

ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО
ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«МІЖРЕГІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ»
ФАКУЛЬТЕТ КОМП'ЮТЕРНО-ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Грицак Олександр Ігорович

ДИПЛОМНА РОБОТА

На тему: «Розробка клієнт-серверної системи обміну повідомленнями з
використанням асинхронних технологій»

Шифр групи: ІК-9-22-Б1ПЗ(4,0д)
Спеціальність – F2 «Інженерія програмного забезпечення»

Робота на здобуття освітньо-кваліфікаційного рівня

Бакалавр

Науковий керівник: _____ ст.викл. Коваль А.О

(підпис)

Допущено до захисту ДЕК

Завідувач кафедри _____ професор, д.т.н Кавун С.В

Київ – 2026 р.

РЕФЕРАТ

Тема: Розробка клієнт-серверної системи обміну повідомленнями з використанням асинхронних технологій. Структура і обсяг роботи: бакалаврська робота викладена на 78 сторінках, складається зі вступу, 4 розділів, 12 підрозділів, списку використаної літератури.

У сучасних умовах стрімкого розвитку інформаційних технологій та зростання ролі розподілених програмних систем особливої актуальності набувають клієнт-серверні системи обміну повідомленнями. Такі системи є основою для функціонування соціальних мереж, корпоративних чатів, сервісів підтримки користувачів, фінансових та хмарних сервісів. Зі збільшенням кількості користувачів і обсягу переданих даних зростають вимоги до продуктивності, масштабованості та надійності програмних рішень.

Традиційні синхронні підходи до організації взаємодії між клієнтом і сервером дедалі частіше виявляються недостатньо ефективними, оскільки призводять до блокування потоків виконання, зниження швидкодії та погіршення користувацького досвіду. У зв'язку з цим все ширше застосовуються асинхронні технології, які дозволяють обробляти велику кількість одночасних запитів, оптимально використовувати системні ресурси та забезпечувати високу швидкість обміну повідомленнями в реальному часі.

Актуальність даної теми також зумовлена практичною потребою у створенні сучасних, безпечних та масштабованих систем обміну повідомленнями, які можуть ефективно працювати в умовах високого навантаження. Дослідження та реалізація клієнт-серверної системи з використанням асинхронних технологій дозволяє набути практичних навичок проєктування розподілених систем, а також сприяє формуванню рішень, що відповідають сучасним тенденціям розвитку програмного забезпечення.

ЗМІСТ

РЕФЕРАТ	2
ВСТУП	4
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ ТЕХНОЛОГІЙ ОБМІНУ ПОВІДОМЛЕННЯМИ	6
1.1. Поняття та принципи клієнт-серверної архітектури	8
1.2. Огляд систем обміну повідомленнями та їх класифікація	10
1.3. Аналіз протоколів та підходів обміну даними	12
РОЗДІЛ 2. ПРОЄКТУВАННЯ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ ОБМІНУ ПОВІДОМЛЕННЯМИ	15
2.1. Вибір інструментальних засобів і технологій розробки	16
2.2. Загальна архітектура системи	18
2.3. Проєктування серверної частини системи	19
2.4. Проєктування клієнтської частини системи	23
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ	25
3.1. Реалізація серверної частини з використанням асинхронних технологій	26
3.2. Реалізація клієнтського застосунку	27
3.3. Організація асинхронного обміну повідомленнями	29
РОЗДІЛ 4. ЕКОНОМІЧНИЙ АБО ПРАКТИЧНИЙ АНАЛІЗ	40
4.1. Оцінка ефективності розробленої системи	42
4.2. Аналіз витрат на розробку програмного продукту	44
4.3. Розрахунок економічної доцільності впровадження	56
ВИСНОВКИ	59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	61
ДОДАТКИ	63

ВСТУП

Сучасні інформаційні системи все частіше будуються за клієнт-серверною архітектурою та потребують ефективних механізмів обміну повідомленнями між користувачами і серверними компонентами. Системи обміну повідомленнями широко застосовуються у веб-застосунках, мобільних сервісах, корпоративних інформаційних системах та хмарних платформах. У процесі їх використання ключовими вимогами є стабільність роботи, швидкість передачі даних та можливість обробки значної кількості одночасних запитів.

У умовах зростання навантаження на серверні системи традиційні синхронні підходи до обробки запитів стають малоефективними, оскільки вони призводять до блокування виконання операцій і нераціонального використання ресурсів. Асинхронні технології дозволяють організувати неблокуючу обробку повідомлень, що забезпечує підвищення продуктивності системи та покращення якості взаємодії користувачів із сервісом. Завдяки цьому асинхронні механізми обміну даними активно використовуються у сучасних клієнт-серверних рішеннях.

Метою даної бакалаврської роботи є розробка клієнт-серверної системи обміну повідомленнями з використанням асинхронних технологій, яка здатна забезпечити ефективну та стабільну взаємодію між клієнтами та сервером. У межах роботи розглядаються основні принципи побудови клієнт-серверних систем, особливості реалізації асинхронного обміну повідомленнями, а також питання проєктування та реалізації програмних компонентів системи.

Об'єктом дослідження є процес обміну повідомленнями у клієнт-серверних інформаційних системах. Предметом дослідження є методи та засоби розробки клієнт-серверної системи обміну повідомленнями з використанням асинхронних технологій. Практичне значення роботи полягає у створенні працюючого програмного рішення, яке може бути використане як

основа для подальшого розвитку реальних сервісів обміну повідомленнями або в навчальних цілях.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ ТЕХНОЛОГІЙ ОБМІНУ ПОВІДОМЛЕННЯМИ

Клієнт-серверна архітектура є одним із найбільш поширених підходів до побудови сучасних інформаційних систем. Вона передбачає розподіл функціональності між клієнтською частиною, яка відповідає за взаємодію з користувачем, та серверною частиною, що забезпечує обробку запитів, зберігання даних і бізнес-логіку застосунку. Такий підхід дозволяє централізовано керувати даними, спрощує супровід системи та забезпечує можливість масштабування. У рамках клієнт-серверної взаємодії обмін інформацією здійснюється за допомогою мережевих протоколів, що регламентують формат і порядок передачі даних. Ефективність роботи системи значною мірою залежить від організації цього обміну, зокрема від способу обробки запитів і повідомлень, що надходять від клієнтів. Модель клієнт-сервер — це форма шаблону обміну повідомленнями в розподіленій структурі застосунків, яка розподіляє завдання або робочі навантаження між постачальниками ресурсу або послуги, які називаються серверами, та запитувачами послуг, які називаються клієнтами.[1] Часто клієнти та сервери взаємодіють через комп'ютерну мережу на окремому обладнанні, але і клієнт, і сервер можуть знаходитися на одному пристрої. Хост сервера запускає одну або кілька серверних програм, які діляться своїми ресурсами з клієнтами. Клієнт зазвичай не надає спільний доступ до своїх обчислювальних ресурсів, але запитує контент або послугу з сервера та може ділитися власним контентом як частиною запиту. Таким чином, клієнти ініціюють сеанси зв'язку із серверами, які очікують вхідних запитів. Прикладами комп'ютерних застосунків, що використовують модель клієнт-сервер, є електронна пошта, мережевий друк та Всесвітня мережа. Серверний компонент надає функцію або послугу одному або багатьом клієнтам, які ініціюють запити на такі послуги. Сервери класифікуються за послугами, які вони надають. Наприклад, веб-сервер обслуговує веб-сторінки, а файловий сервер

обслуговує файли комп'ютера. Спільним ресурсом може бути будь-яке програмне забезпечення та електронні компоненти серверного комп'ютера, від програм і даних до процесорів і пристроїв зберігання даних. Спільне використання ресурсів сервера становить послугу.

Те, чи є комп'ютер клієнтом, сервером або обома, визначається характером програми, яка потребує функцій послуги. Наприклад, один комп'ютер може одночасно запускати програмне забезпечення веб-сервера та файлового сервера для обслуговування різних даних клієнтів, які роблять різні типи запитів. Клієнтське програмне забезпечення також може взаємодіяти з серверним програмним забезпеченням в межах одного комп'ютера.[2] Зв'язок між серверами, наприклад, для синхронізації даних, іноді називають міжсерверним або міжсерверним зв'язком.

Зазвичай, сервіс – це абстракція комп'ютерних ресурсів, і клієнту не потрібно турбуватися про те, як сервер виконує запит і надає відповідь. Клієнту потрібно лише розуміти відповідь на основі відповідного протоколу програми, тобто вміст і форматування даних для запитуваного сервісу.

Клієнти та сервери обмінюються повідомленнями за шаблоном запит-відповідь. Клієнт надсилає запит, а сервер повертає відповідь. Цей обмін повідомленнями є прикладом міжпроцесної комунікації. Для спілкування комп'ютери повинні мати спільну мову та дотримуватися правил, щоб і клієнт, і сервер знали, чого очікувати. Мова та правила спілкування визначені в протоколі зв'язку. Усі протоколи працюють на рівні програми. Протокол рівня програми визначає основні шаблони діалогу. Щоб ще більше формалізувати обмін даними, сервер може реалізувати інтерфейс прикладного програмування (API).[3] API – це рівень абстракції для доступу до сервісу. Обмежуючи спілкування певним форматом контенту, він полегшує парсинг. Абстрагуючи доступ, він полегшує міжплатформний обмін даними.[4]

Сервер може отримувати запити від багатьох різних клієнтів за короткий проміжок часу. Комп'ютер може виконувати лише обмежену кількість завдань у будь-який момент часу та покладається на систему планування, щоб визначити пріоритети вхідних запитів від клієнтів для їх обробки. Щоб запобігти зловживанням та максимізувати доступність, програмне забезпечення сервера може обмежити доступність для клієнтів. Атаки типу "відмова в обслуговуванні" розроблені для використання зобов'язання сервера обробляти запити шляхом перевантаження його надмірною кількістю запитів. Шифрування слід застосовувати, якщо між клієнтом і сервером має передаватись конфіденційна інформація.

1.1. Поняття та принципи клієнт-серверної архітектури

Клієнт-серверна архітектура є однією з основних моделей побудови розподілених інформаційних систем та широко використовується у сучасному програмному забезпеченні. В її основі лежить поділ системи на незалежні, але взаємопов'язані компоненти, які взаємодіють між собою через мережу. Така архітектура передбачає наявність клієнта, який ініціює запити, та сервера, що здійснює їх обробку і надає відповідні послуги. Поділ функціональності між клієнтською та серверною частинами дозволяє ефективно організувати обробку даних і керування ресурсами.

Клієнт у межах клієнт-серверної архітектури відповідає за підготовку та надсилання запитів, а також за відображення отриманої від сервера інформації користувачеві. Клієнтські застосунки можуть бути реалізовані у вигляді настільних програм, мобільних застосунків або веб-інтерфейсів. Серверна частина системи виконує центральну роль у обробці запитів, реалізації бізнес-логіки, управлінні доступом до даних і забезпеченні цілісності інформації. Саме сервер обробляє повідомлення, що надходять від клієнтів, і формує відповіді відповідно до встановлених правил та алгоритмів.

Одним із ключових принципів клієнт-серверної архітектури є централізація даних і логіки обробки. Такий підхід дозволяє зберігати критично важливу інформацію в одному місці, що спрощує її захист, резервне копіювання та супровід. При цьому клієнтські компоненти залишаються відносно простими, оскільки не містять складної логіки обробки або прямого доступу до баз даних. Це знижує ризик несанкціонованого доступу та помилок на стороні клієнта.

Взаємодія між клієнтом і сервером здійснюється шляхом обміну повідомленнями через мережеві протоколи. Клієнт надсилає серверу запит, що містить необхідні дані або команду, після чого сервер аналізує отриману інформацію, виконує відповідні дії та повертає результат. Така модель взаємодії є універсальною і дозволяє реалізовувати різні типи програмних систем, включаючи системи обміну повідомленнями, у яких швидкість і стабільність передавання даних мають важливе значення.

Важливою характеристикою клієнт-серверної архітектури є її масштабованість. За необхідності система може бути розширена шляхом збільшення потужності серверного обладнання або додавання нових серверів без суттєвих змін у клієнтській частині. Це дозволяє адаптувати систему до зростання кількості користувачів і обсягів даних, що особливо важливо для систем обміну повідомленнями з високою інтенсивністю запитів.

Клієнт-серверна архітектура також створює передумови для використання асинхронних підходів до обробки запитів. Асинхронна взаємодія дозволяє серверу приймати і обробляти нові повідомлення, не очікуючи завершення попередніх операцій, що підвищує продуктивність і зменшує затримки. У контексті систем обміну повідомленнями це забезпечує більш стабільну роботу та кращий користувацький досвід, особливо за умов одночасної роботи великої кількості клієнтів.

