

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут фізики, математики та інформаційних
технологій

Кафедра інформаційних технологій та систем

Матієвський Володимир Валерійович

**АНАЛІЗ МОЖЛИВОСТЕЙ HTTP/3 ТА РОЗРОБКА ДОДАТКУ НА
ЙОГО ЗАСАДАХ**

**кваліфікаційна робота
здобувача вищої освіти другого (магістерського) рівня
освітньої програми «Комп'ютерні мережі»
за спеціальністю 123 Комп'ютерна інженерія**

Особистий підпис



_____ Володимир МАТІЄВСЬКИЙ

Науковий керівник

Світлана ПЕРЕЯСЛАВСЬКА,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри

Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Полтава – 2023

АНОТАЦІЯ

Матієвський В. В.

Тема: Аналіз можливостей http/3 та розробка додатку на його засадах.

Спеціальність: 123 «Комп'ютерна інженерія».

Установа: ЛНУ імені Тараса Шевченка, 2023р.

Магістерська робота містить: 68 с., 21 рис., 4 табл., 41 джерел, 2 додатки.

Об'єкт дослідження – протокол передачі гіпертексту (HTTP), як базовий для діяльності всесвітньої павутини (WWW).

Предмет дослідження – особливості та можливості протоколу передачі гіпертексту версії 3.0 (HTTP/3.0) та створення додатку на його засадах.

Мета роботи - зробити аналіз можливостей протоколу HTTP/3 та розробити програмний засіб, який його використовує.

Результати роботи – в роботі проаналізовано розвиток протоколу HTTP у часі, досліджено основні методи запитів, заголовки запитів та відповідей, проаналізовано особливості HTTP/2.0 які підвищують ефективність WWW. Зроблена порівняльна характеристика протоколів TCP та QUIC, досліджено протокол HTTP/3.0 та можливості використання мови Python для праці із HTTP/3.0. Створено додаток на засадах HTTP/3.0

Ключові слова: HTTP/3.0 QUIC TCP WWW AIOQUIC HYPERCORN.

ANNOTATION

Matiievskyi Volodymyr

Theme: Analysis of http/3 capabilities and application development based on it.

Specialty: 123 "Computer Engineering".

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2023 year.

Master's work of: 68 p., 21 im, 4 tables, 41 sources 2 supplement.

A research object of: – Hypertext Transfer Protocol (HTTP) as a base for World Wide Web activity (WWW).

The article of research – features and capabilities of hypertext transfer protocol version 3.0 (HTTP/3.0) and creation of the application on its basis.

An aim of research is – analyze the capabilities of the HTTP/3 protocol and develop a software tool that uses it

Job performances – the paper analyzes the development of the HTTP protocol over time, explores the main methods of requests, request and response headers, analyzes the features of HTTP/2.0 that increase the efficiency of the WWW. A comparative description of the TCP and QUIC protocols was made, the HTTP/3.0 protocol and the possibilities of using Python to work with HTTP/3.0 were investigated. Created an application based on HTTP/3.0.

Keywords: HTTP/3.0 QUIC TCP WWW AIOQUIC HYPERCORN.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	5
ВСТУП.....	6
РОЗДІЛ 1 ПРОТОКОЛ HTTP ТА ЙОГО ОНТОГЕНЕЗ.....	9
1.1. Зародження WWW: протокол HTTP (0.9, 1.0)	9
1.2. Стандартизація протоколу HTTP/1.1	12
1.3. Підвищення ефективності у версії HTTP/2.0	21
1.4. Висновки до розділу 1	27
РОЗДІЛ 2 ОГЛЯД ОСНОВНИХ ОСОБЛИВОСТЕЙ HTTP/3.0	29
2.1. Огляд протоколу QUIC	29
2.2. Покращення продуктивності в HTTP/3.0.....	42
2.3. Практичні рекомендації по розгортанню HTTP/3.0	50
2.4. Висновки до розділу 2	55
РОЗДІЛ 3 РОЗРОБКА ДОДАТКА НА ЗАСАДАХ HTTP/3.....	58
3.1. Опис бібліотек на Python, які реалізують HTTP/3.0	58
3.2. Опис додатка побудованого на основі HTTP/3.0.....	60
3.3. Висновки до розділу 3	63
ВИСНОВКИ	64
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	65
ДОДАТОК А.....	69
ДОДАТОК Б	72

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

WWW	World Wide Web
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
QUIC	Quick UDP Internet Connections
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
TLS	Transport Layer Security
IETF	Internet Engineering Task Force
MIME	Multipurpose Internet Mail Extensions
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
CDN	Content Delivery Network
RTT	Round-Trip Time
HoL	Head-Of-Line blocking
CID	Connection IDentifier (QUIC)
TTFB	Time To First Byte

ВСТУП

Протокол передачі гіпертексту (HTTP) використовується для доступу до переважної більшості ресурсів в Інтернеті, від веб-магазинів до соціальних мереж і платформ для спільної роботи. HTTP зародився на початку 90-х, а його перша версія (HTTP 1.0) була створена в 1996 році. Другу версію (HTTP/2) було стандартизовано у 2014 році, вона включала значні зміни в механізмах кадрування протоколу. HTTP/3 є третьою версією HTTP і зараз знаходиться на завершальній стадії стандартизації в IETF так в 2022 отримав статус «Proposed Standard». HTTP/3 в деяких умовах показує переваги продуктивності та покращення безпеки порівняно з HTTP/2. Одна з основних змін полягає в тому, що HTTP/3 замінює TCP як транспортний рівень на користь QUIC, транспортного протоколу на основі UDP, спочатку запропонованого Google і наразі стандарту IETF. Крім того, HTTP/3 запроваджує ефективніший механізм стиснення заголовків і використовує TLS 1.3 (або новішої версії) для покращення безпеки.

Очікується, що HTTP/3 займе місце HTTP/2 у найближчі кілька років, і деякі провідні інтернет-компанії вже підтримують його станом на кінець 2022 року, наприклад CloudFlare CDN, Fastly і Facebook.. Що ще важливіше, вплив протоколу на продуктивність Інтернету ще не було широко виміряно. Такі зусилля важливі для зовнішньої перевірки переваг протоколу, які були оцінені лише кількома постачальниками послуг, які його розгорнули.

Дослідники вказують, що спочатку HTTP був розроблений без зосередження на безпеці та надійності; це одна з головних мотивацій розробки HTTP/2, однак його прийняття запровадило нові види атак.

Наступна основна версія HTTP, а саме HTTP/3, є оновленням HTTP/2 з точки зору продуктивності, надійності та безпеки; водночас він базується на протоколі QUIC і значною мірою змінює спосіб взаємодії веб-браузерів і серверів, враховуючи, що він використовує UDP як протокол транспортного

рівня замість TCP, що робить його потенційним джерелом додаткових проблем безпеки.

Питання вивчення можливостей HTTP/3 та проблем безпеки, які можуть виникнути при його масовому використанні цікавить багато науковців Е. Чацоглу (Efstratios Chatzoglou), Д. Стенберг (D. Stenberg), Р. Маркс (R. Marx) [25], Г. Перна (G. Perna) та інші.

Таким чином тема дослідження «Аналіз можливостей http/3 та розробка додатку на його засадах» є актуальною для вивчення.

Об'єкт дослідження – протокол передачі гіпертексту (HTTP), як базового для діяльності всесвітньої павутини (WWW).

Предмет дослідження – особливості та можливості протоколу передачі гіпертексту версії 3.0 (HTTP/3.0) та створення додатку на його засадах.

Мета роботи - зробити аналіз можливостей протоколу HTTP/3 та розробити програмний засіб, який його використовує.

Для досягнення поставленої мети необхідно вирішити наступні **завдання**:

- 1) проаналізувати історичний розвиток протоколу HTTP та можливості його практичного застосування;
- 2) дослідити HTTP/1.1 транзакції, методи запитів, основні заголовки запитів та відповідей, зробити аналіз архітектури web (проксі, шлюзи, тунелі);
- 3) дослідити особливості HTTP/2.0, які впливають на підвищення ефективності праці WWW та зробити порівняльний аналіз першої та другої версії протоколу;
- 4) зробити огляд нового транспортного протоколу QUIC;
- 5) дослідити покращення продуктивності, які надає HTTP/3.0 та на основі проведеного аналізу сформулювати рекомендації по його використанню;
- 6) дослідити можливості використання мови Python для праці із HTTP/3.0 та створити додаток на засадах HTTP/3.0.

Методи дослідження. Поставлені завдання вирішені наступними методами: абстрагування й конкретизація, аналіз, синтез, порівняння. Також можливо виділити теоретичні методи: аналіз науково-технічних джерел з проблем дослідження, емпіричні методи: порівняльний аналіз різних версій протоколу HTTP, а також протоколів TCP та QUIC.

У першому розділі розглянуто розвиток протоколу HTTP від простого текстового однорядкового протоколу (версія 0.9) до складного бінарного (версія 2.0). Виконано огляд літератури з аналізом питань пов'язаних із архітектурою WWW, базою протоколу HTTP (методи та заголовки, запити та відповіді, приведено порівняння версій 1.1 та 2.0.

У другому розділі проаналізовано можливості протоколу QUIC, визначено основні недоліки TCP, які зумовили появу нового транспортного протоколу. Досліджено особливості HTTP/3.0, які повинні підвищити ефективність праці WWW.

У третьому розділі Розглянуто основні бібліотеки для мови програмування Python, які дозволяють працювати із HTTP/3.0 та створена програма, яка працює із його використанням.

РОЗДІЛ 1 ПРОТОКОЛ HTTP ТА ЙОГО ОНТОГЕНЕЗ

1.1. Зародження WWW: протокол HTTP (0.9, 1.0)

HTTP (протокол передачі гіпертексту) є основним протоколом Всесвітньої павутини (www). Розроблений Тімом Бернерсом-Лі та його командою в період з 1989 по 1991 роки, протокол HTTP зазнав багатьох змін, які допомогли зберегти його простоту, одночасно сформувавши його гнучкість. HTTP розвинувся з протоколу, призначеного для обміну переважно текстовими файлами в лабораторному середовищі, до сучасного інтернет-лабіринту, який передає зображення та відео у високій роздільній здатності та 3D.

У 1989 році, працюючи в CERN, Тім Бернерс-Лі написав пропозицію створити гіпертекстову систему через Інтернет. Побудований на основі існуючих протоколів TCP та IP, він (WWW) складався з 4 основних складових частин (рис. 1.1):

- текстовий формат для представлення гіпертекстових документів, HyperText Markup Language (HTML);
- протокол для обміну цими документами, а само протокол передачі гіпертексту (HTTP);
- клієнт для відображення (і редагування) цих документів, перший веб-браузер;
- сервер для надання доступу до документа, рання версія httpd.



Рисунок 1.1 Основні частини WWW

Відмітимо, що два із них це браузер (агент) та сервер по суті є програмами які обробляють запити керуючись у своєму спілкуванні протоколом HTTP, та четверта частина описує структуру текстової інформації HTML.

6 серпня 1991 року Тім Бернерс-Лі написав у загальнодоступній групі новин alt.hypertext [4] про працю WWW. Зараз це вважається офіційним початком Всесвітньої павутини як публічного проєкту.

Протокол HTTP, який використовувався на тих ранніх етапах, був дуже простим. Пізніше його назвали HTTP/0.9 і іноді називають однорядковим протоколом. Це підмножина повного протоколу HTTP.

Із запитом не передається інформація профілю клієнта. Майбутні протоколи HTTP повинні бути сумісними з цим протоколом. Цей обмежений протокол був дуже простим і завжди може бути використаним, коли не потрібні можливості повного протоколу, який є зворотно сумісним.

Протокол використовує звичайний стиль протоколу telnet у стилі Інтернету на каналі TCP-IP[3].

HTTP/0.9 був надзвичайно простим: запити складалися з одного рядка й починалися з єдиного можливого методу GET, за яким слідував шлях до ресурсу. Повну URL-адресу не було включено, оскільки протокол, сервер і порт не були потрібні після підключення до сервера (рис. 1.2).

```
GET /firstpage.html
```

Рисунок 1.2 Запит у протоколі http/0.9

Відповідь також була надзвичайно простою: вона складалася лише з самого файлу (рис. 1.3).

```
<html>
  Our first HTML page
</html>
```

Рисунок 1.3 Відповідь у протоколі http/0.9

На відміну від наступних версій, HTTP-заголовків не було. Це означало, що можна було передавати лише файли HTML. Не було жодного статусу чи кодів помилок. У разі виникнення проблеми створювався окремий HTML-файл із описом проблеми для користувача.

Як уже зазначалось вище HTTP/0.9 був дуже обмеженим, але браузерери та сервери швидко зробили його більш універсальним:

- інформація про версії надсилалася в кожному запиті (HTTP/1.0 було додано до рядка GET);
- рядок коду статусу також було надіслано на початку відповіді. це дозволяло самому браузеру розпізнавати успіх або невдачу запиту та відповідним чином адаптувати свою поведінку;
- для передавання метаданих було введено поняття HTTP-заголовків як для запитів, так і для відповідей;
- завдяки заголовку Content-Type можна було передавати документи, відмінні від простих файлів HTML.

Між 1991-1995 роками усі ці нововведення були введені за принципом пробного перегляду. Сервер і браузер додавали функцію та перевіряли, чи зацікавить вони розробників. Проблеми сумісності були поширеними. З метою вирішення цих проблем у листопаді 1996 року було опубліковано інформаційний документ, який описував загальні практики. Він був відомий як RFC 1945 [2] і визначав HTTP/1.0.

В цій специфікації вже була визначена основна термінологія: запит, відповідь, з'єднання, повідомлення, ресурс, агент, проксі, тунель та інші. Описано основний синтаксис (заголовки, кодування, формат дати, основні статус коди та методи).

Процес удосконалення принципів побудови взаємодії сервера та агенту приводив до подальшого покращення та кристалізації у вигляді стандартів, які затверджені IETF.

1.2. Стандартизація протоколу HTTP/1.1

У 2014 році було створено RCF 7230 (Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing) [11], яка визначила синтаксис та семантику повідомлень HTTP, керування з'єднаннями та працю кеша. У 2022 році це документ замінено на три стандарти (рис. 1.4): RFC 9112[10] у якому визнається . синтаксис повідомлень HTTP/1.1, розбір (аналіз) повідомлень, керування з'єднаннями та питання безпеки, RFC 9110[9] описує загальну архітектуру HTTP, встановлює загальну термінологію та визначає аспекти протоколу, спільні для всіх версій основні елементи протоколу, механізми розширення та схеми "http" і "https" Uniform Resource Identifier (URI) та третя частина RFC 9111[8] HTTP-кеші та асоційовані поля заголовка, які керують поведінкою кешу.

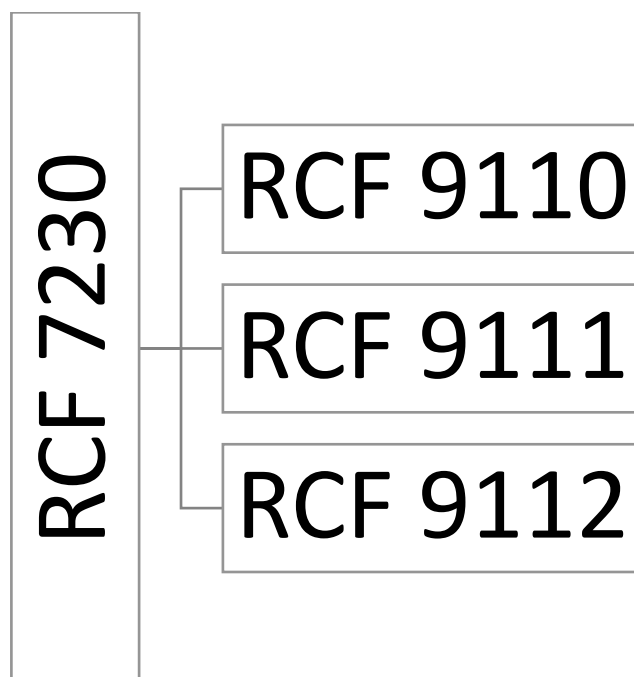


Рисунок 1.4 Стандарти, які описують основні моменти HTTP/1.1

На основі вивчення вище зазначених стандартів та [41, 12] ми можемо описати основні відомості, які потрібно знати, для усвідомленого використання версії 1.1

Веб-вміст живе на веб-серверах. Веб-сервери використовують протокол HTTP, тому їх часто називають HTTP-серверами. Ці HTTP-сервери зберігають

дані та надають їх, а рази запиту (request) клієнтів HTTP. Клієнти надсилають HTTP-запити на сервери, а сервери повертають запитувані дані у HTTP-відповідях (responses), як показано на рис. 1.5. Разом HTTP-клієнти та HTTP-сервери складають основні компоненти Всесвітньої мережі.

