Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

(повне найменування вищого навчального закладу)

Факультет *Інфокомунікацій*

(повне найменування інституту, факультету (відділення))

Кафедра *Інформаційно-мережної інженерії*

(повна назва кафедри (предметної, циклової комісії))

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)

(рівень вищої освіти)

Дослідження особливостей оптимізації та надійності взаємодії серверної і клієнтської частин мережних WEB-додатків

(тема)

Виконав: студент *2* курсу, групи *ІМІм-19-2*

Спеціальності *172 Телекомунікації та радіотехніка*

(код і повна назва спеціальності)

Тип програми *освітньо-наукова*

(освітньо-професійна або освітньо-наукова)

Освітня програма *Інформаційно-мережна інженерія*

(повна назва освітньої програми)

Лялічев В.Д.

(прізвище та ініціали)

Керівник *доц. Бондар Д. В.*

(посада, прізвище, ініціали)

Допускається до захисту

Зав. Кафедри ________ *Безрук В. М.*

(підпис) (прізвище, ініціали)

2021 р.

Не містить відомостей, заборонених до відкритого публікування

Керівник ______________ /*Бондар Д. В.*/

Студент _____________ /*Лялічев В.Д.*/

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

(повне найменування вищого навчального закладу)

Факультет *Інфокомунікацій*

Кафедра *Інформаційно-мережної інженерії*

Рівень вищої освіти *другий (магістерський)*

Спеціальність *172 Телекомунікації та радіотехніка*

(код і повна назва)

Тип програми *освітньо-наукова*

(освітньо-професійна або освітньо-наукова)

Освітня програма *Інформаційно-мережна інженерія*

(повна назва)

ЗАТВЕРДЖУЮ

Зав. кафедри «ІМІ»

_______________ проф. Безрук В.М.

(підпис)

«___» __________________ 2021 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові *Лялічеву Владиславу Дмитровичу*

(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження особливостей оптимізації та надійності взаємодії серверної і клієнтської частин мережних WEB-додатків*

керівник роботи *доц., к.т.н. Бондар Дмитро Вадимович*

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ВНЗ від «*12*» *березня* 2021 р. № *350 Ст*

2. Термін подання студентом роботи до екзаменаційної комісії *21.05.* 2021 р.

3. Вихідні дані до роботи *Огляд існуючих клієнт-серверних архітектур; технології зв'язку в мережних WEB-додатків; огляд оптимізації та надійності зв'язку; мережний додаток та взаємодія клієнта та сервера; результати випробовування мережного додатку*

4. Перелік питань, що потрібно опрацювати в роботі

Вступ

1 Клієнт-серверна архітектура

2 Технології зв'язку між клієнтом та сервером

3 Оптимізація та надійність зв'язку

4 Застосування технології WebSocket до клієнт-серверного додатку

Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів)

Слайди у форматі Power Point (вступ, модель Клієнт-Сервер,багато потокова обробка різних клієнтів, TCP-з'єднання, технології зв'язку між клієнтом на сервером, REST проти WebSockets, принципи оптимізації, надійність в сучасних WEB-додатків, вибір технологій для сучасного web-додатку, зовнішній вигляд web-додатку,висновки)

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз завдання та літературних джерел	12.03.21-15.03.21	
2	Виконання розділу 1	16.03.21-20.03.21	
3	Виконання розділу 2	21.03.21-01.04.21	
4	Виконання розділу 3	02.04.21-15.04.21	
5	Виконання розділу 4	16.04.21-01.05.21	
6	Висновки	02.05.21-03.05.21	
7	Оформлення пояснювальної записки	04.05.21	

Дата видачі завдання *18 березня* 20 21 р.

Студент _______________ Лялічев В. Д.
(підпис) (прізвище та ініціали)

Керівник роботи _______________ доц. Бондар Д. В.
(підпис) (посада, прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 68 с., 26 рис., 7 посилань, 4 додатків.

Мета роботи – дослідити необхідність забезпечення оптимізації та надійності зв'язку між клієнтом та сервером. Розглянути всі нюанси при створенні мережних WEB-додатків з використанням сучасних технологій, на прикладі мережного додатку. Додаток виконано у вигляді клієнт-серверної архітектури, де клієнтом виступає веб-сторінка браузера.

У сучасному світі архітектура клієнт-сервер зустрічається майже на кожній сторінці веб-браузеру. Важко згадати час, коли існували інші архітектури. Нескінченний потік інформації зберігається у базах, проходячи через фільтрацію на серверах. А потім достається за потрібними атрибутами пошуку.

У представленій кваліфікаційній роботі було створено WEB-додаток на архітектурі клієнт–сервер та розглянуто такі технології: JavaScript, NodeJS, React, WebSocket. Проведено тестування додатка. Додаток створювався за допомогою JavaScript IDE by JetBrains - WebStorm.

ЗВ'ЯЗОК, КЛІЄНТ, КОРИСТУВАЧ, ДОДАТОК, ЗАПИТ, АРХІТЕКТУРА

THE ABSTRACT

Explanatory note: 68 pp., 26 fig., 7 reference, 4 app.

The purpose of the work is to investigate the need to ensure optimization and reliability of communication between the client and the server. Consider all the nuances of creating network WEB-applications using modern technologies, for example, a network application. The application is made in the form of a client-server architecture, where the client is a web page of the browser.

In today's world, client-server architecture is found on almost every page of a web browser. It is difficult to recall a time when other architectures existed. An endless stream of information is stored in databases, passing through filtering on servers. And then get the right search attributes.

In the presented qualification work the WEB-application on client-server architecture was created and the following technologies were considered: JavaScript, NodeJS, React, WebSocket. The application has been tested. The application was created using JavaScript IDE by JetBrains - WebStorm.

COMMUNICATION, CLIENT, USER, APPLICATION, REQUEST, ARCHITECTURE

ЗМІСТ

С.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 7

ВСТУП 8

1 КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА 9

1.1 Технологія «Клієнт - сервер»....................9

1.2 Порівняння між TCP та UDP....................10

1.3 Використання моделі клієнт-сервер у реальному світі....................12

1.4 Розвиток архітектури....................13

2 ТЕХНОЛОГІЇ ЗВ'ЯЗКУ МІЖ КЛІЄНТОМ ТА СЕРВЕРОМ 16

2.1 Аналіз існуючих технологій....................16

2.1.1 HTTP протокол....................16

2.1.2 Архітектурний стиль REST....................17

2.1.3 Опитувач HTTP Polling....................17

2.1.4 Потокове передавання HTTP Streaming....................19

2.1.5 Server Sent Events (SSE)....................20

2.1.6 Пушери HTTP/2 Server Push....................20

2.1.7 Технологія WebSockets....................21

2.2 REST проти Websockets....................23

2.3 Зв’язок між серверами....................26

2.4 Що використовувати для API....................28

3 ОПТИМІЗАЦІЯ ТА НАДІЙНІСТЬ ЗВ'ЯЗКУ 31

3.1 Принципи оптимізації....................31

3.1.1 Виконання пошуку DNS....................31

3.1.2 Встановлення зв'язку....................32

3.1.3 Переговори щодо рукостискання SSL....................32

3.1.4 Надсилання запиту....................33

3.1.5 Час очікування....................34

3.1.6 Отримання відповіді....................35

3.2 Надійність в сучасних WEB додатків....................35

3.2.1 Підтвердження операції (ACK)....................36

3.2.2 Час очікування....................37

3.2.3 Порядкові номери (SEQ)....................37

3.2.4 Протокол конвеєризації..38
3.2.5 Протокол Go-Back-N (GBN)...39
3.2.6 Вибіркове повторення (SR)..40
3.2.7 Забезпечення надійності..41
4 ЗАСТОСУВАННЯ ТЕХНОЛОГІЇ WEBSOCKET ДО КЛІЄНТ-СЕРВЕРНОГО ДОДАТКУ 42
4.1 Опис додатку...42
4.2 Розробка серверної частини Web-додатку...43
4.2 Написання клієнтської частини Web-додатку......................................44
ВИСНОВКИ 49
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 50
ДОДАТОК А – КОД СЕРВЕРНОЇ ЧАСТИНИ ДОДАТКУ НА МОВІ JAVASCRIPT З ЗАСТОСУВАННЯМ NODEJS 51
ДОДАТОК Б – КОД КЛІЄНТСЬКОЇ ЧАСТИНИ НАПИСАНИЙ НА МОВІ JAVASCRIPT З ЗАСТОСУВАННЯМ REACTJS 52
ДОДАТОК В – ТЕЗИ ДОПОВІДІ ДО 25-ГО МІЖНАРОДНОГО МОЛОДІЖНОГО ФОРУМУ «РАДІОЕЛЕТРОНІКА І МОЛОДЬ У XXI СТОЛІТТІ» 57
ДОДАТОК Г – СЛАЙДИ ПРЕЗЕНТАЦІЇ 62

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

HTML	HyperText Markup Language	Мова розмітки гіпертекстових документів
HTTP	HyperText Transfer Protocol	Протокол передачі даних
JS	JavaScript	Мова програмування
JSON	JavaScript Object Notation	Формат файлу для обміну даними
TCP	Transmission Control Protocol	Протокол передачі даних
URL	Uniform Resource Locator	Адреса ресурсу
XML	Extensible Markup Language	Розширювана мова розмітки

ВСТУП

Поширення інформаційних систем постійно збільшується, але вони стають дедалі складнішими. З часом комп’ютерні ресурси почали швидко еволюціонувати в ідеї та механізми, які почали рости та розвиватися.

Саме для забезпечення максимальної вигоди та продуктивності інформаційних систем була винайдена архітектура клієнт-сервер, яка допомогла вирішити багато проблем на той час.

Термін "клієнт-сервер" відноситься до такого архітектурного програмного комплексу, в якому його функціональні частини взаємодіють з найпростішою схемою, клієнт дає попит, сервер дає відповідь.

Якщо ми розглянемо кожну частину взаємодії з цього комплексу, один з них, конкретний клієнт, робить активну роботу, тобто вони утворюють певні вимоги, а інший, сервер, відповідає.

Як розвиток інформаційних систем, ці завдання можуть відрізнятися, наприклад, розробка блоків одночасно для виконання функцій сервера та функцій клієнтів у порівнянні з іншими блоками.

Щоб стати сучасною архітектурою, взаємодія клієнт-сервер пройшло довгий шлях, починаючи з централізованої системи архітектурних додатків, які були популярні в 70-х роках минулого століття. Згодом такі системи перейшли на новий рівень, рівень персональних комп'ютерів і локальних задач на цих машинах.

Розглянемо як саме відбувається це спілкування та що саме для цього потрібно.

1 КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА

1.1 Технологія «Клієнт - сервер»

З розвитком Інтернет-технологій Інтернет стає все більш важливим у нашому житті, так що він навіть стає важливим елементом. У той же час застосування Мережі ніколи не обмежувалось лише комп’ютерами; він був відкритий для всіх видів інтелектуальних цифрових пристроїв, таких як мобільні. Крім того, архітектура Інтернету - це модель клієнт-сервер, яка представлена на рисунку 1.1. Як результат, спілкування між сервером і клієнтом - це перше, що нас повинно турбувати. Тільки на основі успішного та безперебійного спілкування веб-технологія може рухатись вперед, а веб-програми застосовуватимуться до всіх типів пристроїв, щоб вони могли допомогти людям [1].

