М. В. Васильківський, Д. В. Михалевський
Г. Г. Бортник

ТЕХНОЛОГІЇ МЕРЕЖЕВОГО ПРОГРАМУВАННЯ та ПАРАЛЕЛЬНОЇ ОБРОБКИ ДАНИХ

Міністерство освіти і науки України
Вінницький національний технічний університет

**М. В. Васильківський, Д. В. Михалевський,
Г. Г. Бортник**

Технології мережевого програмування та паралельної обробки даних

Електронний навчальний посібник

Вінниця
ВНТУ
2026

УДК 621.0
В19

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (Протокол № 10 від 26.02.2026 р..)

Рецензенти:

В. Г. Крижановський, доктор технічних наук, професор
С. М. Захарченко, кандидат технічних наук, професор
О. В. Осадчук, доктор технічних наук, професор

Васильківський, М. В.

В19 Технології мережевого програмування та паралельної обробки даних: навчальний посібник [Електронний ресурс] / Васильківський М. В., Михалевський Д. В., Бортник Г. Г. – Вінниця : ВНТУ, 2026. – (PDF, 134 с.)

ISBN 978-617-8927-01-1 (PDF)

У навчальному посібнику розглянуто сучасні підходи до проєктування, реалізації та оптимізації мережевих застосунків у середовищі багатопотокової обробки інформації. Висвітлено принципи функціонування стека протоколів TCP/IP, особливості мережевого програмування мовою C++, методи синхронізації потоків і керування паралельними процесами. Значну увагу приділено підвищенню продуктивності, використанню сучасних бібліотек і фреймворків, а також питанням інформаційної безпеки та застосуванню захищених протоколів зв'язку. Теоретичний матеріал доповнено практичними прикладами програмного коду та рекомендаціями щодо розроблення мережевих застосунків.

Навчальний посібник призначений для здобувачів вищої освіти ІТ-спеціальностей, аспірантів, викладачів і фахівців у галузі мережевого та паралельного програмування.

УДК 621.0

ISBN 978-617-8927-01-1(PDF)

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API (Application Programming Interface) – інтерфейс програмування застосунків

Boost.Asio – бібліотека C++ для асинхронного введення/виведення

CPU (Central Processing Unit) – центральний процесор

C++ – мова програмування загального призначення

DDoS (Distributed Denial of Service) – розподілена відмова в обслуговуванні

DNS (Domain Name System) – система доменних імен

DoS (Denial of Service) – відмова в обслуговуванні

FTP (File Transfer Protocol) – протокол передавання файлів

HTTP (HyperText Transfer Protocol) – протокол передавання гіпертексту

HTTPS (HyperText Transfer Protocol Secure) – захищений протокол передавання гіпертексту

I/O (Input/Output) – введення/виведення даних

IP (Internet Protocol) – інтернет-протокол

IPC (Inter-Process Communication) – міжпроцесна взаємодія

JSON (JavaScript Object Notation) – формат обміну даними

LAN (Local Area Network) – локальна обчислювальна мережа

MAN (Metropolitan Area Network) – міська обчислювальна мережа

NAT (Network Address Translation) – перетворення мережевих адрес

OS (Operating System) – операційна система

OSI (Open Systems Interconnection) – еталонна модель взаємодії відкритих систем

POSIX (Portable Operating System Interface) – стандарт інтерфейсів операційних систем

QoS (Quality of Service) – якість обслуговування

RAM (Random Access Memory) – оперативна пам’ять

RPC (Remote Procedure Call) – віддалений виклик процедур

SDK (Software Development Kit) – набір засобів розробника

SSL (Secure Sockets Layer) – протокол захищеного з’єднання

STL (Standard Template Library) – стандартна бібліотека шаблонів C++

TBB (Threading Building Blocks) – бібліотека Intel для паралельного програмування

TCP (Transmission Control Protocol) – протокол керування передаванням

TLS (Transport Layer Security) – протокол захисту транспортного рівня

TOCTOU (Time Of Check To Time Of Use) – вразливість «час перевірки — час використання»

UDP (User Datagram Protocol) – протокол користувацьких датаграм

URL (Uniform Resource Locator) – уніфікований локатор ресурсів

WAN (Wide Area Network) – глобальна обчислювальна мережа

XML (eXtensible Markup Language) – розширювана мова розмітки

ВСТУП

Сучасні інформаційно-комунікаційні системи функціонують на основі складних мережевих технологій, що забезпечують надійне, швидке та безпечне передавання даних між розподіленими обчислювальними ресурсами. Розвиток інтернет-технологій, хмарних сервісів, мобільних застосунків та систем Інтернету речей зумовлює зростання вимог до продуктивності, масштабованості та надійності мережевого програмного забезпечення. У цьому контексті особливої актуальності набуває підготовка фахівців, здатних проєктувати, реалізовувати та оптимізувати мережеві додатки з використанням сучасних протоколів і технологій паралельної обробки даних.

