

Було вирішено розглянути 2 мови системного рівня – Rust та Go.

2.2.1 Rust

Rust – це компільована мова програмування загального призначення, яка поєднує дві парадигми програмування - функціонального та процедурного з об’єктною системою. Об'єкто-орієнтоване програмування не підтримується, проте дозволяє реалізувати більшість концепцій ООП за допомогою інших абстракцій.

Основною особливістю є управління пам'яттю на основі концепції «володіння», що доволяє відмовитись від механізму «збирання сміття» аналогічно до мови C++.

Переваги мови:

1. Швидкість – Rust можна порівняти за швидкістю C++.
2. Безпечність – при роботі з пам’яттю компілятор слідкує за відсутністю нульових посилань.
3. Паралелізм – вбудована підтримка мультипотокової та конкурентної моделі.
4. Спільнота – хоча мова нова, вона має велику спільноту. Саме тому Rust має розвинену екосистему бібліотек, що компенсує головний недолік даної мови – складність.
5. Активно розвивається – мова постійно отримує нові версії.

Проте мова має надзвичайно високий поріг входу для програміста, хоча і забезпечує відсутність великої кількості помилок на етапі компіляції.

Дана мова була спроектована за принципом «абстракція з нульовою ціною». В ній присутні високорівневі функціональні можливості без

втрати швидкодії. Крім того, всі відповідності правилам «володіння», перевірки на відсутність гонки за даними і інших можливих помилок відбуваються лише на етапі компіляції, що забезпечує достатній рівень безпеки без накладних витрат при роботі.

Підходить для реалізації сервера статичних файлів, оскільки надає можливості для майже прямого управління пам’яттю, має високу швидкодію, підтримує асинхронну модель.

2.2.2 Go

Go – це компільована мова програмування загального призначення, яка підтримує лише процедурний підхід з об’єктною системою.

Основною особливістю є підтримка на рівні мови конкурентності. Реалізовує власні легкі потоки. Реалізовує підхід CSP (communicating sequential processes) для розробки конкурентних програм, яка схожа за ідеєю на іншу конкуренту модель – акторну.

Основний принцип CSP – це розділення програми на функціональні модулі, що дозволяє зменшити розмір глобального стану. Для того, щоб передавати дані між модулями використовується засіб синхронізація – канали.

Переваги мови:

1. Простота – мова має низький поріг входу, що забезпечується простим синтаксисом.
2. Вбудована підтримка конкурентності на основі CSP.
3. Швидкість розробки – простота та вбудована підримка конкурентності дозволяє розробляти веб-сервіси швидко, вони придатні до масштабування і досить легко підтримуються.
4. Розвинена екосистема модулів, що

5. Поставляється с утилітою, яка здатна знаходити в коді можливі гонки за даними.

Недоліки:

1. Відсутність «узагальненого програмування», що призводить до дублювання коду.
2. Відсутність управління пам'яттю, що робить реалізацію кеша проблемною.

Мова Go підтримує конкурентність та асинхронність за допомогою програмно реалізованих потоків - goroutine. Хоча мова і має вказівники та посилання, управління пам’яттю повністю перебирає на себе. Працює повільніше за розглянутий Rust.

ВИСНОВКИ ДО РОЗДІЛУ 2

В другому розділі було розглянуто паралельний та конкурентний підходи до обробки одночасних запитів, проаналізовано їх переваги та недоліки. Також були розглянуті мови програмування для реалізації, а саме – Rust та Go. Після аналізу було зроблено такі висновки:

1. Для розробки сервера було обрано конкурентну модель тому, що даний підхід чудово масштабується на багатоядерних системах, є економнішим в плані використання процесора та пам'яті за мультипотоковість за рахунок обмеженої кількості потоків. Конкурентних підхід мінімізує потребу в глобальному стані, що зменшує вірогідність появи гонки станів.
2. Для роботи з диском та мережею оптимальним є використання асинхронних викликів, що дозволяє не блокувати потік при операціях введення та виведення.
3. Була обрана мова програмування Rust, оскільки має вбудовану підтримку асинхронного підходу, розвинену роботу з пам'яттю і є швидшим за конкурентів.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1. Опис задачі

Для того, щоб статичний сервер відповідав сучасним вимогам, необхідно реалізувати підтримку протоколів HTTP/1.1 та HTTP/2, а також можливість шифрування за допомогою протоколів SSL та TLS. Необхідною вимогою також є здатність витримувати значне навантаження багатьох одночасних з’єднань. Необхідно застосувати знайдені в результаті досліджень методи оптимізації роботи, основним серед яких є кешування контенту в оперативній пам’яті. Джерелами контенту можуть бути файли з локальної файлової системи або за посиланням. Кешування файлів в пам’яті програми обов’язкове за вимогою та по можливості, якщо доступна вільна пам’ять. Всі інші файли, які не потрапили в кеш, мають щоразу читатися з диску або по посиланню. Реалізований сервер має масштабуватися на багатоядерних системах. Крім того, необхідно реалізувати можливість гнучкої конфігурації.

Серед досліджених методів оптимізації слід застосувати асинхронний підхід до операцій введення та виведення, кеш в пам’яті програми, який адаптується для кількість операцій читання з диску або по посиланню, що суттєво прискорює роботу сервера.

Продуктивність сервера має бути достатньою для того, щоб конкурувати с існуючими рішеннями.

3.2 Опис моделі паралельної обробки

В результаті дослідження, описаному у попередньому розділі, був обраний конкурентний підхід для обробки одночасних запитів, оскільки дана модель масштабується на багатоядерних системах, є економною в плані використання процесора та пам'яті та надає можливість реалізувати кеш в оперативній пам'яті. Для обробки з'єднань та читання файлів доречно використовувати асинхронний підхід, при якому блокування потоку відсутнє, що дозволяє оптимально використовувати ресурси операційної системи.

Для реалізації була обрана мова програмування системного рівня Rust, яка демонструє найкращі результати при роботі з пам'яттю. Крім того, ця мова дозволяє розробнику розпоряджатися пам'яттю програми власноруч, що дозволяє зробити кеш статичних файлів максимально гнучким. Завжди відомо, що знаходиться в даний момент в кеші і не потрібно чекати, поки garbage collector підбере видалені файли, що робить використання пам'яті оптимальним [19].

Rust підтримує конкурентність на основі системних потоків, але нас більше цікавлять легкі програмні потоки. І така можливість в мові присутня за допомогою сторонніх бібліотек – futures, tokio, mio. Крім того, компілятор має механізм, котрий перевіряє відсутність гонок станів у програмі.

Rust надає як високорівневий, так і низькорівневий інтерфейси для роботи з мережею. Стороння бібліотека tokio надає можливість виконувати всі операції з мережею та диском у неблокуючому вигляді.

Для підтримки протоколів HTTP/1.1 та HTTP/2 була обрана стороння бібліотека Actix, яка також підтримує інтеграцію с протоколами шифрування SSL/TLS.