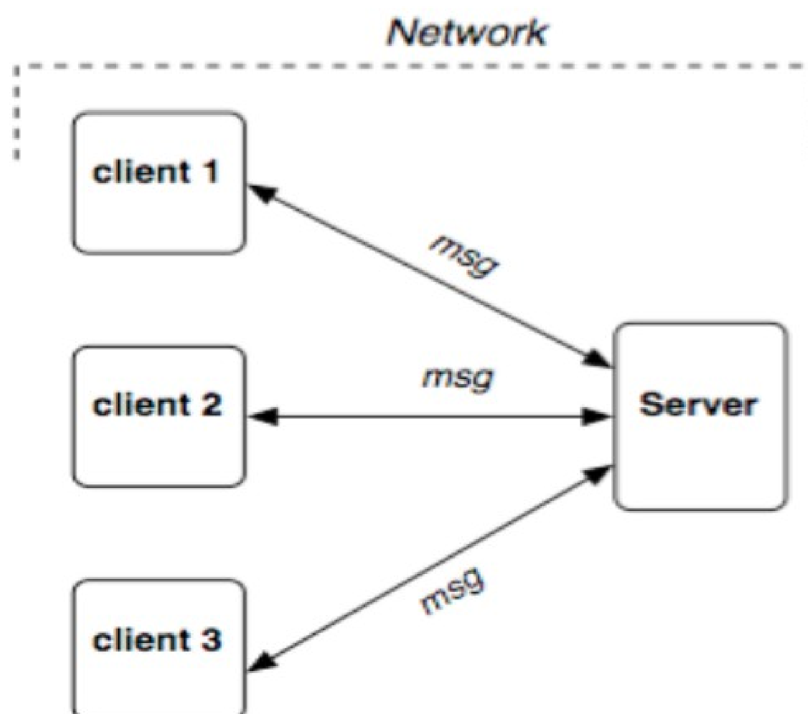

Рисунок 1.1 – Структура моделі Клієнт-Сервер

1.2 Порівняння між TCP та UDP

Для здійснення комунікації, по-перше, нам слід визначити, як передавати повідомлення між сервером і клієнтом, тобто протокол, який є одним з видів мови, який можна зрозуміти терміналам, які підключаються до Інтернету. Подібно до того, як ми вивчили одну нову мову, включаючи її синтаксис та структуру речень, так що ми можемо скласти повне речення та абзац, для термінального зв'язку ця мова називається протоколом. Перша частина, яку я хочу представити, стосується різних протоколів передачі даних, що використовуються між клієнтом та сервером для різних цілей [2]. По-друге, поки термінал знає мову, що використовується комунікацією, коли йому є що передати, він може створити пакет на основі своєї мови та протоколу. Звичайно, у конкретній реалізації комунікації, особливо, коли ми хочемо досягти її за допомогою програмування, пакету недостатньо для передачі, нам потрібна інша структура, щоб закінчити пакувати наше повідомлення, яке називається сокетом. Сокет є основним операційним блоком, який підтримує мережевий зв'язок.

Так само, як я коротко згадував вище, протокол можна визначити як правила, домовленості та стандарти, що регулюють синтаксис, семантику та синхронізацію зв'язку. Іншими словами, протоколи - це набори правил (або послідовність подій), які контролюють або забезпечують переважно надійну та впізнавану передачу інформації між кінцевими точками зв'язку. Існує так багато протоколів передачі даних, які використовуються у спілкуванні через різні цілі. Тобто, пристрої можуть бути підключені, але не підтримувати зв’язок без протоколу. Тут я познайомлю з двома найпоширенішими протоколами: «Протокол користувацьких датаграм (UDP) та Протокол управління передачею (TCP)» [2]. UDP не має попереднього зв'язку для встановлення спеціального каналу передачі, щоб це була проста модель передачі з мінімальним механізмом протоколу. Крім того, це основна та

суттєва різниця між UDP та TCP. Для TCP, перш ніж два термінали встановлять з'єднання, вони виконають процес, який називається рукостисканням, тобто процес узгодження. Після рукостискання два термінали досягнуть згоди, включаючи, але не обмежуючись цим, швидкість передачі інформації, алфавіт кодування, паритет, процедуру переривання та інші функції протоколу або обладнання. В результаті рукостискання TCP встановить специфічний зв'язок між двома терміналами для цього зв'язку, так що TCP буде коштувати більше часу, залежно від стану мережевого трафіку, ніж UDP, щоб закінчити його, однак, TCP одночасно є набагато надійнішим, ніж UDP. Тож, чи використовує наша передача протокол TCP або UDP, це компроміс, ми повинні враховувати перевантаження мережі та ступінь безпеки.

Основа для порівняння	TCP	UDP
Значення	TCP встановлює зв'язок між комп'ютерами перед передачею даних	Введіть дані безпосередньо на цільовий комп'ютер, не перевіряючи, чи готова система приймати чи ні
Розширюється на	Протокол управління передачею	Протокол дейтаграм користувача
Тип з'єднання	Підключення орієнтоване	З'єднання менше
Швидкість	Повільно	Швидкий
Надійність	Висока надійність	Ненадійний
Розмір заголовка	20 байт	8 байт
Подяка	Він займає підтвердження даних і має можливість повторної передачі, якщо користувач просить.	Він ні приймає підтвердження, ні повторно передає втрачені дані.
Налаштування протокольного з'єднання	Орієнтований на з'єднання, з'єднання має бути встановлено до передачі	Без підключення, дані надсилаються без налаштування
Інтерфейс даних до програми	На основі потоку	-оснований
Повторні передачі	Доставка усіх даних керується	Не виконується
Функції, що надаються для управління потоком даних	Управління потоком за допомогою протоколу розсувного вікна	Немає
Накладні витрати	Низький, але більший, ніж UDP	Дуже низько
Придатність кількості даних	Невелика до помірної кількості даних	Невеликі до величезних обсягів даних
Здійснено понад	Програми, де важлива надійна передача даних.	Застосування, коли важлива швидкість доставки даних.
Програми та протоколи	FTP, Telnet, SMTP, IMAP etcetera.	DNS, BOOTP, DHCP, TFTP etcetera.

Рисунок 1.2 – Порівняння між TCP та UDP

1.3 Використання моделі клієнт-сервер у реальному світі

Після вищезазначеного введення протоколу передачі, ми повинні почати цікавитися, як застосувати ці протоколи до реального використання, тобто як реалізувати мережевий зв'язок за допомогою програмування. Найпоширеніший метод - це сокет, який є мережевим інтерфейсом, кінцевою точкою міжпроцесорного потоку зв'язку через комп'ютерну мережу. Насправді існує безліч різновидів сокетів, але в наш час більшість комунікацій між комп’ютерами базуються на протоколі Інтернету, який цей документ вводить, так що більшість мережевих сокетів називаються сокетами Інтернету [1]. Сокет містить у собі адресу та дані про себе. Адреса сокета включає локальну IP-адресу та віддалену IP-адресу, які є адресами нашого комп'ютера та цільового комп'ютера в мережі та номером порту, який використовується для надсилання та отримання даних. Що стосується даних, це не тільки те, що комп’ютер хоче передати, але й деякі елементи, що використовуються для синхронізації, є частиною протоколу [1]. Однак ми не можемо керувати сокетами безпосередньо, оскільки це просто абстрактне поняття, тому нам потрібно через інтерфейс прикладного програмування (API) керувати сокетами, включаючи надсилання, отримання даних та функції. Варто зазначити, що API не є однаковим для різних платформ та мов програмування. Я працюю над одним проектом комп’ютерного спілкування, програмуючи C, а іншим проектом Android, який також включає спілкування між одним сервером та багатьма клієнтами, що використовують Java. Незважаючи на те, що всі використовувані мною Інтернет-протоколи є TCP, досягнення різняться між собою, програмування C потребує більш складних висловлювань, оскільки воно є більш базовим, ніж Java, Java API інкапсулює більше операцій, але мені доводиться реалізовувати багато функцій самостійно в програмуванні C, що також є різницею між мовою орієнтованих об'єктів та структурою мови програмування [2].

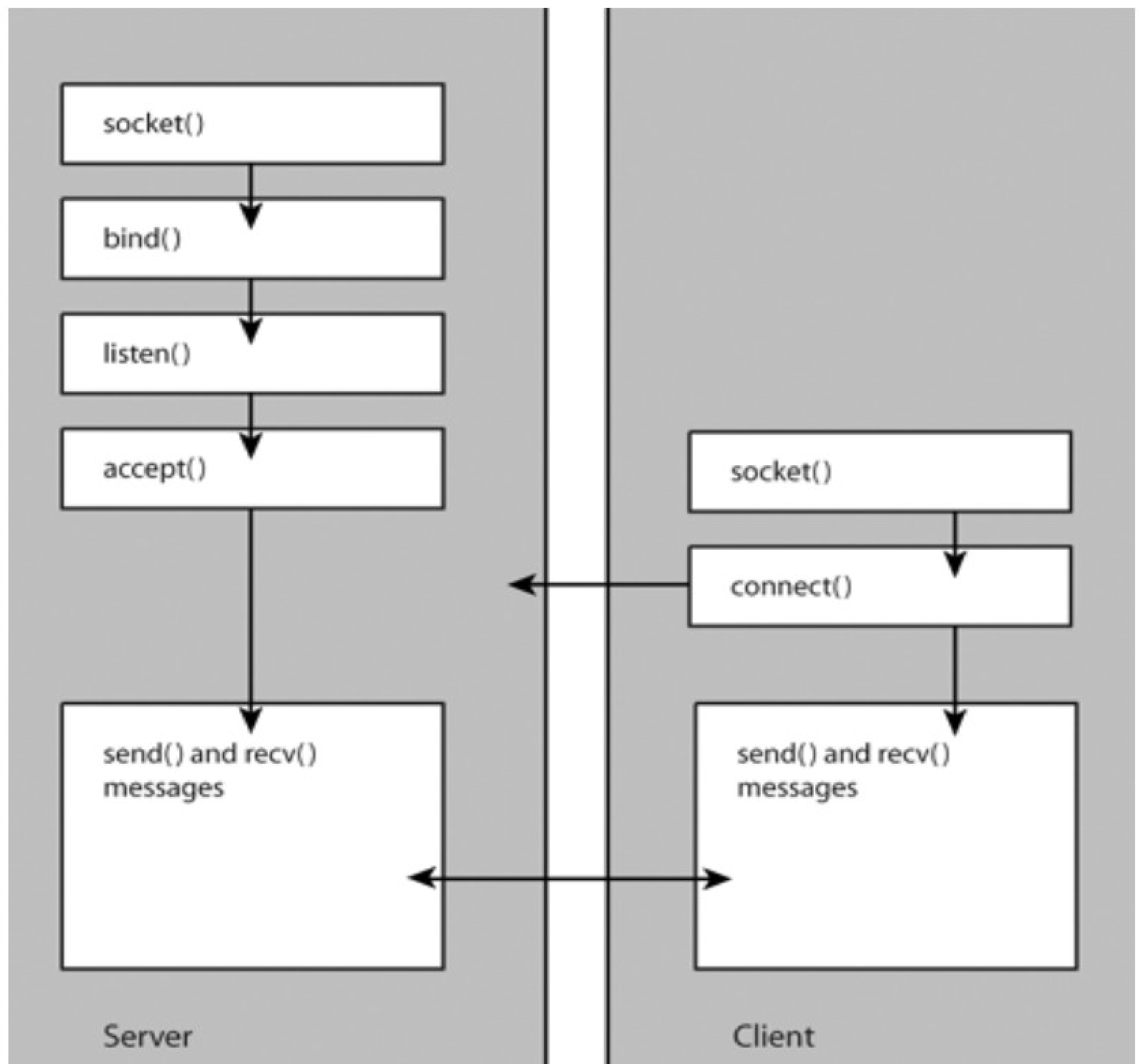

Рисунок 1.3 – TCP-з'єднання

1.4 Розвиток архітектури

Крім того, якщо я закінчив спілкування між одним сервером та кількома клієнтами, іншим фактором, який ми повинні враховувати, є ефективність. Оскільки мені потрібно мати справу з кількома клієнтами, якщо ми обробляємо запити по одному, загальний час витрат буде N разів за один час виконання. Однак, якщо ми можемо зробити це паралельно, загальний час може бути майже рівним одному часу виконання. Ще один курс, який я проходжу в цьому семестрі, - це Операційна система, яка вводить, як вирішувати різні питання чи різні частини в одному питанні за