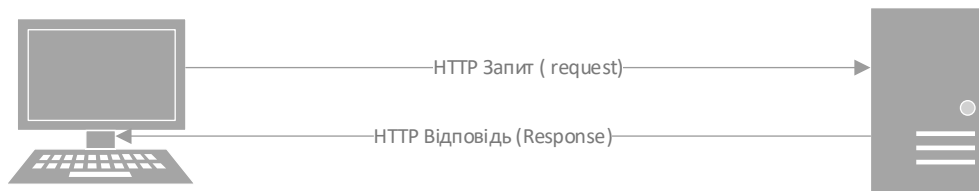


Рисунок 1.5 Взаємодія веб-клієнта та веб-сервера

Веб-сервери розміщують веб-ресурси. Веб-ресурс є джерелом веб-контенту. Найпростішим типом веб-ресурсу є статичний файл у файловій системі веб-сервера. Ці файли можуть містити будь-що: це можуть бути текстові файли, файли HTML, файли Microsoft Word, файли Adobe Acrobat, файли зображень PNG, або будь-який інший формат.

Однак ресурси не обов'язково мають бути статичними файлами. Ресурси також можуть бути програмним забезпеченням, яке створює вміст на вимогу.

Оскільки в Інтернеті розміщено багато тисяч різних типів даних, HTTP ретельно позначає кожен об'єкт, що транспортується через Інтернет, міткою формату даних, що називається типом MIME (багатоцільові розширення Інтернет-пошти). Він спочатку був розроблений для вирішення проблем, що виникають під час переміщення повідомлень між різними системами електронної пошти. MIME настільки добре працював для електронної пошти, що HTTP прийняв його для опису та позначення власного мультимедійного вмісту.

Веб-сервери додають тип MIME до всіх даних об'єктів HTTP. Коли веб-браузер повертає об'єкт із сервера, він переглядає відповідний тип MIME, щоб перевірити, чи знає він, як обробляти об'єкт. Більшість браузерів можуть працювати з сотнями популярних типів об'єктів: відображати файли зображень, аналізувати та формувати файли HTML, відтворювати

аудіофайли через динаміки комп'ютера або запускати зовнішнє програмне забезпечення для обробки спеціальних форматів.

Кожен ресурс веб-сервера має назву, тому клієнти можуть вказати, які ресурси їх цікавлять. Ім'я ресурсу сервера називається уніфікованим ідентифікатором ресурсу або URI. Ідентифікатори URI схожі на поштові адреси в Інтернеті, які однозначно ідентифікують і визначають місцезнаходження інформаційних ресурсів по всьому світу.

Уніфікований покажчик ресурсу (URL) є найпоширенішою формою ідентифікатора ресурсу. URL-адреси описують конкретне розташування ресурсу на певному сервері. Вони розповідають, як саме отримати ресурс із точного фіксованого місця.

Другий різновид URI — це уніфіковане ім'я ресурсу, або URN. URN служить унікальною назвою для певної частини вмісту, незалежно від того, де наразі знаходиться ресурс. Ці незалежні від розташування URN дозволяють ресурсам переміщатися з місця на місце. URN також дозволяють отримувати доступ до ресурсів за допомогою кількох протоколів доступу до мережі, зберігаючи однакові назви. Наприклад DOI.

Розглянемо більш докладно структуру HTTP транзакцій.

HTTP-транзакції не потребують використання всіх заголовків. По суті, можна виконувати деякі HTTP-запити, взагалі не надаючи жодного заголовка.

HTTP-запити мають такі загальні компоненти:

Перший рядок повідомляє клієнту, який метод використовувати, до якої сутності (документа) його застосовувати та яку версію HTTP використовує клієнт. Можливі методи HTTP 1.1: GET, POST, HEAD, PUT, LINK, UNLINK, DELETE, OPTIONS і TRACE. HTTP 1.0 не підтримує метод OPTIONS або TRACE. Не всі методи повинні підтримуватися сервером.

URL-адреса вказує розташування документа, до якого потрібно застосувати метод. Кожен сервер може мати свій власний спосіб перетворення рядка URL-адреси в певну форму придатного для використання ресурсу. Наприклад, URL може представляти документ для передачі клієнту. Або URL-

адреса може фактично зіставлятися з програмою, вихідні дані якої надсилаються клієнту.

Нарешті, останній запис у першому рядку вказує версію HTTP, яку використовує клієнт.

Загальні заголовки повідомлень – це додаткові заголовки, які використовуються як у запиті клієнта, так і у відповіді сервера. Вони вказують загальну інформацію, таку як поточний час або шлях через мережу, яку використовують клієнт і сервер.

Заголовки запитів повідомляють серверу більше інформації про клієнта. Клієнт може ідентифікувати себе та користувача на сервері та вказати бажані формати документів, які він хотів би бачити на сервері.

Заголовки сутності використовуються, коли сутність (документ) збираються надіслати. Вони вказують інформацію про сутність (метайнформацію), таку як схеми кодування, довжина, тип і походження.

На рис. 1.6 представлено запит (request).

```
GET / HTTP/1.1
Host: luguniv.edu.ua
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:108.0) Gecko/20100101 Firefox/108.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif
,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: qtrans_front_language=ua;
PHPSESSID=ilqlnoig5eisd70sqlpl2e8st1
Upgrade-Insecure-Requests: 1
DNT: 1
Sec-GPC: 1
```

Рисунок 1.6 Заголовок запиту HTTP/1.1 на сайт ЛНУ імені Тараса Шевченка

У відповіді сервера загальний заголовок і заголовки сутності такі самі, як ті, що використовуються в запиті клієнта. Тіло сутності схоже на те, що використовується в запиті клієнта, за винятком того, що воно використовується як відповідь.

Перша частина першого рядка вказує на версію HTTP, яку використовує сервер. Сервер докладе всіх зусиль, щоб відповідати найбільш сумісній версії HTTP, яку використовує клієнт. Код статусу вказує на результат запиту, а пояснення причини є зрозумілим для людини описом коду статусу.

Заголовок відповіді повідомляє клієнту про конфігурацію сервера. Він інформує клієнта про підтримувані методи, запитує авторизацію або повідомляє клієнту повторити спробу пізніше.

```
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Mon, 09 Jan 2023 20:58:03 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Set-Cookie: qtrans_front_language=ua; expires=Tue, 09-Jan-
2024 20:58:02 GMT; Max-Age=31536000; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-
check=0, pre-check=0
Pragma: no-cache
Link: <http://luguniv.edu.ua/index.php?rest_route=/>;
rel="https://api.w.org/"
Link: <https://wp.me/P5gvnJ-1WW>; rel=shortlink
Content-Encoding: gzip
```

Рисунок 1.7 Заголовок відповіді HTTP/1.1 з сайту ЛНУ імені Тараса Шевченка

Метод клієнтського запиту — це команда або запит, який веб-клієнт надсилає серверу. Ми можемо розглядати метод як декларацію намірів клієнта. Звичайно, є винятки, але ось деякі узагальнення:

Запит GET вважають таким, що означає, що користувач просто має намір отримати ресурс на сервері. Цей ресурс може бути вмістом статичного файлу або викликати програму, яка генерує дані.

Запит HEAD означає, що просто потрібна деяка інформація про документ, але не потрібен сам документ.

Запит POST повідомляє, що користувач надає певну власну інформацію (зазвичай використовується для заповнення форм). Зазвичай це певним чином змінює стан сервера. Наприклад, він може створити запис у базі даних.

PUT використовується для надання нового або замінного документа для зберігання на сервері.

DELETE використовується для видалення документа на сервері.

TRACE вимагає, щоб проксі-сервери декларували себе в заголовках, щоб клієнт міг дізнатися шлях, яким пройшов документ (і таким чином визначити, де щось могло бути спотворено або втрачено). Це використовується для налагодження протоколу.

OPTIONS використовується, коли клієнт хоче знати, які інші методи можна використати для цього документа (або для сервера в цілому).

CONNECT використовується, коли клієнту потрібно спілкуватися з сервером HTTPS через проксі-сервер.

Більш докладний розгляд цих команд виходить за межі цього дослідження, але зауважимо, що у HTTP/2.0 HTTP/3.0 зовнішні команди залишаються тими самими, але змінюється їх спосіб реалізації, і про це більш докладно, ми довідаємось пізніше.

Початковий рядок відповіді сервера вказує на версію HTTP, тризначний код статусу та зрозумілий опис результату. Коди стану згруповані таким чином:

Таблиця 1.1 Коди відповіді HTTP

Коди	Значення
100-199	Інформаційні
200-299	Успішний запит

300-399	Перенаправлення
400-499	Помилка при обробці запита
500-599	Помилка сервера

HTTP визначає лише кілька конкретних кодів у кожному діапазоні, хоча з розвитком HTTP ці діапазони стануть більш заповненими.

Якщо клієнт отримує код відповіді, який він не розпізнає, він повинен зрозуміти його основне значення з його числового діапазону. У доданку А представлено найбільш вживані коди.

Далі повернемося ще раз до заголовків.

Існує чотири типи заголовків HTTP [41]:

Загальні заголовки вказують загальну інформацію, таку як дата або те, чи потрібно підтримувати з'єднання. Вони використовуються як клієнтами, так і серверами.

Заголовки запитів використовуються лише для запитів клієнтів. Вони передають конфігурацію клієнта та бажаний формат документа на сервер.

Заголовки відповідей використовуються лише у відповідях сервера. Вони описують конфігурацію сервера та інформацію про запитану URL-адресу.

Заголовки сутностей описують формат документа даних, що надсилаються між клієнтом і сервером. Хоча заголовки сутностей найчастіше використовуються сервером під час повернення запитуваного документа, вони також використовуються клієнтами під час використання методів POST або PUT.

Заголовки з усіх чотирьох категорій можуть бути вказані в будь-якому порядку. Імена заголовків нечутливі до регістру, тому заголовок Content-type також часто записується як Content-Type.

Розглянемо деякі із заголовків.

Заголовок Cache-control визначає бажану поведінку системи кешування, яка використовується в проксі-серверах. Наприклад:

```
Cache-control: no-cache
```

І клієнти, і сервери можуть використовувати заголовок Cache-control, щоб указати параметри для кешу або запитати певні види документів із кешу. Директиви кешування вказані у списку, розділеному комами.

Вказує потрібні параметри для цього з'єднання, але не для подальших з'єднань через проксі. Наприклад:

```
Connection: close
```

Опція закриття означає, що або клієнт, або сервер бажають завершити з'єднання (тобто це остання транзакція). Параметр Keep-alive означає, що клієнт бажає залишати з'єднання відкритим. За замовчуванням HTTP 1.1 використовує постійні з'єднання, тобто з'єднання не закривається автоматично після транзакції.

Заголовок Pragma визначає директиви для систем проксі та шлюзів. Оскільки між клієнтом і сервером може існувати багато проксі-систем, заголовки Pragma повинні проходити через кожен проксі. Коли заголовок Pragma досягає сервера, програмне забезпечення сервера може проігнорувати його.

Заголовок Transfer-Encoding вказує, що повідомлення закодовано. Це не те саме, що кодування вмісту (заголовок тіла сутності, обговорюватиметься пізніше), оскільки кодування передачі є властивістю повідомлення, а не тіла сутності. Наприклад:

```
Transfer-Encoding: chunked
```

У специфікації HTTP 1.1 єдиним підтримуваним методом кодування є фрагментація (chunked).

Кодування фрагментів передачі кодує повідомлення як серію фрагментів, за якими йдуть заголовки об'єктів.

Дані заголовка клієнта передають серверу конфігурацію клієнта та бажані формати документів. Заголовки запитів використовуються в повідомленні клієнта для надання інформації про клієнта.

Вони визначають типи носіїв, які клієнт бажає приймати. Наприклад:

```
Accept: text/*, image/gif
```

Кілька типів носіїв можна вказати через кому.

Визначаються мови, яким надає перевагу клієнт. Якщо клієнт хоче вказати параметри для певної мови, це робиться в заголовку Accept-Language. Якщо сервер містить один і той самий документ кількома мовами, він надішле документ мовою, яку вподобає клієнт, якщо вона доступна. Наприклад:

```
Accept-language: en
```

Можна вказати кілька мов, розділивши їх комами.

Розгляд усіх можливих заголовків виходить за рамки дослідження їх можливо вивчити в [9].

Завершуємо цей підрозділ стислим оглядом архітектури Web.

Проксі

Давайте почнемо з огляду проксі-серверів HTTP, важливих будівельних блоків для веб-безпеки, інтеграції програм і оптимізації продуктивності.

Як показано на рисунку 1.8, проксі знаходиться між клієнтом і сервером, отримуючи всі HTTP-запити клієнта та ретранслюючи запити на сервер (можливо, після зміни запитів). Ці програми діють як проксі для користувача, звертаючись до сервера від імені користувача.

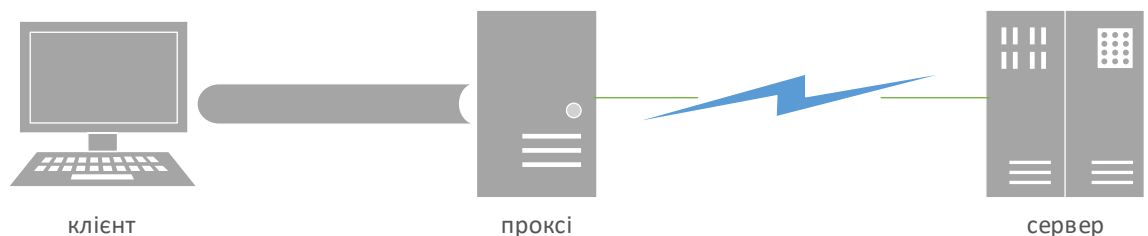


Рисунок 1.8 Візуалізація праці проксі сервера

Проксі-сервери часто використовуються для безпеки, діючи як надійні посередники, через які протікає весь веб-трафік. Проксі також можуть фільтрувати запити та відповіді; наприклад, для виявлення програмних вірусів у корпоративних завантаженнях або для фільтрації вмісту для дорослих від учнів школи.

Веб-кеш або проксі-сервер кешування — це особливий тип проксі-сервера HTTP, який зберігає копії популярних документів, які проходять через

проксі. Наступний клієнт, який запитує той самий документ, може бути обслугований з особистої копії кеша, ще одна назва CDN.

Шлюзи — це спеціальні сервери, які виконують роль посередників для інших серверів (рис. 1.9). Вони часто використовуються для перетворення трафіку HTTP на інший протокол. Шлюз завжди отримує запити так, ніби він був вихідним сервером для ресурсу. Клієнт може не знати, що він спілкується зі шлюзом.

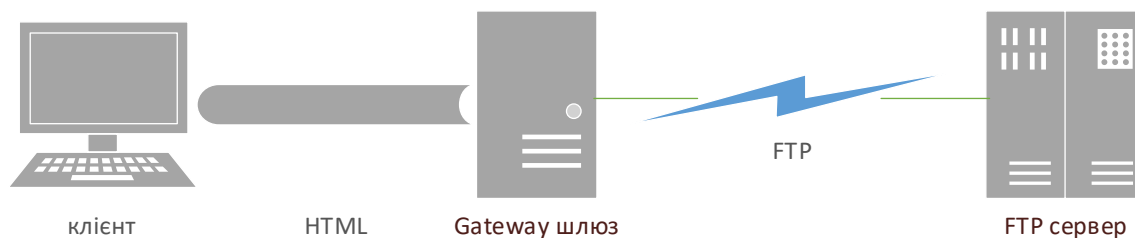


Рисунок 1.9 Шлюз HTTP/FTP

Тунелі — це програми HTTP, які після налаштування сліпо передають необроблені дані між двома з'єднаннями. HTTP-тунелі часто використовуються для передачі не-HTTP-даних через одне або кілька HTTP-з'єднань без перегляду даних.

Агенти користувачі (user agents) (або просто агенти) — це клієнтські програми, які роблять HTTP-запити від імені користувача. Будь-яка програма, яка надсилає веб-запити, є агентом HTTP. Поки що ми говорили лише про один тип агента HTTP: веб-браузери (Chrome, Edge, Firefox). Але є багато інших типів агентів користувача (curl).

1.3. Підвищення ефективності у версії HTTP/2.0

Коли HTTP був створений він сприймався як досить простий і зрозумілий протокол, але час довів хибність такої думки.

Природа протоколу HTTP 1.1, що містить безліч дрібних деталей і параметрів, доступних для пізніших розширень, призвела до розвитку екосистеми програмного забезпечення, де майже жодна реалізація не реалізує все — і навіть неможливо точно сказати, що таке «все». Це призвело до ситуації,

коли функції, які спочатку мало використовувалися, мали дуже мало реалізацій, а ті, які реалізували ці функції, потім дуже мало використовували їх [35].

HTTP 1.1 важко повністю використати всю потужність і продуктивність TCP. HTTP-клієнти та браузерери мають бути дуже творчими, щоб знайти рішення, які зменшують час завантаження сторінки.