3.3 Опис структури програми

Написаний програмний продукт складається з низки модулів, що мають різне функціональне призначення:

1. Модуль config містить реалізацію парсингу та валідації конфігурації. Основна структура Config має методи parse_from_string та parse_from_file для зручного парсингу файлу конфігурації з різних джерел. На сервері передбачено можливість задавати як джерело директорію, то цей модуль також включає функціонал для того, щоб отримати з директорії рекурсивно всі файли з повними шляхами до них, оскільки при роботі на сервер необхідно завантажити саме окремі файли. Для обробки та валідації використовується стороння бібліотека serde, яка дозволяє перетворювати текстові yaml файли відразу в екземпляри структур.
2. Модуль ssl інкапсулює в собі логіку конфігурування обробника для з'єднань по протоколам SSL та TLS. Інтерфейс даного модуля складається лише з однієї функції – create_ssl_builder. Підтримка протоколів безпечної передачі працює на основі системної бібліотеки OpenSSL, для інтегрування в сервер було використано бібліотеку rust-openssl. Для ініціалізації обробника, необхідно мати сгенеровані приватний та публічний ключ в форматі pem. Конфігурацією для протоколів SSL та TLS обрано intermediate.
3. Модуль compression надає інтерфейс для стиснення файлів. Оскільки всі сучасні клієнти підтримують стиснення базовими алгоритмами, наприклад, gzip, то є сенс кешувати стиснені дані. Таким чином буде отримано економію оперативної пам'яті та процесорного часу, оскільки відпадає необхідність застосовувати стиснення на кожен запит. Даний модуль містить структуру CompressionManager, якій для ініціалізації необхідно вказати

алгоритм для стиснення. Після ініціалізації екземпляр структури надає простий інтерфейс для стиснення – метод compress, який приймає масив байтів та повертає стиснений масив.

4. Модуль FileManager інкапсулює логіку роботи з файлами. Структура, яка описує файл на сервері виглядає так:

```
struct File {
        bytes: Option<Bytes>,
        source: String,
        in_memory: bool,
        file_type: FileType,
}
```

Поле file_type описує джерело файлу та може приймати значення FileType.FromFileSystem та FileType.FromURL – з локальної файлової системи та по посиланню відповідно. Поле in_memory описує, чи повинен файл знаходитись в кеші. Поле source – це шлях до файлу на диску або посилання в залежності від значення file_type. Поле bytes відповідає за зберігання контенту файлу. Це поле має тип Option<Bytes>, що можливе значення None, що говорить про відсутність файлу в кеші.

Інша важлива структура у цьому модулі – FileManager, яка керує файлами та забезпечує кешування необхідних файлів в пам’яті. Надає наступний інтерфейс для використання:

- Метод get, який по шляху при наявності файлу оптимальним чином повертає вміст файлу.
- Метод load_files для того, щоб завантажити необхідні файли в кеш.
- Метод invalidate для того, щоб привести кеш в актуальний стан.

Останній модуль також включає в себе функції для асинхронного читання файлів з диску та по посиланню. Для читання з диску:

```
pub fn load_file_from_url_async(url: String)
        -> Box<Future<Item=Bytes, Error=Error>> {
    let mut client : Client  = Client::default();

    Box::new( x: client.get(&url)
        .send()
        .map_err( f: |_| ())
        .and_then( f: |mut response : ClientResponse<Box<Stream<Item=..., Error=...>>> | {
            response.body().map_err( f: |_| ()).and_then( f: |bytes : Bytes | {
                result( r: Ok(bytes))
            })
        }).or_else( f: move |err| {
            log!("Error during downloading file {:?}, {:?}", &url, err);
            result( r: Err(
                Error {
                    description: err.to_string()
                }
            ))
        })
    )
}
```

Рисунок 3.1 Приклад асинхронної функції для завантаження

Дана функція приймає посилання на файл в мережі та повертає фьючер, який в майбутньому буде можливо розпакувати в масив байтів. Фьючер має бути обернутий в структуру Box, оскільки компілятор не здатний вивести розмір типажа Future. Типаж – це аналог інтерфейсів в інших об’єкто-орієнтованих мовах. Оскільки в обраній мові об'єкто-орієнтоване програмування не підтримується, проте можливо реалізувати більшість концепцій ООП за допомогою інших абстракцій. Типаж – саме одна з таких абстракцій.

Їх об’єднує головний модуль – main. Файл main.rs містить точку входу програми. В даному модулі ініціалізується логування, отримуємо конфігурацію та на її основі створюємо сервер та ініціалізуємо кеш.

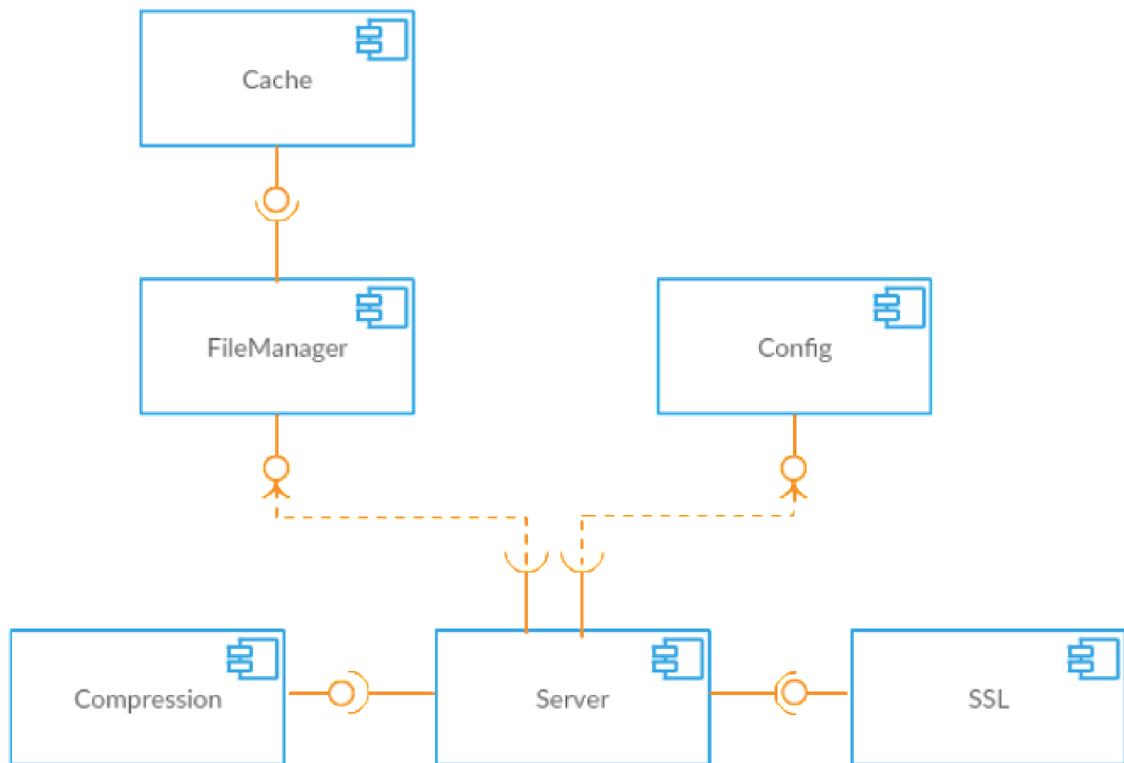

Рисунок 3.1 - Діаграма компонентів

Модулі сервера спроектовані так, щоб мати найменшу зв’язність, а отже їх можна використовувати в інших проектах.