допомогою багатопотокової роботи. В результаті, коли я розробив шаблон одного сервера, який повинен зменшити перевантаження через занадто багато клієнтів, я вирішив досягти цього за допомогою багатопоточності на стороні сервера. Точніше кажучи, коли один клієнт хоче підключитися до сервера, сервер відкриє для нього новий потік, цей потік просто належить клієнту і він обробляє кожен запит. Тим часом у моєму проекті є така функція, як MSN, яка потребує зв'язку між двома різними потоками. Для цього сервер буде зберігати свій конкретний ідентифікатор та свій потік для кожного клієнта, щоб сервер міг знайти цільовий потік за своїм ідентифікатором, а потім надіслати йому повідомлення від вихідного клієнта. Крім того, те, що я хочу сказати, - це багатопотоковість, може бути використана для вирішення різних частин в одному питанні. Найосновнішим є обчислення добутку двох матриць. Звичайно, якщо розмір матриці невеликий, нам не потрібно про це турбуватися, ми просто обробляємо її рядок за рядком і стовпець за стовпцем. Однак, якщо вони занадто великі, цей метод буде коштувати багато часу, тому ми можемо призначити один потік одному рядку та одному стовпцю, а потім ми можемо підвищити ефективність, обчислюючи кожен рядок і стовпець одночасно.

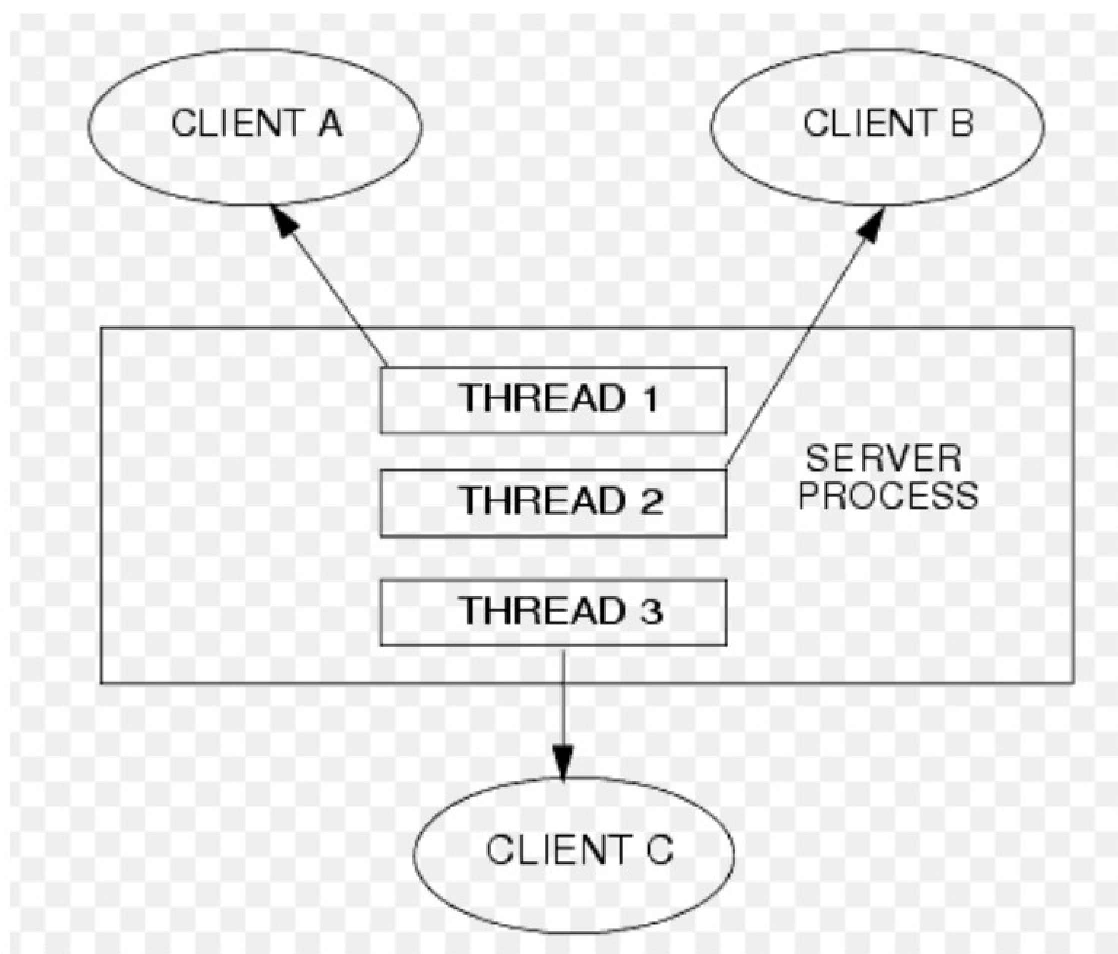

Рисунок 1.4 – Багатопотокова обробка різних клієнтів

Одним словом, якщо ми хочемо практично спроектувати та закінчити мережевий проект, перше, що потрібно, - це зрозуміти протокол Інтернету, а по-друге, ми пакуємо його в сокет і відправляємо. Нарешті, слід врахувати ефективність сервера, щоб багатопотокові імпортувати в проект.

2 ТЕХНОЛОГІЇ ЗВ'ЯЗКУ МІЖ КЛІЄНТОМ ТА СЕРВЕРОМ

2.1 Аналіз існуючих технологій

Існує так багато класифікацій для API. Але що стосується веб-спілкування, ми можемо виділити два найважливіших типи API - API веб-служб (наприклад, SOAP, JSON-RPC, XML-RPC, REST) та веб-інтерфейс API. Але що це насправді означає? Давайте поринемо у світ протоколів веб-комунікацій та обговоримо, як вибрати найкращі механізми API наприкінці.

2.1.1 HTTP протокол

HTTP є основним комунікаційним протоколом Всесвітньої павутини. HTTP функціонує як протокол відповіді на запит у обчислювальній моделі клієнт-сервер. HTTP/1.1 - найпоширеніша версія HTTP, що використовується в сучасних веб-браузерах та серверах. У порівнянні з ранніми версіями HTTP, ця версія може реалізувати критичну оптимізацію продуктивності та вдосконалення функцій, таких як постійні та конвеєрні з'єднання, послідовні передачі, нові поля заголовків у тілі запиту/відповіді тощо [3]. Серед них, наступні два заголовки дуже помітні, оскільки більшість сучасних удосконалень HTTP покладаються на ці два заголовки.

- Заголовок Keep-Alive для встановлення політик довготривалого спілкування між хостами (період очікування та максимальна кількість запитів для обробки підключення);
- Заголовок Upgrade, щоб переключити з'єднання в розширений режим протоколу, такий як HTTP/2.0 (h2, h2c) або Websockets (websocket).

2.1.2 Архітектурний стиль REST

Архітектурний стиль REST (REpresentational State Transfer) на сьогоднішній день є найбільш стандартизованим способом структурування веб-API для запитів. REST - це суто архітектурний стиль, заснований на декількох принципах. API, що дотримуються принципів REST, називаються RESTful API. API REST використовують модель запиту/відповіді, де кожне повідомлення від сервера є відповіддю на повідомлення від клієнта. Загалом, RESTful API використовує HTTP як транспортний протокол [4]. У таких випадках для пошуку потрібно використовувати запити GET. Запити PUT, POST та DELETE слід використовувати відповідно для мутації, створення та видалення (уникайте використання запитів GET для оновлення інформації).

2.1.3 Опитувач HTTP Polling

У HTTP Polling клієнт опитує сервер, що вимагає нової інформації, дотримуючись одного з наведених нижче механізмів. Опитування використовується переважною більшістю додатків сьогодні, і більшість випадків воно застосовується із практикою RESTful. На практиці HTTP Short Polling використовується дуже рідко, але HTTP Long Polling та Periodic Polling завжди є дефолтним вибором.

- HTTP Short Polling: Простіший підхід. Багато запитів обробляються, коли вони надходять на сервер, створюючи багато трафіку (використовує ресурси, але звільняє їх, як тільки відповідь буде відправлена назад). Оскільки кожне з'єднання відкрите лише на короткий проміжок часу, багато з'єднань можуть бути мультиплексованими за часом;

```
00:00:00 C-> Is the cake ready?
00:00:01 S-> No, wait.
00:00:01 C-> Is the cake ready?
00:00:02 S-> No, wait.
00:00:02 C-> Is the cake ready?
00:00:03 S-> Yeah. Have some lad.
00:00:03 C-> Is the other cake ready?
```

Рисунок 2.1 – Приклад HTTP Short Polling

– HTTP Long Polling: Один запит надходить на сервер, а клієнт чекає відповіді. Сервер тримає запит відкритим, доки не з'являться нові дані (вони не вирішені та ресурси заблоковані). Ви отримуєте повідомлення без затримок, коли відбувається подія сервера. Більш складні та використовувані серверні ресурси [3];

```
12:00 00:00:00 C-> Is the
cake ready?
12:00 00:00:03 S-> Yeah.
Have some lad.
12:00 00:00:03 C-> Is the
other cake ready?
```

Рисунок 2.2 – Приклад HTTP Long Polling

– HTTP Periodic Polling**:** Існує заздалегідь визначений проміжок часу між двома запитами. Це вдосконалена/керована версія опитування. Ви можете зменшити споживання сервера, збільшивши часовий проміжок між двома запитами. Але якщо вам потрібно негайно повідомити про подію сервера, це невдалий варіант.

Рисунок 2.3 – Приклад HTTP Periodic Polling

2.1.4 Потокове передавання HTTP Streaming

Клієнт робить HTTP-запит, а сервер подає відповідь невизначеної довжини (це як нескінченне опитування). Потокове передавання HTTP є продуктивним, простим у використанні та може бути альтернативою WebSockets.