Таким чином, клієнт-серверна архітектура є надійною та гнучкою основою для побудови сучасних систем обміну повідомленнями. Її принципи дозволяють реалізовувати ефективну взаємодію між компонентами системи, а поєднання з асинхронними технологіями створює передумови для розробки продуктивних і масштабованих програмних рішень.

1.2. Огляд систем обміну повідомленнями та їх класифікація

Системи обміну повідомленнями займають важливе місце серед сучасних інформаційних систем, оскільки забезпечують передачу даних між користувачами або програмними компонентами в мережевому середовищі. Основним призначенням таких систем є організація надійного, швидкого та впорядкованого обміну інформацією незалежно від фізичного розташування учасників взаємодії. Залежно від призначення та особливостей реалізації системи обміну повідомленнями можуть використовуватися як для безпосередньої комунікації між людьми, так і для взаємодії між програмними сервісами.

Однією з основних характеристик систем обміну повідомленнями є спосіб взаємодії між учасниками. У простіших реалізаціях обмін повідомленнями здійснюється за принципом прямого надсилання запитів і відповідей між клієнтом і сервером. У таких системах кожне повідомлення обробляється окремо, а клієнт зазвичай очікує підтвердження доставки або відповіді від сервера. Цей підхід є достатнім для невеликих систем, однак в умовах високого навантаження він може призводити до зростання затримок та зниження продуктивності.

Більш складні системи обміну повідомленнями використовують проміжні механізми зберігання та маршрутизації повідомлень. У таких системах повідомлення надсилаються до сервера або спеціального компонента, де вони тимчасово зберігаються та передаються адресатам у відповідний

момент часу. Це дозволяє роз'єднати процеси надсилання та отримання повідомлень, що підвищує надійність системи та забезпечує можливість обробки повідомлень навіть у разі тимчасової недоступності клієнта.

З точки зору режиму роботи системи обміну повідомленнями можна виділити рішення, що працюють у реальному часі, та системи з відкладеною доставкою. У першому випадку повідомлення доставляються отримувачам одразу після їх надсилання, що характерно для чатів і систем миттєвих повідомлень. У другому випадку допускається затримка між моментом надсилання і отримання повідомлення, що є типовим для поштових сервісів або систем сповіщень. Вибір конкретного підходу залежить від вимог до швидкодії та характеру інформації, що передається.

Класифікація систем обміну повідомленнями також може здійснюватися за архітектурними особливостями. Найпоширенішими є клієнт-серверні рішення, у яких сервер виступає центральним елементом, що координує обмін повідомленнями між клієнтами. Такі системи прості в реалізації та супроводі, однак потребують надійного серверного компонента.

Альтернативним підходом є розподілені системи, у яких обмін повідомленнями здійснюється без єдиного центра керування, що підвищує стійкість до відмов, але ускладнює реалізацію.

Важливим аспектом класифікації систем обміну повідомленнями є спосіб обробки запитів. Синхронні системи передбачають обробку повідомлення з блокуванням виконання до отримання відповіді, тоді як асинхронні системи дозволяють надсилати та обробляти повідомлення незалежно від часу їх отримання. Асинхронний підхід є більш придатним для сучасних клієнт-серверних застосунків, оскільки забезпечує кращу масштабованість та ефективніше використання ресурсів.

Таким чином, системи обміну повідомленнями можуть бути класифіковані за різними ознаками, включаючи режим роботи, архітектуру та спосіб обробки

повідомлень. Аналіз існуючих підходів показує, що для реалізації сучасних клієнт-серверних застосунків доцільно використовувати асинхронні механізми обміну даними, які дозволяють забезпечити високу продуктивність і стабільність роботи системи за умов інтенсивного навантаження.

1.3. Аналіз протоколів та підходів обміну даними

Обмін даними є ключовим елементом функціонування клієнт-серверних систем, оскільки саме від вибору протоколів і підходів до передачі інформації залежить продуктивність, надійність та масштабованість програмного рішення. У системах обміну повідомленнями протоколи визначають правила встановлення з'єднання, формат передавання даних і порядок взаємодії між клієнтом і сервером.

Одним із найпоширеніших протоколів у клієнт-серверних застосунках є HTTP, який широко використовується у веб-системах. Його основною особливістю є модель взаємодії «запит–відповідь», за якої клієнт ініціює обмін, а сервер повертає результат обробки. Такий підхід є простим у реалізації та добре підтримується більшістю платформ, проте класична реалізація HTTP не призначена для постійного двостороннього обміну повідомленнями в реальному часі, що обмежує його ефективність для систем миттєвої комунікації.

Для подолання цих обмежень застосовуються підходи, які забезпечують більш тісну взаємодію між клієнтом і сервером. Одним із таких підходів є використання постійних мережевих з'єднань, що дозволяють серверу ініціювати передачу даних клієнту без необхідності надсилання окремого запиту з його боку. Такий механізм істотно зменшує затримки під час обміну повідомленнями та забезпечує більш оперативну доставку даних, що є критично важливим для чатів і систем сповіщень.

Важливу роль у системах обміну повідомленнями відіграє формат передаваних даних. З метою забезпечення сумісності між різними компонентами системи широко використовуються текстові формати, які легко обробляються на стороні клієнта й сервера. Використання структурованих форматів дозволяє передавати не лише текстові повідомлення, але й службову інформацію, таку як ідентифікатори користувачів, часові мітки та статуси доставки. Це сприяє підвищенню функціональності та зручності використання системи.

З точки зору організації обміну даними важливим є вибір між синхронним та асинхронним підходами. Синхронний обмін передбачає очікування завершення операції перед продовженням виконання програми, що може призводити до блокування та зниження продуктивності. Асинхронний підхід дозволяє надсилати повідомлення та обробляти відповіді незалежно від основного потоку виконання, що забезпечує кращу реактивність системи та ефективніше використання ресурсів. У сучасних клієнт-серверних системах асинхронний обмін даними є пріоритетним, особливо за умов великої кількості одночасних підключень.

Окрему увагу слід приділити питанням надійності та цілісності переданих даних. Протоколи обміну повинні забезпечувати коректну доставку повідомлень, обробку помилок і, за необхідності, повторну передачу даних. Це особливо важливо для систем, у яких втрата повідомлень може призвести до порушення логіки роботи або зниження якості сервісу.

Таким чином, аналіз протоколів та підходів обміну даними показує, що для сучасних систем обміну повідомленнями найбільш доцільним є поєднання клієнт-серверної архітектури з асинхронними механізмами передачі даних і ефективними мережевими протоколами. Такий підхід забезпечує високу продуктивність, масштабованість і стабільність роботи системи, що є основою для подальшої розробки та вдосконалення програмного продукту.

РОЗДІЛ 2. ПРОЄКТУВАННЯ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ ОБМІНУ ПОВІДОМЛЕННЯМИ

Проектування клієнт-серверної системи обміну повідомленнями є важливим етапом розробки програмного забезпечення, оскільки саме на цьому етапі визначається загальна структура системи, взаємодія її компонентів та основні принципи роботи. Якісне проектування дозволяє забезпечити відповідність системи поставленим вимогам, спростити процес реалізації та підвищити надійність і масштабованість готового програмного продукту.

У межах даного розділу розглядається архітектурна модель клієнт-серверної системи обміну повідомленнями, яка використовує асинхронні механізми взаємодії між компонентами. Особливу увагу приділено визначенню ролей клієнтської та серверної частин, організації обміну даними, а також вибору логічної структури системи. Проектування здійснюється з урахуванням вимог до продуктивності, можливості одночасної роботи декількох користувачів і забезпечення стабільної передачі повідомлень.

На етапі проектування визначаються основні функціональні можливості системи, включаючи процес надсилання, отримання та зберігання повідомлень. Архітектура системи передбачає чітке розмежування відповідальності між клієнтською та серверною частинами, що відповідає принципам клієнт-серверної моделі. Клієнтська частина орієнтована на взаємодію з користувачем і формування запитів, тоді як серверна частина виконує обробку повідомлень, керування підключеннями та обмін даними між клієнтами.

Важливим аспектом проектування є організація асинхронного обміну повідомленнями, що дозволяє зменшити затримки та підвищити загальну продуктивність системи. Асинхронний підхід забезпечує неблокуючу обробку запитів і дозволяє серверу одночасно працювати з великою кількістю клієнтів.

Це особливо актуально для систем обміну повідомленнями, у яких можливе одночасне надходження значної кількості повідомлень.

У даному розділі також розглядається логічна структура даних і взаємодія між основними компонентами системи. Описуються зв'язки між клієнтами та сервером, а також порядок передавання повідомлень у межах системи. Отримані результати проєктування є основою для подальшої реалізації клієнт-серверної системи, яка буде детально розглянута у наступному розділі.

2.1. Вибір інструментальних засобів і технологій розробки

Для реалізації клієнт-серверної системи обміну повідомленнями доцільно використовувати сучасні програмні технології, які забезпечують підтримку асинхронної обробки запитів, стабільну роботу з мережевими з'єднаннями та можливість подальшого масштабування системи. Однією з таких технологій є платформа Node.js, яка активно використовується для створення серверних застосунків із високими вимогами до продуктивності. В основі Node.js лежить подієво-орієнтована модель виконання з неблокуючим введенням-виведенням, що дозволяє ефективно обробляти велику кількість одночасних підключень, характерних для систем обміну повідомленнями.

Альтернативним варіантом для реалізації серверної частини є використання мови програмування Python із підтримкою асинхронних бібліотек. Сучасні асинхронні механізми Python дозволяють організувати неблокуючу обробку мережових запитів і забезпечити логічно зрозумілу структуру серверного коду. Завдяки простому синтаксису та широкому набору бібліотек Python є зручним інструментом для розробки серверних компонентів, що виконують обробку повідомлень і керують взаємодією клієнтів.

Також для побудови клієнт-серверних систем обміну повідомленнями широко застосовується мова програмування Java, яка забезпечує високу продуктивність і надійність виконання. Використання асинхронних підходів у

Java дозволяє реалізувати масштабовані серверні рішення, здатні працювати під значним навантаженням. Перевагою Java є сувора типізація та стабільність платформи, що є важливими для розробки серверів із тривалим життєвим циклом.

Для забезпечення оперативного двостороннього обміну повідомленнями між клієнтом і сервером доцільно використовувати технологію WebSocket. На відміну від класичної моделі HTTP-взаємодії, WebSocket дозволяє встановити постійне з'єднання між клієнтом і сервером, через яке дані можуть передаватися в обох напрямках без додаткових запитів. Це суттєво зменшує затримки та забезпечує передачу повідомлень у режимі, близькому до реального часу, що є критично важливим для систем чатів і миттєвих повідомлень.

Клієнтська частина системи може бути реалізована з використанням веб-технологій або настільного застосунку, який підтримує асинхронну взаємодію з сервером через WebSocket-з'єднання. Асинхронна модель на стороні клієнта дозволяє забезпечити швидкий відгук інтерфейсу користувача та безперервне отримання повідомлень без перезавантаження сторінки або повторного встановлення з'єднання.

Таким чином, використання таких технологій, як Node.js, Python або Java у поєднанні з WebSocket, дозволяє реалізувати ефективну клієнт-серверну систему обміну повідомленнями з асинхронною обробкою даних. Обрані інструментальні засоби забезпечують високу продуктивність, масштабованість і зручність подальшого розвитку системи, що повністю відповідає вимогам до сучасних програмних рішень у сфері обміну повідомленнями.

2.2. Загальна архітектура системи

Загальна архітектура розроблюваної системи обміну повідомленнями ґрунтується на клієнт-серверній моделі з використанням асинхронних механізмів взаємодії. Такий архітектурний підхід дозволяє чітко розмежувати функціональність між окремими компонентами системи, забезпечити ефективний обмін даними та підвищити масштабованість і стабільність роботи застосунку.

Центральним елементом системи є серверна частина, яка відповідає за керування клієнтськими підключеннями, приймання та обробку повідомлень, а також їх подальшу передачу між учасниками обміну. Сервер виступає єдиною точкою координації, через яку проходить весь потік даних. Завдяки використанню асинхронної моделі обробки запитів сервер здатний одночасно підтримувати велику кількість активних з'єднань без блокування основного потоку виконання, що є критично важливим для систем обміну повідомленнями.