3.4 Опис реалізації кешування

Основною частиною програмної реалізації є кешування найважливіших файлів в оперативній пам’яті і по можливості всіх інших. При старті сервера всі файли, у яких в конфігурації задано директиву in_memory: true завантажуються в пам’ять. Завантаження як з диску, так і по посиланню виконується за допомогою асинхронних викликів, що робить час ініціалізації кеша швидким і обмеженим лише можливостями диску та мережі. Після завантаження всіх необхідних файлів сервер переходить в робочий стан. Крім того, кеш інвалідує збережені дані,

наприклад, при зміні файлу по посиланню завантажує актуальну версію, або перевіряє чи не змінився файл на диску.

Кеш реалізований як частина модуля file_manager. Даний модуль включає структуру FileManager, яка надає інтерфейс для ініціалізації, читання, заміни та видалення файлів. Структура повністю інкапсулює всю логіку роботи кешування, що робить її безпечнішою та передбачуваною у використанні.

Кеш є спільним для всіх потоків у додатку. При одночасному читанні одного файлу проблем не виникне. Проте іноді кеш необхідно інвалідувати для того, щоб підтримувати його в актуальному стані, що призводить до перезапису деяких файлів. Тому структуру FileManager необхідно синхронізувати.

Синхронізація кеша досягається за допомогою використання спеціалізованої структури даних. Конкурентна хеш таблиця регулює паралельній доступ до даних. При читанні даних таблиця не блокується повністю, а лише розподіляє блокування для обмеження запису. Дана структура складається з контейнерів пар ключ-значення. Під час читання з такого контейнера блокується будь-яка спроба змінити його. При мутації такого контейнера, він блокується як на читання, так і на запис. А оскільки запис файлів відбувається дуже рідко, то дана структура є найоптимальнішим варіантом серед можливих. Проте, у обраної структури є недолік – реалокація пам’яті. Якщо таблиця майже заповнена, то вона повністю блокується до моменту завершення реалокації. При розробці така ситуація була врахована і для таблиці при створенні виділяється достатня кількість контейнер для всіх обов’язкових і потенціальних файлів.

Ще одна критична проблема, яка може статися при роботі сервера з кешуванням – це підкачка сторінок або swapping, коли операційна система переміщує частину даних з пам’яті на диск, звільняючи пам’ять

для завантаження інших даних. Це може значно сповільнити роботу сервера. Тому для запобіганню даної проблеми розмір кеша дозволяється конфігурувати за допомогою директиви max_memory_size, яка приймає значення в мегабайтах. Якщо розмір стартових файлів перевищує ліміт, то сервер повідомить про це.

Одна з найскладніших частин реалізації – це динамічне кешування, коли сервер на основі статистики, яка збирається по кожному файлу, при наявності вільної пам'яті кеша, завантажує найбільш необхідні в пам'ять. З періодом, який конфігурується, актор збирає статистику по кількості запитів всіх файлів і за алгоритмом оновлює кеш.

Для визначення, які файли необхідно кешувати, вводиться важливими є дві величини:

1. Об'єм файлу на диску
2. Сумарний об'єм, який дорівнює добутку кількості запитів файлу та його об'єм на диску.

Таким чином, кешування файлів з найбільшим сумарним об'ємом є найбільш оптимальним. Отримано класичну «задачу про рюкзак», оскільки об'єм – це вага предмета, сумарний об'єм – ціна, місткість рюкзака – доступний об'єм оперативної пам'яті, яка не має поліноміального рішення [20]. Було обрано алгоритм жадібний алгоритм, як найпростіший в реалізації, але модуль спроектований таким чином, щоб в майбутньому було просто змінювати один алгоритм рішення даної задачі на інший.

Розроблений статичний сервер має змогу кешувати файли не тільки з файлової системи, а і за посиланням. Таким чином, при ініціалізації виконується запит на завантаження файлу, який потім складається в кеш. Якщо недостатньо вільної пам'яті, то вміст буде збережено на диску.

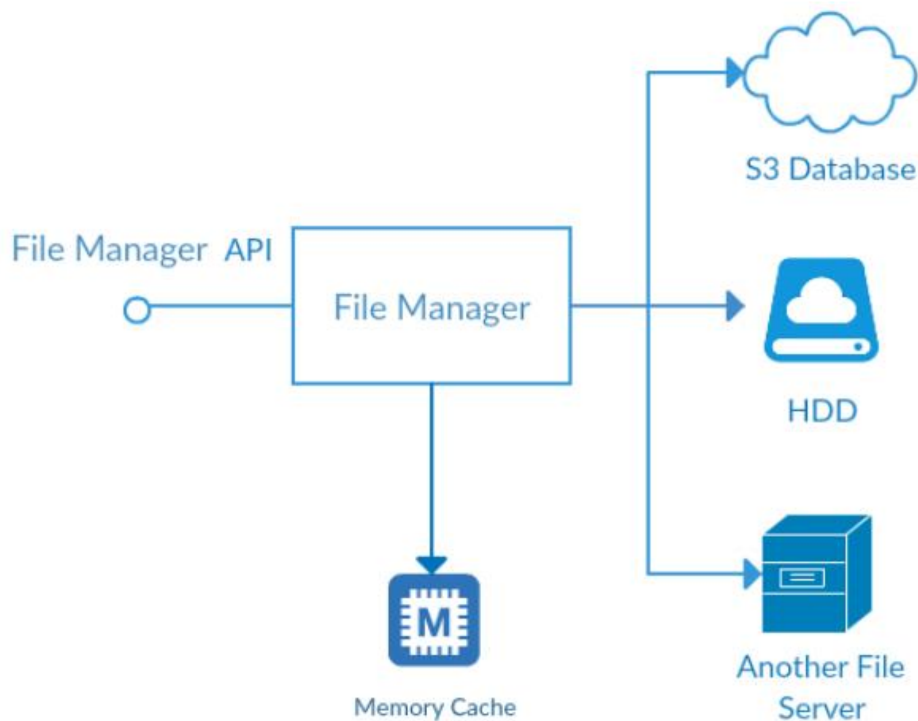

Рисунок 3.2 - Взаємодія File Manager модуля з сховищами

3.5 Опис конфігурації

Гнучка конфігурація є важливою вимогою до сервера статичних файлів. Серед доступних форматів конфігурації було обрано формат серіалізації yaml, оскільки має декілька переваг перед іншими:

1. Даний формат візуально легшим за JSON та XML, він просто читається і є зрозумілішим, тому користувач буде почувати себе комфортніше.
2. Специфікація даної серіалізації передбачає посилання на інші елементи файлу за допомогою «якорів».
3. Rust має бібліотеку, яка надає можливість перетворювати yaml файли в строго типізовані структури даних.

Хоча парсинг даного формату є повільнішим за JSON, проте файл конфігурації необхідний лише на початку роботи.