Проблема: посередники можуть перервати з’єднання (наприклад, час очікування, посередники, що обслуговують інші запити круглим способом). У таких випадках він не може гарантувати повну реальність [4].

```
00:00:00 CLIENT-> I need cakes
00:00:01 SERVER-> Wait for a moment.
00:00:01 SERVER-> Cake-1 is in process.
00:00:02 SERVER-> Have cake-1.
00:00:02 SERVER-> Wait for cake-2.
00:00:03 SERVER-> Cake-2 is in process.
00:00:03 SERVER-> You must be enjoying cake-1.
00:00:04 SERVER-> Have cake-2.
00:00:04 SERVER-> Wait for cake-3.
00:00:05 CLIENT-> Enough, I'm full.
```

Рисунок 2.4 – Приклад HTTP Streaming

2.1.5 Server Sent Events (SSE).

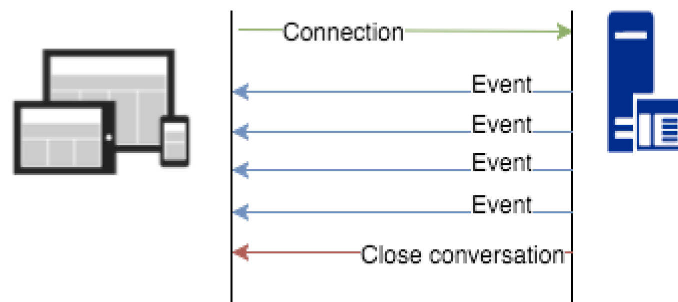

Рисунок 2.5 – Приклад SSE

- З’єднання SSE можуть лише надсилати дані до браузера. (зв’язок здійснюється від сервера до браузера, браузери можуть підписатися лише на оновлення даних, створені сервером, але не можуть відправляти будь-які дані на сервер);
- Приклади програм: оновлення Twitter, котирування акцій, оцінки в крикет, повідомлення в браузер;
- Проблема №1: Деякі браузери не підтримують SSE;
- Проблема №2: Максимальна кількість відкритих підключень обмежена 6 або 8 через HTTP/1.1 (на основі версії браузера). Якщо ви використовуєте HTTP/2, проблеми не буде, оскільки одного TCP-з'єднання достатньо для всіх запитів (завдяки мультиплексній підтримці в HTTP/2).

2.1.6 Пушери HTTP/2 Server Push

- Механізм для сервера заздалегідь попередньо надсилати ресурси (таблиці стилів, сценарії, медіа) до кешу клієнта;
- Приклади програм: стрічки соціальних мереж, програми на одній сторінці;
- Проблема №1: Посередники (проксі-сервери, маршрутизатори, хости) можуть вирішити не надсилати належним чином інформацію клієнту за призначенням вихідного сервера;

– Проблема №2: Зв’язки не відкриваються на невизначений час. З’єднання може бути розірвано в будь-який час, навіть коли відбувається процес просування вмісту. Після закриття та повторного відкриття це з’єднання не може продовжуватися з того місця, де воно залишилось;
– Проблема №3: Деякі браузери/посередники не підтримують Push Server.

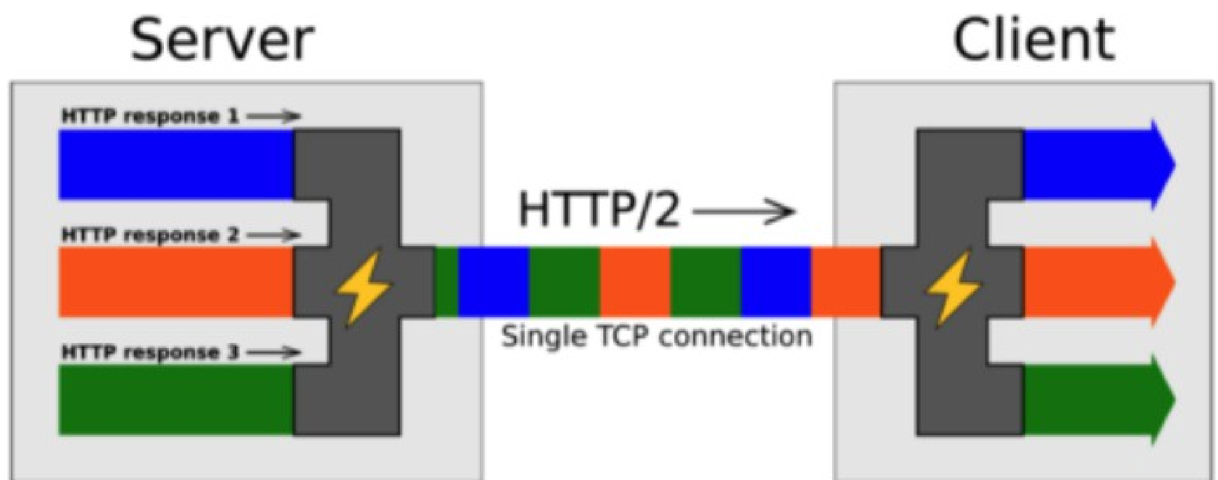

Рисунок 2.6 – Приклад HTTP/2 Server Push

2.1.7 Технологія WebSockets

WebSockets дозволяють як серверу, так і клієнту передавати повідомлення в будь-який час без будь-якого відношення до попереднього запиту. Однією з помітних переваг використання WebSockets є те, що майже кожен браузер підтримує WebSockets [6].

WebSocket вирішує кілька проблем з HTTP:

– Двонаправлений протокол - будь-який клієнт/сервер може надіслати повідомлення іншій стороні (у HTTP запит завжди ініціюється клієнтом, а відповідь обробляється сервером - що робить HTTP односпрямованим протоколом);

- Повнодуплексний зв’язок - клієнт і сервер можуть одночасно спілкуватися один з одним самостійно;
- Єдине TCP-з'єднання - після оновлення HTTP-з'єднання на початку клієнт і сервер спілкуються через те саме TCP-з'єднання протягом усього життєвого циклу з'єднання WebSocket;

```
00:00:00 CLIENT-> I need cakes
00:00:01 SERVER-> Wait for a moment.
00:00:01 CLIENT-> Okay, cool.
00:00:02 SERVER-> Have cake-1.
00:00:02 SERVER-> Wait for cake-2.
00:00:03 CLIENT-> What is this flavor?
00:00:03 SERVER-> Don't you like it?
00:00:04 SERVER-> Have cake-2.
00:00:04 CLIENT-> I like it.
00:00:05 CLIENT-> But this is enough.
```

Рисунок 2.7 – Приклад WebSocket зв’язку

Приклади програм: програми для обміну миттєвими повідомленнями/чату, ігри, інтерфейси адміністратора [4].

Хоча WebSockets, як кажуть, підтримується у кожному браузері, у посередників теж можуть бути винятки:

- Несподівана поведінка посередників: якщо ваші з’єднання WebSocket проходять через проксі-сервери/брандмауери, ви, можливо, помічали, що такі зв’язки постійно не працюють. Завжди використовуйте захищені веб-розетки (WSS), щоб різко зменшити такі збої. Тому будьте обережні і готуйтеся до них, використовуючи WSS і повертаючись до підтримуючого протоколу;
- Посередники, які не підтримують WebSockets: Якщо з якихось причин протокол WebSocket недоступний, переконайтесь, що

ваше з'єднання автоматично відновлюється до відповідного довгого опитування.

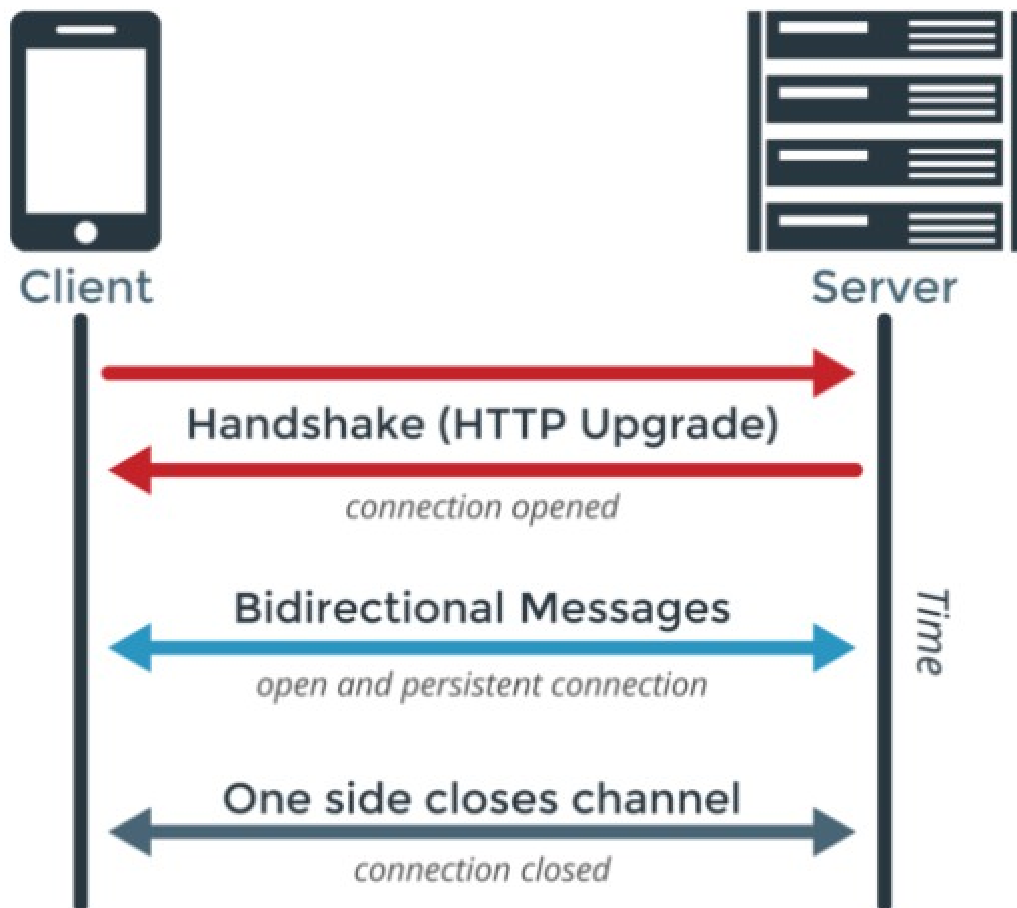

Рисунок 2.8 – Зв'язок WebSocket між сервером та клієнтом

2.2 REST проти Websockets

Якщо ви проводите перевірку продуктивності для REST та Websockets, ви можете виявити, що Websockets працюють краще, коли присутні великі навантаження. Це не обов'язково означає, що REST неефективний. Моя особиста думка полягає в тому, що порівнювати REST з Websockets - це все одно, що порівнювати яблука з апельсинами. Ці дві функції вирішують дві різні проблеми, і їх не можна порівняти з таким простим тестом на перфоманс.

Перший графік (рис. 2.9) показує час у мілісекундах, необхідний для обробки N повідомлень для постійного розміру корисного навантаження.