Дивлячись на тенденції для деяких із найпопулярніших веб-сайтів сьогодні та на те, що потрібно для завантаження їхніх головних сторінок, виявляється чітка закономірність. З роками обсяг даних, які потрібно отримати, поступово зріс до 1,9 МБ. У цьому контексті важливішим є те, що для відображення кожної сторінки в середньому потрібно понад 100 окремих ресурсів.

HTTP 1.1 дуже чутливий до затримки, частково через те, що конвеєрна передача HTTP все ще повна проблем, щоб залишатися вимкненою для великого відсотка користувачів.

Ці та інші обставини сприяли виникненню в 2007 році HTTPbis – робочої групи для створення нової версії протоколу HTTP, яка у своїй роботі відштовхується від SPDY – це протокол, розроблений і очолюваний Google. Вони, звичайно, розробили його відкрито та запросили всіх до участі, але було очевидно, що вони виграли, контролюючи як популярну реалізацію браузера, так і значну кількість серверів із добре використовуваними службами. Тому для відкритої конкуренції потрібно були зробити певні зміни.

Давайте зануримося в специфіку протоколу та концепції, які складають HTTP/2.

HTTP/2 є бінарним, що значно полегшує фреймування. Визначити початок і кінець фреймів — одна зі справді складних речей у HTTP 1.1 і, власне, у текстових протоколах загалом (див. Рисунок 1.10). Завдяки відходу від необов'язкових пробілів і різних способів написання того самого, реалізація стає простішою.

Крім того, це значно полегшує відокремлення фактичних частин протоколу від фрейму, який у HTTP/1 змішаний до плутанини [30].

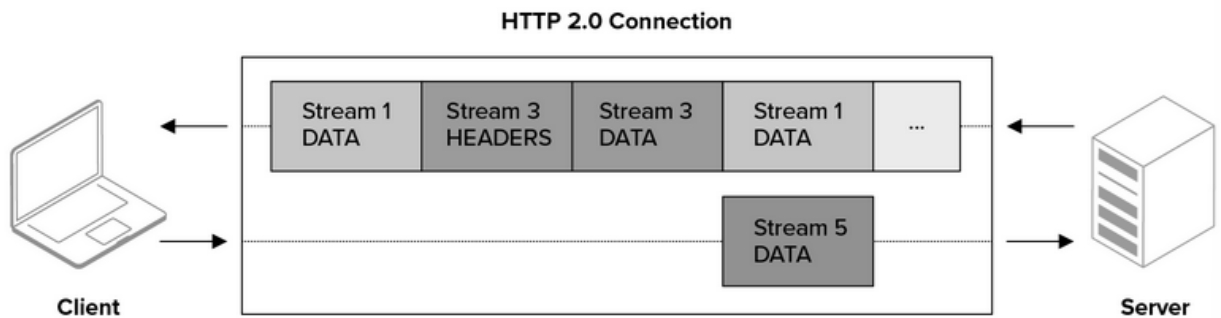


Рисунок 1.10 Бінарний формат фреймів HTTP/2.0 [40]

За допомогою HTTP/2 браузері спілкуються з серверами через двонаправлені потоки з кількома повідомленнями, кожне з яких складається з кількох кадрів і при цьому однакові заголовки не передаються дивиться Рисунок 1.11. Cloudflare повідомляє про значну економію пропускну здатності лише завдяки HPACK: 76% стиснення для вхідних заголовків; Зниження загального вхідного трафіку на 53%; зниження загального вихідного трафіку на 1,4–15%.

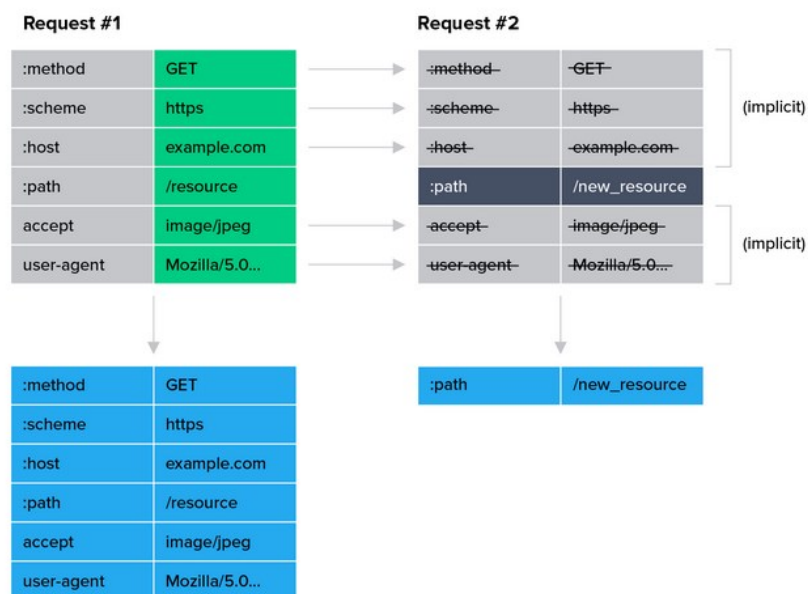


Рисунок 1.11 Візуалізація стиснення заголовків в HTTP/2.0 [40]

Той факт, що протокол підтримує стиснення та часто працює через TLS, також зменшує цінність тексту, оскільки зовнішній спостерігач все одно не побачить текст при передачі. Просто потрібно звикнути до ідеї використання

чогось на зразок інспектора Wireshark, щоб з'ясувати, що саме відбувається на рівні протоколу в HTTP/2.

HTTP/2 надсилає двійкові кадри. Існують різні типи кадрів, які можна надсилати, і всі вони мають однакові налаштування: довжина, тип, прапори, ідентифікатор потоку та корисне навантаження кадру.

Ідентифікатор потоку, пов'язує кожен кадр, надісланий через HTTP/2, із «потокom». Потік — це незалежна двонаправлена послідовність кадрів, якими обмінюються клієнт і сервер у рамках з'єднання HTTP/2.

Одне підключення HTTP/2 може містити кілька одночасно відкритих потоків, причому будь-яка кінцева точка чергує кадри з кількох потоків. Потоки можна встановлювати та використовувати в односторонньому порядку або спільно використовувати клієнтом або сервером, і вони можуть бути закриті будь-якою кінцевою точкою. Важливим є порядок, у якому кадри надсилаються в межах потоку. Одержувачі обробляють кадри в порядку їх отримання.

Кожен потік також має пріоритет (також відомий як «вага»), який використовується, щоб повідомити одноранговому вузлу, які потоки вважати найважливішими, у випадку, якщо існують обмеження ресурсів, які змушують сервер вибирати, які потоки надсилати першими [27].

Пріоритетні ваги та залежності можна динамічно змінювати під час виконання, що має дозволити браузерам переконатися, що коли користувачі прокручують сторінку, повну зображень, браузер може вказувати, які зображення є найважливішими, або, якщо користувач перемикає вкладки, він може визначати пріоритети у новому набору потоків, які стають у центрі уваги після зміни увагу користувача.

HTTP є протоколом без стану. Коротше кажучи, це означає, що кожен запит повинен містити стільки деталей, скільки потрібно серверу для обслуговування цього запиту, без того, щоб сервер мав зберігати багато інформації та метаданих із попередніх запитів. Оскільки HTTP/2 не змінює цю парадигму, він має працювати так само.

Це робить HTTP повторюваним. Коли клієнт запитує багато ресурсів з одного сервера, як-от зображення з веб-сторінки, буде велика серія запитів, які виглядають майже однаково. Серія майже ідентичних речей вимагає стиснення.

Хоча кількість об'єктів на веб-сторінку зростає (як згадувалося раніше), використання файлів cookie та розмір запитів також з часом зростає. Файли cookie також потрібно включати в усі запити, часто одні й ті самі в кількох запитах [24].

Розміри запитів HTTP 1.1 фактично стали настільки великими, що іноді вони перевищують початкове вікно TCP, через що вони надсилаються дуже повільно, оскільки їм потрібен повний зворотний зв'язок, щоб отримати ACK від сервера до того, як буде отримано повний запит. Це ще один аргумент на користь компресії.

Одним із недоліків HTTP 1.1 є те, що коли HTTP-повідомлення було надіслано з Content-Length певного розміру, користувач не може просто зупинити його. Звичайно, він часто може роз'єднати TCP-з'єднання, але це відбувається за рахунок необхідності знову обговорювати нове рукоштовування TCP.

Також існує функція, відома як «кеш-пам'ять». Ідея полягає в тому, що якщо клієнт запитує ресурс X, сервер може знати, що клієнт, ймовірно, також захоче отримати ресурс Z, і надсилає його клієнту без запиту. Він допомагає клієнту, поміщаючи Z в його кеш, щоб він був там, коли користувач цього забажає.

HTTP/2 зменшує кількість необхідних мережевих обходів і повністю усуває дилему головного блокування лінії завдяки мультиплексуванню та швидкому відкиданню небажаних потоків [14].

Це дозволяє велику кількість паралельних потоків, які проходять навіть через найбільш фрагментовані сайти.

З правильним використанням пріоритетів у потоках набагато більше шансів, що клієнти дійсно отримають важливі дані перед менш важливими

даними. Узявши все це разом, можливо сказати, що існують дуже високі шанси, що це призведе до швидшого завантаження сторінок і більшої чутливості веб-сайтів. Коротко кажучи: користувачі отримають кращий веб-досвід.

Щодо використання HTTP/2.0 станом на кінець із боку серверів згідно [31] 39,9% вебсайтів використовує HTTP/2.0 дивиться Рисунок 1.12, що стосується браузерів, то відповідно до [15], 96,65% клієнтів можуть використовувати його.

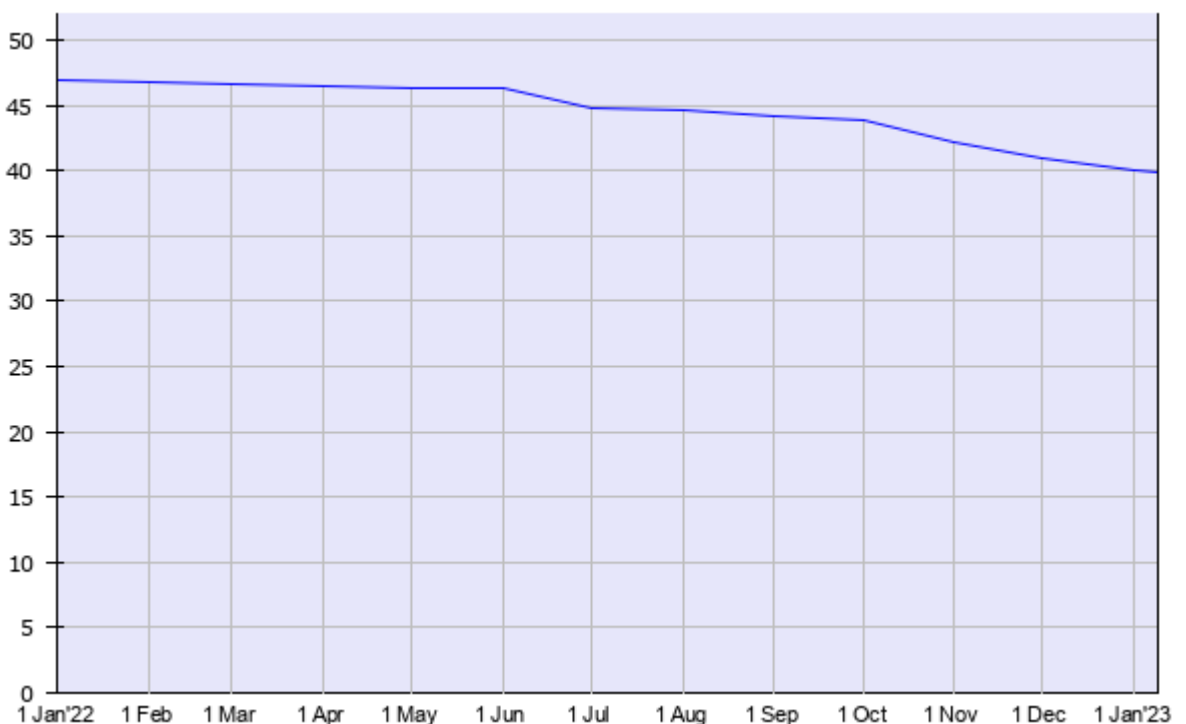


Рисунок 1.12 Використання HTTP/2.0 в 2022 році [31]

Щодо швидкості то, наприклад, згідно дослідження [7] HTTP/2 не завжди перевершує HTTP/1.1 за продуктивністю. Натомість є оптимальний діапазон для кожного типу протоколу. Ці діапазони свідчать про ефективність HTTP/2 гірше на сторінках, більшого за середнього розміру.

Інші дослідники зазначають, що по параметру «розмір заголовків запиту клієнта та відповіді сервера»: контрольні тести HTTP/2 демонструють, що використання механізму стиснення заголовків значно зменшує розмір заголовка. По параметру «розмір повідомлення відповіді сервера» відповідь сервера HTTP/2 була більшою за розміром, вона забезпечує надійніше

шифрування для покращеної безпеки. Кількість використаних TCP-з'єднань: HTTP/2 використовує менше мережевих ресурсів, обробляючи декілька одночасних запитів (мультиплексування), а отже, зменшують затримку.

Зробимо порівняльну характеристику HTTP/1.1 та HTTP/2.0 у таблиці 1.2

Таблиця 1.2 Порівняльна характеристика HTTP/1.1 та HTTP/2.0

HTTP1.x	HTTP2
SSL не потрібен, але рекомендований	SSL не потрібен, але рекомендований
Повільне шифрування.	Ще швидше шифрування.
Один запит клієнт-сервер на TCP-з'єднання	Багатохостове мультиплексування. Виникає на кількох хостах одночасно
Без стиснення заголовка.	Стиснення заголовка за допомогою вдосконалених алгоритмів, які покращують продуктивність і безпеку.
Пріоритезації потоку немає.	Використано вдосконалені механізми пріоритезації потоку.

1.4. Висновки до розділу 1

HTTP — це протокол Всесвітньої павутини. З кожною веб-транзакцією запускається HTTP. HTTP стоїть за кожним запитом на веб-документ або графіку, кожним клацанням гіпертекстового посилання та кожним надсиланням форми. Мережа — це розповсюдження інформації через Інтернет, і для цього використовується протокол HTTP.

HTTP є корисним, оскільки він забезпечує стандартизований спосіб для комп'ютерів спілкуватися один з одним. HTTP визначає, як клієнти запитують дані та як сервери відповідають на ці запити.

З роками веб-сторінки ставали складнішими. Деякі з них навіть були самостійними програмами. Було відображено більше візуальних засобів масової інформації, а також збільшено обсяг і розмір JS-сценаріїв, що додають

інтерактивність. Набагато більше даних було передано через значно більше запитів HTTP, і це створило більшу складність і накладні витрати на з'єднання HTTP/1.1. Щоб вирішити це, на початку 2010-х років Google запровадив експериментальний протокол SPDY. Цей альтернативний спосіб обміну даними між клієнтом і сервером зацікавив розробників, які працюють як над браузерами, так і над серверами. SPDY визначив підвищення швидкості реагування та вирішив проблему передачі дублікатів даних, послуживши основою для протоколу HTTP/2.

Протокол HTTP/2 відрізняється від HTTP/1.1 кількома моментами:

Це бінарний протокол, а не текстовий. Його неможливо прочитати та створити вручну. Незважаючи на цю перешкоду, це дозволяє реалізувати вдосконалені методи оптимізації.

Це мультиплексний протокол. Паралельні запити можна надсилати через одне з'єднання, усуваючи обмеження протоколу HTTP/1.x.

Він стискає заголовки. Оскільки вони часто подібні в наборі запитів, це усуває дублювання та накладні витрати на передані дані.

Він дозволяє серверу заповнювати дані в кеш-пам'яті клієнта за допомогою механізму, який називається сервером push.

На кінець 2022 приблизно 40% сайтів використовували HTTP/2.0 і ця частка поступово зменшується [31].

РОЗДІЛ 2 ОГЛЯД ОСНОВНИХ ОСОБЛИВОСТЕЙ HTTP/3.0

2.1. Огляд протоколу QUIC

Як ми з'ясували HTTP/2 було стандартизовано в 2015, і виникає питання, чому виникла потреба у ще одній специфікації HTTP/3. Для відповіді на це питання нам потрібно подивитися на стек протоколів, який використовується при праці інтернет.

HTTP/2 працює на основі TCP. TCP є основним протоколом, який забезпечує важливі послуги, такі як надійність і порядок доставки іншим протоколам, таким як HTTP. Це також одна з причин, чому ми можемо продовжувати використовувати Інтернет з багатьма одночасними користувачами, оскільки цей протокол розумно обмежує використання смуги пропускання кожного користувача відповідно до його частки.