Приклад конфігурації:

```yaml
http:
  host: 0.0.0.0
  port: 8080

ssl:
  enabled: true
  host: 0.0.0.0
  port: 4433
  cert_path: ssl/cert.pem
  key_path: ssl/key.pem

runtime:
  workers: 4
  max_memory_size: 1000  # in Mb

content:
  files:
    /project:
      path: src
      in_memory: true
```

Рисунок 3.3 Приклад конфігураційного файлу

Конфігурація розділена на 4 секції:

1. Секція http відповідає за налаштування сервера для підтримки з'єднань за допомогою протоколу HTTP, серед яких поки доступні адреса, порт для сервера.
2. Секція ssl відповідає за налаштування сервера для підтримки з'єднань за допомогою безпечного протоколу HTTPS. В даній секції мають бути вказані шляхи до приватного та публічного ключів. Крім того, доступна опція підтримки HTTP/2.0 з'єднань. Вона знаходиться в цій секції, оскільки даний протокол за специфікацією працює лише по захищеному каналу.
3. Секція runtime включає в себе робочі налаштування. Директива workers вказує скільки потоків необхідно створити при старті сервера. Ці потоки серверу необхідні для того, щоб виконувати

незалежні задачі одночасно. Наприклад, один із цих потоків буде завантажувати файл з віддаленого сервера, як інший – формувати відповідь на запит. Директива max_memory_size – дозволяє встановити ліміт, за який розмір кеша сервера не має виходити.

4. Остання секція content відповідає за контент, який буде віддаватися сервером. Директива files дозволяє описати файли, які мають обслуговуватися сервером та по якому шляху вони будуть доступні. Серед важливих директив – in_memory, яка вказує, чи необхідно файл кешувати.

Більше детально конфігурація буде розглянута в наступному розділі.

3.6 Інструкція для користувача

Маючи вже готовий продукт слід пояснити основні моменти та як конфігурувати проект для роботи найбільш оптимальним чином.

3.6.1 Вимоги до програмного оточення

Оскільки даний програмний продукт знаходиться на стадії розробки, коректність роботи сервера не досліджена на операційній системі Windows. Проте Rust і всі бібліотеки, які використовувалися, є кросплатформеними, що, теоретично, дозволяє запускати сервер на будь-якій операційній системі.

Для запуску сервера необхідна інстальована в системі бібліотека для роботи протоколів шифрування SSL та TLS. Для дистрибутиву Ubuntu ці пакети доступні за назвами libssl1.0.0 та libssl-dev.

Завдяки тому, що Rust при компіляції використовує статичне зв'язування, всі інші необхідні бібліотеки знаходяться в бінарному файлі, але призводить до збільшення об'єму файлу.

3.6.2 Компіляція в бінарний файл

Для того, щоб скомпілювати актуальну версію програми, був написаний Dockerfile, який дозволяє створити бінарний файл програми без інсталювання компілятора, пакетного менеджера та системних залежностей:

```
FROM rust:1.34.2-slim-stretch
RUN apt-get update
RUN apt-get install pkg-config openssl libssl-dev -y
WORKDIR /usr/server
ADD src src
ADD Cargo.toml Cargo.toml
ADD Cargo.lock Cargo.lock
RUN cargo install --path .
CMD "echo 'compiled file is /usr/server/server'"
```

Для того, щоб виконати компіляцію з допомогою даного сценарію, необхідно мати встановлену в системі утиліту Docker.

Необхідно виконати команду в корені проекту:

```
docker build –t server-image .
docker run –rm --name server server-image
docker cp server:/usr/server/server: path_to_host.
```

В результаті отримаємо готовий до використання бінарний файл з назвою «server».

3.6.3 Конфігурація проекту

Конфігураційний файл складається з 4 секцій, структура яких вже була розглянута в даному розділі. Необхідно розглянути як використовувати сервер у реальних ситуаціях:

1. По-перше, необхідно задати директиви address та port секції http. Порт обов’язково повинен бути вільним. Для того, щоб перевірити, що порт вільний в Linux можна скористатися командою «lsof –i :<порт>» Якщо користувач не зацікавлений у використанні незахищеного з’єднання, то дану секцію можна не вказувати у конфігурації.
2. Для конфігурування https призначена друга секція. Також є опція підтримки HTTP/2.0 – http2_enabled. В даній секції порт не повинен співпадати з вказаним портом секції http. Обов’язково необхідно вказати key_path та cert_path, які вказають на приватний та публічний ключі відповідно.
3. Частина runtime має всього дві директиви – workers та max_memory_size. Перша директива відповідає за кількість потоків у пулі, який ініціалізується при старті програми. Оптимальне значення залежить від від кількості логічних ядер центрального процесора. Занадто великі значення не тільки не прискорять, а можливо сповільнять програму. Поле max_memory_size відповідає за об’єм кеша в оперативній пам’яті. Це значення не має перевищувати реальний об’єм оперативної пам’яті системи. При визначенні значення цього параметру, потрібно враховувати, що сервер може витрачати оперативну пам’ять навіть для операцій без

кешу. Наприклад, читання файлу з диску потребує вільної оперативної пам'яті. Якщо вона відсутня, то можлива ситуація, що операційна система почне переносити сегменти пам'яті на диск, що дуже сповільнить роботу сервера. Крім того, в системі працюють інші додатки, які теж використовують оперативну пам'ять. Наприклад, при об'єму оперативної пам'яті в 4 гігабайти, на кеш оптимально буде виділити 2 гігабайти. Інших 2 гігабайти буде достатньо для роботи системи і інших програм.

4. Остання секція призначена для конфігурування контента. Директива root описує корінь пошуку файлів, якщо шлях до нього є відносним. Наприклад, якщо root має значення /usr/static/, а шлях до статичного файлу images/img.png, то сервер шукатиме файл за шляхом /usr/static/images/img.png. Якщо директиву не вказати, то за її значення приймається шлях запуску програми.
5. Піддиректива files є ключ-значення структурою, де ключ – це шлях, по якому буде доступний файл. Наприклад, якщо ключ має значення /static, а шлях - ./images/img.png, то файл буде доступний за шляхом /static/images/img.png. Значення – це саме опис файлу. Він може включати обов'язково директиву url або path. У випадку першої, сервер буде завантажувати файл за посиланням, другої – з локальної файлової системи. Директива in_memory означає наявність файлу в кеші.

При налаштуванні слід враховувати, що відбувається початкова валідація розміру кешу. Якщо кеш не здатний вмістити всі обов'язкові файли, то статичний сервер повідомить про помилку і не завершить роботу.

Якщо у описі файлу вказане поле url, то сервер буде виступати в ролі проксі. Якщо такий файл не є кешованим, то сервер збереже його на диск для того, щоб зменшити кількість запитів на сторонні ресурси.

Директива path допускає як значення шлях до директорії, в такому випадку будуть доступні всі файли, розміщені в ній.