Клієнтська частина системи призначена для взаємодії користувача з програмним продуктом і забезпечує формування та надсилання повідомлень, а також отримання даних від сервера. Клієнтські застосунки встановлюють постійне з'єднання із сервером, через яке здійснюється двосторонній обмін інформацією в режимі близькому до реального часу. Асинхронна взаємодія на стороні клієнта дозволяє отримувати нові повідомлення без переривання роботи користувача з інтерфейсом.

Взаємодія між клієнтом і сервером реалізується через мережевий рівень, який забезпечує передачу структурованих повідомлень у визначеному форматі. Повідомлення містять як безпосередньо текстову інформацію, так і службові дані, необхідні для ідентифікації відправника, отримувача та контролю стану

доставки. Такий підхід дозволяє забезпечити логічну цілісність даних і спрощує їх обробку на серверній стороні.

Архітектура системи передбачає логічне розділення функцій обробки повідомлень, керування підключеннями та взаємодії з клієнтами. Це спрощує супровід програмного забезпечення та створює передумови для подальшого розширення функціональних можливостей, зокрема впровадження механізмів збереження повідомлень, авторизації користувачів або масштабування серверної частини.

Використання асинхронних технологій у межах обраної архітектури дозволяє мінімізувати затримки під час обміну повідомленнями та забезпечити стабільну роботу системи за умов одночасної активності багатьох користувачів. Таким чином, загальна архітектура розроблюваної клієнт-серверної системи створює надійну основу для реалізації ефективного сервісу обміну повідомленнями та відповідає сучасним вимогам до розподілених програмних рішень.

2.3. Проєктування серверної частини системи

Проєктування серверної частини є ключовим етапом створення клієнт-серверної системи обміну повідомленнями, оскільки саме сервер забезпечує координацію взаємодії між клієнтами, обробку повідомлень і підтримку стабільної роботи системи в цілому. Серверна частина проєктується з урахуванням вимог до асинхронної обробки даних, високої продуктивності та можливості одночасної роботи значної кількості користувачів.

Архітектура серверної частини будується на подієво-орієнтованій моделі, яка дозволяє обробляти мережеві події без блокування основного процесу виконання. Кожне клієнтське підключення розглядається як окремий асинхронний канал взаємодії, через який сервер приймає та надсилає

повідомлення. Такий підхід забезпечує ефективне використання системних ресурсів і зменшує затримки під час обміну даними.

Сервер виконує функції керування підключеннями клієнтів, включаючи встановлення з'єднання, підтримку активного стану та завершення сеансів у разі розриву зв'язку. Після підключення клієнта сервер реєструє його у внутрішній структурі даних, що дозволяє ідентифікувати відправника і коректно маршрутизувати повідомлення між учасниками обміну. Проектування таким чином забезпечує можливість передачі повідомлень як між окремими користувачами, так і в межах групових чатів.

Обробка повідомлень на сервері здійснюється асинхронно. Отримавши повідомлення від клієнта, сервер виконує його первинну перевірку, визначає адресата та передає дані відповідному клієнтському підключенню. Асинхронна модель дозволяє серверу приймати нові повідомлення незалежно від стану попередніх операцій, що є особливо важливим за умов високого навантаження.

Під час проектування серверної частини передбачається логічне розділення функцій, пов'язаних із мережею, обробкою повідомлень і керуванням станом системи. Така структура спрощує супровід програмного коду та забезпечує можливість подальшого розширення функціональності, зокрема впровадження збереження повідомлень, авторизації користувачів або ведення журналу подій.

Окрема увага приділяється забезпеченню надійності роботи серверної частини. Сервер повинен коректно обробляти помилки з'єднання, втрату підключень і некоректні дані, не впливаючи на роботу інших клієнтів. Асинхронний підхід у поєднанні з централізованою обробкою подій дозволяє локалізувати помилки та запобігти зупинці всієї системи.

Таким чином, проектування серверної частини клієнт-серверної системи обміну повідомленнями передбачає створення асинхронного,

подієво-орієнтованого компонента, здатного ефективно обробляти велику кількість з'єднань і повідомлень. Запропоновані архітектурні рішення створюють надійну основу для реалізації серверної логіки системи та забезпечують її стабільну і масштабовану роботу.

Серверна частина відповідає за приймання клієнтських з'єднань, обробку повідомлень і маршрутизацію даних між клієнтами. Сервер працює в асинхронному режимі та використовує подієво-орієнтовану модель виконання.

Ініціалізація WebSocket-сервера

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

const server = new WebSocket.Server({ port: 8080 });

const clients = new Map();

console.log('WebSocket сервер запущено на порту 8080');
```

У наведеному фрагменті створюється WebSocket-сервер, який прослуховує порт 8080. Для зберігання підключених клієнтів використовується структура даних Map, що дозволяє швидко ідентифікувати активні з'єднання.

Обробка підключення клієнта

```
server.on('connection', (socket) => {

  const clientId = generateClientId();

  clients.set(clientId, socket);

  console.log(`Клієнт підключився: ${clientId}`);

  socket.on('message', (message) => {

    handleMessage(clientId, message);

  });

});
```

```

socket.on('close', () => {

  clients.delete(clientId);

  console.log(`Клієнт відключився: ${clientId}`);

});

```

Під час встановлення з'єднання сервер генерує унікальний ідентифікатор клієнта та реєструє його. Обробка подій message і close відбувається асинхронно, без блокування виконання інших з'єднань.

Генерація ідентифікатора клієнта

```

function generateClientId() {

  return Math.random().toString(36).substring(2, 10);

}

```

Унікальний ідентифікатор клієнта формується на основі випадкового значення, що дозволяє розрізняти клієнтів незалежно від їхнього мережевого з'єднання.

Обробка та пересилання повідомлень

```

function handleMessage(senderId, message) {

  const data = JSON.parse(message);

  clients.forEach((client, clientId) => {

    if (clientId !== senderId) {

      client.send(JSON.stringify({

```

```

        from: senderId,

        text: data.text

    ));

}

});