Це «накладення» протоколів один на інший зроблено для того, щоб дозволити легке повторне використання їх функцій. Протоколи вищого рівня (такі як HTTP) не потребують повторної реалізації складних функцій (таких як шифрування), оскільки протоколи нижчого рівня (такі як TLS) уже роблять це за них. Як інший приклад, більшість додатків в Інтернеті внутрішньо використовують TCP, щоб гарантувати, що всі їхні дані передаються в повному обсязі. З цієї причини TCP є одним із найбільш широко використовуваних і розгорнутих протоколів в Інтернеті [37].

TCP був наріжним каменем Інтернету протягом десятиліть, але він почав демонструвати свій вік наприкінці 2000-х років. Його передбачувана заміна, новий транспортний протокол під назвою QUIC, досить відрізняється від TCP декількома ключовими моментами, тому запустити HTTP/2 безпосередньо поверх нього буде дуже важко. Таким чином, сам HTTP/3 є відносно невеликою адаптацією HTTP/2, щоб зробити його сумісним з новим протоколом QUIC, який включає більшість нових функцій. Стек протоколів представлено на Рисунок 2.1 [33]

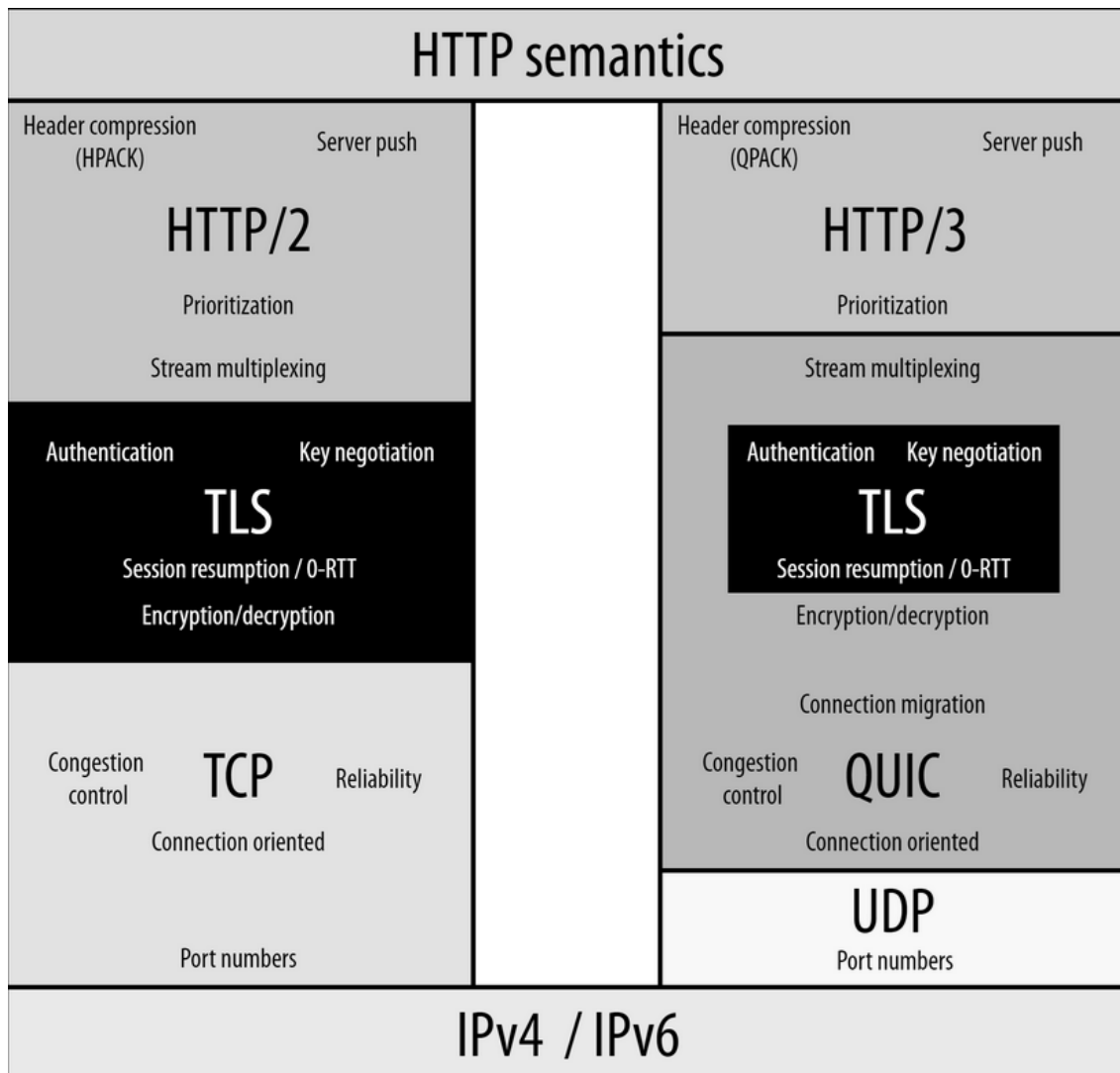


Рисунок 2.1 Порівняння стеків HTTP/2 та HTTP/3[33]

QUIC потрібен тому, що TCP, який існує з перших днів Інтернету, не створювався з оглядом на максимальну ефективність. Наприклад, TCP вимагає «рукоштовування» для встановлення нового з'єднання. Це робиться для того, щоб переконатися, що клієнт і сервер існують і що вони бажають і можуть обмінюватися даними. Однак для цього також потрібен повний обхід мережі, перш ніж щось ще можна буде зробити на з'єднанні. Якщо клієнт і сервер географічно віддалені, час кожного зворотного зв'язку (RTT) може тривати понад 100 мілісекунд, спричиняючи помітні затримки.

Як другий приклад, TCP бачить усі дані, які він транспортує, як один «файл» або потік байтів, навіть якщо ми фактично використовуємо його для передачі кількох файлів одночасно (наприклад, під час завантаження веб-сторінки, що складається з багато ресурсів). На практиці це означає, що якщо

TCP-пакети, що містять дані одного файлу, будуть втрачені, то всі інші файли також будуть відкладені, доки ці пакети не будуть відновлені.

Це називається блокуванням на початку лінії (HoL). Хоча на практиці ці неефективності цілком можна контролювати вона помітно впливає на протоколи вищого рівня, такі як HTTP.

Були спроби оновити TCP, щоб покращити деякі з цих проблем і навіть запровадити нові функції продуктивності. Наприклад, TCP Fast Open [1] позбавляє від накладних витрат рукописання, дозволяючи протоколам вищого рівня надсилати дані з самого початку. Інша спроба називається MultiPath TCP [39]. Тут ідея полягає в тому, що ваш мобільний телефон зазвичай має як Wi-Fi, так і стільниковий зв'язок (4G) і з'єднання не переривається при переході із одної мережі до іншої.

Реалізувати ці розширення TCP не дуже складно. Однак фактично розгорнути їх у масштабі Інтернету надзвичайно складно. Оскільки TCP дуже популярний, майже кожен підключений пристрій має власну реалізацію протоколу. Якщо ці реалізації занадто старі, не мають оновлень або мають помилки, розширення практично не можна буде використовувати. Іншими словами, всі реалізації повинні знати про розширення, щоб воно було корисним.

Це не було б великою проблемою, якби ми говорили лише про пристрої кінцевого користувача (наприклад, комп'ютер або веб-сервер), оскільки їх можна відносно легко оновити вручну. Однак багато інших пристроїв знаходяться між клієнтом і сервером, які також мають свій власний код TCP (приклади включають брандмауери, балансувальники навантаження, маршрутизатори, сервери кешування, проксі-сервери тощо).

Ці проміжні блоки часто важче оновлювати, а іноді вони суворіші щодо того, що вони приймають. Наприклад, якщо пристрій є брандмауером, він може бути налаштований на блокування всього трафіку, що містить невідомі розширення(формати). На практиці виявляється, що величезна кількість

активних проміжних блоків робить певні припущення щодо TCP, які більше не відповідають новим розширенням.

Отже, може знадобитися від років до десяти років, перш ніж буде оновлено достатню кількість реалізацій TCP (проміжного блоку), щоб фактично використовувати розширення у великих масштабах. Можна сказати, що розвинути TCP стало практично неможливо.

У результаті стало зрозуміло, що для вирішення цих проблем нам знадобиться замінити протокол TCP, а не пряме оновлення. Однак через величезну складність функцій TCP та їх різноманітних реалізацій створення чогось нового, але кращого з нуля було б монументальною справою. Тому на початку 2010-х років цю роботу було вирішено відкласти.

QUIC працює на основі ще одного протоколу, який називається протоколом дейтаграм користувача (UDP). В ідеалі QUIC мав би бути повністю незалежним новим транспортним протоколом, який працює безпосередньо поверх IP у стеку протоколів, показаному на Рисунок 2.1

Багато джерел стверджують, що HTTP/3 побудований поверх UDP через продуктивність. Вони кажуть, що HTTP/3 швидший, оскільки, як і UDP, він не встановлює з'єднання та не чекає повторної передачі пакетів. Ці твердження неправильні. Як ми вже говорили вище, UDP використовується QUIC і, отже, HTTP/3 головним чином тому, що це полегшить їх розгортання, оскільки він уже відомий і реалізований (майже) усіма пристроями в Інтернеті. [22]

Крім UDP, QUIC по суті заново реалізує майже всі функції, які роблять TCP таким потужним і популярним (але дещо повільнішим) протоколом. QUIC абсолютно надійний, використовує підтвердження для отриманих пакетів і повторних передач, щоб переконатися, що втрачені все ще надходять. QUIC також усе ще встановлює з'єднання та має дуже складне рукописання.

Нарешті, QUIC також використовує так звані механізми керування потоком і перевантаженням, які запобігають перевантаженню відправником мережі чи одержувача, але також роблять TCP повільнішим, ніж UDP. Ключовим є те, що QUIC реалізує ці функції розумнішим і продуктивнішим

способом, ніж TCP. Він поєднує десятиліття досвіду розгортання та найкращі практики TCP з деякими основними новими функціями.

У QUIC є кілька нових конкретних функцій і можливостей (0-RTT дані, міграція підключення, більша стійкість до втрати пакетів і повільних мереж).

Однак усі ці нові речі в основному зводяться до чотирьох основних змін:

QUIC глибоко інтегрується з TLS.

QUIC підтримує кілька незалежних потоків байтів.

QUIC використовує ідентифікатори підключення.

QUIC використовує фрейми.

Далі ми розглядаємо кожен із цих моментів більш докладно.

TLS (протокол безпеки транспортного рівня) відповідає за захист і шифрування даних, що надсилаються через Інтернет. Коли ми використовуємо HTTPS, відкриті дані HTTP спочатку шифруються за допомогою TLS, а потім транспортуються за допомогою TCP.

На початку існування Інтернету шифрування трафіку було досить дорогим з точки зору обробки. Крім того, це також не вважалося необхідним для всіх випадків використання. Історично склалося так, що TLS був повністю окремим протоколом, який за бажанням можна використовувати поверх TCP. Ось чому ми розрізняємо HTTP (без TLS) і HTTPS (з TLS) [34].

З часом наше ставлення до безпеки в Інтернеті, звісно, змінилося на «захищено за замовчуванням». Таким чином, хоча HTTP/2 теоретично може працювати безпосередньо через TCP без TLS, жоден (популярний) веб-браузер насправді не підтримує цей режим. У певному сенсі постачальники браузерів пішли на свідомий компроміс для більшої безпеки за рахунок продуктивності.

Враховуючи цю чітку еволюцію до постійного використання TLS (особливо для веб-трафіку), не дивно, що дизайнери QUIC вирішили вивести цю тенденцію на новий рівень. Замість того, щоб просто не визначати режим відкритого тексту для HTTP/3, вони вирішили глибоко впровадити шифрування в сам QUIC. У той час як перші версії QUIC для Google використовували спеціальні налаштування для цього, стандартизований QUIC

використовує безпосередньо сам існуючий TLS 1.3 (візуалізацію можливо побачити на Рисунок 2.2). Для цього він ніби порушує типове чітке розділення між протоколами в стеку протоколів [25]

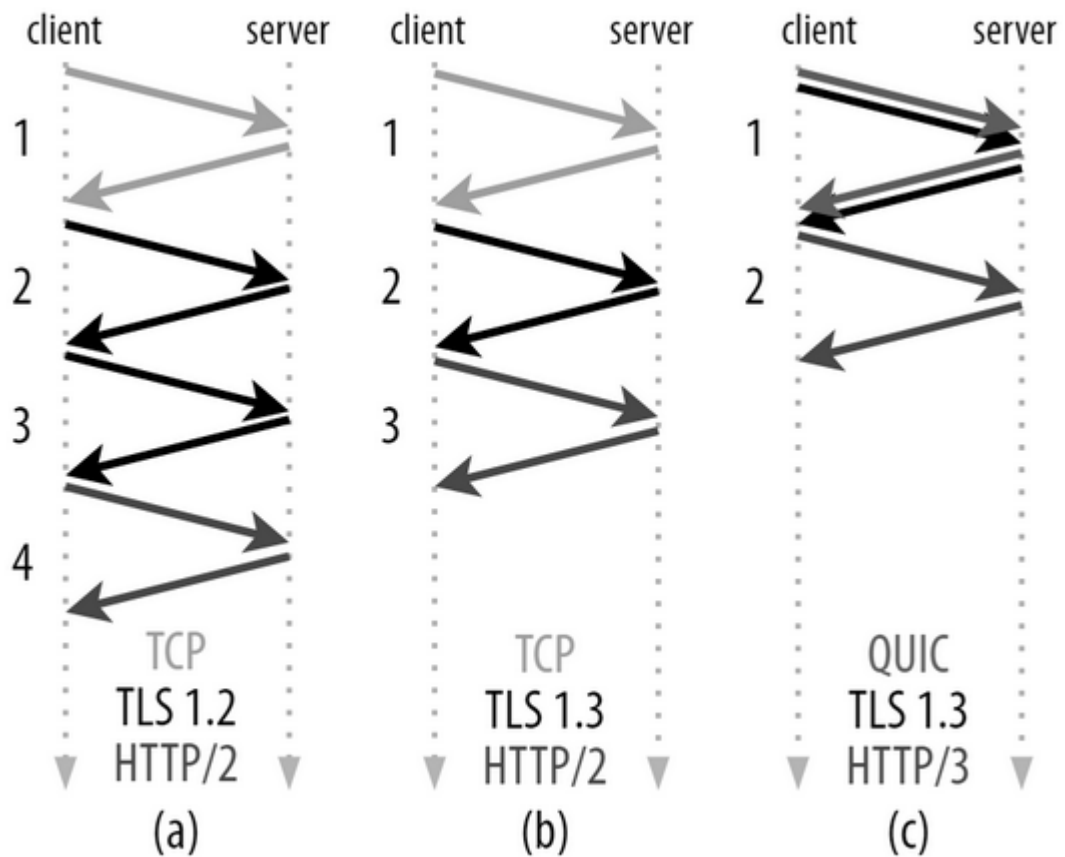


Рисунок 2.2 Тривалість рукоштовування TLS, TCP і QUIC

Хоча TLS 1.3 все ще може працювати незалежно поверх TCP, замість цього QUIC ніби інкапсулює TLS 1.3. Іншими словами, немає можливості використовувати QUIC без TLS; QUIC (і побудований на ньому HTTP/3) завжди повністю зашифрований. Крім того, QUIC також шифрує майже всі поля заголовків своїх пакетів; інформація транспортного рівня (така як номери пакетів, які ніколи не шифруються для TCP) більше не читається посередниками в QUIC (навіть деякі прапори заголовків пакетів зашифровані) дивиться на Рисунок 2.3.