3.6.4 Запуск серверу

Для запуска сервера достатньо запустити бінарний файл і вказати аргументом шлях до конфігурації:

./server ~/configs/my_config.yml

Якщо параметр не передавати, то сервер спробує знайти його в директорії запуску.

```
→ project git:(master) ✗ sudo docker run --rm -it --net=host pro
[2019-05-31T23:57:09Z INFO  static_server] Running with config Config
 { http: HttpConfig { host: "0.0.0.0", port: "80" }, ssl: SslConfig {
 enabled: true, host: "0.0.0.0", port: "443", cert_path: "ssl/cert.pe
m", key_path: "ssl/key.pem" }, runtime: RuntimeConfig { workers: 4 },
 content: ContentConfig { root_path: ".", files: {"/med.txt": FileCon
fig { in_memory: true, path: Some("/data/med.txt"), url: None }, "/bi
g.txt": FileConfig { in_memory: true, path: Some("/data/big.txt"), ur
l: None }, "/small.txt": FileConfig { in_memory: true, path: Some("/d
ata/small.txt"), url: None }} } }
Loading from disk /data/med.txt
Loading from disk /data/big.txt
Loading from disk /data/small.txt
[2019-05-31T23:57:09Z INFO  actix_server::builder] Starting 4 workers
[2019-05-31T23:57:09Z INFO  actix_server::builder] Starting server on
 0.0.0.0:80
[2019-05-31T23:57:09Z INFO  actix_server::builder] Starting server on
 0.0.0.0:443
[2019-05-31T23:57:14Z INFO  static_server] Got request /small.txt, 20
0, wrote 27269 bytes
[2019-05-31T23:57:26Z INFO  static_server] Got request /bigtxt, 404
[2019-05-31T23:57:26Z INFO  static_server] Got request /favicon.ico,
404
[2019-05-31T23:57:29Z INFO  static_server] Got request /big.txt, 200,
 wrote 983302 bytes
[2019-05-31T23:57:29Z INFO  static_server] Got request /favicon.ico,
404
```

Рисунок 3.3 - Приклад роботи сервера

ВИСНОВКИ ДО РОЗДІЛУ 3

У третьому розділі дипломної роботи виконано опис основних складових розробленого програмного забезпечення. Також було описано абстракції мови програмування Rust для реалізації концепцій ООП. Було наведено приклад реалізації асинхронної функції для роботи з мережею. У розділі наведено інструкції до використання та компіляції серверу та детально описано конфігурацію сервера. У результаті виконання даного розділу були зроблені наступні висновки:

1. Обрана мова програмування є оптимальною для розробки програм, які вимагають прямою роботи з пам’яттю. Вдалося реалізувати розподілений для потоків кеш, який є повністю безпечним при паралельних операціях читання та запису. Вдалося уникнути глобального блокування сховища файлів при зміні вмісту за допомогою використання синхронізованих структур даних та спеціальних примітивів. За рахунок того, що мова надає низькорівневий інтерфейс для роботи з операційною системи, і в той же час існують високорівневі обгортки, вдалося знайти баланс між рівнем абстракції та швидкістю роботи.

2. Асинхронний підхід, за допомогою якого реалізоване читання з диску та запити на інші ресурси, є достатньо складним для розуміння та написання програм, проте сторонні бібліотеки значно спрощують розробку.

3. Під час розробки програми було розроблено алгоритм перерозподілення файлів кеша. Даний алгоритм не є оптимальним, проте вирішує поставлені проблеми. Для роботи алгоритму було реалізовано підрахунок кількості запитів кожного конкретного файлу та асинхронну задачу, яка через деякий період оновлює вміст кешу.

4 НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ

При розробці сервера була застосована оптимізація – кешування файлів в пам’яті програми. Одна з вимог при розробці сервера – це конкурування з існуючими рішенням по продуктивності.

Тестування проводилось на двох розділених віртуальних машинах. Затримка мережі в середньому становить 5 мс. Такий підхід дозволяє змоделювати реальну ситуацію. Віртуальні машини працюють на базі Google Cloud Platform та мають 2 ядра з частотою 2 ГГц та 4 гігабайти оперативної пам’яті.

Тестування проводилося за допомогою утиліти Gatling. Дане рішення має гнучку конфігурацію, підтримку HTTPS та HTTP/2.0. Крім того, існує можливість створювати цілі сценарії запитів. Крім того, конфігурація описується за допомогою мови програмування Scala, що дозволяє створювати складні сценарії.

Тестування проводиться для файлів розміром 30 кілобайт, 300 кілобайт і 1000 кілобайт. Для кожного випадку проводиться 3 тести та обирається найкращий. Така система обрана тому, що всі похибки при вимірюванні продуктивності програми є додатними, а отже найкращий результат є найточнішим, при умові, що вхідні данні не змінюються [21].

4.1 Тестування без шифрування

Оскільки реалізації протоколів SSL та TLS, які використовуються у розробленому рішенні та Nginx різні і, можливо, одна з них швидша за іншу, то доречно провести тестування без використання шифрування. Такий підхід дозволить максимально точно дослідити ефективність кешування в оперативній пам’яті.

При тестуванні наступні враховуються наступні значення:

1. Середній час – повний час, від початку до кінця запиту.
2. 50 перцентиль – середній час серед тих запитів, які потрапляють у 50% найшвидших. 75 перцентиль та 95 перцентиль аналогічно.
3. Успішні обробки – описує, скільки запитів завершились успішно.

Таблиця 4.1 – Тестування на файлі розміром 30 кілобайт

Сервер	**Середній час, мс**	**50 перцентиль, мс**	**75 перцентиль, мс**	**95 перцентиль, мс**	**Успішні обробки, %**
Сервер з кешем	29	21	23	49	100
Nginx	33	23	25	65	100

По результатах помітно, що в середньому реалізований статичний сервер швидше приблизно на 10%. Присутній розрив в 25% на 95 перцентилі.

Таблиця 4.2 – Тестування на файлі розміром 300 кілобайт

Сервер	**Середній час, мс**	**50 перцентиль, мс**	**75 перцентиль, мс**	**95 перцентиль, мс**	**Успішні обробки, %**
Сервер з кешем	58	42	46	144	100
Nginx	61	46	48	166	100

Результати є аналогічними до попередніх.

Таблиця 4.3 – Тестування на файлі розміром 1000 кілобайт

Сервер	Середній час, мс	50 перцентиль, мс	75 перцентиль, мс	95 перцентиль, мс	Успішні обробки, %
Сервер з кешем	89	70	71	281	100
Nginx	95	70	74	308	100

Результати є аналогічними до попередніх.

4.2 Тестування з шифруванням

Тепер необхідно протестувати разом із шифруванням:

Таблиця 4.4 – Тестування на файлі розміром 30 кілобайт

Сервер	Середній час, мс	50 перцентиль, мс	75 перцентиль, мс	95 перцентиль, мс	Успішні обробки, %
Сервер з кешем	79	49	55	279	100
Nginx	80	42	57	321	100

Середній час запиту повністю співпадає. Присутній невеликий виграш на 95 перцентилі.