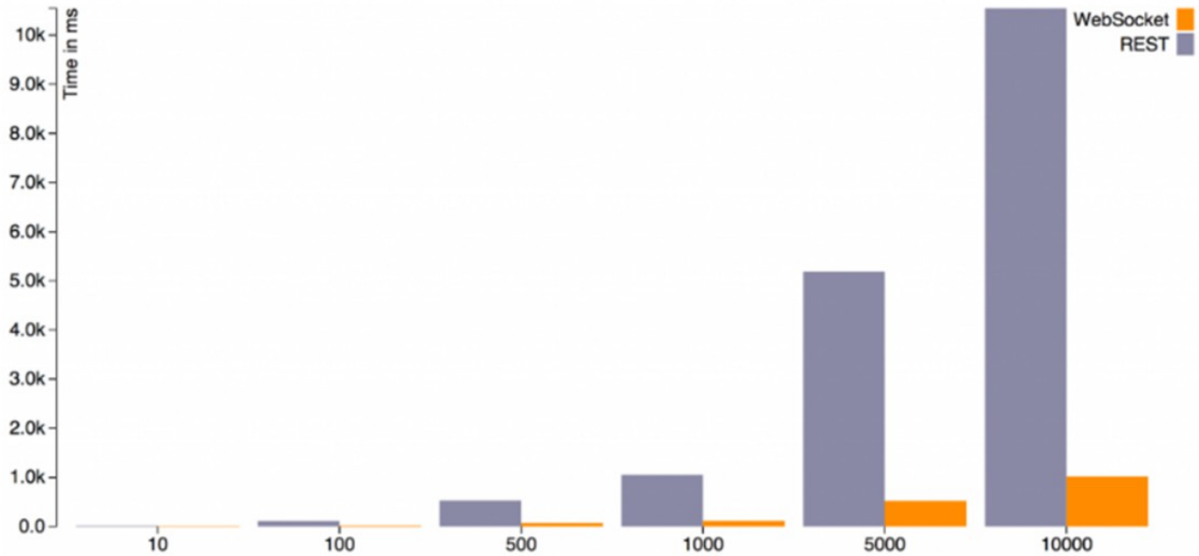

Рисунок 2.9 – Кількість повідомлень (корисне навантаження = 1000 байт)

Постійне корисне навантаження, збільшення кількості повідомлень показує нам, що немає необхідності використовувати REST (рис. 2.10).

Messages	REST (in ms)	WebSocket (in ms)	x times
10	17	13	1.31
100	112	20	5.60
500	529	68	7.78
1000	1050	115	9.13
5000	5183	522	9.93
10000	10547	1019	10.35

Рисунок 2.10 – Результати збільшення кількості повідомлень

Другий графік (рис. 2.11) показує час, необхідний для обробки фіксованої кількості повідомлень, змінюючи розмір корисного навантаження.

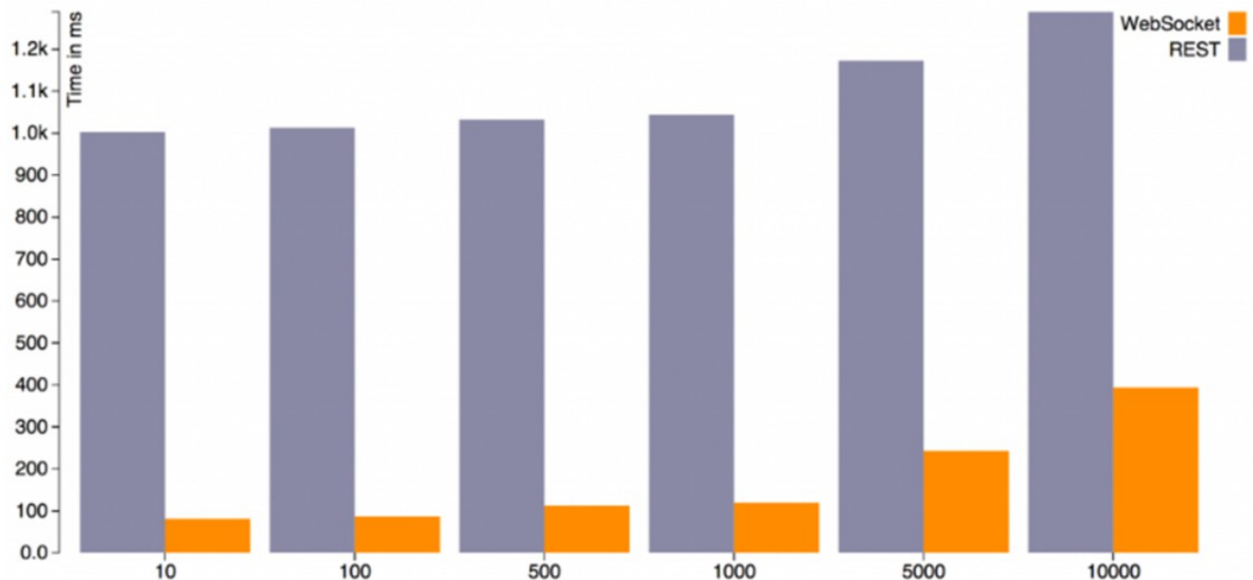

Рисунок 2.11 – Корисне навантаження (кількість повідомлень = 1000)

Постійна кількість повідомлень, збільшення корисного навантаження показує нам, що немає необхідності використовувати REST (рис. 2.12).

Payload (in bytes)	REST (in ms)	WebSocket (in ms)	x times
10	1003	81	12.38
100	1013	87	11.64
500	1032	113	9.13
1000	1044	119	8.77
5000	1173	243	4.83
10000	1289	394	3.27

Рисунок 2.12 – Результати збільшення корисного навантаження

На першому графіку (рис. 2.9) накладні витрати REST зростають у порівнянні з кількістю повідомлень, оскільки стільки підключень TCP потрібно ініціювати і розірвати, а також що багато заголовків HTTP потрібно відправити та отримати. На другому графіку (рис. 2.11) додаткові витрати на обробку запиту/відповіді для кінцевої точки REST є мінімальними, і більшу частину часу витрачають на ініціювання/припинення з'єднання та вшанування семантики HTTP [3].

Однак тепер ви повинні розуміти, що WebSockets - це чудовий вибір для обробки довготривалого двоспрямованого потоку даних майже в режимі реального часу, тоді як REST чудово підходить для випадкових комунікацій. Використання WebSockets - це значні інвестиції, отже, це надмірна кількість випадкових підключень.

2.3 Зв’язок між серверами

Якщо ви хочете отримати дані з вашого API про зміну даних, опитування має бути першим варіантом, який вам спадає на думку. Але що стосується спілкування між серверами, неефективність опитування коштує нам багато (в середньому 98,5% опитувань марно витрачаються).

Веб-хуки є порятунком цієї проблеми. Тут пам’ятайте, що спілкування, як правило, відбувається між серверами. По-перше, вузол відправника заздалегідь реєструє URL-адресу зворотного виклику у вузлі / номерах приймача. Коли подія відбувається на стороні відправника, веб-хук спрацьовує і надсилає об'єкт події з новими даними як запит HTTP POST на вузол/адреси одержувача, використовуючи URL-адреси зворотного виклику, зареєстровані в кожному з них.

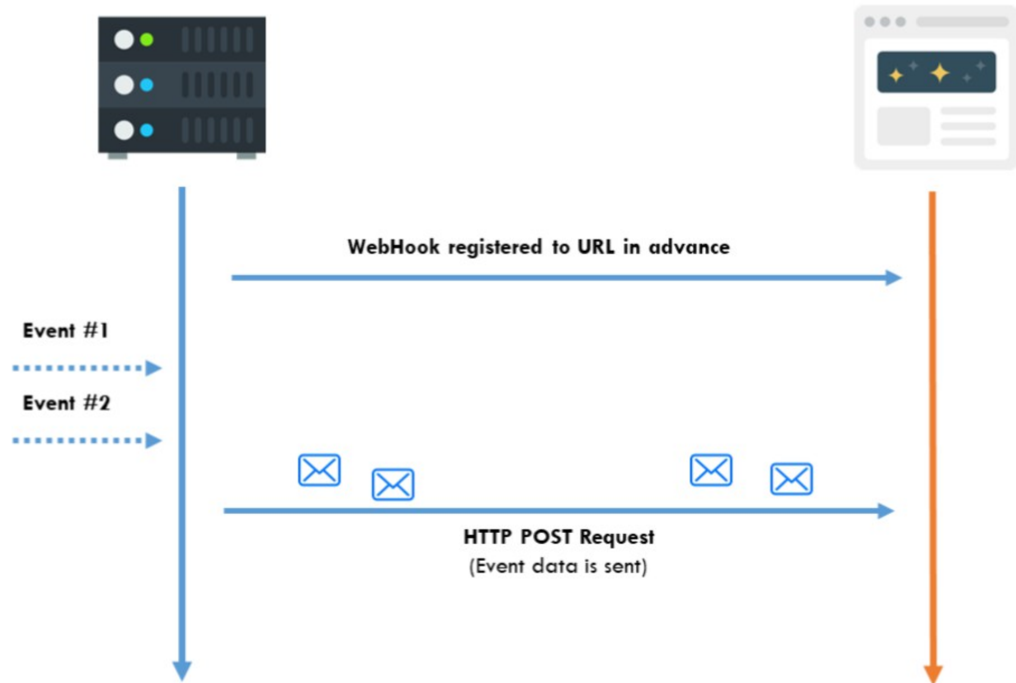

Рисунок 2.13 – Приклад роботи Веб-хуків

Класна річ полягає в тому, що навантаження на сервер для вузлів відправника та одержувача може бути різко зменшено за допомогою веб-хуків. Це забезпечує кращу взаємодію з користувачем, тоді як розробники можуть використовувати кінцеві точки вашого сервісу для значущих речей, не витрачаючи на опитування.

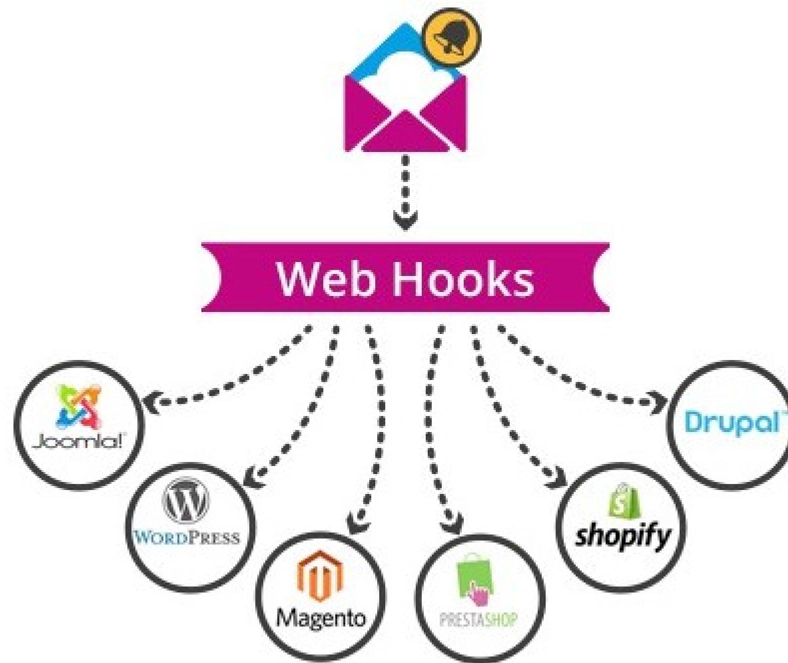

Рисунок 2.14 – Візуалізація роботи Веб-хуків

Веб-хуки зазвичай використовуються для надсилання сповіщень та змін стану між серверами, коли відбувається подія. Наприклад, коли користувач скасовує підписку за допомогою клацання кнопки в електронному листі, він потрапляє на сервер і відбувається подія відписки користувача, ця подія запускає відповідні веб-хуки, і вони повідомляють всі сервери, що користувач зараз відмовився від своїх послуг.