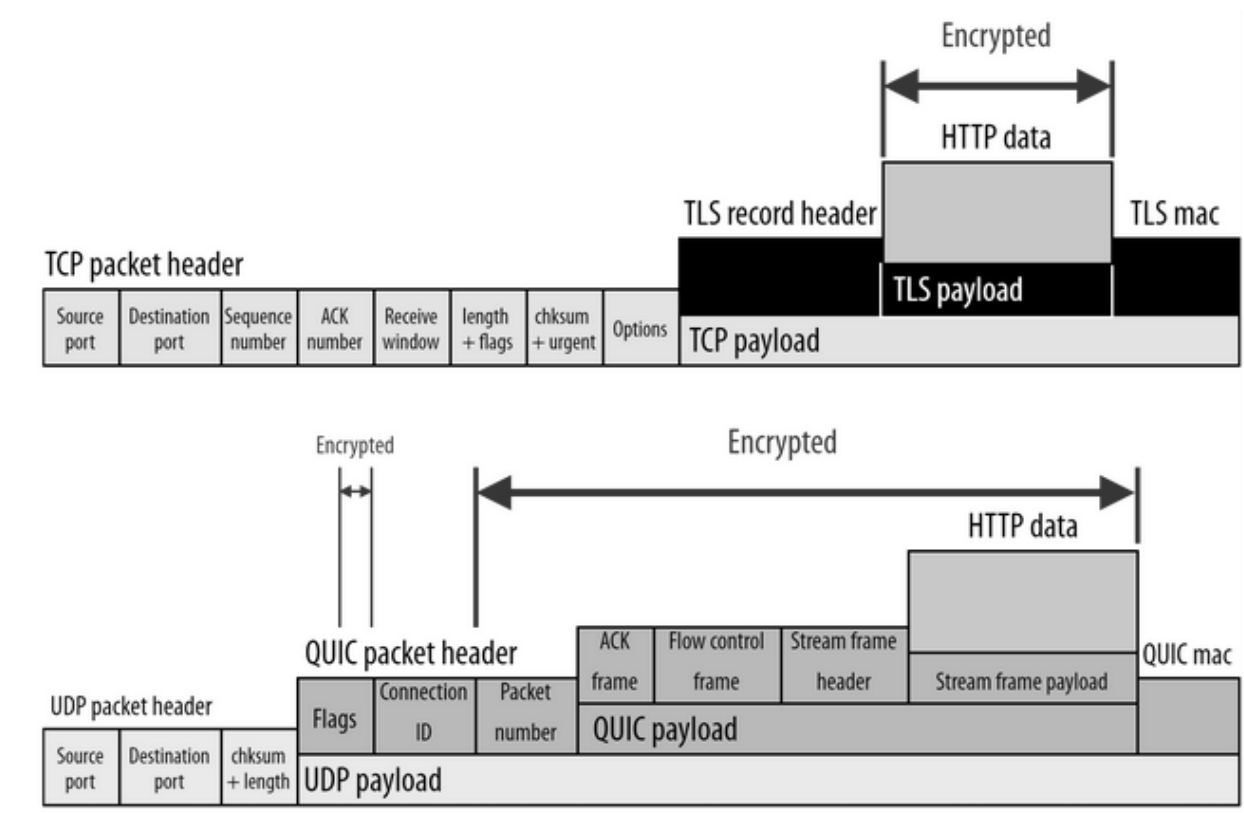


Рисунок 2.3 Шифрування в TCP та UDP

Для всього цього QUIC спочатку використовує зв'язок TLS 1.3 більш-менш так, як це було б з TCP для встановлення математичних параметрів шифрування. Однак після цього QUIC бере на себе контроль і шифрує пакети самостійно, тоді як з TLS-over-TCP TLS виконує власне шифрування. Ця, здавалося б, невелика різниця являє собою фундаментальну концептуальну зміну в напрямку постійного шифрування, яке застосовується на нижчих рівнях протоколу.

Усе це приводить до наступних переваг. QUIC більш безпечний для своїх користувачів. Немає способу запустити відкритий текст QUIC, тому у зломисників і підслуховувачів також менше можливостей для прослуховування. Налаштування підключення QUIC відбувається швидше. У той час як для TLS-over-TCP обидва протоколи потребують власних окремих рукописок, QUIC натомість об'єднує транспортне та криптографічне рукопискування в одне, заощаджуючи час з'єднання (див. рис. 2.2 вище). QUIC може розвиватися легше. Оскільки він повністю зашифрований, проміжні

пристрої в мережі більше не можуть спостерігати та інтерпретувати його внутрішню роботу, як вони можуть із TCP.

Однак, окрім цих переваг, існують також деякі потенційні недоліки розширеного шифрування [21]:

Багато мереж можуть блокувати QUIC.

Компанії можуть захотіти заблокувати його на своїх брандмауерах, оскільки виявлення небажаного трафіку стає важчим. Інтернет-провайдери та проміжні мережі можуть блокувати його, оскільки такі показники, як середні затримки та відсотки втрат пакетів, більше недоступні, що ускладнює виявлення та діагностику проблем.

QUIC має вищі витрати на шифрування.

QUIC шифрує кожен окремий пакет за допомогою TLS, тоді як TLS-over-TCP може шифрувати декілька пакетів одночасно. Це потенційно робить QUIC повільнішим для сценаріїв з високою пропускну здатністю.

QUIC робить Інтернет більш централізованим.

Друга велика відмінність між TCP і QUIC трохи більш технічна

Для HTTP/1.1 процес завантаження ресурсу досить простий, оскільки кожному файлу надається власне TCP-з'єднання та завантажується повністю. Наприклад, якщо у нас є файли А, В і С, у нас буде три TCP-з'єднання. Перший побачить потік байтів АААА, другий ВВВВ, третій СССС (кожне повторення літери є TCP-пакетом). Це працює, але також дуже неефективно, оскільки кожне нове підключення має певні витрати.

На практиці браузері накладають обмеження на кількість одночасних з'єднань (і, отже, на кількість файлів, які можна завантажити паралельно) — як правило, від 6 до 30 на одне завантаження сторінки. Потім з'єднання повторно використовуються для завантаження нового файлу, коли попередній буде повністю передано. Зрештою ці обмеження почали перешкоджати веб-продуктивності на сучасних сторінках, які часто завантажують більше 30 ресурсів.

Покращення цієї ситуації було однією з головних цілей HTTP/2. Протокол робить це, не відкриваючи нове TCP-з'єднання для кожного файлу, а завантажуючи різні ресурси через одне TCP-з'єднання. Це досягається шляхом «мультиплексування» різних потоків байтів.

Для наших трьох прикладів файлів ми отримаємо єдине з'єднання TCP, а вхідні дані можуть виглядати як AABVSSAABVSS (хоча можливі й інші схеми впорядкування). Це здається досить простим і справді працює досить добре, роблячи HTTP/2 зазвичай таким же або трохи швидшим, ніж HTTP/1.1, але з набагато меншими витратами Рисунок 2.4.

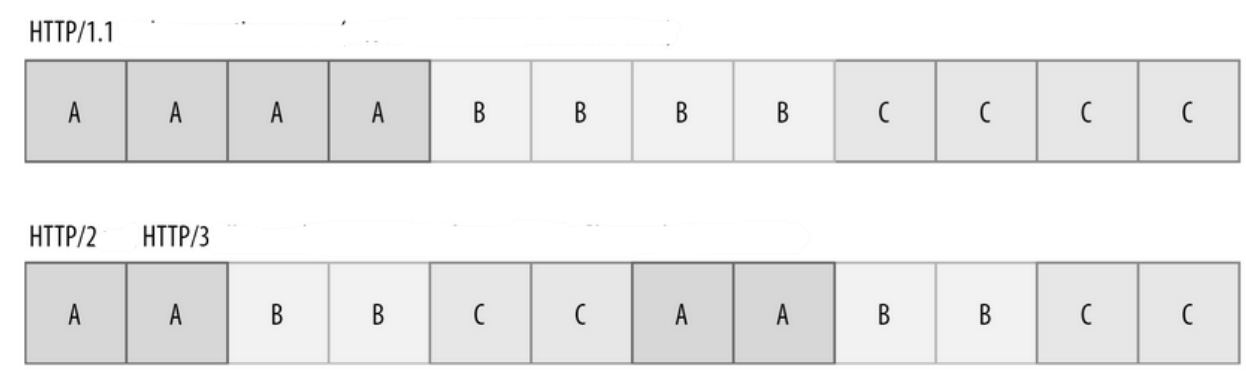


Рисунок 2.4 Візуалізація мультиплексування в HTTP/3

Однак існує проблема на стороні TCP. Оскільки TCP є набагато старішим протоколом і не призначений лише для завантаження веб-сторінок, він не знає про А, В або С. Внутрішньо TCP вважає, що транспортує лише один файл, Х, і це не так. Проблема полягає в тому, що він розглядає дані як XXXXXXXXXXXXX, а насправді це AABVSSAABVSS на рівні HTTP. У більшості ситуацій це не має значення (і це робить TCP досить гнучким), але ситуація змінюється, коли у мережі втрачаються пакети [22].

Припустімо, що третій пакет TCP втрачено (той, що містить перші дані для файлу В), але всі інші дані доставлено. TCP справляється з цією втратою шляхом повторної передачі нової копії втрачених даних у новому пакеті.

Оскільки логіка повторної передачі відбувається на рівні TCP, і TCP не знає про А, В і С. Натомість TCP вважає, що частину одного файлу Х було втрачено, і, отже, він вважає, що він повинен утримувати решту даних Х від

обробки, доки прогалина не буде заповнена. TCP цього не знає, через що все працює повільніше, ніж могло б бути. Ця неефективність є прикладом проблеми «блокування голови лінії (HoL).

Вирішення блокування HoL на транспортному рівні було однією з головних цілей QUIC. На відміну від TCP, QUIC чітко усвідомлює, що він мультиплексує кілька незалежних потоків байтів. Він, звичайно, не знає, що транспортує CSS, JavaScript і зображення; він просто знає, що потоки окремі. Таким чином, QUIC може виконувати логіку виявлення втрати пакетів і відновлення на основі кожного потоку.

Третім головним удосконаленням у QUIC є те, що з'єднання можуть зберігатися довше.

В Інтернеті IP-адреси використовуються для маршрутизації пакетів між двома унікальними машинами. Однак просто мати IP-адреси для вашого телефону та сервера недостатньо, тому що обидва хочуть мати можливість запускати кілька мережевих програм на кожному кінці одночасно.

Ось чому кожному окремому з'єднанню також призначається номер порту на обох кінцевих точках, щоб розрізняти з'єднання та програми, до яких вони належать. Серверні програми зазвичай мають фіксований номер порту залежно від їх функції (наприклад, порти 80 і 443 для HTTP(S) і порт 53 для DNS), тоді як клієнти зазвичай вибирають свої номери портів (напів)випадково для кожного з'єднання.

Таким чином, щоб визначити унікальне з'єднання між машинами та програмами, нам потрібні ці чотири речі, так званий 4-кортеж: IP-адреса клієнта + порт клієнта + IP-адреса сервера + порт сервера.

У TCP підключення ідентифікуються лише 4-кортежем. Отже, якщо змінюється хоча б один із цих чотирьох параметрів, з'єднання стає недійсним і його необхідно відновити (включаючи нове рукошестикання). Щоб зрозуміти це, уявимо проблему паркування: зараз хтось використовує свій Iphone у будівлі з Wi-Fi. Таким чином, у нього є IP-адреса в цій мережі Wi-Fi, а потім

виходить на вулицю і підключається до 4G мережі, при цьому протокол TCP не має можливості зрозуміти цю ситуацію, представлена на Рисунок 2.5

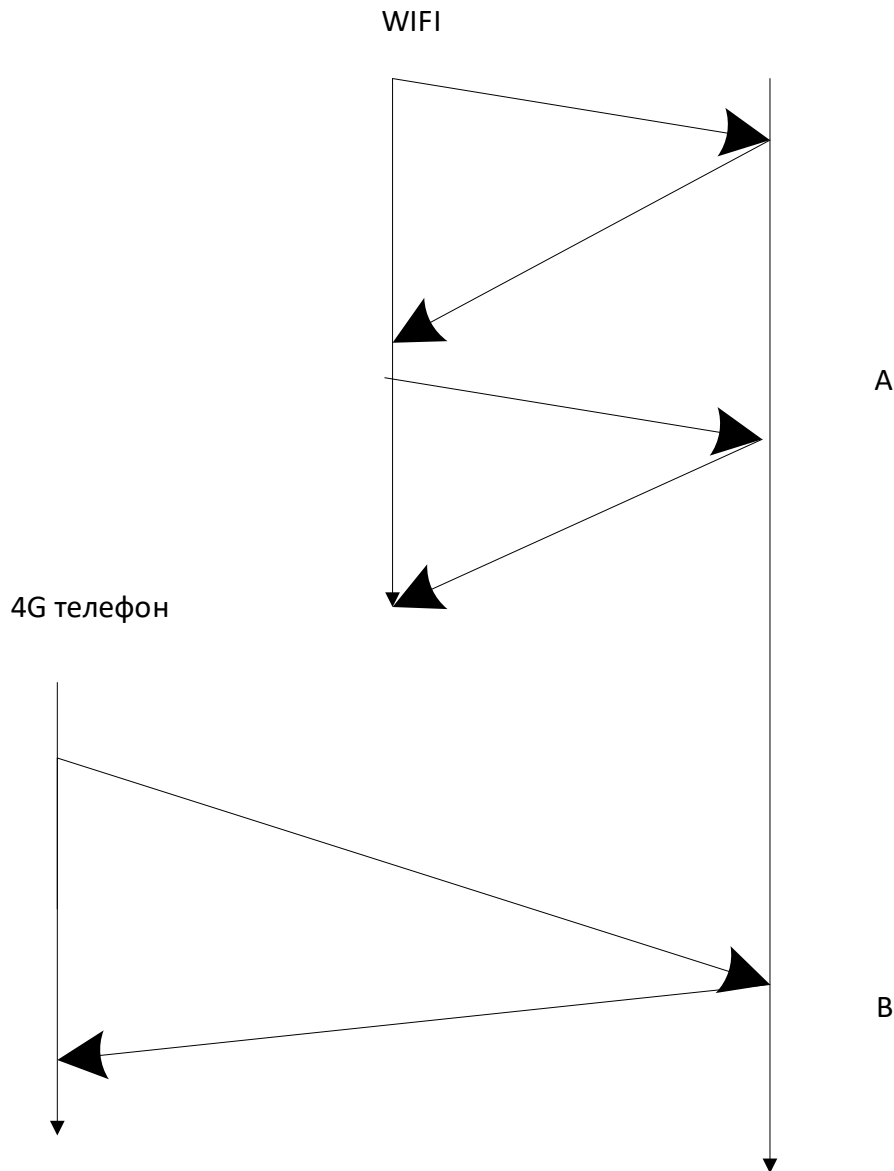


Рисунок 2.5 Проблема паркування з TCP: як тільки клієнт отримує нову IP-адресу, сервер більше не може підключити його до з'єднання.

Оскільки TCP був винайдений ще до того, як ми навіть мріяли про стільникові мережі та смартфони, не існує, наприклад, механізму, який би дозволив клієнту повідомляти серверу про зміну IP-адрес. Немає навіть способу «закрити» з'єднання, оскільки команда скидання TCP або `fin`, надіслана старому 4-кортежу, більше навіть не досягне клієнта. Таким чином, на практиці кожна зміна мережі означає, що існуючі з'єднання TCP більше не можна використовувати.

Для встановлення нового з'єднання потрібно виконати нове рукоштовнання TCP (і, можливо, TLS), і, залежно від протоколу на рівні програми, дії в процесі потрібно буде перезапустити. Наприклад, якщо завантажувати великий файл через HTTP, тоді цей файл, можливо, доведеться повторно запитувати з самого початку

Вкрай важливо, оскільки цей CID визначено на транспортному рівні в самому QUIC, він не змінюється під час переходу між мережами. Щоб зробити це можливим, CID міститься на початку кожного пакета QUIC, якщо повернутися до Рисунок 2.5 у на початку ситуація А встановлюється QUIC з'єднання і ми отримуємо CID (ід з'єднання) і коли у нас виникає ситуація В приєднання через телефон, хоч IP адреса буде різною, але CID тим самим і тому сервер розпочне комунікацію із клієнтом без зайвої роботи.

Існують інші проблеми, які необхідно подолати за допомогою CID. Наприклад, якби ми справді використовували лише один CID, хакерам і перехоплювачам було б надзвичайно легко стежити за користувачем у мережах і, як наслідок, визначити його (приблизне) фізичне місцезнаходження. Щоб уникнути цього кошмару конфіденційності, QUIC змінює CID щоразу, коли використовується нова мережа.

Що насправді відбувається всередині, так це те, що клієнт і сервер узгоджують загальний список (випадково згенерованих) CID, які всі відображаються в одному концептуальному «з'єднанні».

Наприклад, вони обидва знають, що CID K, C і D насправді відображаються на з'єднанні X. Таким чином, хоча клієнт може позначати пакети K на Wi-Fi, він може переключитися на використання C на 4G. Ці загальні списки узгоджуються повністю зашифрованими в QUIC, тому потенційні злоумисники не знатимуть, що K і C насправді є X, але клієнт і сервер знатимуть це, і вони зможуть підтримувати з'єднання.

Останнім аспектом QUIC є те, що він спеціально створений для легкого розвитку. Це досягається кількома різними способами. По-перше, як обговорювалося, той факт, що QUIC майже повністю зашифрований, означає,

що нам потрібно лише оновити кінцеві точки (клієнти та сервери), а не всі проміжні пристрої, якщо ми хочемо розгорнути новішу версію QUIC. На це все одно потрібен час, але зазвичай це місяці, а не роки.

По-друге, на відміну від TCP, QUIC не використовує один фіксований заголовок пакета для надсилання всіх метаданих протоколу. Натомість QUIC має короткі заголовки пакетів і використовує різноманітні «фрейми» (на кшталт мініатюрних спеціалізованих пакетів) усередині корисного навантаження пакета для передачі додаткової інформації. Існує, наприклад, кадр ACK (для підтвердження), кадр NEW_CONNECTION_ID (щоб допомогти налаштувати міграцію підключення) і кадр STREAM (для передачі даних).