Таблиця 4.5 - Тестування на файлі розміром 300 кілобайт

Сервер	Середній час, мс	50 перцентиль, мс	75 перцентиль, мс	95 перцентиль, мс	Успішні обробки, %
Сервер з кешем	168	78	257	501	100
Nginx	126	64	86	487	100

У даному вимірюванні кращі результати покає Nginx. На 50 перцентилі розрив невеликий, проте на 75 – в декілька разів.

Таблиця 4.6 - Тестування на файлі розміром 1000 кілобайт

Сервер	Середній час, мс	50 перцентиль, мс	75 перцентиль, мс	95 перцентиль, мс	Успішні обробки, %
Сервер з кешем	486	412	821	1086	100
Nginx	395	102	800	1078	100

Результати за середнім часом майже аналогічні до попередніх.

ВИСНОВКИ ДО РОЗДІЛУ 4

Було проведено тестування та порівняння швидкодії розробленого програмного рішення та Nginx. Nginx був обраний, оскільки вважається найшвидшим сервером для обробки статичних файлів [22]. В розділі було описано умови тестування та використане програмне забезпечення. Було проведено тестування для файлів різного розміру с шифруванням та без шифрування. Для кожного випадку було проведено по 3 тести та обрано найкращі результати, оскільки всі похибки при вимірюванні продуктивності програми є додатними, а отже найкращий результат є найточнішим, при умові, що вхідні данні не змінюються.

Без використання шифрування розроблений сервер показує приріст в 10-20% за будь-якого розміру файлу. Важливе уточнення - операційна система сервера мала достатню кількість вільної пам’яті для використання кешування сторінок. Проте, з використанням протоколів SSL/TLS розроблений сервер показує значно гірші результати. Різниця в роботі реалізації шифрування повністю нівелює приріст від вбудованого кешування. Однією з причин може бути неоптимальна використана реалізація протоколів SSL/TLS для Rust. Інше припущення – власна реалізація вказаних протоколів Nginx підготовлює файли для шифрування, проте документація не розкриває даний деталей. Схожий механізм має Apache HTTP Server [23].

ВИСНОВКИ

В результаті виконання бакалаврської роботи був проведений детальний аналіз існуючих рішень, використаних оптимізацій для обслуговування статичних файлів, та на основі досліджень було розроблено сервер статичних файлів з низькорівневим кешуванням.

Аналіз існуючих рішень доводить, що при обслуговуванні статичного контенту найбільшу роль відіграє модель паралельної обробки запитів сервера. Найоптимальнішою як в теорії, так і на практиці виявилася асинхронна модель, оскільки саме вона найбільше розрахована для програм з інтенсивним введенням та виведенням. Головною темою дослідження була реалізація кешування контенту в пам'яті програми. Існуючі рішення покладаються на кешування сторінок операційною системою, а в даній роботі було проведено дослідження впровадження кешування на рівні програми.

При розробці сервера статичних файлів був реалізований кеш в пам’яті програми, який дозволяє зменшити кількість звернень до диску та мережі, що значно прискорює обробку запитів. Оскільки оперативна пам’ять обмежена, була реалізована гнучка конфігурація, яка дозволяє забезпечити наявність найбільш необхідних файлів в кеші сервера.

Було проведене порівняльне навантажувальне тестування розробленого програмного продукту та Nginx, оскільки Nginx є найшвидшим серед аналогів сервером. Без використання протоколів шифрування було отримано зменшення середнього часу обробки запиту на 15%, що доводить ефективність кешування в пам’яті програми. З використанням шифрування – результати значно гірші. Причиною може бути власна реалізація протоколів у Nginx, яка оптимізована для роботи саме з цим сервером.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Saniyeh Deylami, Yaghoub Farjami. Optimizing Web programs Response in Cloud Using Pre-processing, Case study Nginx, Varnish - Signal and Data Processing, 2018

2. Документація до модуля Nginx ngx_http_core_module [Електронний ресурс]. - Режим доступу: https://www.nginx.org/ru/docs/http/ngx_http_core_module.html

3. Linux Programmer's Manual. System call sendfile description [Електронний ресурс]. - Режим доступу: http://man7.org/linux/man-pages/man2/sendfile.2.html

4. Martin Fjordvald (Author), Clement Nedelc. Nginx HTTP Server: Harness the power of Nginx to make the most of your infrastructure and serve pages faster than ever before – Packt, 2018. – 251 с.

5. MDN web docs. Keep-alive [Електронний ресурс]. - Режим доступу: https://developer.mozilla.org/docs/Web/HTTP/Headers/Keep-Alive

6. Liquid Web. Apache MPMs Explained [Електронний ресурс]. - Режим доступу: https://www.liquidweb.com/kb/apache-mpms-explained/

7. Apache Module mod_file_cache [Електронний ресурс] // - Режим доступу: https://httpd.apache.org/docs/2.4/mod/mod_file_cache.html

8. Configuring Apache Content Caching [Електронний ресурс]. - Режим доступу: https://www.digitalocean.com/community/tutorials/how-to-configure-apache-content-caching-on-ubuntu-14-04

9. Why use CDN? [Електронний ресурс]. - Режим доступу: https://gtmetrix.com/why-use-a-cdn.html

10. Взгляд на Content Delivery Network [Електронний ресурс]. - Режим доступу: https://habr.com/ru/company/webzilla/blog/236511/

11. RFC 2616. Caching in HTTP // - Режим доступу: https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html

12. Linux Programmer's Manual. System call read description [Електронний ресурс]. - Режим доступу: http://man7.org/linux/man-pages/man2/read.2.html

13. Linux Programmer's Manual. System call write description [Електронний ресурс]. - Режим доступу: http://man7.org/linux/man-pages/man2/write.2.html

14. Linux Programmer's Manual. System call readahead description [Електронний ресурс]. - Режим доступу: man7.org/linux/man-pages/man2/readahead.2.html

15. Benjamin Cassell, Tyler Szepesi. Jim Summers. Disk Prefetching Mechanisms for Increasing HTTP Streaming Video Server Throughput // ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 2018

16. Peter Bright. Moore’s law really is dead this time [Електронний ресурс]. - Режим доступу: https://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/

17. George Popov, Nikos Mastorakis, Valeri Mladenov. Calculation of the acceleration of parallel programs as a function of the number of threads - Technical University of Sofia, 2010

18. An Exploration Into Concurrent Programming and Concurrency Models [Електронний ресурс]. - Режим доступу: https://medium.com/one-datum-at-a-time/an-exploration-into-concurrent-programming-and-concurrency-models-b22144950a88

19. Rust garbage collector [Електронний ресурс] // - Режим доступу: https://words.steveklabnik.com/borrow-checking-escape-analysis-and-the-generational-hypothesis

20. Exact methods for the knapsack problem and its generalizations [Електронний ресурс]. - Режим доступу: https://www.sciencedirect.com/science/article/abs/pii/0377221787901652

21. Improve benchmark performance with simple optimizations [Електронний ресурс]. - Режим доступу: https://searchitoperations.techtarget.com/tip/Improve-benchmark-performance-with-simple-optimizations

22. Web server performance comparison [Електронний ресурс]. - Режим доступу: https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison

23. Apache Module mod_ssl [Електронний ресурс]. - Режим доступу: https://httpd.apache.org/docs/2.4/mod/mod_ssl.html#sslsessioncache