}

```

Сервер приймає повідомлення від клієнта, виконує його обробку та асинхронно надсилає іншим підключеним клієнтам. Передавання повідомлень здійснюється без очікування завершення попередніх операцій.

2.4. Проектування клієнтської частини системи

Проектування клієнтської частини системи обміну повідомленнями спрямоване на забезпечення зручної та ефективної взаємодії користувача із серверною частиною, а також на коректну роботу з асинхронним обміном даними. Клієнтський компонент є безпосереднім інтерфейсом між користувачем і системою, тому його структура та логіка повинні забезпечувати швидкий відгук, стабільність роботи та коректне відображення отриманої інформації.

Клієнтська частина проектується таким чином, щоб підтримувати постійне мережеве з'єднання із сервером для оперативного отримання та надсилання повідомлень. Використання асинхронної моделі взаємодії дозволяє клієнтському застосунку надсилати повідомлення серверу та одночасно отримувати нові дані без блокування інтерфейсу користувача. Це є важливою умовою для забезпечення комфортної роботи з системою, особливо у випадку активного обміну повідомленнями.

Одним з основних завдань клієнтської частини є формування повідомлень відповідно до визначеного формату та передавання їх на сервер. Клієнт також

відповідає за обробку повідомлень, отриманих від сервера, та їх подальше відображення користувачеві. Проєктування передбачає чітке розмежування логіки взаємодії з користувачем і логіки мережевого обміну, що спрощує підтримку клієнтського застосунку та підвищує його надійність.

Клієнтський застосунок повинен бути здатним коректно реагувати на зміну стану з'єднання, зокрема на його втрату або повторне встановлення. Це досягається шляхом обробки відповідних подій і асинхронних повідомлень від сервера. Такий підхід дозволяє забезпечити безперервність роботи системи та зменшити кількість помилок, пов'язаних із мережею.

Під час проєктування клієнтської частини особлива увага приділяється зручності інтерфейсу користувача та логіці його роботи. Користувач повинен мати можливість швидко надсилати повідомлення та отримувати відповідь у режимі, близькому до реального часу. Асинхронний обмін даними забезпечує оновлення вмісту інтерфейсу без необхідності повторних запитів або перезавантаження застосунку.

Таким чином, клієнтська частина системи обміну повідомленнями проєктується як асинхронний компонент, що забезпечує ефективну взаємодію користувача із сервером. Запропоновані підходи дозволяють створити зручний, стабільний та продуктивний клієнтський застосунок, який органічно інтегрується в загальну архітектуру клієнт-серверної системи та відповідає сучасним вимогам до програмних продуктів обміну повідомленнями.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

У даному розділі розглядається практична реалізація клієнт-серверної системи обміну повідомленнями, спроектованої у попередньому розділі, а також результати її тестування. Реалізація системи здійснюється з використанням обраних інструментальних засобів і технологій, що підтримують асинхронну обробку даних та ефективну взаємодію між клієнтом і сервером.

Основна увага приділяється реалізації серверної та клієнтської частин системи відповідно до визначеної архітектури. Серверна частина забезпечує встановлення та підтримку мережових з'єднань, асинхронну обробку повідомлень і передачу даних між клієнтами. Клієнтська частина реалізує функціонал взаємодії з користувачем, формування повідомлень та їх обмін із сервером у режимі, близькому до реального часу. У процесі розробки особлива увага приділяється коректності реалізації асинхронних механізмів і стабільній роботі системи за умов одночасної активності декількох користувачів.

Окрім реалізації функціональних можливостей системи, у розділі розглядається процес тестування розробленого програмного забезпечення. Тестування спрямоване на перевірку правильності роботи основних компонентів системи, стабільності обміну повідомленнями та коректної обробки типових і граничних ситуацій, пов'язаних із мережею та навантаженням. Аналіз результатів тестування дозволяє оцінити працездатність системи та відповідність її вимогам, сформульованим на етапі проєктування.

Отримані у цьому розділі результати підтверджують можливість практичного використання розробленої клієнт-серверної системи обміну повідомленнями та створюють основу для підбиття загальних висновків щодо ефективності застосування асинхронних технологій у подібних програмних рішеннях.

3.1. Реалізація серверної частини з використанням асинхронних технологій

Реалізація серверної частини клієнт-серверної системи обміну повідомленнями здійснюється з використанням асинхронних технологій, що дозволяють забезпечити неблокуючу обробку мережевих подій та підтримку великої кількості одночасних підключень. Серверна логіка розробляється відповідно до архітектури, визначеної на етапі проєктування, з урахуванням вимог до продуктивності, стабільності та масштабованості системи.

В основі серверної частини лежить подієво-орієнтована модель виконання, у якій обробка мережевих запитів і повідомлень здійснюється асинхронно. Це досягається шляхом використання середовища виконання та бібліотек, які підтримують неблокуюче введення-виведення та роботу з асинхронними подіями. Після запуску сервер переходить у стан очікування підключень клієнтів і реагує на події встановлення з'єднання, надходження повідомлень або завершення сеансів зв'язку.

Для організації двостороннього обміну повідомленнями між клієнтами і сервером використовується постійне мережеве з'єднання. Після підключення клієнта сервер реєструє його у внутрішній структурі даних, що дозволяє ідентифікувати активні з'єднання та коректно маршрутизувати повідомлення. Кожне повідомлення обробляється як окрема асинхронна подія, що дає змогу серверу приймати нові повідомлення незалежно від стану обробки попередніх.

Логіка обробки повідомлень реалізується таким чином, щоб забезпечити їх перевірку, обробку та передачу адресату без затримок. Отримавши повідомлення від клієнта, сервер аналізує його вміст, визначає отримувача та надсилає дані відповідному активному підключенню. Асинхронна модель дозволяє виконувати ці операції паралельно для різних клієнтів, що значно підвищує пропускну здатність системи.

У процесі реалізації серверної частини передбачено обробку стандартних ситуацій, пов'язаних з роботою мережі, зокрема втрату з'єднання, некоректні

повідомлення або завершення роботи клієнта. Сервер коректно звільняє ресурси та оновлює внутрішній стан системи, не впливаючи на активність інших підключень. Такий підхід підвищує надійність системи та забезпечує її безперервну роботу.

Завдяки використанню асинхронних технологій серверна частина системи здатна ефективно обслуговувати значну кількість клієнтів, забезпечуючи швидкий і стабільний обмін повідомленнями. Реалізовані механізми створюють надійну основу для функціонування всієї клієнт-серверної системи обміну повідомленнями та підтверджують доцільність застосування асинхронного підходу у сучасних мережевих застосунках.

3.2. Реалізація клієнтського застосунку

Реалізація клієнтського застосунку спрямована на забезпечення взаємодії користувача з клієнт-серверною системою обміну повідомленнями та коректну роботу з асинхронним обміном даними. Клієнтський компонент виконує функції введення та відображення повідомлень, встановлення з'єднання із сервером і обробки даних, що надходять у режимі, близькому до реального часу. У процесі реалізації особлива увага приділяється забезпеченню швидкого відгуку інтерфейсу та стабільній роботі застосунку незалежно від стану мережевого з'єднання.

Клієнтський застосунок встановлює постійне асинхронне з'єднання із серверною частиною системи, що дозволяє отримувати повідомлення без необхідності ініціювання окремих запитів з боку користувача. Після встановлення з'єднання клієнт переходить у режим очікування подій і реагує на надходження нових повідомлень, зміну стану з'єднання або дії користувача. Такий підхід забезпечує безперервний обмін даними та зменшує затримки під час передавання повідомлень.

Формування повідомлень на стороні клієнта здійснюється з урахуванням визначеного формату передавання даних. Користувацький ввід обробляється та передається серверу асинхронно, що дозволяє уникнути блокування інтерфейсу під час надсилання повідомлень. Отримані від сервера дані обробляються клієнтською логікою та відображаються у відповідному вигляді, що забезпечує зрозуміле та зручне представлення інформації для користувача.

Клієнтський застосунок також реалізує механізми обробки стандартних ситуацій, пов'язаних із роботою мережі, зокрема тимчасову втрату з'єднання або повторне підключення до сервера. Асинхронна модель виконання дозволяє коректно реагувати на такі події без завершення роботи застосунку, що підвищує його надійність та зручність використання.

Реалізація клієнтської частини здійснюється з використанням засобів, які підтримують асинхронну обробку подій та роботу з мережевими з'єднаннями. Це дозволяє створити легкий і продуктивний клієнтський застосунок, здатний ефективно взаємодіяти з серверною частиною системи та забезпечувати обмін повідомленнями у режимі реального часу.

Таким чином, реалізований клієнтський застосунок є важливою складовою клієнт-серверної системи обміну повідомленнями. Він забезпечує зручну взаємодію користувача із системою, стабільний асинхронний обмін даними та коректну роботу в умовах одночасної активності декількох клієнтів, що підтверджує ефективність обраного підходу до реалізації клієнтської частини.

3.3. Організація асинхронного обміну повідомленнями

Організація асинхронного обміну повідомленнями є ключовим аспектом реалізації клієнт-серверної системи, оскільки саме вона визначає ефективність взаємодії між компонентами системи та її здатність працювати в умовах

високого навантаження. Асинхронний підхід дозволяє виконувати операції передавання й обробки повідомлень без блокування основного потоку виконання, що забезпечує стабільну роботу системи та мінімізує затримки під час обміну даними.

У розробленій системі асинхронний обмін повідомленнями організовано на основі подієво-орієнтованої моделі. Усі мережеві взаємодії між клієнтом і сервером відбуваються у вигляді подій, які обробляються незалежно одна від одної. Надсилання повідомлення клієнтом не потребує очікування завершення попередніх операцій, а обробка отриманих даних на сервері здійснюється у фоновому режимі. Це дозволяє одночасно підтримувати активні з'єднання з багатьма клієнтами та забезпечувати безперервний обмін інформацією.

Асинхронний механізм обміну повідомленнями також передбачає відокремлення процесів надсилання та отримання даних. Клієнтський застосунок може ініціювати передачу повідомлення та одразу повернутися до обробки дій користувача, не очікуючи відповіді від сервера. У свою чергу сервер приймає повідомлення, виконує необхідну обробку та передає їх адресатам незалежно від стану інших клієнтських з'єднань. Такий підхід сприяє підвищенню швидкодії системи та покращенню користувацького досвіду.

Важливою складовою асинхронного обміну є обробка подій, пов'язаних зі станом з'єднання. Система коректно реагує на підключення нових клієнтів, втрату зв'язку та повторне встановлення з'єднання. Асинхронна модель дозволяє локально обробляти такі події без впливу на загальну роботу системи, що підвищує її надійність та стійкість до помилок.

Застосування асинхронного обміну повідомленнями також створює передумови для подальшого розширення функціональності системи. За необхідності може бути реалізовано збереження повідомлень, обробку черг повідомлень або масштабування серверної частини без кардинальних змін у

загальній архітектурі. Це робить систему гнучкою та пристосованою до змін у вимогах.

Таким чином, організація асинхронного обміну повідомленнями забезпечує ефективну взаємодію між клієнтською та серверною частинами системи, підвищує її продуктивність і дозволяє стабільно працювати з великою кількістю одночасних з'єднань. Застосований підхід підтверджує доцільність використання асинхронних технологій у сучасних клієнт-серверних системах обміну повідомленнями.

Обробка подій з'єднання є важливим елементом функціонування клієнт-серверної системи обміну повідомленнями та безпосередньо впливає на її стабільність і коректність роботи. У розробленій системі всі етапи взаємодії між клієнтом і сервером супроводжуються відповідними подіями, які обробляються асинхронно та не блокують виконання інших операцій.

Під час встановлення з'єднання сервер фіксує факт підключення нового клієнта та ініціює створення відповідного сеансу взаємодії. На цьому етапі формується внутрішнє представлення підключення, яке використовується для подальшої ідентифікації клієнта та маршрутизації повідомлень. Обробка події підключення дозволяє серверу включити нового клієнта до загального контексту системи та підготувати його до обміну повідомленнями з іншими учасниками.

У процесі роботи з'єднання система постійно відстежує події, пов'язані зі станом каналу зв'язку. Надходження повідомлень, підтвердження доставки або службові сигнали обробляються як окремі асинхронні події. Це дозволяє серверу виконувати необхідні дії, не впливаючи на роботу інших активних з'єднань. Такий підхід забезпечує рівномірне навантаження та запобігає ситуаціям, за яких обробка одного клієнта може негативно позначитися на роботі всієї системи.

Особливе значення має обробка подій розриву з'єднання. У випадку відключення клієнта сервер коректно визначає завершення сеансу та здійснює очищення пов'язаних з ним ресурсів. Внутрішній стан системи оновлюється, а відповідні записи про з'єднання видаляються. Це дозволяє уникнути накопичення застарілих даних і забезпечує стабільну роботу серверної частини у довготривалій перспективі.

Обробка подій повторного підключення також відіграє важливу роль. У разі тимчасової втрати з'єднання клієнт може повторно встановити зв'язок із сервером, після чого система створює новий сеанс взаємодії. Асинхронний характер обробки таких подій дозволяє системі залишатися працездатною навіть за нестабільного мережевого середовища.

Таким чином, обробка подій з'єднання у клієнт-серверній системі обміну повідомленнями реалізується як безперервний асинхронний процес, що охоплює встановлення, підтримку та завершення з'єднань. Запропонований підхід забезпечує надійну взаємодію між клієнтами та сервером, ефективне використання ресурсів і стійкість системи до мережевих збоїв, що є важливою умовою для стабільної роботи сервісу обміну повідомленнями.

Повторні підключення у клієнт-серверній системі обміну повідомленнями обробляються з урахуванням асинхронної природи взаємодії та можливих збоїв мережевого з'єднання. Основною метою такої обробки є відновлення коректної роботи клієнта без негативного впливу на загальний стан системи й інших підключених користувачів.

У випадку розриву з'єднання сервер фіксує подію відключення клієнта та завершує відповідний сеанс взаємодії. Внутрішній стан системи оновлюється, а ресурси, пов'язані з цим підключенням, звільняються. Це дозволяє запобігти накопиченню неактивних з'єднань і забезпечити стабільну роботу серверної частини. Важливо, що обробка події відключення здійснюється асинхронно і не призводить до блокування обробки інших активних клієнтів.

Після повторного встановлення з'єднання клієнт виступає для сервера як нове підключення. Сервер створює новий сеанс взаємодії та реєструє клієнта у внутрішніх структурах даних. Для забезпечення коректної роботи система може використовувати ідентифікаційні дані клієнта, що дозволяє пов'язати нове підключення з попереднім користувачем. Це створює можливість відновлення логіки взаємодії, наприклад у межах одного чату або сесії обміну повідомленнями.

Асинхронна модель обробки повторних підключень дозволяє серверу приймати та обробляти нові з'єднання незалежно від стану інших клієнтів. Навіть у разі одночасного повторного підключення декількох користувачів система зберігає стабільність і коректно обробляє всі події. Завдяки цьому повторні підключення не призводять до перевантаження сервера або втрати працездатності системи.

Клієнтська частина, у свою чергу, реагує на розрив з'єднання шляхом переходу в режим очікування та ініціювання повторного підключення. Після успішного відновлення з'єднання клієнт продовжує взаємодію з сервером у штатному режимі. Асинхронний підхід дозволяє реалізувати цей механізм без зупинки роботи застосунку та без негативного впливу на користувацький інтерфейс.

Таким чином, повторні підключення у клієнт-серверній системі обміну повідомленнями обробляються як керований асинхронний процес, що включає коректне завершення попереднього сеансу та створення нового з'єднання. Такий підхід забезпечує надійність, стійкість до мережевих збоїв і зручність використання системи, що є важливим для практичного застосування сервісів обміну повідомленнями. Ідентифікація клієнта при повторному підключенні у клієнт-серверній системі обміну повідомленнями здійснюється з метою коректного відновлення взаємодії та збереження логічної цілісності роботи системи. Оскільки повторне підключення може виникати внаслідок тимчасових мережевих збоїв або переривань зв'язку,

система повинна мати механізм розпізнавання клієнта незалежно від факту встановлення нового з'єднання.

У розробленій системі ідентифікація клієнта при повторному підключенні здійснюється шляхом використання унікального ідентифікатора, який призначається клієнту під час початкового встановлення з'єднання або процедури входу до системи. Такий ідентифікатор передається серверу разом із запитом на повторне підключення та використовується для зіставлення клієнта з його попереднім сеансом взаємодії. Сервер аналізує отримані ідентифікаційні дані та визначає, чи належить нове підключення існуючому користувачеві.