Зразки програм: Сповіщення про те, що новий користувач реєструється або поточний користувач оновлює існуючі налаштування профілю. Проблема: Розробникам важко налаштовувати веб-хуки та масштабувати служби HTTP.

2.4 Що використовувати для API

Яку техніку використовувати, залежить від того, що має більший сенс у контексті вашої заявки. Звичайно, ви можете скористатися деякими хитрощами, щоб імітувати поведінку однієї технології з іншою, але, як

правило, переважно використовувати ту, яка більше відповідає вашій моделі спілкування, якщо вона використовується в книзі.

HTTP/1.1 проти HTTP/2: це транспортні протоколи. Можливість мультиплексування в HTTP/2 чудова, але ще не скрізь підтримується. У таких випадках переконайтеся, що відмова від HTTP/2 та HTTP/1.1 не створить жодного безладу у вашій програмі. Для передачі даних HTTP/1.1 все ще є чудовим вибором [6].

RESTful API: Поки що RESTful API - це нормально для веб-додатків. Але трапляються дискусії щодо вивчення кращих шляхів. Прикладом є така концепція - «Замінити RESTful API на JSON-Pure». Мені ця ідея подобається, насправді JSON зручний для розробників, і ви можете з ним творити чудеса. Але ви знаєте, це розробляють концепції.

JSON проти XML: використовуйте JSON. Це тенденція, і з нею також зручно мати справу.

HTTP Polling: це старий, але все ще чудовий вибір для роботи з API. Якщо ваші дані змінюються часто або в режимі реального часу, не використовуйте коротке опитування, просто використовуйте WebSockets, кращу технологію для реального часу. Завжди використовуйте довгий та періодичний опитування належним чином (з принципами REST).

HTTP Streaming: Це добре для таких додатків, як новини/стрічки соціальних медіа, таблиці/табло, твіти тощо. Але на практиці люди використовують WebSockets, ніж HTTP Streaming [4].

HTTP/2 Server Push чудово підходить для надсилання ресурсів клієнту більш керованим способом. Але всі посередники та браузери цього не підтримують. Переконайтеся, що ви грамотно обробляєте такі відкиди.

Server-sent події, що надсилаються сервером, є досить новою технологією, яка ще не підтримується усіма основними браузерами, тому поки що це не варіант для серйозної веб-програми корпоративного рівня.

WebSockets забезпечують розширений протокол для здійснення двонаправленого, повнодуплексного зв'язку. Наявність двостороннього

каналу є більш привабливим для таких речей, як ігри, програми обміну повідомленнями, інструменти співпраці, інтерактивний досвід (включаючи мікро-взаємодії), а також для випадків, коли вам потрібні оновлення в реальному часі в обох напрямках [3].

Веб-хуки відрізняються від усіх вищезазначених технологій, оскільки це вирішує цілком конкретну проблему. Якщо ваші сервери потребують частого зв'язку та/або двонаправленого зв'язку, перейдіть за WebSockets. Якщо ваші сервери зрідка спілкуються, використовуйте дзвінки REST. Якщо вашим серверам потрібно взаємодіяти односпрямовано на тригер подій, перейдіть за веб-хуками. Не використовуйте опитування для перевірки змін даних або стану, це марнотратство.

3 ОПТИМІЗАЦІЯ ТА НАДІЙНІСТЬ ЗВ'ЯЗКУ

3.1 Принципи оптимізації

Обговоримо загальну архітектуру фронтенда, як забезпечити попередню завантаження необхідних ресурсів і збільшити ймовірність того, що вони в кеші. Трохи поміркуємо, як віддавати ресурси з бекенд і коли можна обмежитися статичними сторінками замість інтерактивного клієнтського додатку [5].

3.1.1 Виконання пошуку DNS

Пошук DNS - це процес перетворення доменного імені (наприклад, example.com) на IP-адресу (наприклад, 93.184.216.119). Для веб-запиту це означає отримання IP-адреси сервера, що розміщує веб-сайт, для даного імені домену.

Ваш браузер запускає веб-запит, запитуючи його вирішувач, де знаходиться доменне ім’я. Якщо IP-адреса кешована, вирішувач негайно повертає IP-адресу. Якщо IP-адреса не кешована, або якщо термін дії кешу закінчився (як визначається значенням TTL в зоні DNS домену), засіб розпізнавання запитує у кореневих DNS-серверів IP-адресу.

Використання надійного DNS-сервера має важливе значення для отримання якнайшвидшого дозволу IP-адреси доменного імені. Рекомендовано використовувати сервери імен A2 Hosting для вашого домену [5].

Крім того, встановлення правильних значень Time to Live (TTL) для записів домену у вашій зоні DNS є хорошим способом забезпечити, щоб кореневі сервери DNS не постійно шукали, де знаходиться ваш сервер. Значення TTL має бути встановлене менше доби, але більше години.

Якщо ви знаєте, що скоро будете використовувати нову IP-адресу, ви можете заздалегідь знизити значення TTL, щоб переміщення відбулося швидше. Після завершення переміщення встановіть для значення TTL значення від 4 до 6 годин.

3.1.2 Встановлення зв'язку

Браузер і сервер повинні встановити зв’язок, перш ніж вони зможуть спілкуватися між собою. Це з'єднання відоме як рукостискання TCP, яке складається з триступеневого процесу: SYN (синхронізоване), SYN-ACK (синхронізоване підтвердження) та ACK (підтвердження). Браузер відправляє SYN, сервер відповідає SYN-ACK, а браузер відповідає ACK. Коли цей трикроковий процес закінчується, веб-серверу Apache присвоюється потік веб-браузера [5].

Найкращий спосіб забезпечити швидший час підключення - це переконатися, що ваш сервер розташований відносно близько до вашої клієнтської бази. Між сервером і браузером існує багато пакетів, які рухаються вперед і назад, тому, якщо фізична відстань між браузером і сервером занадто велика, затримка кожного пакета збільшує час з'єднання.

Наприклад, якщо ваші клієнти знаходяться в Європі, спробуйте скористатися сервером, який знаходиться в Європі.

3.1.3 Переговори щодо рукостискання SSL

Під час рукостискання SSL браузер і сервер встановлюють зашифроване спілкування між собою за допомогою багатоетапного процесу:

1. Браузер надсилає серверу інформацію про свою версію SSL та налаштування, необхідні серверу для відкриття з'єднання SSL. Сервер надсилає ту саму інформацію браузеру, а також його сертифікат.
2. Браузер перевіряє, чи справжній сертифікат сервера. Потім браузер створює тимчасовий секретний ключ, зашифрований

відкритим ключем сервера, а потім надсилає зашифрований ключ серверу.

3. Сервер використовує свій приватний ключ для дешифрування тимчасового секретного ключа, створеного браузером, а потім клієнт і сервер генерують головний секретний ключ із тимчасового ключа.
4. І браузер, і сервер використовують головний секретний ключ для генерації сеансових ключів, які є симетричними ключами, що використовуються для шифрування та дешифрування інформації, якою обмінюються під час сеансу SSL. Ці ключі також використовуються для перевірки того, що дані не були змінені під час транспортування.
5. Браузер і сервер повідомляють один одному, що відтепер усі зв'язки між ними будуть зашифровані, узгодження SSL завершено, і запити можна надсилати.

Як і для часу підключення, найкращий спосіб покращити продуктивність SSL - це стратегічне розташування сервера поблизу вашої клієнтської бази. Наприклад, якщо ваші клієнти знаходяться в Європі, спробуйте скористатися сервером, який знаходиться в Європі.

3.1.4 Надсилання запиту

Браузер генерує запит до сервера, який включає URL-адресу, файли cookie, сеанси та заголовки, що описують, які форми відповідей браузер прийме від сервера. Браузер також надсилає будь-які дані форми, включаючи завантаження файлів. Якщо для з'єднання використовується SSL, запит шифрується.

Потім сервер надсилає підтвердження, що отримав запит.

Щоб покращити ефективність запиту, ви можете:

- Обмежте обсяг інформації, що зберігається у файлах cookie. Це зменшує розмір запиту, коли означає, що його можна надіслати швидше;
- Надсилайте необхідну інформацію лише у веб-формах, не включаючи зайвих прихованих полів. Ви можете використовувати сеансові файли cookie для зберігання інформації про користувача на сервері між запитами форм, а не передавати змінні у веб-форми.

3.1.5 Час очікування

Час очікування - це час між моментом, коли сервер підтверджує запит, і часом, коли браузер отримує перший байт, що містить початок заголовків відповіді від сервера. Цей крок включає обробку сценаріїв та дзвінки до бази даних.

Деякі сценарії повертають заголовок відповіді перед будь-якою обробкою або викликами бази даних, як правило, це позначається надзвичайно коротким часом очікування менше 10 мілісекунд. Коли заголовки надсилаються перед вмістом, час, який зазвичай відповідає часу очікування, додається до часу отримання або завантаження.

Якщо ваш сайт використовує популярну CMS (систему управління вмістом), таку як WordPress, Drupal або Magento, переконайтеся, що на вашому сайті немає активних плагінів чи розширень. Наприклад, на WordPress ви можете використовувати інструменти GTMetrix, щоб визначити, які плагіни або розширення тем уповільнюють роботу вашого сайту. New Relic - це ще один інструмент, який ви можете використовувати на будь-якому сайті, щоб допомогти визначити повільні функції та запити до бази даних [5].

Незалежно від швидкості програми, найбільший приріст продуктивності можна отримати за допомогою механізму кешування. Механізми кешування зберігають копію відображеного HTML-коду для

кожної сторінки вашого сайту [5]. Вони подають свіжий вміст через певний проміжок часу або видаляють кешовану копію сторінки, коли сторінку було змінено. Пам'ятайте, що перша особа, яка відвідає сторінку після закінчення терміну дії кеш-пам'яті, може мати довший час очікування, поки вона подаватиме новий вміст та зберігатиме нову копію кешу.

Багато CMS мають плагіни, які можуть автоматично кешувати вміст сайту.

3.1.6 Отримання відповіді

Це стільки часу, скільки потрібно браузеру для завантаження вмісту, сформованого сервером. Пам'ятайте, що це значення може бути більшим, ніж очікувалося, якщо сервер надсилає інформацію негайно, коли вона генерується, замість того, щоб генерувати весь вміст, а потім надсилати його.

Щоб підвищити продуктивність, увімкніть мініфікацію у вихідних файлах та використовуйте стиснення Gzip.