В основному це робиться для оптимізації, оскільки не кожен пакет містить усі можливі метадані (і тому заголовок пакету TCP зазвичай витрачає досить багато байтів — див. також зображення вище). Однак дуже корисним побічним ефектом використання фреймів є те, що в майбутньому визначати нові типи фреймів як розширення QUIC буде дуже легко. Дуже важливим є, наприклад, кадр DATAGRAM, який дозволяє надсилати ненадійні дані через зашифроване з'єднання QUIC [33].

Підсумовуємо інформацію щодо QUIC та TCP у таблицю 2.1

Таблиця 2.1 Порівняльна характеристика QUIC та TCP

	TCP	QUIC
Базовий протокол	+	- (базується на UDP)
Congestion control (Контроль заторів)	+	+
Reliability (Надійність)	+	+
Head-of-line blocking (блокування початку передачі)	-	+
Connection migration (Міграція підключення)	-	+
multiplexing	-	+

2.2. Покращення продуктивності в HTTP/3.0

У цьому пункті ми дослідимо масштаб покращень продуктивності, які QUIC і HTTP/3 повинні надати для процесу завантаження веб-сторінок.

Оскільки ми маємо справу з мережевими протоколами, то в основному розглянемо мережеві аспекти, з яких два найважливіші: затримка та пропускна здатність.

Затримку можна приблизно визначити як час, необхідний для надсилання пакета з точки А (скажімо, клієнта) до точки Б (сервера). Він фізично обмежений швидкістю світла або, практично, швидкістю передачі сигналів по дротах або на відкритому повітрі. Це означає, що затримка часто залежить від фізичної, реальної відстані між А і В.

На землі це означає, що типові затримки є концептуально малими, приблизно від 10 до 200 мілісекунд. Однак це лише один спосіб: відповіді на пакети також повинні повертатися. Двосторонню затримку часто називають часом зворотного зв'язку (RTT).

Завдяки таким функціям, як контроль перевантажень (о ній читайте нижче), часто потрібно досить багато проходжень, щоб завантажити хоча б один файл. Таким чином, навіть низькі затримки менше 50 мілісекунд можуть призвести до значних затримок. Це одна з головних причин існування мереж доставки контенту (CDN).

Часто використовується метафора про трубу для транспортування води. Довжина каналу - це затримка, тоді як ширина каналу - це пропускна здатність. Однак в Інтернеті ми зазвичай маємо довгий ряд з'єднаних каналів, деякі з яких можуть бути ширшими за інші (що призводить до так званих вузьких місць у найвужчих ланках). Таким чином, наскрізна смуга пропускання між точками А і В часто обмежена найповільнішими підрозділами.

Одним з аспектів продуктивності є те, наскільки ефективно транспортний протокол може використовувати повну (фізичну) смугу пропускання мережі (тобто приблизно скільки пакетів за секунду можна

надіслати або отримати). Це, у свою чергу, впливає на швидкість завантаження ресурсів сторінки.

Інша проблема полягає в тому, що ми не знаємо заздалегідь, яка буде максимальна пропускна здатність. Це часто залежить від вузького місця десь у наскрізному з'єднанні, але ми не можемо передбачити або знати, де це буде. В Інтернеті також (поки що) немає механізмів, щоб сигналізувати про пропускну здатність каналів до кінцевих точок.

Таким чином, з'єднання не знає, яку пропускну здатність воно може безпечно чи справедливо використовувати наперед, і ця пропускна здатність може змінюватися, коли користувачі приєднуються до мережі, виходять із неї та використовують її. Щоб вирішити цю проблему, ТСП постійно намагатиметься виявити доступну пропускну здатність з часом за допомогою механізму, що називається контролем перевантаження.

На початку з'єднання він надсилає лише кілька пакетів (на практиці це становить від 10 до 100 пакетів, або близько 14–140 КБ даних) і чекає один зворотний зв'язок, доки одержувач не надішле підтвердження цих пакетів. Якщо всі вони підтверджені, це означає, що мережа може обробляти цю швидкість надсилання, і можливо спробувати повторити процес, але з більшою кількістю даних (на практиці швидкість надсилання зазвичай подвоюється з кожною ітерацією).

Таким чином, швидкість надсилання продовжує зростати, доки деякі пакети не будуть підтверджені (що вказує на втрату пакетів і перевантаження мережі). Ця перша фаза зазвичай називається «повільним стартом». Після виявлення втрати пакета ТСП зменшує швидкість надсилання та (через деякий час) знову починає збільшувати швидкість надсилання, хоча й (набагато) меншими кроками. Ця логіка зменшення, а потім зростання повторюється для кожної втрати пакета згодом. Зрештою це означає, що ТСП постійно намагатиметься досягти своєї ідеальної справедливої частки пропускну здатності. Цей механізм зображено на Рисунок 2.6 [13].

Це надзвичайно спрощене пояснення контролю заторів. На практиці діє багато інших факторів, таких як розвантаження буфера, коливання RTT через перевантаження та той факт, що кілька одночасних відправників повинні отримати свою справедливую частку пропускної здатності. Таким чином, існує багато різних алгоритмів контролю перевантажень, і сьогодні все ще винаходять багато, і жоден не працює оптимально в усіх ситуаціях.

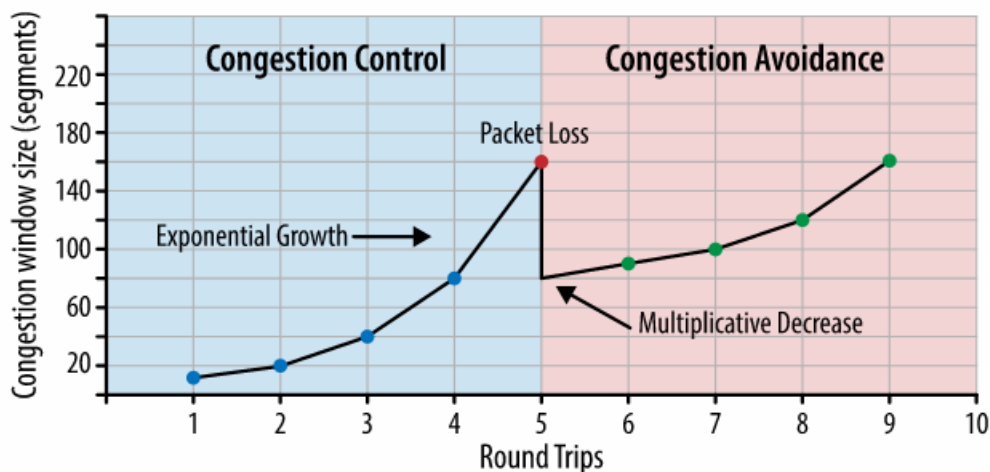


Рисунок 2.6 Контроль заторів і уникнення заторів [13]

Для завантаження веб-сторінки цей підхід із повільним запуском також може вплинути на такі показники, як перше відображення вмісту, оскільки лише невеликий обсяг даних (від десятків до кількох сотень КБ) можна передати під час перших кількох двосторонніх передачі. Само звідси існує рекомендація зберігати важливі дані меншими ніж 14 КБ [26].

QUIC, теоретично, менше страждає від втрати пакетів (і пов'язаного блокування голови рядка (HOL)), оскільки він обробляє втрату пакетів у кожному потоці байтів ресурсу незалежно. Крім того, QUIC працює через протокол UDP, який, на відміну від TCP, не має вбудованої функції контролю перевантаження; це дозволяє спробувати надсилати з будь-якою швидкістю, і не передає повторно втрачені дані.

Це призвело до багатьох статей, які стверджували, що QUIC також не використовує контроль перевантаження, що замість цього QUIC може почати надсилати дані з набагато вищою швидкістю через UDP ось чому QUIC набагато швидший за TCP.

Насправді згідно стандарту RCF9002 [21] QUIC фактично використовує дуже схожі методи керування пропускнуою здатністю, що й TCP. Він також починається з нижчої швидкості надсилання та збільшується з часом, використовуючи підтвердження як ключовий механізм для вимірювання пропускнуої здатності мережі. Це (серед інших причин) тому, що QUIC має бути надійним, щоб бути корисним для чогось, наприклад HTTP, тому що він має бути справедливим щодо інших QUIC.

Однак це не означає, що QUIC не може бути (трохи) розумнішим щодо керування пропускнуою здатністю, ніж TCP. Це головним чином тому, що QUIC є більш гнучким і легшим для розвитку, ніж TCP.

TCP зазвичай реалізується в ядрі операційної системи (ОС), безпечному та більш обмеженому середовищі, яке для більшості ОС навіть не є відкритим кодом. Таким чином, налаштування логіки перевантаження зазвичай виконується лише декількома вибраними розробниками, і розвиток відбувається повільно.

На відміну від цього, більшість реалізацій QUIC наразі виконується в «користувальницькому просторі» і створюються з відкритим кодом, це дозволяє заохотити до експериментів набагато ширшу групу розробників.

Іншим конкретним прикладом є пропозиція розширення частоти відкладеного підтвердження для QUIC [20]. Хоча за замовчуванням QUIC надсилає підтвердження для кожних 2 отриманих пакетів, це розширення дозволяє кінцевим точкам підтверджувати, наприклад, кожні 10 пакетів. Показано, що це дає значні переваги у швидкості в супутникових і дуже високошвидкісних мережах, оскільки накладні витрати на передачу пакетів підтвердження знижуються. Додавання такого розширення для TCP займе багато часу, щоб прийняти його, тоді як для QUIC його набагато легше розгорнути.

Другий аспект продуктивності полягає в тому, скільки проходжень потрібно, перш ніж користувач зможе надіслати корисні дані HTTP (скажімо, ресурси сторінки) через нове з'єднання. Дехто стверджує, що QUIC на два-три

рази швидше, ніж TCP + TLS, але ми побачимо, що насправді це лише один графічно відображення на Рисунок 2.2

QUIC був розроблений з урахуванням TLS із самого початку, і, таким чином, поєднує транспортні та криптографічні рукостискання в одному механізмі. Це означає, що для завершення зв'язку QUIC знадобиться лише одне проходження в обидві сторони, що на один зворотний зв'язок менше, ніж TCP + TLS 1.3

Хоча збільшення швидкості на одну «поїздку» туди й назад приємно, це навряд чи дивно. Особливо у швидких мережах (скажімо, RTT менше 50 мілісекунд) це буде ледь помітно, хоча повільні мережі та підключення до віддалених серверів принесуть трохи більше зиску.

Можливо задатися питанням, чому нам взагалі потрібно чекати рукостискання. Чому ми не можемо надіслати HTTP-запит під час першого повернення? Це головним чином тому, що, якби ми це зробили, перший запит було б надіслано незашифрованим, його міг би прочитати будь-який перехоплювач, що, очевидно, не підходить для конфіденційності та безпеки. Таким чином, нам потрібно дочекатися завершення криптографічного рукостискання, перш ніж надсилати перший запит HTTP.

Тут на практиці використовується хитрий прийом. Якщо відомо, що користувачі часто переглядають веб-сторінки протягом короткого часу після свого першого відвідування. Можливо використовувати початкове зашифроване з'єднання для завантаження другого з'єднання в майбутньому. Простіше кажучи, протягом свого життя перше з'єднання використовується для безпечної передачі нових криптографічних параметрів між клієнтом і сервером. Потім ці параметри можна використовувати для шифрування другого з'єднання з самого початку, не чекаючи завершення повного рукостискання TLS. Цей підхід називається «відновлення сеансу» (session resumption). Рисунок 2.7

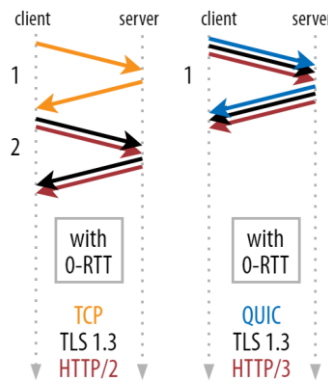


Рисунок 2.7 Налаштування підключення TCP + TLS проти QUIC 0-RTT [29]

Під час використання 0-RTT QUIC навіть не може використовувати цю отриману передачу так добре через безпеку. Щоб зрозуміти це, нам потрібно зрозуміти одну з причин, чому існує рукостискання TCP. По-перше, це дозволяє клієнту переконатися, що сервер дійсно доступний за вказаною IP-адресою, перш ніж надсилати йому будь-які дані вищого рівня.

По-друге, що має вирішальне значення тут, це дозволяє серверу переконатися, що клієнт, який відкриває з'єднання, насправді є тим, хто і де вони кажуть, перш ніж надсилати йому дані.

Для уникнення відображення або посилення атаки, і це важливий спосіб, за допомогою якого хакери виконують розподілені атаки на відмову в обслуговуванні (DDoS). QUIC має бути консервативним у відповідях на запити 0-RTT, обмежуючи кількість даних, які він надсилає у відповідь, доки не буде перевірено, що клієнт є справжнім клієнтом, а не жертвою. Інакше кажучи, QUIC має максимальний «коефіцієнт посилення» три, що було визначено як прийнятний компроміс між корисністю продуктивності та ризиком безпеки.

Третя функція продуктивності робить QUIC швидшим під час передачі між мережами, зберігаючи наявні з'єднання без змін. Хоча це справді працює, такий тип зміни мережі трапляється не так часто, і з'єднання все одно потребують скидання швидкості надсилання.

Оскільки між припиненням роботи мережі 1 і доступністю мережі 2 зазвичай проходить деякий час, відеопрограми можуть відкривати кілька підключень (по одному на мережу), синхронізуючи їх до того, як стара мережа

повністю припинить роботу. Користувач все одно помітить процес перемикання, але це не призведе до повного відключення відео.

Ця функціональність базується на протоколі QUIC та була розглянута у розділі 2.1.

Четверта функція продуктивності призначена для пришвидшення QUIC у мережах із великою кількістю втрат пакетів шляхом пом'якшення проблеми блокування голови лінії (HoL). Хоча теоретично це вірно, але на практиці, надає лише незначні переваги для продуктивності завантаження веб-сторінки.

Втрата пакетів TCP може призвести до затримки даних для кількох ресурсів, що передаються, оскільки абстракція байтового потоку TCP розглядає всі дані як частину одного файлу. QUIC, з іншого боку, глибоко усвідомлює, що існує кілька одночасних потоків байтів, і може впоратися з втратою на основі кожного потоку. Однак, ці потоки насправді не передають дані паралельно.

Деякі ресурси можуть блокувати візуалізацію. Це стосується файлів CSS і деяких JavaScript в елементі head HTML. Поки ці файли завантажуються, браузер не може відобразити сторінку (або, наприклад, виконати новий JavaScript).

Більше того, для використання файли CSS і JavaScript потрібно завантажити повністю (хоча їх часто можна поступово аналізувати та скомпілювати). Таким чином, ці ресурси потрібно завантажити якнайшвидше з найвищим пріоритетом.

Керування пріоритетом є у HTTP/3.

Досі ми в основному говорили про нові функції продуктивності в QUIC проти TCP. HTTP/3 насправді є HTTP/2-over-QUIC, і тому в новій версії не було представлено жодних реальних великих нових функцій. Це не схоже на перехід від HTTP/1.1 до HTTP/2, який був набагато масштабнішим і впроваджував такі нові функції, як стиснення заголовків, пріоритезація потоку та надсилання даних на сервер. Усі ці функції все ще є в HTTP/3, але є деякі важливі відмінності в тому, як вони реалізовані під капотом [36].

Тому внутрішню механіку та реалізацію функцій довелося змінити для HTTP/3. Конкретним прикладом є стиснення заголовків HTTP, яке зменшує накладні витрати на повторювані великі заголовки HTTP (наприклад, файли cookie та рядки агента користувача). У HTTP/2 це було зроблено за допомогою налаштування HPACK, тоді як для HTTP/3 це було перероблено до більш складного QPACK. Обидві системи забезпечують однакову функцію (тобто стиснення заголовка), але зовсім різними способами [23].

Є сервер push. Ця функція дозволяє серверу надсилати HTTP-відповіді, не чекаючи попереднього явного запиту на них. Теоретично це може забезпечити чудовий приріст продуктивності. На практиці, однак, виявилось, що це важко використовувати правильно та непослідовно реалізувати.

QUIC і HTTP/3 продовжуватимуть розвиватися та ставати швидшими в найближчі роки.

Розробники Cloudflare дослідили швидкодію HTTP/3.0 та отримали наступні результати [38].

Покращення запуску сеансу означають, що «підключення» до серверів починаються набагато швидше, а це означає, що браузер починає бачити дані швидше. Щоб виміряти покращення завдяки підтримці 0-RTT, ми запустили кілька тестів вимірювання часу до першого байта (TTFB). У середньому з HTTP/3 дослідники побачили, що перший байт з'являється через 176 мс. З HTTP/2 ми бачимо 201 мс, тобто HTTP/3 вже працює на 12,4% краще.