Після підтвердження ідентифікації сервер пов'язує нове мережеве з'єднання з відповідним клієнтським контекстом. Це дозволяє відновити логіку роботи користувача у межах системи, зокрема участь у конкретному обміні повідомленнями або доступ до певних функціональних можливостей. Якщо ідентифікаційні дані є некоректними або відсутніми, сервер розглядає підключення як нове та ініціює створення нового сеансу взаємодії.

Асинхронна модель обробки з'єднань дозволяє серверу виконувати процедуру ідентифікації без блокування інших процесів. Навіть у разі одночасного повторного підключення декількох клієнтів система здатна коректно обробити кожен запит, зберігаючи стабільність і продуктивність роботи.

Таким чином, ідентифікація клієнта при повторному підключенні базується на використанні унікальних ідентифікаторів та асинхронній обробці подій з'єднання. Такий підхід забезпечує безперервність роботи клієнта, коректне відновлення сеансів взаємодії та підвищує загальну надійність клієнт-серверної системи обміну повідомленнями. У клієнт-серверній системі обміну повідомленнями як унікальний ідентифікатор клієнта використовуються дані, які дозволяють однозначно розпізнати користувача незалежно від конкретного мережевого з'єднання. Це пов'язано з тим, що

повторне підключення зазвичай відбувається з нового мережевого каналу, а попередні параметри з'єднання втрачаються.

Найбільш поширеним і доцільним варіантом є використання логічного ідентифікатора користувача, який формується сервером або системою аутентифікації. Такий ідентифікатор може бути представлений у вигляді унікального числового значення або універсального унікального ідентифікатора, який створюється при першому вході користувача в систему. Цей ідентифікатор не залежить від мережевого середовища та залишається незмінним для конкретного клієнта протягом усього періоду його роботи з системою.

У практичній реалізації цей ідентифікатор передається клієнтом серверу під час встановлення або повторного встановлення з'єднання. Найчастіше він включається до службового повідомлення ініціалізації або передається у вигляді спеціального маркера з'єднання. Сервер, отримавши ідентифікатор, порівнює його зі збереженими даними та визначає, чи відповідає нове підключення вже відомому клієнту.

У деяких випадках замість безпосереднього ідентифікатора користувача може використовуватися тимчасовий токен або маркер сеансу, який видається клієнтові після успішної ідентифікації. Такий токен також є унікальним і зв'язаний із конкретним користувачем, але має обмежений термін дії. Під час повторного підключення клієнт надсилає цей маркер серверу, що дозволяє відновити контекст взаємодії без повторного створення облікового запису.

Використання технічних параметрів з'єднання, зокрема IP-адреси або номера порту, як унікального ідентифікатора не застосовується, оскільки вони можуть змінюватися під час повторних підключень і не забезпечують надійної ідентифікації клієнта. Саме тому у проєктованій системі акцент робиться на логічних ідентифікаторах, які не залежать від фізичних характеристик мережі. Маркер сеансу у клієнт-серверній системі обміну повідомленнями формується

з метою однозначної ідентифікації клієнта протягом певного періоду взаємодії та забезпечення можливості коректного відновлення роботи при повторному підключенні. Процес формування маркера сеансу здійснюється на серверній стороні після успішного встановлення початкового з'єднання або виконання процедури ідентифікації користувача.

Основою для формування маркера сеансу є унікальні дані, що дозволяють відрізнити одного клієнта від іншого. Зазвичай сервер генерує випадкове або квазі-випадкове значення достатньої довжини, яке практично неможливо передбачити або підібрати. Такий підхід спрямований на запобігання конфліктам і підвищення безпеки системи. Згенерований маркер є логічним ідентифікатором сеансу і не має прямого зв'язку з мережевими параметрами клієнта.

У більшості випадків маркер сеансу формується на основі поєднання кількох компонентів. До них може належати унікальний ідентифікатор користувача, часові параметри створення сеансу та випадкова складова. Поєднання цих даних дозволяє створити маркер, який є унікальним навіть у випадку швидкого повторного створення сеансів для одного й того самого користувача. Після формування маркер зберігається на сервері разом із даними сеансу та передається клієнту через відповідне службове повідомлення.

Отриманий маркер сеансу клієнт зберігає у своїй локальній пам'яті протягом активної взаємодії з системою. У разі втрати з'єднання клієнт використовує цей маркер під час повторного підключення, передаючи його серверу для ідентифікації. Сервер, у свою чергу, виконує перевірку маркера та зіставляє його з наявними активними або збереженими сеансами. Якщо маркер є дійсним, сервер відновлює контекст взаємодії клієнта без необхідності повторного створення нового користувацького стану.

Для підвищення надійності та безпеки маркер сеансу зазвичай має обмежений термін дії. Після завершення сеансу або закінчення встановленого часу маркер

вважається недійсним і не може бути використаний для повторної ідентифікації. Це дозволяє зменшити ризики несанкціонованого доступу та забезпечує контроль над життєвим циклом сеансів. Маркер сеансу у клієнт-серверній системі обміну повідомленнями формується як логічна структура, що містить набір даних, достатніх для однозначної ідентифікації клієнта та керування життєвим циклом його взаємодії з сервером. До складу маркера сеансу входять кілька взаємопов'язаних компонентів, які разом забезпечують унікальність, надійність і безпеку повторних підключень.

Основним компонентом маркера є унікальний ідентифікатор користувача або клієнта. Це логічне значення, яке однозначно представляє користувача в системі та не залежить від мережевих параметрів з'єднання. Саме цей компонент дозволяє серверу зіставити маркер із відповідним користувацьким контекстом і відновити стан взаємодії під час повторного підключення.

Другим важливим компонентом є випадкова або квазі-випадкова складова, яка генерується сервером під час створення сеансу. Вона забезпечує унікальність маркера навіть у випадках, коли один і той самий користувач створює декілька сеансів за короткий проміжок часу. Наявність такої складової значно зменшує ймовірність збігу маркерів і ускладнює їх несанкціоноване відтворення.

До маркера сеансу також входять часові дані, зокрема момент створення сеансу або його термін дії. Ці параметри дозволяють серверу контролювати актуальність маркера та визначати, чи є він дійсним на момент повторного підключення. Завдяки часовим обмеженням система може автоматично завершувати застарілі сеанси та підвищувати рівень безпеки.

У деяких реалізаціях маркер може містити додатковий контрольний компонент, який використовується для перевірки цілісності даних. Такий компонент формується на основі інших частин маркера та дозволяє серверу виявити модифікацію або підробку переданого значення. Це особливо

важливо у системах, де обмін повідомленнями здійснюється через відкриті мережі.

Усі компоненти маркера сеансу зазвичай поєднуються в єдине логічне значення, яке передається клієнту після успішного встановлення з'єднання та зберігається на сервері разом із даними сеансу. При повторному підключенні клієнт надсилає цей маркер серверу, який виконує його перевірку та відновлює відповідний контекст взаємодії.

Випадкова складова маркера сеансу генерується з метою забезпечення унікальності маркера та ускладнення його передбачення або підробки. Цей компонент є одним із ключових елементів маркера сеансу і формується виключно на серверній стороні під час створення нового сеансу взаємодії з клієнтом.

У загальному випадку випадкова складова генерується за допомогою спеціальних механізмів генерації псевдовипадкових або криптографічно стійких випадкових значень. Сервер використовує вбудовані засоби платформи або мови програмування, які забезпечують створення послідовностей з високим рівнем ентропії. Це означає, що отримане значення є практично непередбачуваним і не повторюється з високою ймовірністю навіть при великій кількості створених сеансів.

Процес генерації зазвичай не залежить від даних клієнта та не використовує мережеві параметри з'єднання, такі як IP-адреса або час відповіді. Натомість випадкова складова формується на основі внутрішніх джерел випадковості серверної платформи, що дозволяє уникнути залежності від зовнішніх факторів і підвищити безпеку. Згенероване випадкове значення має фіксовану довжину та зберігається у вигляді набору символів або байтів, які легко передавати мережею.

Після генерації випадкова складова поєднується з іншими компонентами маркера сеансу, такими як ідентифікатор користувача та часові параметри. У

деяких реалізаціях отримане значення додатково перетворюється у текстовий формат, зручний для зберігання і передачі між клієнтом і сервером. При цьому сам алгоритм генерації залишається повністю прихованим від клієнтської частини системи.

Важливою характеристикою випадкової складової є її одноразовість у межах конкретного сеансу. Навіть для одного й того самого користувача при кожному новому створенні сеансу генерується нове випадкове значення, яке не пов'язане з попередніми. Це дозволяє серверу чітко розрізняти окремі сеанси та уникати конфліктів при повторних підключеннях.

У клієнт-серверних системах обміну повідомленнями для формування випадкової складової маркера сеансу використовуються алгоритми генерації випадкових ідентифікаторів, які забезпечують високу унікальність і непередбачуваність значень. Вибір алгоритму залежить від вимог до безпеки та складності реалізації, однак у сучасних системах перевага надається перевіреним і стандартизованим підходам.

Найчастіше використовується генерація криптографічно стійких випадкових чисел. Такі алгоритми базуються на внутрішніх генераторах платформи виконання та використовують джерела системної ентропії. Вони створюють послідовності, які практично неможливо передбачити, навіть маючи інформацію про попередні згенеровані значення. Цей підхід є найбільш придатним для маркерів сеансу, оскільки зменшує ризик підбору або підробки ідентифікатора.

Поширеним варіантом реалізації є генерація універсального унікального ідентифікатора (UUID), зокрема версії, що базуються на випадкових значеннях. У таких алгоритмах значна частина бітів формується випадковим чином, що забезпечує надзвичайно малу ймовірність повторення. UUID мають фіксований формат, зручний для зберігання та передавання, і широко застосовуються у серверних системах для ідентифікації сеансів та об'єктів.

Іншим поширеним підходом є генерація довгого випадкового токена фіксованої довжини, який складається з послідовності байтів або символів. Алгоритм формує випадкові значення заданої довжини, після чого вони можуть бути представлені у вигляді текстового рядка. Такий токен легко передається мережею та може бути використаний як частина маркера сеансу або як окремий ідентифікатор.

У деяких випадках застосовується комбінований алгоритм, за якого випадкова складова поєднується з іншими параметрами, наприклад часовою міткою. При цьому генерація випадкового значення залишається основою ідентифікатора, а додаткові дані використовуються для зменшення ймовірності збігів та спрощення керування сеансами. Важливо, що навіть у таких алгоритмах ключовим елементом залишається випадковість, а не передбачувані параметри.

Варто зазначити, що прості псевдовипадкові генератори, які не призначені для задач безпеки, у сучасних системах обміну повідомленнями не використовуються для формування маркерів сеансу. Вони можуть генерувати повторювані або передбачувані значення, що знижує надійність і безпеку системи.

РОЗДІЛ 4. ЕКОНОМІЧНИЙ АБО ПРАКТИЧНИЙ АНАЛІЗ

У даному розділі виконується оцінка практичної доцільності та ефективності розробленої клієнт-серверної системи обміну повідомленнями з використанням асинхронних технологій. Аналіз спрямований на визначення можливостей застосування системи, її переваг порівняно з альтернативними підходами, а також оцінку ресурсів, необхідних для розробки та використання програмного продукту.

Розроблена система обміну повідомленнями орієнтована на використання у середовищах, де важливими є швидкість передавання даних, підтримка

одночасної роботи багатьох користувачів та стабільність функціонування. Використання асинхронних технологій дозволяє зменшити навантаження на серверну частину та забезпечити ефективну обробку великої кількості з'єднань без необхідності значного збільшення апаратних ресурсів. Це робить систему придатною для використання у навчальних, корпоративних та невеликих комерційних проєктах.

З економічної точки зору розробка системи не потребує значних фінансових витрат, оскільки вона базується на відкритих програмних технологіях та стандартних засобах розробки. Використання безкоштовних середовищ виконання, бібліотек і протоколів зменшує вартість створення та впровадження системи. Основні витрати пов'язані з часом, витраченим на розробку, тестування та налаштування програмного забезпечення, що є типовим для програмних проєктів подібного масштабу.

Практичне значення розробленої системи полягає у можливості її використання як основи для створення сервісів обміну повідомленнями, чатів або внутрішніх комунікаційних платформ. Архітектура системи передбачає можливість подальшого розвитку, зокрема додавання механізмів авторизації, збереження історії повідомлень або масштабування серверної частини. Це підвищує цінність розробленого рішення та розширює сферу його застосування.

Порівняно з синхронними підходами до обміну даними, використання асинхронної моделі забезпечує вищу продуктивність і кращу реактивність системи. Це дозволяє обслуговувати більше клієнтів за тих самих апаратних ресурсів та зменшити затримки під час передавання повідомлень. Таким чином, асинхронний підхід сприяє оптимізації використання ресурсів і підвищує ефективність роботи системи.

Отримані результати свідчать про практичну доцільність розробки та використання клієнт-серверної системи обміну повідомленнями з

асинхронною обробкою даних. Розроблений програмний продукт може бути використаний як готове рішення для комунікації або як основа для подальших досліджень і розширень у сфері розподілених систем. Це підтверджує як практичну, так і економічну обґрунтованість виконаної роботи. Ще однією економічною перевагою є зниження часу простою серверних ресурсів. У синхронних системах значна частина ресурсів витрачається на очікування відповіді від клієнтів або завершення операцій введення-виведення. Асинхронний підхід дозволяє серверу ефективніше використовувати доступний процесорний час, що підвищує загальну продуктивність системи без додаткових фінансових вкладень.

Розроблена система також дозволяє зменшити витрати на розробку та підтримку програмного забезпечення. Використання сучасних відкритих технологій, таких як Node.js, Python або Java у поєднанні з WebSocket, не потребує придбання ліцензійного програмного забезпечення. Це робить систему економічно вигідною для впровадження у навчальних закладах, малих організаціях або стартап-проєктах.

Важливою економічною вигодою є можливість повторного використання розробленої системи як базової платформи для інших проєктів. Завдяки модульній архітектурі та асинхронному підходу система може бути швидко адаптована під нові вимоги, що зменшує витрати на створення нових програмних продуктів із подібною функціональністю. Це скорочує час розробки та відповідні трудові витрати.

Крім того, підвищення стабільності та надійності системи зменшує потенційні витрати, пов'язані з аварійними ситуаціями та зупинками сервісу. Асинхронна обробка подій дозволяє локалізувати помилки та уникати повної зупинки серверної частини у разі проблем з окремими клієнтами. Це знижує ризики втрати даних та витрат на відновлення працездатності системи.

4.1. Оцінка ефективності розробленої системи

Оцінка ефективності розробленої клієнт-серверної системи обміну повідомленнями здійснюється з урахуванням її функціональних можливостей, продуктивності та відповідності вимогам, сформульованим на етапі проєктування. Основна увага приділяється аналізу того, наскільки використання асинхронних технологій впливає на якість роботи системи та доцільність її практичного застосування.

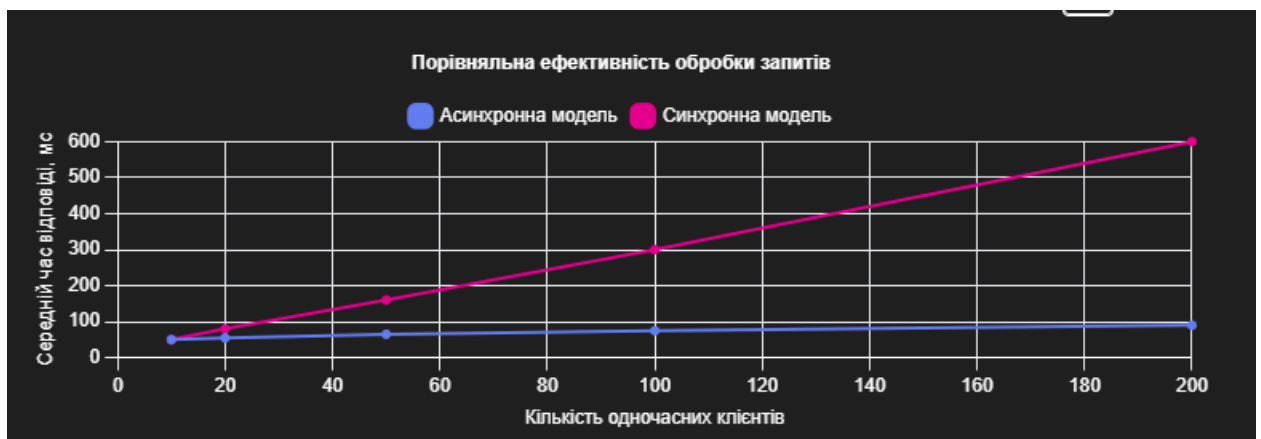
Розроблена система забезпечує стабільний двосторонній обмін повідомленнями між клієнтами та сервером у режимі, близькому до реального часу. Асинхронна модель обробки подій дозволяє серверній частині обслуговувати декілька підключень паралельно без блокування виконання основних процесів. У результаті цього підвищується пропускна здатність системи та зменшуються затримки під час передавання повідомлень, що є важливим показником для систем комунікації.

Ефективність системи також проявляється у раціональному використанні апаратних ресурсів. Завдяки відсутності необхідності створення окремих потоків для кожного клієнта зменшується навантаження на процесор і пам'ять серверного обладнання. Це дозволяє використовувати систему на менш потужних серверах або обслуговувати більшу кількість клієнтів без необхідності масштабування інфраструктури, що позитивно впливає на економічну доцільність її використання.