Ви можете запустити свій HTML-код через мініфікатор, перш ніж завантажувати його на веб-сервер. Мініфікатор, такий як HTMLTidy, видаляє коментарі та непотрібні пробіли з документа, щоб переконатися, що він є якомога компактнішим. CssTidy та JSTidy зменшують розмір файлів CSS та JavaScript. У Wordpress W3 Total Cache забезпечує мініфікацію, об'єднуючи всі файли CSS в один файл CSS, а також файли JavaScript. одночасно видаляючи пробіли та всі дубльовані правила.

Щоб використовувати стиснення Gzip, переконайтеся, що ви використовуєте модуль mod_deflate.

3.2 Надійність в сучасних WEB додатків

Надійні протоколи передачі даних (RDT, RDP) - це алгоритмічні заходи, що забезпечують надійну передачу даних через мережу, яка може спричинити втрату та/або пошкодження даних. RDT включає послідовності

та змінні на стороні відправника та одержувача для перевірки, підтвердження та повторної передачі даних, коли це необхідно.

Метою протоколів RDT є надання послуг мережевого та лінійного зв'язку, щоб послуги прикладного та транспортного рівня могли гарантувати доставку даних.

Надійні протоколи передачі даних повинні вирішувати дві основні проблеми втрати даних та пошкодження даних. У мережевих комунікаціях такі типи помилок зазвичай виникають на фізичному мережевому обладнанні під час буферизації, розповсюдження та передачі.

Для вирішення цих проблем відправникам та одержувачам потрібен спосіб опосередкованої комунікації щодо отримання та перевірки даних, що передаються [5]. У таких випадках, коли дані були втрачені або пошкоджені, протоколи RDT диктують, що дані слід передавати повторно. На найосновнішому рівні це вводить ще одне занепокоєння щодо дублікатів даних у суміші.

Втрата даних та корупція - це дві основні проблеми, які RDT прагне вирішити. Занепокоєння щодо дублікатів даних виникає лише через дії, вжиті для вирішення цих перших двох проблем. Таким чином, розгляд питань збитків та корупції є розумним місцем для початку розгляду основної функції RDT.

3.2.1 Підтвердження операції (ACK)

Коли система надсилає дані в іншу систему, RDT пропонує вжити деяких заходів для забезпечення успішного надходження даних. Одним з основних підходів для досягнення цього є включення такої системи підтвердження, щоб отримувач міг повідомити відправника "так, я отримав ваше повідомлення".

Як же надіслати таке підтвердження? Додавання зайвого біта в переданий сегмент даних досягає саме цього. Одержувач може надіслати відправникові повідомлення, тепер із "1", що вказує на позитивне

підтвердження ("повідомлення отримано, відправити наступне повідомлення") або "0", що вказує на негативне підтвердження ("останнє повідомлення було пошкоджене, повторно надішліть".

Такі протоколи, як Протокол управління передачею (TCP), інкапсулюють дані, інтегровані із заголовками, причому одне поле заголовка зарезервовано спеціально для цих типів підтверджень (ACK). Система ACK інформує відправника про втрату даних або пошкодження, коли отримується негативний ACK (NAK) або отримуються дублікати ACK.

3.2.2 Час очікування

Що відбувається, якщо приймач ніколи не отримує пакет на ACK або NAK? Такий може бути випадок втрати пакетів у мережі. У цьому сценарії відправник може застрягти нескінченно довго, чекаючи ACK/NAK, який ніколи не надійде.

Щоб пом'якшити цей випадок, RDT вимагає введення змінної таймера на стороні відправника. Коли відправник передає сегмент даних, він запускає таймер. Якщо таймер досягає порогу до отримання ACK/NAK, відправник повторно передає дані [5].

Додавання змінної таймера допомагає справитись із втратою даних, але є ще один дуже обмежуючий фактор: відправник обмежується відправленням одного сегмента даних, очікуванням на ACK/NAK/тайм-аутом, а потім відправленням або повторною передачею іншого сегмента даних.

Для кращого використання доступної пропускної здатності мережі повинна існувати система надсилання декількох сегментів даних, одночасно відстежуючи, які з них отримані, пошкоджені або втрачені на шляху.

3.2.3 Порядкові номери (SEQ)

Коли відправники передають дані через мережу, чекаючи зворотного зв’язку ACK від одержувачів, виникає інша проблема: як відправник знає, на які дані посилається ACK? Наприклад, скажімо, було передано 3 сегменти

даних, а відправник отримав 2 ACK і 1 NAK. Який сегмент потрібно буде повторно передавати?

У протоколі, обмеженому простими ACK/NAK, не було б ніякого способу, якщо відправник не обмежився надсиланням окремого сегмента даних одночасно. Цей підхід був використаний у ранніх мережевих комунікаціях і відомий як протокол зупинки і очікування. Безумовно, протоколи зупинки і очікування кращі за нічого, але вони можуть значно обмежити пропускну здатність мережі, якщо вони реалізовані випадково.

Одним із підходів до вирішення цього обмежувального фактора є введення нової змінної в сегмент даних: порядковий номер (SEQ). Ця змінна, яка тепер включена як 1-бітове поле в заголовках TCP, може дозволити відправнику передавати кілька сегментів і чекати ACK для кожного.

Один з підходів, відомий як протокол змінних бітів, передбачає, що відправник просто чергує між 0 і 1 для кожного сегмента, переданого через мережу. Отримувач, у свою чергу, відповість ACK та значенням SEQ або 0 або 1, вказуючи останню дійсну отриману послідовність [5].

Ця система підтвердження відома як автоматичний повторний запит (ARQ). Відправник може визначити, які пакети потрібно повторно відправити, і автоматично це зробити. Протокол змінних бітів - це дуже спрощений підхід до ARQ, який створює більш дієву систему RDT, але має деякі серйозні обмеження.

Наприклад, якщо для SEQ-номера можна використовувати лише два унікальних значення, як можна одночасно передавати більше двох сегментів? Це питання стосується теми, відомої як конвеєризація, за допомогою якої RDT передає кілька пакетів одночасно, обробляє кілька ACKS і ретранслює різними способами.

3.2.4 Протокол конвеєризації

Конвеєризація - це протокол ARQ, який дозволяє значно збільшити швидкість передачі. У сучасних протоколах, таких як TCP, діапазон

доступних порядкових номерів становить 232 (0 -4 294 967 295 безцільних цілих чисел). Однак існує ще кілька підходів щодо того, як слід обробляти ретрансляцію пошкоджених або втрачених сегментів даних [5].

Розглянемо такий сценарій: відправник передає 5 пакетів із позначкою SEQ0-SEQ4 і чекає ACK від приймача. Приймач відповідає ACK для SEQ0, SEQ1 та SEQ4, вказуючи, що SEQ2 та SEQ3 мали проблеми. Що має статися в цей момент?

Одним із варіантів буде повторна передача лише SEQ2 та SEQ3. Іншим варіантом буде повторна передача всього діапазону SEQ0-SEQ4. Ці підходи характерні для конкретних типів ARQ, відомих як Go-Back-N та Selective Repeat. Кожне з них визначає, як обліковуються послідовності та як/коли дані повинні повторно передаватися.

3.2.5 Протокол Go-Back-N (GBN)

Go-Back-N - це протокол ковзного вікна, за допомогою якого відправник підтримує вікно пакетів, які можуть надсилатися по мережі. Наприклад, якщо відправник має 30 загальних пакетів, які мають число SEQ від 0 до 29, вікно передачі розміром 5 може включати SEQ5-SEQ9 лише в певний час.

Щоб краще зрозуміти поняття вікна, як воно використовується тут, розгляньте визначення меж:

- Мінімальне значення - це номер SEQ з останнього відправленого пакета, який не був ACK. Це називається базовим числом;
- Максимальним значенням є SEQ базової + N для вікна розміром n. У нашому прикладі вище розмір вікна 5 дав би максимум + 5. Це представляє номер SEQ останнього пакета, дозволеного для передачі.

Протокол GBN диктує, що відправник повинен відповісти на 3 унікальні події:

1. Дані програми, отримані з вищевказаного рівня;
2. ACK, отриманий від приймача;
3. Подія очікування для раніше переданого пакета.

Механіка GB дещо проста: відправник передає кілька пакетів і чекає на ACK від приймача. ACK включають номер SEQ, що вказує останній отриманий пакет. Оскільки відправник отримує ACK, базове значення (що представляє мінімум вікна) ковзає вперед, таким чином, “дозволяючи” збільшені SEQ пронумеровані сегменти.

З боку одержувача пакети, які надходять не в порядку, просто відкидаються. У багатьох випадках це становить значну втрату мережевих ресурсів. Головною перевагою цього підходу є усунення необхідності зберігати пакети, що не належать до послідовності, у певній формі буфера на стороні приймача. Це зменшує використання ресурсів на стороні одержувача та спрощує реалізацію.

3.2.6 Вибіркове повторення (SR)

Вибіркове повторення подібне до GBN, але вирішує проблему зайвих мережевих передач. Наприклад, там, де приймачі GBN відкидають пакети, що не належать до послідовності, і повторно передають сигнали цілих діапазонів послідовностей, SR може вимагати повторної передачі конкретних пакетів.

Цей підхід очевидні переваги, але має деякі недоліки, такі як додаткова складність та споживання ресурсів. Однією з таких додаткових складностей є необхідність одержувача відстежувати та розпізнавати окремі пакети незалежно від того, отримані вони в порядку чи ні.

Це вимагає буферизації пакетів, що не входять в послідовність, доки не будуть отримані такі тимчасові пакети з номерами SEQ до тих, що надходять.

Подібно до GBN, Selective Repeat диктує, що відправник вживає заходів після таких подій:

1. Потрібні дані, отримані від рівня додатку вище, з інструкцією щодо передачі;
2. Час очікування для вже переданих пакетів;
3. ACK, отриманий для вже переданих пакетів.

3.2.7 Забезпечення надійності

Надійні протоколи даних підтримують послуги транспортного рівня, необхідні для багатьох сучасних мережевих комунікацій. Використання підтверджень, таймерів, порядкових номерів, буферів та розумних алгоритмів допомагає забезпечити надходження повідомлень відправників до одержувачів по всьому світу [5].

Основи RDT можуть бути забезпечені так званими підходами зупинки і очікування, що використовують такі конструкції, як змінні бітові протоколи. Хоча діє в концепції; ці підходи не стосуються потреби у надійній пропускній здатності, доступній у багатьох сучасних мережах. Повернення-N та вибіркове повторення пропонують рішення для цих проблем, хоча все одно вимагають старанного впровадження, щоб забезпечити оптимальну роботу в певній області умов/вимог мережі.

4 ЗАСТОСУВАННЯ ТЕХНОЛОГІЇ WEBSOCKET ДО КЛІЄНТ-СЕРВЕРНОГО ДОДАТКУ

4.1 Опис додатку

Я покажу вам мінімальний код, необхідний для запуску чату в реальному часі за допомогою React, Node та WebSocket.

Звідти вже можна пограти, дослідити та додати власні ідеї та особливості до цієї програми.

Врешті-решт ми отримаємо програму, яка виглядає приблизно як на рисунку 4.1.