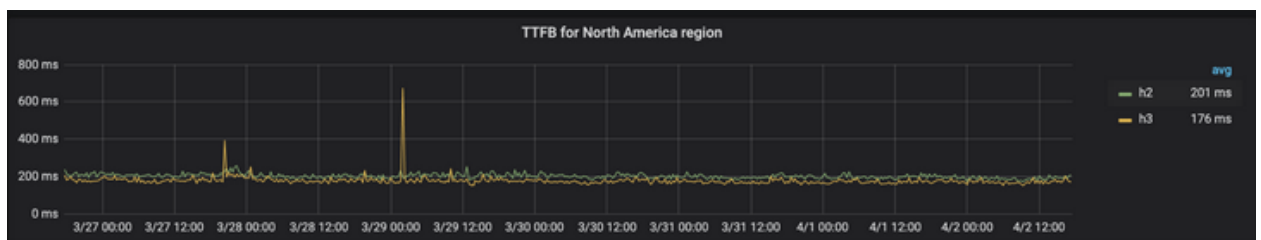


Рисунок 2.8 TTFB для регіону North America [38]

Але це у синтетичних тестах, якщо вимірювати повне завантаження реальної сторінки blog.cloudflare.com то різниця між HTTP/2.0 була менше ніж статистична значимість в умовах наявності гарного каналу зв'язку. Більше докладно із результатами можливо ознайомиться в [38]

У додатку Б приведено візуалізація різниці між різними версіями HTTP.

2.3. Практичні рекомендації по розгортанню HTTP/3.0

Найбільша різниця між HTTP/1.1 і HTTP/2 полягала в переході від 6 до 30 паралельних TCP-з'єднань на одне базове TCP-з'єднання. HTTP/3 продовжує цей підхід, але «просто» перемикається з одного TCP на одне з'єднання QUIC. Ця різниця сама по собі не має особливого ефекту (вона в основному зменшує накладні витрати на стороні сервера).

На практиці перехід до встановлення єдиного підключення був досить складним, оскільки багато сторінок було розділено на різні імена хостів і навіть сервери (наприклад, `img1.example.com` і `img2.example.com`). Це пояснюється тим, що браузері відкривають лише до шести з'єднань для кожного окремого імені хоста, тому наявність кількох дозволяла більше з'єднань. Без змін у цьому налаштуванні HTTP/1.1 HTTP/2 усе одно відкриватиме кілька з'єднань, зменшуючи ефективність роботи інших функцій, таких як пріоритезація [16].

У HTTP/1.1 можливо мати лише один активний ресурс на з'єднання, що призводило до блокування головного рядка (HoL) на рівні HTTP. Оскільки кількість з'єднань була обмежена лише 6–30, об'єднання ресурсів (де менші підресурси об'єднуються в один більший ресурс) було найкращою практикою протягом тривалого часу.

Однак за допомогою HTTP/2 єдине з'єднання мультиплексує ресурси, тож у користувача може бути набагато більше невиконаних запитів на файли (іншими словами, один запит більше не займає одне з небагатьох дорогоцінних з'єднань).

Запити файлів все ще не такі дешеві в HTTP/2, як передбачалося спочатку. Нарешті, відсутність вбудованих ресурсів має додаткову вартість затримки, оскільки файл потрібно запитувати. Це в поєднанні з пріоритетами

та проблемами з сервером означає, що навіть сьогодні все ще краще вбудувати частину критичного CSS. Усе це, звісно, також актуально для HTTP/3.

Щоб мати можливість завантажувати декілька файлів за одне з'єднання, потрібно якось їх мультиплексувати. Мультиплексування керується за допомогою системи визначення пріоритетів. Ось чому важливо мати якомога більше ресурсів, запитуваних на тому самому з'єднанні, щоб мати можливість правильно розставити пріоритети один для одного.

Це, у свою чергу, означало, що деякі інші рекомендації для HTTP/2 — наприклад, зменшене групування, оскільки запити дешеві, і зменшене шардування сервера, щоб оптимально використовувати єдине з'єднання.

На жаль, пересічний веб-розробник, не можете багато чого зробити, тому що це в основному проблема в самих браузерах і серверах.

Тим не менш, фундаментальні концепції визначення пріоритетів не зміняться. Все одно не можливо використовувати такі методи, як попереднє завантаження, не розуміючи, що відбувається всередині, тому що це все одно може неправильно розставити пріоритети ресурсів.

Ось деякі конкретні моменти, які здебільшого справедливі як для HTTP/2, так і для HTTP/3 [5]:

Розробникам можливо рекомендувати розділяти ресурси приблизно на одне-три підключення на критичному шляху (якщо користувачі здебільшого не користуються мережами з низькою пропускну здатністю), використовуючи за потреби попереднє підключення та попередню вибірку dns (dns-prefetch).

Логічно згрупувати підресурси за шляхом чи функцією або за частотою змін. П'ять-десять ресурсів JavaScript і п'ять-десять CSS-ресурсів на одну сторінку досить ефективно. Вбудовування критичного CSS може бути хорошою оптимізацією.

Використовувати складні функції, такі як попереднє завантаження, економно.

Використовувати сервер, який належним чином підтримує пріоритезацію HTTP/2.

Причому потрібно перевіряти, що TLS 1.3 увімкнено на веб-сервері HTTP/2.

QUIC і HTTP/3 є досить складними протоколами. Реалізація їх з нуля потребувала б прочитання і розуміння сотень сторінок, розташованих на більш ніж семи документах.

Кілька компаній працюють над впровадженням QUIC і HTTP/3 з відкритим кодом уже понад п'ять років, тож у нас є кілька зрілих і стабільних варіантів на вибір їх перелік можливо знайти в [6].

Деякі з найбільш важливих і стабільних представлені у таблиці 2.2:

Таблиця 2.2 Основні реалізації QUIC та HTTP/3

Мова реалізації	Реалізації
Python	Aioquic
Go	quic-go
C, C++	mvfst (Facebook), MsQuic, (Microsoft), (Google), ngtcp2, LSQUIC (Litespeed), picoquic, quicly (Fastly)

Однак багато із цих реалізацій в основному піклуються про HTTP/3 і QUIC; самі по собі не є повноцінними веб-серверами. Коли справа доходить до типових серверів (наприклад, NGINX, Apache, Node.js), все було трохи повільніше.

По-друге, багато серверів залежать від сторонніх бібліотек TLS, таких як OpenSSL. Знову ж таки, це тому, що TLS дуже складний і має бути безпечним, тому найкраще повторно використовувати наявні, перевірені напрацювання. Однак, незважаючи на те, що QUIC інтегрується з TLS 1.3, він використовує його значно інакше, ніж взаємодія TLS і TCP.

Наразі можливо рекомендувати використовувати Hypercorn, Caddy, H₂O, Litespeed.

Звернемо увагу на деякі важливі нюанси:

Навіть «повна підтримка» означає «настільки добре, наскільки це можливо на даний момент», не обов'язково «готове до виробництва». Наприклад, багато реалізацій ще не повністю підтримують міграцію з'єднання, 0-RTT, натискання сервера або пріоритезацію HTTP/3.

Щодо використання HTTP/3.0 станом на кінець із боку серверів згідно [32] 25,1% вебсайтів використовує HTTP/3.0 дивиться Рисунок 2.9, що стосується браузерів, то відповідно до [18], 73,79% клієнтів можуть використовувати третю версію HTTP.

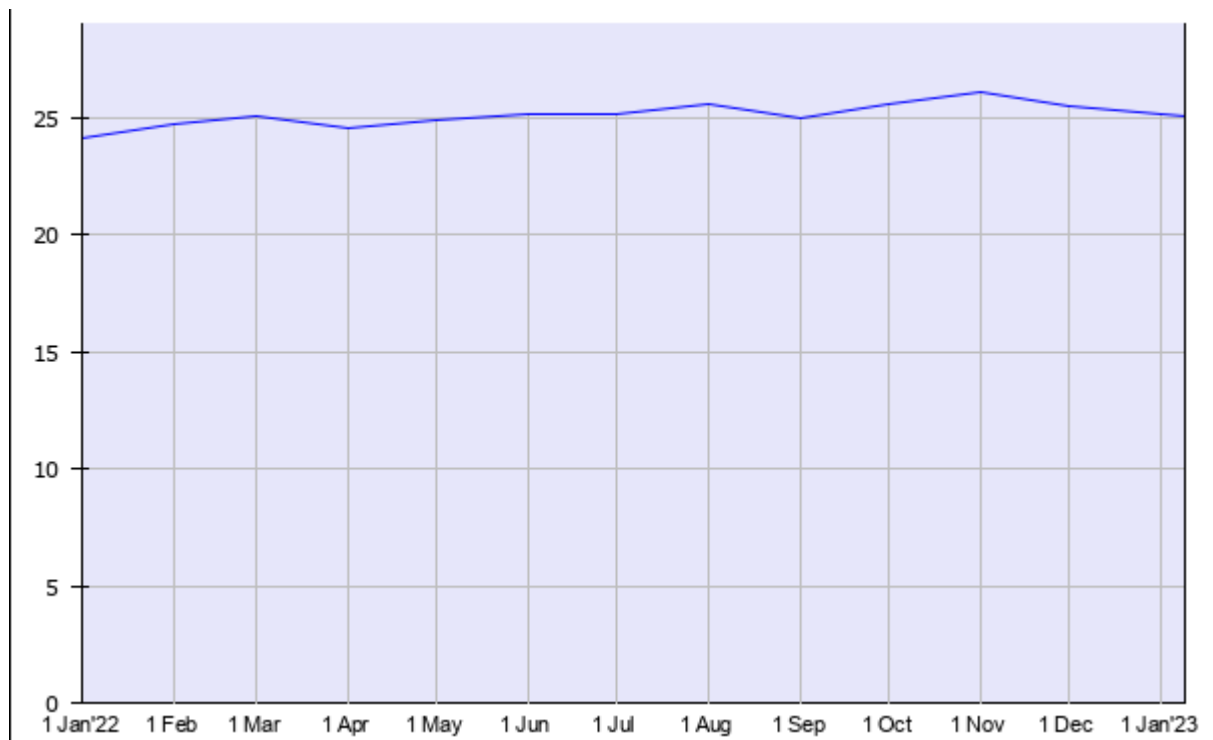


Рисунок 2.9 Використання HTTP/3.0 в 2022 році [32]

Також потрібно потурбуватися про конфігурацію мережі оскільки UDP часто використовується для атак і не є критичним для нормальної повсякденної роботи, окрім DNS, багато (корпоративних) мереж і брандмауерів майже повністю блокують протокол. Таким чином, UDP, ймовірно, має бути явно дозволено до/від ваших серверів HTTP/3. QUIC може працювати на будь-якому порту UDP, але очікується, що порт 443 (який зазвичай також використовується для HTTPS через TCP) буде найпоширенішим.

Однак багато мережевих адміністраторів не бажають просто дозволяти UDP оптом. Натомість вони можуть дозволити QUIC через UDP. Проблема полягає в тому, що, QUIC майже повністю зашифрований. Це включає метадані рівня QUIC, такі як номери пакетів, а також, наприклад, сигнали, які вказують на закриття з'єднання.

Однак завдяки шифруванню QUIC брандмауери можуть набагато менше виконувати цю логіку відстеження на рівні підключення, а кілька бітів, які вони можуть перевірити, є відносно складними.

Таким чином, багато постачальників брандмауерів наразі рекомендують блокувати QUIC, доки вони не зможуть оновити своє програмне забезпечення.

Також розглянемо декілька інструментів, які мають спеціальну підтримку для нового протоколу.

Lighthouse — це автоматизований інструмент із відкритим кодом для покращення якості веб-сторінок. Його можливо запустити його на будь-якій веб-сторінці, загальнодоступній або з вимогою автентифікації. Він має аудит продуктивності, доступності, прогресивних веб-додатків, SEO тощо.

Користувач надає Lighthouse URL-адресу для перевірки, програма запускає серію перевірок сторінки, а потім створює звіт про те, наскільки добре сторінка. Після цього використовуйте невдалі перевірки як індикатори того, як покращити сторінку. Кожен аудит має довідковий документ, у якому пояснюється, чому аудит важливий, а також як це виправити.

По-друге, є WebPageTest. Цей проєкт дозволяє завантажувати сторінки через реальні мережі з реальних пристроїв по всьому світу, а також дозволяє додавати емуляцію мережі на рівні пакетів, включаючи такі аспекти, як втрата пакетів. Таким чином, WebPageTest концептуально займає найкраще місце для порівняння продуктивності HTTP/2 і HTTP/3. Однак він дійсно вже може завантажувати сторінки через новий протокол.

Інструмент Wireshark має розширену підтримку QUIC, а також може експериментально аналізувати HTTP/3. Це дозволяє спостерігати, які пакети QUIC і HTTP/3 фактично передаються по дроту. Однак для того, щоб це

працювало, потрібно отримати ключі розшифровки TLS для даного з'єднання, які більшість реалізацій (включно з Chrome і Firefox) дозволяють отримати за допомогою змінної середовища SSLKEYLOGFILE. Хоча це може бути корисним для деяких речей, справді з'ясувати, що відбувається, особливо для довгих з'єднань, може спричинити за собою багато ручної роботи. Також розробнику знадобиться досить глибоке розуміння внутрішньої роботи протоколів.

2.4. Висновки до розділу 2

Наступна основна версія HTTP, HTTP/3, має таку саму семантику, як і попередні версії HTTP, але використовує QUIC замість TCP для частини транспортного рівня. У жовтні 2022 року 26% усіх веб-сайтів використовували HTTP/3 [32].

QUIC розроблено для забезпечення значно меншої затримки для HTTP-з'єднань. Як і HTTP/2, це мультиплексований протокол, але HTTP/2 працює через одне з'єднання TCP, тому виявлення втрати пакетів і повторна передача, оброблена на рівні TCP, може блокувати всі потоки. QUIC запускає кілька потоків через UDP і реалізує виявлення втрати пакетів і повторну передачу незалежно для кожного потоку, тому в разі виникнення помилки блокується лише потік із даними в цьому пакеті.

Тобто розвитку WWW потрібен був не HTTP/3, а скоріше «TCP/2», і ми отримали HTTP/3 «безкоштовно» в процесі. Основні функції HTTP/3, швидше встановлення з'єднання, менше блокування HoL, міграція з'єднань тощо, походять від QUIC.

HTTP/3 швидший за HTTP/2 не лише тому, що замінили TCP на UDP. Швидше було переосмислено та впроваджено набагато вдосконалену версію TCP під назвою QUIC, але для полегшення розгортання QUIC, його запускають його через UDP.

QUIC за замовчуванням глибоко зашифрований. Це не тільки покращує його характеристики безпеки та конфіденційності, але також сприяє його розгортанню та розвитку. Це робить протокол трохи важчим для праці із ним, але, натомість, дозволяє інші оптимізації, такі як швидше встановлення з'єднання.

TCP ніколи не створювався для передачі кількох незалежних файлів через одне з'єднання. Оскільки це саме те, що потрібно для веб-перегляду, це призвело до багатьох неефективностей протягом багатьох років. QUIC вирішує цю проблему, роблячи багато байтові потоки основною концепцією на транспортному рівні та обробляючи втрату пакетів на основі кожного потоку.

У TCP підключення визначаються чотирма параметрами, які можуть змінюватися, коли кінцеві точки змінюють мережу. Таким чином, ці підключення іноді потрібно перезапускати, що призводить до деякого простою. QUIC додає ще один параметр, який називається ідентифікатором підключення. І клієнт, і сервер QUIC знають, які ідентифікатори з'єднань відображаються на які з'єднання, і тому є більш стійкими до змін мережі.

Хоча QUIC тепер стандартизовано, його дійсно слід розглядати як QUIC версії 1 (що також чітко зазначено в запиті на коментарі (RFC)), і існує чіткий намір досить швидко створити версію 2 і більше. Крім того, QUIC дозволяє легко визначати розширення, тому можна реалізувати ще більше випадків використання.

QUIC і HTTP/3 мають великий потенціал веб-продуктивності, але головним чином для користувачів у повільних мережах. Якщо пересічний відвідувач використовує швидкісну кабельну або стільникову мережу, він, ймовірно, не отримає особливої користі від нових протоколів. Однак навіть у країнах і регіонах із зазвичай швидкими висхідними каналами найповільніші від 1% до навіть 10% від загальної аудиторії (так званий 99-й або 90-й перцентиль) все ще потенційно можуть отримати значний вигреш. Це пов'язано з тим, що HTTP/3 і QUIC головним чином допомагають вирішити

дещо незвичайні, але потенційно серйозні проблеми, які можуть виникнути в сучасному Інтернеті.

Швидше встановлення з'єднання QUIC за допомогою 0-RTT насправді є радше мікрооптимізацією, ніж новою революційною функцією. Порівняно з найсучаснішим налаштуванням TCP + TLS 1.3, це заощадить щонайбільше одну «поїздку» туди й назад. Обсяг даних, який фактично можна надіслати під час першого зворотного зв'язку, додатково обмежується низкою міркувань безпеки.