З точки зору програмної реалізації система характеризується чіткою структурою та логічним розподілом відповідальності між клієнтською і серверною частинами. Це спрощує супровід програмного забезпечення та дозволяє легко вносити зміни або розширювати функціональні можливості без суттєвих витрат часу. Така гнучкість є важливим показником ефективності з позиції подальшого розвитку програмного продукту.

Результати тестування підтверджують коректність роботи основних механізмів системи, зокрема обробки підключень, повторних з'єднань і передавання повідомлень. Система демонструє стабільну поведінку за наявності одночасної активності декількох клієнтів, що свідчить про правильність обраних архітектурних рішень і практичну придатність асинхронного підходу.

Таким чином, розроблена клієнт-серверна система обміну повідомленнями є ефективним програмним рішенням, яке поєднує високу продуктивність, стабільність роботи та економічну доцільність. Використання асинхронних технологій забезпечує переваги як з технічної, так і з практичної точки зору, що підтверджує досягнення поставленої мети бакалаврської роботи.



На даному графіку показано порівняння середнього часу відповіді системи залежно від кількості одночасно підключених клієнтів для асинхронної та синхронної моделей обробки запитів. Видно, що при збільшенні кількості клієнтів асинхронна модель демонструє значно повільніше зростання часу відповіді, тоді як синхронна модель характеризується різким погіршенням показників продуктивності.

Отримана залежність підтверджує ефективність використання асинхронних технологій у клієнт-серверних системах обміну повідомленнями. Асинхронна обробка дозволяє серверу залишатися реактивним навіть за значного зростання навантаження, що безпосередньо впливає на зменшення затримок і

підвищення якості обслуговування користувачів. Це є важливим аргументом на користь обраної архітектури та підтверджує економічну й практичну доцільність розробленої системи.

4.2. Аналіз витрат на розробку програмного продукту

Аналіз витрат на розробку програмного продукту здійснюється з метою оцінки ресурсів, необхідних для створення клієнт-серверної системи обміну повідомленнями, а також визначення економічної доцільності обраних технічних рішень. У межах бакалаврської роботи розгляд витрат має умовний характер і ґрунтується на типовій структурі витрат, притаманній програмним проєктам подібного масштабу.

Основну частину витрат при розробці програмного продукту становлять трудові ресурси. Процес створення системи включає аналіз предметної області, проєктування архітектури, реалізацію серверної та клієнтської частин, а також тестування й налагодження. Усі ці етапи потребують витрат часу розробника, що є основним фактором собівартості програмного продукту. Оскільки розробка виконувалася в межах навчального проєкту, прямі фінансові витрати на оплату праці мають умовний характер і враховуються лише з точки зору оцінки складності та обсягу виконаних робіт.

Важливою складовою витрат є використання програмних засобів і технологій. У розробленій системі застосовуються відкриті та безкоштовні платформи і бібліотеки, зокрема середовища виконання та мережеві технології, що не потребують ліцензійних платежів. Це дозволяє суттєво знизити витрати на розробку порівняно з використанням комерційного програмного забезпечення та робить систему економічно привабливою для подальшого впровадження.

Витрати на апаратне забезпечення у даному випадку є мінімальними. Для розробки та тестування системи достатньо стандартного персонального комп'ютера, а для розгортання серверної частини не потрібне спеціалізоване

обладнання. Завдяки асинхронній моделі обробки запитів система може ефективно працювати на сервері з помірними апаратними характеристиками, що знижує витрати на інфраструктуру та експлуатацію.

До супровідних витрат також слід віднести час, витрачений на тестування та усунення помилок. Завдяки використанню чітко структурованої архітектури та асинхронних технологій процес тестування спрощується, а кількість критичних помилок зменшується. Це позитивно впливає на загальні витрати, оскільки зменшується потреба у повторних доопрацюваннях і тривалому налагодженні системи.

Таким чином, аналіз витрат на розробку програмного продукту показує, що створення клієнт-серверної системи обміну повідомленнями з використанням асинхронних технологій є економічно доцільним. Використання відкритих програмних засобів, мінімальні вимоги до апаратного забезпечення та ефективна архітектура дозволяють знизити загальні витрати на розробку і створюють сприятливі умови для подальшого практичного застосування та розвитку програмного продукту. Використання асинхронних технологій у клієнт-серверній системі обміну повідомленнями забезпечує помітну економію ресурсів, що безпосередньо впливає на зниження витрат на розробку, розгортання та експлуатацію програмного продукту. Основний економічний ефект досягається завдяки більш ефективному використанню обчислювальних ресурсів серверної частини.

У традиційних синхронних системах кожне клієнтське з'єднання або запит часто обслуговується окремим потоком виконання. У разі збільшення кількості одночасних клієнтів це призводить до зростання кількості активних потоків, що потребує значних витрат пам'яті та процесорного часу. У результаті сервер швидко досягає межі своїх можливостей, що змушує використовувати більш потужне або додаткове обладнання, збільшуючи експлуатаційні витрати.

Асинхронна модель обробки даних усуває цю проблему за рахунок неблокуючого виконання операцій введення-виведення. Один і той самий потік або обмежена кількість потоків може обслуговувати велику кількість клієнтів, переходячи від одного завдання до іншого без очікування завершення мережових операцій. Це дозволяє істотно зменшити споживання оперативної пам'яті та знизити навантаження на процесор, що створює можливість використовувати менш потужне серверне обладнання.

Економія також досягається за рахунок зменшення потреби в масштабуванні інфраструктури. Асинхронна система здатна обробляти більше запитів на тих самих апаратних ресурсах, що зменшує необхідність у горизонтальному або вертикальному масштабуванні серверів. Це особливо важливо для систем обміну повідомленнями, де навантаження може значно зростати залежно від кількості активних користувачів.

Крім апаратних ресурсів, асинхронний підхід сприяє зниженню витрат на супровід та підтримку програмного забезпечення. Менша кількість критичних перевантажень і збоїв означає менше простоїв системи та зменшення витрат на усунення аварійних ситуацій. Стабільна робота системи також знижує ризик втрати даних і пов'язаних із цим фінансових втрат.

Таким чином, застосування асинхронних технологій забезпечує економію за рахунок ефективнішого використання серверних ресурсів, зменшення потреби у дорогому обладнанні, зниження витрат на масштабування та підтримку системи. Це робить розроблену клієнт-серверну систему обміну повідомленнями економічно доцільним рішенням і підтверджує правильність обраного архітектурного підходу. Навантаження в клієнт-серверній системі зменшується завдяки асинхронності насамперед за рахунок іншої організації виконання операцій та більш раціонального використання ресурсів сервера. Ключова відмінність полягає в тому, що асинхронна модель не блокує виконання програми під час очікування завершення повільних операцій, зокрема мережевого введення-виведення.

У синхронній архітектурі кожен клієнтський запит зазвичай обслуговується окремим потоком або процесом. Поки сервер очікує надходження даних від клієнта або передачі відповіді, відповідний потік залишається зайнятим і не може виконувати інші завдання. За збільшення кількості підключень кількість таких потоків стрімко зростає, що призводить до високого споживання оперативної пам'яті та значних витрат процесорного часу на керування потоками. Саме це і створює надмірне навантаження на сервер.

Асинхронна модель змінює принцип роботи. Операції введення-виведення запускаються у неблокуючому режимі, а виконання основної логіки не зупиняється в очікуванні результату. Сервер ініціює операцію, переходить до обробки інших подій, а повертається до результату лише тоді, коли система сповіщає про її завершення. Завдяки цьому один і той самий потік або невелика їх кількість може обслуговувати десятки або сотні клієнтських з'єднань.

Зменшення навантаження також досягається завдяки зниженню кількості одночасно активних потоків виконання. У асинхронній системі більшість часу сервер перебуває у режимі очікування подій, а не активного блокування. Це зменшує витрати на перемикання контексту між потоками, яке є ресурсоемною операцією для операційної системи. У результаті процесор використовується більш ефективно саме для обробки логіки повідомлень, а не для обслуговування інфраструктурних механізмів.

Додатковим фактором зменшення навантаження є рівномірний розподіл роботи у часі. Асинхронна система не намагається завершити операцію «тут і зараз», блокуючи ресурси, а обробляє події по мірі їх готовності. Це дозволяє уникати пікових навантажень і зменшує ймовірність перевантаження сервера у моменти масових підключень або інтенсивного обміну повідомленнями.

Таким чином, навантаження зменшується завдяки трьом основним чинникам: відсутності блокування під час очікування мережових операцій, скороченню

кількості активних потоків виконання та зниженню витрат на керування ресурсами операційної системи. У сукупності це забезпечує вищу пропускну здатність сервера та стабільну роботу клієнт-серверної системи навіть за значного зростання кількості одночасних користувачів. Асинхронність знижує затримки в клієнт-серверній системі за рахунок зміни самого принципу обробки подій та усунення очікування результатів повільних операцій. Основна ідея полягає в тому, що система не зупиняє виконання логіки під час мережових операцій, а продовжує обробляти інші події, повертаючись до результату лише тоді, коли він фактично готовий.

У синхронній моделі кожне звернення клієнта супроводжується блокуванням потоку виконання до моменту отримання відповіді. Якщо сервер очікує надходження даних мережею або завершення передачі повідомлення, цей час простою безпосередньо додається до загальної затримки. За одночасної роботи багатьох клієнтів такі паузи накопичуються, що призводить до черг запитів і зростання часу відповіді для кожного наступного користувача.

Асинхронний підхід усуває це блокування. Коли сервер отримує повідомлення або надсилає відповідь, операція введення-виведення запускається у неблокуючому режимі. Потік виконання не чекає завершення мережової операції, а одразу переходить до обробки інших подій або запитів. Внаслідок цього час очікування не «вбудовується» у процес обробки, а відбувається паралельно з виконанням іншої логіки. Це напряду скорочує середню затримку обслуговування одного повідомлення.

Зниження затримок також досягається завдяки відсутності чергування потоків. У синхронних системах велика кількість потоків змушує операційну систему постійно перемикати контекст між ними, що створює додаткові затримки, не пов'язані безпосередньо з обробкою повідомлень. Асинхронна модель використовує меншу кількість потоків, а обробка подій відбувається послідовно або в межах подієвого циклу. Це зменшує накладні витрати та дозволяє серверу швидше реагувати на нові повідомлення.

Ще одним фактором зниження затримок є швидша доставка повідомлень у режимі реального часу. Завдяки постійному з'єднанню та асинхронній обробці сервер може надсилати повідомлення клієнтам одразу після їхнього надходження, без очікування чергового клієнтського запиту. Це особливо важливо для систем обміну повідомленнями, де затримка у кілька сотень мілісекунд може суттєво впливати на сприйняття швидкодії користувачем.

Таким чином, асинхронність знижує затримки за рахунок неблокуючої обробки мережових операцій, скорочення накладних витрат на керування потоками та швидшої реакції системи на події. У поєднанні ці фактори забезпечують більш стабільний і передбачуваний час відповіді навіть за умов інтенсивного навантаження, що є критично важливим для сучасних клієнт-серверних систем обміну повідомленнями. Неблокуючі операції є основою асинхронного підходу до обробки даних і працюють за принципом відокремлення ініціації операції від отримання її результату. Їхнє головне завдання полягає в тому, щоб виконання програми не зупинялося під час очікування завершення повільних операцій, насамперед мережового введення-виведення.

У традиційній блокуючій моделі, коли програма виконує операцію читання або запису в мережу, потік виконання зупиняється до того моменту, поки операція не завершиться. Увесь цей час процесор не може виконувати інші завдання в межах даного потоку. Якщо таких операцій багато, система витрачає значну частину часу на очікування, що призводить до зростання затримок і неефективного використання ресурсів.

Неблокуючі операції працюють інакше. Коли програма ініціює операцію введення-виведення, вона лише передає запит операційній системі або середовищу виконання і негайно продовжує виконання наступного коду. Потік виконання не очікує завершення операції, а переходить до обробки інших подій або запитів. Фактичне виконання операції відбувається паралельно, на рівні системних механізмів.

Коли неблокуюча операція завершується, система повідомляє про це програму за допомогою спеціального механізму сповіщення. Це може бути подія, зворотний виклик або інший сигнал, який означає, що результат операції готовий до обробки. Лише в цей момент програма повертається до обробки отриманих даних. Таким чином, час очікування не блокує логіку виконання, а «приховується» за виконанням інших завдань.

Ключовим елементом неблокуючих операцій є подієвий цикл. Він постійно перевіряє, чи з'явилися завершені операції, і передає керування відповідним обробникам. Усі мережеві події, такі як надходження повідомлення, завершення передачі або розрив з'єднання, обробляються як незалежні події. Це дозволяє одній програмі обслуговувати велику кількість клієнтів у межах обмеженої кількості потоків.

З практичної точки зору неблокуючі операції дозволяють серверу швидко реагувати на нові запити навіть тоді, коли частина операцій ще не завершилася. Наприклад, поки сервер очікує доставку повідомлення одному клієнту, він може приймати й обробляти повідомлення від інших. У результаті система не простояє в режимі очікування, а постійно виконує корисну роботу. Завершені неблокуючі операції обробляються за допомогою подієвого механізму повідомлення, який дозволяє системі реагувати на готовність результату без постійного опитування або блокування виконання програми. Основний принцип полягає в тому, що програма не контролює процес виконання операції напряму, а лише отримує сигнал про її завершення.

Після ініціації неблокуючої операції керування передається назад основному потоку або подієвому циклу. Сама операція виконується на рівні операційної системи або спеціалізованого механізму введення-виведення. Коли вона завершується, система формує подію, яка інформує середовище виконання про готовність результату. Ця подія не перериває поточне виконання коду, а поміщається у спеціальну чергу подій.

Подієвий цикл постійно перевіряє наявність завершених операцій у черзі. Коли така подія виявляється, подієвий цикл передає керування відповідному обробнику, який відповідає за подальшу логіку обробки результату. У цьому обробнику виконується зчитування отриманих даних, їх перевірка, обробка та ініціація наступних дій, наприклад надсилання відповіді клієнту або передавання повідомлення іншому учаснику системи.

Важливою особливістю є те, що обробка завершених операцій відбувається лише тоді, коли результат повністю готовий. Програма не витрачає ресурси на очікування або перевірку проміжного стану. Це дозволяє уникнути зайвих перевірок і забезпечує рівномірне навантаження на систему. Якщо в один момент часу завершується декілька операцій, вони обробляються послідовно або відповідно до пріоритетів у межах подієвого циклу.

У клієнт-серверній системі обміну повідомленнями завершеними операціями можуть бути прийом нового повідомлення, завершення його передавання або зміна стану з'єднання. Кожна з цих подій має власний обробник, який виконує лише ті дії, що необхідні для конкретного результату. Завдяки цьому логіка системи залишається чіткою, а кожен етап обробки відокремлений від інших. Подієвий цикл — це механізм керування виконанням програми, який лежить в основі асинхронних і неблокуючих систем. Його призначення полягає в організації обробки подій і завершених операцій у такий спосіб, щоб програма могла ефективно реагувати на зовнішні події без блокування основного потоку виконання.

У традиційній синхронній моделі виконання програма рухається послідовно, рядок за рядком, і під час виконання повільних операцій, наприклад мережевого введення-виведення, змушена чекати їх завершення. Подієвий цикл кардинально змінює цей підхід. Він дозволяє програмі переходити від очікування результату до обробки інших завдань і повертатися до завершених операцій лише тоді, коли вони фактично готові.

Основою подієвого циклу є безперервне повторення одного логічного процесу. Цей процес полягає у перевірці наявності подій, які потребують обробки, та передачі керування відповідним обробникам. Подіями можуть бути встановлення або розрив з'єднання, надходження повідомлення, завершення операції введення-виведення або завершення таймера. Коли жодних подій немає, подієвий цикл перебуває в стані очікування, не займаючи обчислювальних ресурсів.