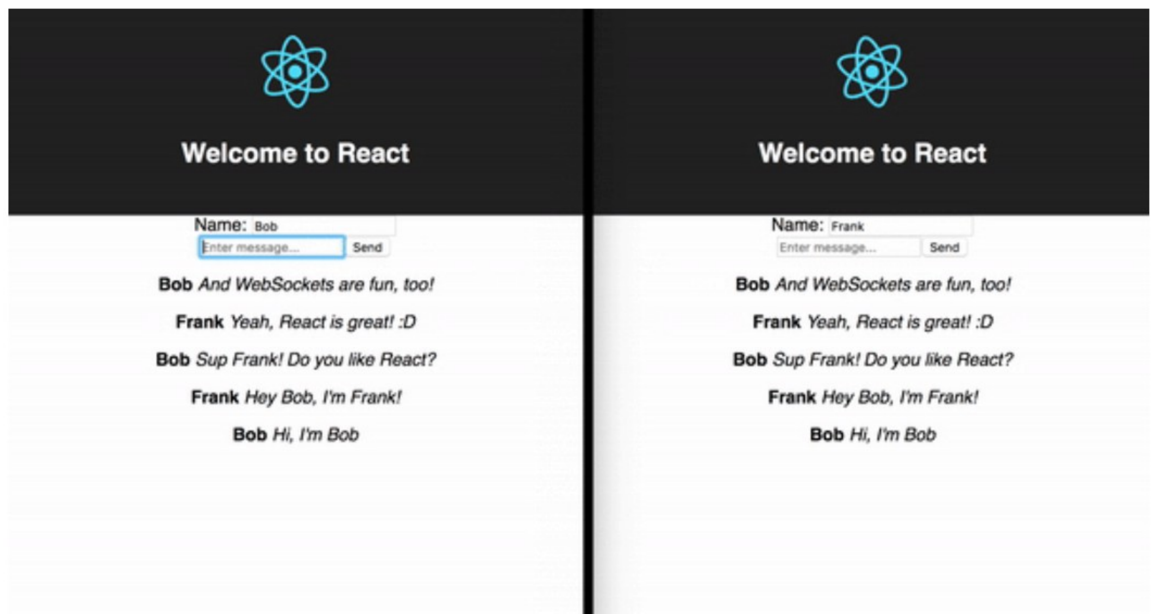

Рисунок 4.1 – Клієнтське відображення чату

Ви бачите програму React, запущену у двох вікнах браузера, які розміщують свої повідомлення у програмі вузла у фоновому режимі, яка транслює всі повідомлення назад до всіх підключених програм.

React (іноді React.js або ReactJS) - JavaScript-бібліотека з відкритим вихідним кодом для розробки призначених для користувача інтерфейсів.

React розробляється і підтримується Facebook, Instagram і співтовариством окремих розробників і корпорацій [7].

React може використовуватися для розробки односторінкових і мобільних додатків. Його мета - надати високу швидкість, простоту і масштабованість. Як бібліотеки для розробки призначених для користувача інтерфейсів React часто використовується з іншими бібліотеками, такими як MobX, Redux і GraphQL.

Node або Node.js - програмна платформа, заснована на движку V8 (здійснює трансляцію JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованого мови в мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями введення-виведення через свій API, написаний на C ++, підключати інші зовнішні бібліотеки, написані на різних мовах, забезпечуючи виклики до них з JavaScript-коду [7]. Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js і десктопні віконні додатки (за допомогою NW.js, AppJS або Electron для Linux, Windows і macOS) і навіть програмувати мікроконтролери (наприклад, tessel, low.js і espruino). В основі Node.js лежить подієво-орієнтоване і асинхронне (або реактивне) програмування з неблокуючим введенням/висновком.

4.2 Розробка серверної частини Web-додатку

Створюємо простий сервер WebSocket, який транслює всі вхідні повідомлення кожному, хто підключений. Отже, у кореневому каталозі проекту виконайте такі команди, щоб створити окремий серверний каталог та встановити ws.

```
mkdir backend
cd backend
yarn add ws
```

Рисунок 4.2 – Створення серверного каталогу

Тоді нам також знадобиться файл actualserver.js.

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 3030 });

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(data) {
    wss.clients.forEach(function each(client) {
      if (client !== ws && client.readyState === WebSocket.OPEN) {
        client.send(data);
      }
    });
  });
});
```

Рисунок 4.3 – Коннект частина сервера

На кожному вхідному повідомленні воно надсилає його назад усім підключеним клієнтам.

4.2 Написання клієнтської частини Web-додатку

Встановлюємо CRA yarn global add create-react-app.

Потім повернемося до кореня проекту, і створюємо нову програму для реагування.

```
create-react-app frontend
cd frontend
yarn add prop-types
```

Рисунок 4.4 – Нова програма для реагування

Тепер у нас є frontend та backend каталог.

Створюємо компонент ChatMessage.js у новому frontend каталозі. Це визначає, як буде виглядати кожне окреме повідомлення чату.

```
import React from 'react'

export default ({ name, message }) =>
  <p>
    <strong>{name}</strong> <em>{message}</em>
  </p>
```

Рисунок 4.5 – Компонент ChatMessage.js

Він виведе ім'я користувача жирним шрифтом, а потім повідомлення.

Створюємо компонент ChatInput.js. Ми покажемо його вгорі в чаті, щоб вводити нові повідомлення.

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

class ChatInput extends Component {
  static propTypes = {
    onSubmitMessage: PropTypes.func.isRequired,
  }
  state = {
    message: '',
  }

  render() {
    return (
      <form
        action="."
        onSubmit={e => {
          e.preventDefault()
          this.props.onSubmitMessage(this.state.message)
          this.setState({ message: '' })
        }}
      >
        <input
          type="text"
          placeholder={'Enter message...'}
          value={this.state.message}
          onChange={e => this.setState({ message: e.target.value })}
        />
        <input type="submit" value={'Send'} />
      </form>
    )
  }
}

export default ChatInput
```

Рисунок 4.6 – Компонент ChatInput.js

Цей компонент отримує один опис onSubmitMessage від компонента чату, який ми збираємось створити, який викликається під час надсилання форми. Крім того, коли форма подається, введення повідомлення знову очищається.

Тепер створюємо компонент Chat.js, який буде центром нашої логіки чату. Він утримує наш стан, управляє зв’язком, а також надсилає та отримує повідомлення.

```
import React, { Component } from 'react'
import ChatInput from './ChatInput'
import ChatMessage from './ChatMessage'

const URL = 'ws://localhost:3030'

class Chat extends Component {
  state = {
    name: 'Bob',
    messages: [],
  }

  ws = new WebSocket(URL)

  componentDidMount() {
    this.ws.onopen = () => {
      // on connecting, do nothing but log it to the console
      console.log('connected')
    }

    this.ws.onmessage = evt => {
      // on receiving a message, add it to the list of messages
      const message = JSON.parse(evt.data)
      this.addMessage(message)
    }

    this.ws.onclose = () => {
      console.log('disconnected')
      // automatically try to reconnect on connection loss
      this.setState({
        ws: new WebSocket(URL),
      })
    }
  }

  addMessage = message =>
    this.setState(state => ({ messages: [message, ...state.messages] }))

  submitMessage = messageString => {
    // on submitting the ChatInput form, send the message, add it to the list and reset the input
    const message = { name: this.state.name, message: messageString }
    this.ws.send(JSON.stringify(message))
    this.addMessage(message)
  }

  render() {
    return (
      <div>
        <label htmlFor="name">
          Name: 
          <input
            type="text"
            id={'name'}
            placeholder={'Enter your name...'}
            value={this.state.name}
            onChange={e => this.setState({ name: e.target.value })}
          />
        </label>
        <ChatInput
          ws={this.ws}
          onSubmitMessage={messageString => this.submitMessage(messageString)}
        />
        {this.state.messages.map((message, index) =>
          <ChatMessage
            key={index}
            message={message.message}
            name={message.name}
          />,
        )}
      </div>
    )
  }
}

export default Chat
```

Рисунок 4.7 – Компонент Chat.js

Розглянемо уважніше компонент Чату. Він створює новий екземпляр WebSocket і додає обробники повідомлень, а також відкриває та закриває з'єднання. Він передає onSubmitMessage проп до нашого ChatInput, який фактично надсилає повідомлення до серверної бази вузла. Крім того, він відображає введення імені, яке ми також надсилаємо на серверну інформацію для ідентифікації користувачів.

Оновимо App.js, щоб він включав компонент Чат, щоб він фактично відображався на екрані.

```
import React, { Component } from 'react'
import logo from './logo.svg'
import './App.css'
import Chat from './Chat'

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <Chat />
      </div>
    )
  }
}

export default App
```

Рисунок 4.8 – Компонент App.js

Тепер запустимо свої команди, запустивши yarn start у frontend каталозі та node server.js у backend каталозі та відкриємо браузер (або кілька вікон браузера) на http://localhost:3000. Побачимо точно таке зображення як на рисунку 4.1.

ВИСНОВКИ

У зв’язку з активним поширенням інтернет–технологій та розповсюдженням веб–сторінок з’являється необхідність у забезпеченні оптимізації та надійності зв’язку. Саме таку оптимізацію і надійність надають сучасні технології обміну інформацією, більшість із яких існує у відкритому доступі.

Завдяки багатої безлічі існуючих технологій, на поточний момент, кожен може обрати, що використовувати для досягнення особистої мети. При вдалому виборі кожен додаток або програма стає швидше за рахунок швидкого обміну збереженої інформації та додаванню нової.

Як показало дослідження сучасного ринку – саме realtime технології знайшли поширене використання за рахунок швидкості передачі даних, та оптимізації, що гарантує безпеку збереженості даних. Також у кваліфікаційній роботі було розглянуто багато різних технологій, виявлено переваги та недоліки.

У другому розділі було проведено роботу по порівнянню існуючих технологій передачі даних у WEB для створення додатку.

Завдяки створенню додатку, у четвертому розділі, було визначено як саме працює архітектура клієнт–сервер. А саме: що потрібно для створення додатку, як посилається запит зі сторони клієнта, як оброблюється цей запит на стороні серверу та як відбувається обмін інформацією.

Додаток було протестовано та не виявлено недоліків.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Client-server architecture / The Editors of Encyclopaedia Britannica, 2019. [Електронний ресурс]/Режим доступу до ресурсу: https://www.britannica.com/technology/client-server-architecture
2. Differences between TCP and UDP / geeks for geeks, 2021. [Електронний ресурс]/Режим доступу до ресурсу: https://www.geeksforgeeks.org/differences-between-tcp-and-udp/
3. Multimedia Networking Technologies, Protocols, & Architectures / Iván Vidal, Ignacio Soto, Albert Banchs, Jaime Garcia-Reinoso, 2019. – 458 с.
4. Patterns in Network Architecture: A Return to Fundamentals / John Day, 2007. – 464 с.
5. Network Inference for Cyber Security in Complex Networks / Chee Wei Tan, 2021. - 200 с.
6. WebSocket: Lightweight Client-Server Communications / Andrew Lombardi, 2015. - 144 с.
7. Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks / David Choi, 2020. - 648 с.