ДОДАТОК А

Текст програми

```
#[macro_use]
extern crate log;
extern crate env_logger;
extern crate futures;

use std::{ process, env };

use actix_rt::System;
use actix_web::{web, App, HttpRequest, HttpResponse, HttpServer,
Error};
use env_logger::Env;

use futures::{future};
use futures::future::{Future, result};

mod config;
mod ssl;
mod file_manager;

use file_manager::FileManager;
use config::{Config};
use ssl::create_ssl_builder;
use std::sync::Arc;

use futures::lazy;
use actix_web::client::Client;

fn init_logger() {
    env_logger::from_env(
        Env::default().default_filter_or("info")
    ).init();
}

fn get_config_from_args() -> Result<String, String> {
    let args: Vec<String> = env::args().collect();

    match args.len() {
        0 => Ok("config.yml".to_owned()),
        1 => Ok(args[0].clone()),
        _ => Err("Unexpected amount of arguments".to_owned())
    }
}
```

```
fn main() -> std::io::Result<()> {
    let mut sys = System::new("main");

    init_logger();

    let config_path = get_config_from_args();
    if let Err(msg) = config_path {
        error!("{}", msg);
        process::exit(1);
    }

    let config = match Config::from_file("config.yml") {
        Ok(config) => config,
        Err(msg) => {
            error!("{}", msg);
            process::exit(2)
        },
    };

    info!("Running with config {:?}", config);

    let builder = create_ssl_builder(
        &config.ssl.key_path, &config.ssl.cert_path
    );

    let file_manager = Arc::new(
        FileManager::from_content_config(
            config.content.clone()
        )
    );


    sys.block_on(file_manager.load_files()).unwrap();

    fn handler(req: &HttpRequest, file_manager: &FileManager)
            -> Box<dyn Future<Item=HttpResponse, Error=Error>> {
        Box::from(
            file_manager.get(req.path())
                .map_err(|e| Error::from(()))
                .and_then(|data| {
                    if let Some(content) = data {
                        HttpResponse::Ok().body(content)
                    } else {
                        HttpResponse::NotFound().finish()
                    }
                }).or_else(|err|
HttpResponse::NotFound().finish())
        )
    }
```

```
    HttpServer::new(move || {
        let file_manager = file_manager.clone();
        App::new()
            .default_service(
                web::route().to(
                    move |req: HttpRequest| handler(&req,
&file_manager)))
    })
    .workers(config.runtime.workers)
    .bind(config.get_http_address())?
    .bind_ssl(config.get_ssl_address(), builder)?
    .start();

    sys.run()
}

use openssl::ssl::{SslAcceptor, SslFiletype, SslMethod,
SslAcceptorBuilder};

pub fn create_ssl_builder(
    key_path: &str, cert_path: &str,
) -> SslAcceptorBuilder {
    let mut builder =
SslAcceptor::mozilla_intermediate(SslMethod::tls()).unwrap();
    builder
        .set_private_key_file(key_path, SslFiletype::PEM)
        .unwrap();
    builder.set_certificate_chain_file(cert_path).unwrap();
    builder
}
use chashmap::{CHashMap, ReadGuard};
use std::sync::{Arc, RwLock};
use crate::config::{ContentConfig};
use bytes::Bytes;
use futures::{Future, future};


mod files;
use crate::file_manager::files::{File, LocalFile, RemoteFile,
LoaderError};
use std::collections::HashMap;
use futures::future::result;

pub struct FileCache {
    files: CHashMap<String, Bytes>,
    memory_size: RwLock<usize>,
}

impl FileCache {
```

```
    pub fn new() -> Self {
        FileCache {
            memory_size: RwLock::from(0),
            files: CHashMap::new(),
        }
    }

    pub fn get(&self, route: &str) -> Option<Bytes> {
        match self.files.get(route) {
            Some(guard) => Some((*guard).clone()),
            None => None,
        }
    }

    pub fn get_current_size(&self) -> usize {
        self.memory_size.read().unwrap().clone()
    }

    pub fn insert(&self, route: String, content: Bytes) ->
Result<usize, ()> {
        let content_len = (&content).len();
        self.files.insert(route.clone(), content);
        let mut write_guard = self.memory_size.write().unwrap();
        *write_guard += content_len;
        println!(
            "Caching: route: {}, len: {}, total_size: {}",
            &route, content_len, *write_guard,
        );
        Ok(content_len)
    }
}

pub struct FileManager {
    file_cache: Arc<FileCache>,
    files: CHashMap<String, Arc<Box<File>>>,
    content_config: Arc<ContentConfig>,
}

impl FileManager {
    pub fn from_content_config(content_config: ContentConfig) ->
Self {
        FileManager {
            file_cache: Arc::new(FileCache::new()),
            content_config: Arc::new(content_config),
            files: CHashMap::new(),
        }
    }

    pub fn load_files(&self) -> Box<dyn Future<Item=Vec<()>,
Error=LoaderError>> {
```

```
        let mut loadings: Vec<Box<dyn Future<Item=(),
Error=LoaderError>>> = vec![];

        for (route, file_config) in
self.content_config.files.iter() {
            let route = (*route).clone();

            let file: Box<File> = if let Some(url) =
&file_config.url {

Box::new(RemoteFile::from_file_config(file_config))
            } else if let Some(path) = &file_config.path {
                Box::new(LocalFile::from_file_config(file_config))
            } else { panic!("invalid file config") };

            let file = Arc::new(file);

            if file.should_be_cached() {
                let file_cache = self.file_cache.clone();
                let route = route.clone();
                let task = file.load()
                    .and_then(move |content| {
                        file_cache.insert(route,
content).unwrap();
                        future::ok(())
                    }).or_else(|_err| {
                        future::ok(())
                    });

                loadings.push(Box::new(task));
            }

            self.files.insert(route, file);
        }

        Box::new(futures::collect(loadings))
    }

    fn get_file(&self, route: &str) -> Option<Arc<Box<File>>> {
        match self.files.get(route) {
            Some(guard) => Some((*guard).clone()),
            None => None,
        }
    }

    pub fn get(&self, route: &str) -> Box<dyn
Future<Item=Option<Bytes>, Error=LoaderError>> {
        match self.file_cache.get(route) {
            Some(content) => Box::new(future::ok(Some(content))),
            None => {
```

```
                let file = self.files.get(route);

                if file.is_none() {
                    return Box::new(future::ok(None));
                }

                Box::new(file.unwrap().load().map(|x| Some(x)))
            }
        }
    }
}
use std::sync::atomic::{AtomicU32, Ordering, AtomicBool};
use std::fs;

use bytes::Bytes;
use futures::Future;
use futures::future::result;

use actix_web::client::Client;

use crate::config::FileConfig;

#[derive(Debug)]
pub enum LoaderError {
    SendRequestError,
    FileLoadingError,
}

pub fn load_file_async(path: &str) -> Box<dyn Future<Item=Bytes,
Error=LoaderError>> {
    let content = fs::read(path);
    Box::new(result(match content {