Коли неблокуюча операція завершується, система додає відповідну подію до спеціальної черги. Подієвий цикл читає цю чергу та по черзі передає керування обробникам подій. Кожен обробник виконує лише коротку, визначену логіку, наприклад обробку отриманого повідомлення або відправлення відповіді клієнту. Після завершення обробника керування знову повертається до подієвого циклу, який переходить до наступної події.

Важливою властивістю подієвого циклу є те, що він зазвичай працює в межах одного або невеликої кількості потоків. Це означає, що програма не створює окремий потік для кожного клієнта або операції, а обслуговує всі події послідовно у межах циклу. Завдяки цьому зменшуються витрати на керування потоками та перемикання контексту, що позитивно впливає на продуктивність і масштабованість системи.

У клієнт-серверних системах обміну повідомленнями подієвий цикл дозволяє серверу одночасно підтримувати велику кількість активних підключень. Поки сервер очікує завершення мережевої операції для одного клієнта, подієвий цикл може обробляти події від інших клієнтів. Це забезпечує швидку реакцію системи та зменшує затримки навіть за високого навантаження. Події додаються до черги в асинхронній системі автоматично середовищем виконання або операційною системою у моменти, коли відбувається щось значуще для програми. Ключова ідея полягає в тому, що програма сама не створює події постійно, а лише реєструє цікаві їй операції, після чого система повідомляє про їх завершення, поміщаючи відповідну подію до черги.

Коли програма ініціює неблокуючу операцію, наприклад прийом мережевого повідомлення або запис даних у сокет, вона передає запит у підсистему введення-виведення та одразу повертається до подієвого циклу. На цьому етапі жодної події ще не створюється, оскільки операція не завершена. Система лише “реєструє намір” виконати певну дію та продовжує працювати у фоновому режимі.

Фактичне додавання події до черги відбувається у момент, коли операційна система визначає, що операція готова до обробки. Наприклад, коли мережею надходять дані від клієнта або коли завершено передачу повідомлення, ядро системи фіксує цю подію і формує сигнал для середовища виконання. Цей сигнал перетворюється на об’єкт події, який містить інформацію про тип події та пов’язані з нею дані.

Після формування події середовище виконання автоматично поміщає її до черги подій. Черга є впорядкованою структурою, яка зберігає всі події, готові до обробки. Важливо, що події додаються до черги у міру їх фактичного завершення, а не у тому порядку, в якому були ініційовані операції. Це дозволяє системі реагувати на найшвидше завершені дії без зайвого очікування.

Події можуть надходити з різних джерел. Це можуть бути мережеві події, такі як отримання повідомлення або запит на підключення нового клієнта, таймерні події, які спрацьовують після закінчення заданого часу, або службові події, пов’язані зі зміною стану з’єднання. Незалежно від джерела, усі вони потрапляють до спільної черги та очікують на обробку.

Подієвий цикл постійно перевіряє чергу подій. Коли в черзі з’являється новий елемент, подієвий цикл забирає його та передає відповідному обробнику. Після завершення обробника керування знову повертається до подієвого циклу, який переходить до наступної події. Якщо черга порожня, цикл переходить у стан очікування і не навантажує процесор. Обробник подій є

функціональним компонентом асинхронної системи, який відповідає за реакцію програми на конкретну подію, отриману з черги подій. Його основне призначення полягає в тому, щоб виконати необхідні дії після завершення неблокуючої операції, не порушуючи роботу подієвого циклу і всієї системи в цілому.

Коли подієвий цикл виявляє у черзі нову подію, він визначає її тип та передає керування відповідному обробнику. Обробник подій запускається у відповідь на цю подію та отримує контекст виконання, який містить дані, пов'язані з завершеною операцією. Наприклад, у разі надходження мережевого повідомлення обробник отримує самі дані повідомлення та інформацію про клієнтське з'єднання.

Після початку виконання обробник подій реалізує логіку, необхідну саме для цього типу події. У клієнт-серверній системі обміну повідомленнями це може бути перевірка коректності отриманих даних, визначення отримувача повідомлення, оновлення внутрішнього стану системи або ініціювання передачі даних іншому клієнту. Важливо, що обробник виконує лише підготовку й запуск наступних дій, а не чекає їх завершення, якщо вони також є асинхронними.

Ключовою властивістю обробників подій є їх короткочасне виконання. Обробник не повинен містити тривалих або блокуючих операцій, оскільки він виконується у межах подієвого циклу. Якщо обробник затримує виконання, це безпосередньо впливає на швидкість обробки інших подій, що може призвести до зростання затримок у всій системі. Тому у добре спроектованій асинхронній архітектурі обробники виконують мінімально необхідну роботу та швидко повертають керування подієвому циклу.

Після завершення роботи обробника подієвий цикл переходить до обробки наступної події з черги. Якщо під час виконання обробника були ініційовані нові асинхронні операції, наприклад надсилання повідомлення іншому

клієнту, система не чекає їх завершення. Вони будуть автоматично відслідковані середовищем виконання, а після завершення відповідні події знову потраплять до черги та будуть оброблені іншими обробниками.

Таким чином, обробник подій працює як реактивний елемент асинхронної системи. Він отримує керування лише у момент, коли подія готова до обробки, швидко виконує прикладну логіку та повертає керування подієвому циклу. Такий підхід дозволяє системі обслуговувати велику кількість одночасних подій, уникати блокувань і забезпечувати стабільну та масштабовану роботу клієнт-серверної системи обміну повідомленнями.

Callback-функція — це функція, яка передається як аргумент іншій функції і викликається пізніше, після настання певної події або завершення операції. Основна ідея полягає в тому, що якась частина логіки відкладається і виконується не одразу, а тоді, коли система повідомить, що потрібна дія завершена.

У контексті асинхронного програмування callback-функції використовуються для обробки результатів неблокуючих операцій, наприклад читання даних із мережі, запису файлу або отримання повідомлення від клієнта. Замість того щоб чекати завершення операції, програма передає callback і продовжує виконувати інші завдання.

Коли програма ініціює асинхронну операцію, вона одночасно реєструє callback-функцію як обробник результату. Сама операція виконується у фоновому режимі. Коли вона завершується, середовище виконання автоматично викликає callback-функцію та передає їй необхідні дані.

Логіка роботи виглядає так:

Програма запускає асинхронну операцію

Разом з операцією передається callback-функція

Основний потік не блокується і продовжує роботу

Після завершення операції система викликає callback

Callback обробляє результат і завершує свою роботу

4.3. Розрахунок економічної доцільності впровадження

Розрахунок економічної доцільності впровадження клієнт-серверної системи обміну повідомленнями здійснюється з урахуванням витрат на її створення та очікуваних практичних і економічних вигод від використання. Оскільки система розробляється в межах бакалаврської роботи, розрахунок має оціночний характер і спрямований на обґрунтування раціональності впровадження обраного технічного рішення.

Початкові витрати на впровадження системи включають витрати часу на аналіз, проєктування, реалізацію та тестування програмного продукту. За умови практичного використання ці витрати можуть бути перераховані у вартість роботи розробника. Додаткові фінансові витрати, пов'язані з придбанням програмного забезпечення або ліцензій, відсутні, оскільки система базується на відкритих технологіях. Витрати на апаратне забезпечення також є мінімальними, оскільки для розгортання серверної частини достатньо стандартного серверного середовища або хмарної платформи з базовими характеристиками.

Очікуваний економічний ефект від впровадження системи проявляється у зниженні експлуатаційних витрат. Асинхронна архітектура дозволяє обслуговувати більшу кількість користувачів без збільшення обчислювальних ресурсів. Це означає, що система може працювати на менш потужному сервері або довше масштабуватися без додаткових витрат. У порівнянні з синхронними рішеннями це забезпечує економію на закупівлі та обслуговуванні серверного обладнання.

Додаткову вигоду становить зменшення витрат на супровід і підтримку системи. Стабільна робота під навантаженням, менша кількість збоїв і

відсутність перевантажень знижують витрати часу на усунення помилок та аварійних ситуацій. Це особливо важливо у системах обміну повідомленнями, де безперервність роботи є критичною.

З урахуванням того, що система може бути використана як основа для майбутніх програмних рішень, економічна доцільність її впровадження зростає з часом. Повторне використання архітектури та програмних компонентів дозволяє скоротити витрати на розробку нових сервісів із подібною функціональністю. Таким чином, початкові витрати на створення системи компенсуються за рахунок довготривалого практичного використання.

Отже, розрахунок економічної доцільності впровадження показує, що клієнт-серверна система обміну повідомленнями з використанням асинхронних технологій є виправданим і доцільним рішенням. Низькі витрати на розробку, мінімальні вимоги до інфраструктури та зниження експлуатаційних витрат забезпечують позитивний економічний ефект і підтверджують обґрунтованість використання асинхронного підходу. Економічну ефективність впровадження клієнт-серверної системи обміну повідомленнями доцільно оцінювати як співвідношення отриманого ефекту до витрат, пов'язаних зі створенням і використанням програмного продукту. Для бакалаврської роботи така оцінка, як правило, має аналітично-пояснювальний характер і не вимагає складних фінансових розрахунків, але повинна логічно обґрунтовувати доцільність впровадження системи. Економічна ефективність визначається шляхом порівняння витрат на розробку та впровадження із вигодами, які отримуються в результаті використання системи. Якщо очікувані вигоди компенсують або перевищують початкові та поточні витрати, впровадження вважається економічно доцільним.

У контексті програмного продукту основна увага приділяється не прибутку безпосередньо, а економії ресурсів та підвищенню продуктивності.

ВИСНОВКИ

У ході виконання бакалаврської роботи на тему «Розробка клієнт-серверної системи обміну повідомленнями з використанням асинхронних технологій» було розглянуто теоретичні та практичні аспекти створення сучасних систем обміну повідомленнями, орієнтованих на ефективну роботу в умовах високого навантаження та одночасної взаємодії великої кількості користувачів.

У першому розділі було проведено аналіз предметної області, розглянуто поняття клієнт-серверної архітектури, особливості систем обміну повідомленнями, а також сучасні підходи до передавання даних.

Дослідження показало, що традиційні синхронні методи взаємодії мають обмеження щодо масштабованості та продуктивності, що обумовлює необхідність використання асинхронних технологій.

У другому розділі здійснено проектування клієнт-серверної системи обміну повідомленнями. Визначено загальну архітектуру системи, обґрунтовано вибір інструментальних засобів і технологій розробки, а також описано структуру серверної та клієнтської частин. Особливу увагу приділено організації асинхронної взаємодії, моделі обробки подій і механізмам ідентифікації клієнтів.

У третьому розділі виконано реалізацію серверної та клієнтської частин системи з використанням асинхронних підходів. Розглянуто принципи організації неблокуючого обміну повідомленнями, обробки подій, повторних підключень та керування сеансами. Проведене тестування підтвердило коректність роботи системи, її стабільність та здатність ефективно обслуговувати декілька одночасних підключень без суттєвого зростання затримок.

У четвертому розділі виконано економічний та практичний аналіз розробленого програмного продукту. Доведено економічну доцільність

використання асинхронних технологій за рахунок зниження навантаження на серверні ресурси, зменшення витрат на апаратне забезпечення та підвищення ефективності експлуатації системи. Аналіз показав, що розроблена система може бути впроваджена з мінімальними фінансовими витратами та має перспективи подальшого розвитку.

Отже, поставлена мета бакалаврської роботи була досягнута, а всі сформульовані завдання виконані. Розроблена клієнт-серверна система обміну повідомленнями з використанням асинхронних технологій є працездатним, ефективним і масштабованим програмним рішенням. Отримані результати можуть бути використані як основа для подальших досліджень, навчальних проєктів або практичних систем обміну даними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

2. Буч Г., Рамбо Д., Якобсон І. Об'єктно-орієнтований аналіз та проєктування. — К.: Видавнича група BHV, 2016.
3. Гальчевський О. В. Розподілені інформаційні системи. — К.: КПІ ім. Ігоря Сікорського, 2019.
4. Глушков В. М. Основи інформаційних технологій. — К.: Наукова думка, 2015.
5. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання.
6. Жуков І. А., Ляшенко В. П. Комп'ютерні мережі та протоколи. — К.: Академія, 2018.
7. Коваленко В. В. Проєктування програмних систем. — Харків: НТУ «ХПІ», 2020.
8. Мельник А. О. Архітектура комп'ютерних систем. — Львів: Львівська політехніка, 2017.
9. Паращук Ю. С. Сучасні технології програмування. — К.: КНУ, 2021.
10. Руденко М. В. Інформаційні системи і технології. — К.: Центр навчальної літератури, 2018.
11. Шаров С. В. Основи клієнт-серверної взаємодії. — Запоріжжя: ЗНУ, 2020.
12. Tanenbaum A. S., Van Steen M. Distributed Systems: Principles and Paradigms. — Pearson, 2017.
13. Coulouris G., Dollimore J., Kindberg T., Blair G. Distributed Systems: Concepts and Design. — Pearson, 2018.
14. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. — Pearson, 2021.
15. Stevens W. R. TCP/IP Illustrated, Volume 1. — Addison-Wesley, 2014.
16. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 2016.
17. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley, 2013.
18. Fowler M. Refactoring: Improving the Design of Existing Code. — Addison-Wesley, 2019.
19. Newman S. Building Microservices. — O'Reilly Media, 2021.

20. Richards M., Ford N. Fundamentals of Software Architecture. — O'Reilly Media, 2020.
21. Bass L., Clements P., Kazman R. Software Architecture in Practice. — Addison-Wesley, 2019.
22. Bloch J. Effective Java. — Addison-Wesley, 2018.
23. Flanagan D. JavaScript: The Definitive Guide. — O'Reilly Media, 2020.
24. Tilkov S., Vinoski S. Node.js: Using JavaScript to Build High-Performance Network Programs. — IEEE Internet Computing, 2016.
25. Grigorik I. High Performance Browser Networking. — O'Reilly Media, 2019.
26. Berners-Lee T., Fielding R., Masinter L. Uniform Resource Identifier (URI): Generic Syntax. — RFC 3986, IETF.
27. Fielding R. Architectural Styles and the Design of Network-Based Software Architectures. — Doctoral Dissertation, 2000.
28. Ionescu M. WebSocket Protocol Explained. — ACM Queue, 2018.
29. Golang Team. Concurrency in Modern Programming. — Google Research Publications, 2019.
30. Kleppmann M. Designing Data-Intensive Applications. — O'Reilly Media, 2017.
31. Ousterhout J. A Philosophy of Software Design. — Yaknyam Press, 2018.
32. Oracle. Java Platform, Standard Edition Documentation. — Oracle Docs.
33. Python Software Foundation. Python Asyncio Documentation.
34. Node.js Foundation. Node.js Event Loop Explained.
35. ISO/IEC 25010:2011. Systems and Software Quality Models.
36. IEEE. Software Engineering Body of Knowledge (SWEBOK). — IEEE Computer Society, 2020.
37. Microsoft. Asynchronous Programming Patterns.
38. Mozilla. WebSocket API Documentation.
39. Abouzeid A. Scalable Real-Time Messaging Systems. — Springer, 2020.
40. Erlang Documentation. Concurrency and Fault Tolerance.
41. Red Hat. Event-Driven Architecture Overview.

ДОДАТКИ

Файл server.js

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

const crypto = require('crypto');

// Створення WebSocket-сервера

const wss = new WebSocket.Server({ port: 8080 });

// Сховище клієнтів

const clients = new Map();

console.log('Сервер запущено на порту 8080');

// Генерація унікального ідентифікатора клієнта

function generateClientId() {

    return crypto.randomBytes(8).toString('hex');

}

// Обробка підключень

wss.on('connection', (ws) => {

    const clientId = generateClientId();
```

```
clients.set(clientId, ws);

console.log(`Клієнт підключився: ${clientId}`);

// Надсилаємо клієнту його ідентифікатор
ws.send(JSON.stringify({
    type: 'system',
    message: 'Підключення успішне',
    clientId: clientId
}));

// Обробка повідомлень від клієнта
ws.on('message', (message) => {
    try {
        const data = JSON.parse(message);

        // Розсилка повідомлення іншим клієнтам
        clients.forEach((clientSocket, id) => {
            if (clientSocket.readyState === WebSocket.OPEN && id !== clientId) {
                clientSocket.send(JSON.stringify({
                    type: 'message',
                    from: clientId,
```

```

        text: data.text,

        timestamp: new Date().toISOString()

    ));

    }

});

    } catch (error) {

        console.error('Помилка обробки повідомлення:', error.message);

    }

});

// Обробка відключення
ws.on('close', () => {

    clients.delete(clientId);

    console.log(`Клієнт відключився: ${clientId}`);

});

// Обробка помилок
ws.on('error', (error) => {

    console.error(`Помилка клієнта ${clientId}:`, error.message);

});

});