Таким чином, ця функція здебільшого сяятиме, якщо користувачі працюють у мережах із дуже високою затримкою (скажімо, супутникові мережі з RTT понад 200 мілісекунд), або якщо користувач зазвичай не надсилає багато даних. Деякими прикладами останнього є веб-сайти з інтенсивним кешуванням, а також односторінкові програми, які періодично отримують невеликі оновлення через API та інші протоколи, такі як DNS-over-QUIC.

HTTP/3 і QUIC є складними протоколами, які покладаються на багато внутрішніх механізмів. Інструменти для тестування HTTP/3: Google Lighthouse, Webpagetest, Wireshark.

РОЗДІЛ 3 РОЗРОБКА ДОДАТКА НА ЗАСАДАХ HTTP/3

3.1. Опис бібліотек на Python, які реалізують HTTP/3.0

aiоquic - це бібліотека для мережевого протоколу QUIC на Python. Вона має мінімальну реалізацію TLS 1.3, стек QUIC і стек HTTP/3 [28, 17].

QUIC стандартизовано в RFC 9000, а HTTP/3 у RFC 9114. aiоquic регулярно перевіряється на взаємодію з іншими реалізаціями QUIC.

aiоquic розроблено для вбудовування в клієнтські та серверні бібліотеки Python, які бажають підтримувати QUIC та/або HTTP/3. Мета полягає в тому, щоб забезпечити загальну кодову базу для бібліотек Python, щоб уникнути дублювання зусиль.

І QUIC, і HTTP/3 API дотримуються шаблону "принесіть свій власний ввід-вивід", залишаючи фактичні операції вводу-виводу користувачеві API. Цей підхід має низку переваг, включаючи можливість тестування коду та можливість інтеграції з різними моделями паралелізму.

Особливості aiоquic

- стек QUIC, що відповідає RFC 9000;
- стек HTTP/3, що відповідає RFC 9114;
- мінімальна реалізація TLS 1.3, що відповідає RFC 8446;
- підтримка IPv4 і IPv6;
- міграція з'єднання та перезв'язування NAT;
- протоколювання секретів трафіку TLS;
- реєстрація подій QUIC у форматі QLOG.

aiоquic має мінімальну реалізацію TLS 1.3, побудовану на бібліотеці криптографії. Це пояснюється тим, що для QUIC потрібні деякі API, які наразі недоступні в основних реалізаціях TLS, таких як OpenSSL: можливість витягувати секрети трафіку; можливість працювати безпосередньо з повідомленнями TLS без використання рівня запису TLS.

QUIC широко використовує криптографічні операції для захисту заголовків пакетів QUIC і шифрування корисних даних пакетів. Ці операції виконуються для кожного окремого пакета і є визначальним фактором продуктивності. З цієї причини вони реалізовані як розширення C, пов'язане з OpenSSL.

QUIC API не виконує вводу-виводу самостійно, залишаючи це користувачеві API. Це дозволяє інтегрувати QUIC у будь-яку програму Python, незалежно від моделі паралелізму.

Connection машина станів керується трьома видами джерел: користувач API, який запитує дані для надсилання (`connect()`, `reset_stream()`, `send_ping()`, `send_datagram_frame()` і `send_stream_data()`); дані, отримані з мережі (`receive_datagram()`); спрацювання таймера (`handle_timer()`)

Configuration конфігурація QUIC. `alpn_protocols` список підтримуваних протоколів ALPN; `connection_id_length` довжина ідентифікаторів локального підключення в байтах і так далі більш докладно можливо дізнатися в [28]

Events базовий клас для подій QUIC. Подія `ConnectionTerminated` запускається, коли з'єднання QUIC припиняється. Подія `HandshakeCompleted` запускається, коли завершується рукошестикання TLS. Подія `StreamDataReceived` запускається щоразу, коли дані надходять у потік.

API HTTP/3 не виконує введення-виведення самостійно, залишаючи це користувачеві API. Це дозволяє інтегрувати HTTP/3 у будь-яку програму Python, незалежно від моделі паралелізму, яку використовує розробник, також має реалізацію `Connection`, `Configuration`, `Events` та `Exceptions` більш докладно в [17]

Для полегшення процесу впровадження HTTP/3.0 можливо використовувати сервера, наприклад `hypercorn`.

`Hypercorn` — це веб-сервер ASGI та WSGI, заснований на бібліотеках `sans-io hyper`, `h11`, `h2` і `wsproto`, натхненний `Gunicorn`. `Hypercorn` підтримує специфікації HTTP/1, HTTP/2, WebSockets (через HTTP/1 і HTTP/2), ASGI та WSGI. `Hypercorn` може використовувати робочі типи `asyncio`, `uvloop` або `trio`.

Hypercorn може додатково обслуговувати поточний проект специфікації HTTP/3 за допомогою бібліотеки aioquic. Щоб увімкнути це, необхідно встановити необов'язковий додаток h3, встановіть `pip hypercorn[h3]`, а потім виберіть швидко прив'язування, наприклад. `hypercorn --quic-bind localhost:4433` [19]

Hypercorn можна встановити через `pip`,

```
$ pip install hypercorn[h3]
```

і вимагає Python 3.7.0 або новішої версії.

Якщо встановлено Hypercorn, фреймворки (або програми) ASGI можна обслуговувати через Hypercorn через командний рядок,

```
$ hypercorn module:app
```

Крім того, Hypercorn можна використовувати програмно,

```
import asyncio
from hypercorn.config import Config
from hypercorn.asyncio import serve

from module import app

asyncio.run(serve(app, Config()))
```

3.2. Опис додатка побудованого на основі HTTP/3.0

Розглянемо приклад використання `hypercorn` для обробки запитів `bareASGI`. Оновлення до HTTP/3 є простим. Для цього потрібно: сервер ASGI, який підтримує (наприклад, Hypercorn); шифрування TLS.

Наступний код запускає веб-сервер, який підтримує:

http на порт 80

https на порт 888

https за допомогою http/3 на порт 8899

Браузер, який може перемикається на http/3, автоматично оновить з'єднання https.

```
import asyncio
import os
import logging

import bareutils.response_code as response_code
from bareasgi import Application, HttpResponse, text_writer
from hypercorn.asyncio import serve
from hypercorn.config import Config

logging.basicConfig(level=logging.DEBUG)

async def http_request_callback(request):
    t = """
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Простой сервер</title>
  </head>
  <body>
    <h1>Ви можете перевірити працю http3</h1>
  </body>
</html>
"""

    h = [
        (b'content-type', b'text/html'),
        (b'content-length', str(len(text)).encode())
    ]

    return HttpResponse(response_code.OK, h, text_writer(t))
```

```
app = Application()
app.http_router.add({'GET'}, '/', http_request_callback)

config = Config()
config.insecure_bind = [ '0.0.0.0:80' ]
config.bind = [ '0.0.0.0:888' ]
config.quic_bind = [ '0.0.0.0:8899' ]
config.keyfile = os.path.expanduser('~/.keys/ser.key')
config.certfile = os.path.expanduser('~/.keys/ser.crt')

asyncio.run(serve(app, config))
```

У цьому коді з початку імпортовано необхідні модулі `asyncio`, `logging`, `os` далі з `bareasgi` імпортовано об'єкти `Application`, `HttpResponse`, та функцію `text_writer`.

Із `hypercorn` імпортовано об'єкт для завдання конфігурації `Config`.

Далі вказали якій події потрібно логіровати, з допомогою директиви `logging.basicConfig(level=logging.DEBUG)`.

Далі створено функцію `http_request_callback`, яка повертає сторінку заглушку.

`app.http_router.add` зв'язує шлях `/` та метод `GET` із функцією `http_request_callback`.

За включення протоколу `HTTP/3.0` відповідають `config.quic_bind = ['0.0.0.0:8899']` `config.keyfile = os.path.expanduser('~/.keys/ser.key')`

`config.certfile = os.path.expanduser('~/.keys/ser.crt')`

Схожі підходи дозволяють обслуговувати будь-яку програму `ASGI` через `HTTP/1`, `HTTP/2` і `HTTP/3` без необхідності змінювати код. Це може бути `Django`, `FastAPI`, `Quart`, `BlackSheep`.

3.3. Висновки до розділу 3

Розглянуто `aioquic` — це бібліотека для мережевого протоколу QUIC на Python. Вона має мінімальну реалізацію TLS 1.3, стек QUIC і стек HTTP/3

Hypercorn — це веб-сервер ASGI та WSGI, заснований на бібліотеках `sans-io hyper`, `h11`, `h2` і `wsproto`, натхненний Gunicorn

Наведено приклад додатку на мові Python, який працює із HTTP/3.0

ВИСНОВКИ

У магістерському дослідженні про аналізовано HTTP/3.0.

Ця версія протоколу передачі гіпертексту (HTTP) знаходиться на завершальній стадії стандартизації IETF. Крім кращої безпеки та підвищеної гнучкості, HTTP/3 застосовує більш ефективну схему стиснення заголовків і замінює TCP на QUIC, транспортний протокол, що передається через UDP, спочатку запропонований Google і наразі також стандартизований. Також він надає кращий рівень безпеки, та певні переваги з точки зору продуктивності, особливо для випадків із високою затримкою або низькою пропускнуою здатністю.

В результаті виконання роботи були отримані наступні результати:

Виконано огляд історичного розвитку протоколу HTTP. На основі версії 1.1 проведено дослідження HTTP транзакцій, основних заголовків як запитів (request) так і відповідей (response), основні методи HTTP та архітектори WWW.

Досліджено особливості HTTP/2.0, які впливають на підвищення ефективності праці WWW та зроблено порівняльний аналіз першої та другої версії протоколу.

Зроблено огляд транспортного протоколу QUIC та його порівняння із TCP.

Визначено архітектурні особливості протоколу HTTP/3.0, які теоретично повинні покращувати працю WWW та надано практичні рекомендації по його розгортанню та використанню.

Наведено приклад додатку на мові Python, який працює із HTTP/3.0 із використання бібліотеки aioquic та серверу Hypercorn.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Andrews, C. The Sad Story of TCP Fast Open / C. Andrews. – 2019 – Режим доступу до ресурсу: <https://squeeze.isobar.com/2019/04/11/the-sad-story-of-tcp-fast-open/>.
2. Berners-Lee, T. Hypertext Transfer Protocol -- HTTP/1.0 (RFC 1945) / Т. Berners-Lee, R. Fielding, H. Frystyk: RFC Editor, 1996.
3. Berners-Lee, T. The Original HTTP as defined in 1991 / Т. Berners-Lee. – 1991 – Режим доступу до ресурсу: <https://www.w3.org/Protocols/HTTP/AsImplemented.html>.
4. Berners-Lee, T. Qualifiers on Hypertext links / Т. Berners-Lee. – 1991 – Режим доступу до ресурсу: <https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt>.
5. Bishop, M. HTTP/3 (RFC 9114) / M. Bishop: RFC Editor, 2022.
6. Denoyelle, A. Implementations of QUIC / A. Denoyelle. – 2022 – Режим доступу до ресурсу: <https://github.com/quicwg/base-drafts/wiki/Implementations>.
7. Eeway, E.H. Evaluating HTTP/1.1 and HTTP/2 Performance with Dependency Graph Properties / Eeway Erika Hsu; MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2017.
8. Fielding, R. HTTP Caching (RFC 9111) / R. Fielding, M. Nottingham, J. Reschke: RFC Editor, 2022.
9. Fielding, R. HTTP Semantics (RFC 9110) / R. Fielding, M. Nottingham, J. Reschke: RFC Editor, 2022.
10. Fielding, R. HTTP/1.1 (RFC 9112) / R. Fielding, M. Nottingham, J. Reschke: RFC Editor, 2022.
11. Fielding, R. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing (RFC 7230) / R. Fielding, J. Reschke: RFC Editor, 2014.
12. Gourley, D. HTTP / D. Gourley, B. Totty. – Beijing, Farnham: O'Reilly, 2002.

13. Grigorik, I. High-performance browser networking / I. Grigorik. – Beijing: O'Reilly, 2013.
14. Grigorik, I. Introduction to HTTP/2 / I. Grigorik. – 2019 – Режим доступа до ресурсу: <https://web.dev/performance-http2/>.
15. HTTP/2 protocol canisue.com . – 2022 – Режим доступа до ресурсу: <https://caniuse.com/http2>.
16. HTTP/3 – Режим доступа до ресурсу: <https://en.wikipedia.org/wiki/HTTP/3>.
17. HTTP/3 API . – 2022 – Режим доступа до ресурсу: <https://aioquic.readthedocs.io/en/latest/h3.html>.
18. HTTP/3 protocol canisue.com . – 2022 – Режим доступа до ресурсу: <https://caniuse.com/http3>.
19. Hypercorn . – 2022 – Режим доступа до ресурсу: <https://github.com/pgjones/hypercorn>.
20. Iyengar, J., et al. Sender Control of Acknowledgement Delays in QUIC / J. Iyengar, I. Swett. – 2020 – Режим доступа до ресурсу: <https://datatracker.ietf.org/doc/html/draft-iyengar-quic-delayed-ack-02>.
21. Iyengar, J. QUIC Loss Detection and Congestion Control (RFC9002) / J. Iyengar, I. Swett: RFC Editor, 2021.
22. Iyengar, J. QUIC: A UDP-Based Multiplexed and Secure Transport (RFC 9000) / J. Iyengar, M. Thomson: RFC Editor, 2021.
23. Krsic, C. QPACK: Field Compression for HTTP/3 (RFC 9204) / C. Krsic, M. Bishop, A. Frindell: RFC Editor, 2022.
24. Ludin, S. Learning HTTP/2 / S. Ludin, J. Garza. – Beijing: O'Reilly, 2017.
25. Same Standards, Different Decisions / R. Marx, J. Herbots, W. Lamotte, P. Quax // SIGCOMM '20: Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC. – New York, NY, USA: ACM, 08102020. – C.14–20.

26. Mishra, A. Understanding speciation in QUIC congestion control / A. Mishra, S. Lim, B. Leong // IMC '22: ACM Internet Measurement Conference: Proceedings of the 22nd ACM Internet Measurement Conference. – New York, NY, USA: ACM, 10252022. – C.560–566.
27. Nottingham, M. Opportunistic Security for HTTP/2 (RFC 8164) / M. Nottingham, M. Thomson: RFC Editor, 2017.
28. Open Source. aioquic . – 2022 – Режим доступа до ресурсу: <https://github.com/aiohttp/aioquic>.
29. A first look at HTTP/3 adoption and performance / G. Perna, M. Trevisan, D. Giordano, I. Drago // Computer Communications. – 2022. – Т.187. – С.115–124.
30. Pollard, B. HTTP/2 in Action / B. Pollard. – Shelter Island, NY: Manning, 2019.
31. Q-Success. Usage statistics of HTTP/2 for websites . – 2022 – Режим доступа до ресурсу: <https://w3techs.com/technologies/details/ce-http2>.
32. Q-Success. Usage statistics of HTTP/3 for websites . – 2022 – Режим доступа до ресурсу: <https://w3techs.com/technologies/details/ce-http3>.
33. QUIC – Режим доступа до ресурсу: <https://en.wikipedia.org/wiki/QUIC>.
34. Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3 (RFC 8446) / E. Rescorla: RFC Editor, 2018.
35. Stenberg, D. http/2 explained / D. Stenberg, 2017.
36. Stenberg, D. HTTP/3 explained / D. Stenberg, 2018.
37. Stewart, R. Stream Control Transmission Protocol (RFC 9260) / R. Stewart, M. Tüxen, K. Nielsen: RFC Editor, 2022.
38. Tellakula, S. Comparing HTTP/3 vs. HTTP/2 Performance / S. Tellakula. – 2020 – Режим доступа до ресурсу: <https://blog.cloudflare.com/http-3-vs-http-2/>.
39. UCLouvain. Linux Kernel MultiPath TCP project . – 2020 – Режим доступа до ресурсу: <https://www.multipath-tcp.org/>.

40. Wojtczak, B. Performance and Efficiency: Working with HTTP/3 / B. Wojtczak. – 2022 – Режим доступа до ресурсу: <https://www.toptal.com/web/performance-working-with-http-3>.
41. Wong, C. HTTP pocket reference / C. Wong. – Sebastopol CA: O'Reilly, 2000. – iii, 75.

ДОДАТОК А

Повний список кодів стану HTTP Complete list of HTTP Status Codes

<https://umbraco.com/knowledge-base/http-status-codes/>

Status code	Meaning
1xx Informational	
100	Continue
101	Switching protocols
102	Processing
103	Early Hints
2xx Successful	
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used
3xx Redirection	
300	Multiple Choices
301	Moved Permanently
302	Found (Previously "Moved Temporarily")
303	See Other
304	Not Modified
305	Use Proxy
306	Switch Proxy
307	Temporary Redirect
308	Permanent Redirect

4xx Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
418	I'm a Teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
425	Too Early
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
451	Unavailable For Legal Reasons
5xx Server Error	
500	Internal Server Error

501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required

ДОДАТОК Б

Візуалізація змін меж різними версіями HTTP.