        Ok(content) => Ok(Bytes::from(content)),
        Err(err) => {
            println!("Error during loading file {:?}", err);
            Err(LoaderError::FileLoadingError)
        }
    }))
}

pub fn load_file_from_url_async(url: String) -> Box<dyn
Future<Item=Bytes, Error=LoaderError>> {
    let client = Client::default();

    Box::new(client.get(&url)
        .send()
        .map_err(|_| ())
        .and_then(|mut response| {
            response.body().map_err(|_| ()).and_then(|bytes| {
                result(Ok(bytes))
```

```
            })
        }).or_else(move |err| {
            println!("Error during downloading file {:?}, {:?}",
&url, err);
            result(Err(LoaderError::SendRequestError))
        })
    )
}

fn store_file_locally(name: &str, content: Bytes) {
    fs::write(format!("./tmp/{}", name), content);
}

pub trait File: Send + Sync {
    fn from_file_config(file_config: &FileConfig) -> Self where
Self: Sized;
    fn incr_counter(&self);
    fn get_counter(&self) -> u32;
    fn drop_counter(&self);
    fn load(&self) -> Box<Future<Item=Bytes, Error=LoaderError>>;
    fn should_be_cached(&self) -> bool;
}

pub struct LocalFile {
    path: String,
    counter: AtomicU32,
    in_memory: bool,
}

impl File for LocalFile {
    fn from_file_config(file_config: &FileConfig) -> Self {
        LocalFile {
            path: file_config.path.clone().unwrap(),
            counter: AtomicU32::new(0),
            in_memory: file_config.in_memory,
        }
    }

    fn incr_counter(&self) {}

    fn get_counter(&self) -> u32 {
        0
    }

    fn drop_counter(&self) {}

    fn load(&self) -> Box<Future<Item=Bytes, Error=LoaderError>> {
        load_file_async(&self.path)
    }
```

```
    fn should_be_cached(&self) -> bool {
        self.in_memory
    }
}

unsafe impl Sync for LocalFile {}

unsafe impl Send for LocalFile {}


pub struct RemoteFile {
    url: String,
    counter: AtomicU32,
    in_memory: bool,
    is_stored_locally: bool,
}

impl File for RemoteFile {
    fn from_file_config(file_config: &FileConfig) -> Self {
        RemoteFile {
            url: file_config.url.clone().unwrap(),
            counter: AtomicU32::new(0),
            in_memory: file_config.in_memory,
            is_stored_locally: false,
        }
    }

    fn incr_counter(&self) {}

    fn get_counter(&self) -> u32 {
        0
    }

    fn drop_counter(&self) {}

    fn load(&self) -> Box<Future<Item=Bytes, Error=LoaderError>> {
        Box::new(result(Ok(Bytes::new())))
    }

    fn should_be_cached(&self) -> bool {
        self.in_memory
    }
}

unsafe impl Sync for RemoteFile {}

unsafe impl Send for RemoteFile {}


use std::{fs, str};
use std::collections::HashMap;
```

```
use serde_yaml;
use serde::Deserialize;

mod utils;
use utils::{format_address, is_dir, get_files_from_dir,
join_path};


#[derive(Debug, Deserialize)]
pub struct HttpConfig {
    pub host: String,
    pub port: String,
}

#[derive(Debug, Deserialize)]
pub struct SslConfig {
    pub enabled: bool,
    pub host: String,
    pub port: String,
    pub cert_path: String,
    pub key_path: String,
}

#[derive(Debug, Deserialize)]
pub struct RuntimeConfig {
    pub workers: usize,
}

#[derive(Debug, Deserialize, Clone)]
pub struct FileConfig {
    pub in_memory: bool,
    pub path: Option<String>,
    pub url: Option<String>,
}

#[derive(Debug, Deserialize, Clone)]
pub struct ContentConfig {
    pub root_path: String,
    pub files: HashMap<String, FileConfig>,
}


#[derive(Debug, Deserialize)]
pub struct Config {
    pub http: HttpConfig,
    pub ssl: SslConfig,
    pub runtime: RuntimeConfig,
    pub content: ContentConfig,
}
```

```
impl Config {
    fn expand_dirs(mut self) -> Self {
        let content = self.content;
        let mut expanded_files: HashMap<String, FileConfig> =
HashMap::new();

        for (route, file_config) in content.files.into_iter() {
            if let Some(path) = &file_config.path {
                if is_dir(&path) {
                    let paths: Vec<String> =
get_files_from_dir(&path);
                    paths.iter().for_each(|path| {
                        expanded_files.insert(
                            join_path(route.clone(),
path.to_owned()),
                            FileConfig {
                                in_memory:
file_config.in_memory.clone(),
                                path: Some(path.to_owned()),
                                url: None,
                            }
                        );
                    });
                } else {
                    expanded_files.insert(route, file_config);
                }
            } else {
                expanded_files.insert(route, file_config);
            }
        }

        let expanded_config = ContentConfig {
            root_path: content.root_path,
            files: expanded_files,
        };

        self.content = expanded_config;
        self
    }

    pub fn parse(config_str: &str) -> Result<Config, String> {
        match serde_yaml::from_str(config_str) {
            Ok(parsed_config) => Ok(parsed_config),
            Err(err) => Err(err.to_string()),
        }
    }

    pub fn get_http_address(&self) -> String {
        format_address(&self.http.host, &self.http.port)
```

```
    }

    pub fn get_ssl_address(&self) -> String {
        format_address(&self.ssl.host, &self.ssl.port)
    }

    pub fn from_file(file_path: &str) -> Result<Config, String> {
        let config_content = match fs::read(file_path) {
            Ok(content) => content,
            Err(err) => {
                let err_msg = format!("Can not read config: {}",
err.to_string());
                return Err(err_msg);
            }
        };

        let config = match
Config::parse(str::from_utf8(&config_content).unwrap()) {
            Ok(config) => Ok(config),
            Err(err) => {
                let err_msg = format!("Config is invalid: {}",
err.to_string());
                Err(err_msg)
            },
        };

        config.map(|config| config.expand_dirs())
    }
}


use std::fs;
use std::path::Path;

pub fn format_address(host: &str, port: &str) -> String {
    format!("{}:{}", host, port)
}

pub fn is_dir(path: &str) -> bool {
    Path::new(path).is_dir()
}

pub fn get_files_from_dir(path: &str) -> Vec<String> {
    if !is_dir(path) {
        panic!("{} is not a directory", path);
    }

    let dir_entries = fs::read_dir(path).unwrap();
    let mut result_items = vec![];

    for entry in dir_entries {
```

```
            let entry = entry.unwrap();
            let path = entry.path().to_str().unwrap().to_owned();
            if is_dir(&path) {
                get_files_from_dir(&path).iter().for_each(|inner_path|
{
                    result_items.push(inner_path.to_owned());
                });
            } else {
                result_items.push(path);
            }
        }

        result_items
    }

    pub fn join_path(config_route: String, path: String) -> String {
        let mut join_char = "";
        if !config_route.ends_with("/") {
            join_char = "/";
        }

        format!("{}{}{}", config_route, join_char, path)
    }
```