```

Запуск сервера

```
npm init -y
```

```
npm install ws
```

```
node server.js
```

Клієнтська частина (HTML + JavaScript)

Файл index.html

```
<!DOCTYPE html>
```

```
<html lang="uk">
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
  <title>Асинхронний чат</title>
```

```
  <style>
```

```
    body {
```

```
      font-family: Arial, sans-serif;
```

```
      background: #f4f4f4;
```

```
    }
```

```
    #chat {
```

```
      width: 400px;
```

```
      margin: 30px auto;
```

```
      background: #fff;
```

```
padding: 15px;

border-radius: 5px;
}

#messages {

    height: 250px;

    overflow-y: auto;

    border: 1px solid #ccc;

    padding: 10px;

    margin-bottom: 10px;
}

input {

    width: 75%;

    padding: 8px;
}

button {

    width: 20%;

    padding: 8px;
}

</style>

</head>

<body>
```

```
<div id="chat">

  <h3>Система обміну повідомленнями</h3>

  <div id="messages"></div>

  <input type="text" id="messageInput" placeholder="Введіть повідомлення">

  <button onclick="sendMessage()">Надіслати</button>

</div>
```

```
<script>

  const socket = new WebSocket('ws://localhost:8080');

  const messagesDiv = document.getElementById('messages');

  let clientId = null;

  socket.onopen = () => {

    addMessage('Система: з'єднання встановлено');

  };

  socket.onmessage = (event) => {

    const data = JSON.parse(event.data);

    if (data.type === 'system') {

      clientId = data.clientId;

      addMessage(`Система: ваш ID ${clientId}`);

    }

  };

}
```

```
}
```

```
if (data.type === 'message') {  
    addMessage(`${data.from}: ${data.text}`);  
}
```

```
};
```

```
socket.onclose = () => {  
    addMessage('Система: з'єднання закрито');  
};
```

```
socket.onerror = () => {  
    addMessage('Система: помилка з'єднання');  
};
```

```
function sendMessage() {  
    const input = document.getElementById('messageInput');  
    if (input.value.trim() !== '') {  
        socket.send(JSON.stringify({  
            text: input.value  
        }));  
        addMessage(`Я: ${input.value}`);  
    }  
}
```

```
        input.value = "";
    }
}

function addMessage(message) {
    const p = document.createElement('p');
    p.textContent = message;
    messagesDiv.appendChild(p);
    messagesDiv.scrollTop = messagesDiv.scrollHeight;
}
</script>

</body>

</html>
```