Навчальний посібник присвячений вивченню основ мережевого програмування, принципів функціонування набору протоколів TCP/IP, методів синхронної та асинхронної комунікації, а також застосуванню багатопотоковості в мережевих додатках мовою програмування C++. У посібнику поєднано теоретичні відомості з практичними прикладами, що сприяє формуванню цілісного уявлення про сучасні підходи до розробки мережевого програмного забезпечення.

Метою навчального посібника є формування у здобувачів освіти системних знань про архітектуру комп’ютерних мереж, принципи взаємодії протоколів, особливості організації мережевого введення/виведення та методи ефективного керування паралельними обчисленнями. Особлива увага приділяється питанням синхронізації потоків, обробки помилок, оптимізації продуктивності та забезпечення надійності багатопотокових мережевих систем.

Посібник структуровано відповідно до логіки поетапного засвоєння матеріалу – від базових протоколів і моделей передавання даних до складних механізмів асинхронної обробки та багатопотоковості. Навчальний матеріал супроводжується прикладами програмного коду, контрольними питаннями та практичними рекомендаціями, що сприяють закріпленню теоретичних знань і розвитку навичок практичного програмування.

Навчальний посібник призначений для здобувачів вищої освіти, які навчаються за спеціальностями у галузі інформаційних технологій, телекомунікацій та комп’ютерної інженерії, а також може бути корисним для викладачів, аспірантів і фахівців, що займаються розробкою та супроводом мережевих систем.

1 НАБІР ПРОТОКОЛІВ TCP/IP

У розділі розглядається набір протоколів TCP/IP, який є базовою архітектурою сучасних інтернет-комунікацій. Описується шарова структура моделі TCP/IP із детальним аналізом функцій та взаємодії ключових протоколів, зокрема IP, TCP та UDP. Наголошується на їхніх відмінних ролях у процесі передавання даних, а також розглядаються протоколи прикладного рівня, такі як HTTP і FTP, із висвітленням їх інтеграції в межах набору TCP/IP. Крім того, подано огляд мережевих інструментів і утиліт, необхідних для аналізу та діагностики мереж на основі TCP/IP, що забезпечує цілісне розуміння цих важливих комунікаційних протоколів.

1.1 Структура моделі TCP/IP

Модель TCP/IP, або набір інтернет-протоколів, є концептуальною основою для розуміння та налаштування мережевих протоколів. Вона бере свій початок із проєкту ARPANET Міністерства оборони США, що зумовило формування надійної моделі для проєктування протоколів у великомасштабних мережах. На відміну від моделі OSI, яка складається із семи рівнів, модель TCP/IP традиційно містить чотири рівні: рівень каналу (Link Layer), інтернет-рівень (Internet Layer), транспортний рівень (Transport Layer) та прикладний рівень (Application Layer). Така структура визначає принципи передавання даних у різнорідних мережах і охоплює всі процеси мережевої взаємодії – від апаратного забезпечення до прикладного програмного забезпечення.

Основною ідеєю моделі TCP/IP є поділ мережевих функцій на окремі рівні, кожен з яких виконує чітко визначені завдання. Така декомпозиція дає змогу модифікувати або розширювати один рівень без впливу на інші, що сприяє масштабованості та гнучкості розробки. Крім того, модель TCP/IP реалізує підхід «знизу вгору», починаючи з фізичних операцій мережі на рівні каналу та завершуючи взаємодією з користувачем на прикладному рівні. Далі детально розглядаються функції кожного з цих рівнів і протоколи, що на них функціонують.

Рівень каналу є фундаментальним рівнем моделі TCP/IP і відповідає за фізичне передавання пакетів даних. Він функціонує на межі між матеріальними апаратними компонентами та абстрактними мережевими рівнями, визначаючи способи фізичного передавання даних різними типами середовищ.

До основних завдань цього рівня належать формування кадрів, адресація та керування протоколами рівня керування доступом до середовища (MAC). До нього належать такі протоколи, як Ethernet, Wi-Fi (IEEE 802.11) та інші, які регламентують взаємодію пристроїв у межах одного мережевого сегмента. Ці протоколи забезпечують виявлення та корекцію помилок передавання даних, синхронізацію пристроїв і середовищ передавання, а також безконфліктну передачу шляхом

застосування механізмів Carrier Sense Multiple Access with Collision Detection (CSMA/CD) або Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA).

Важливими елементами керування на цьому рівні є мережеві інтерфейсні карти (NIC), які забезпечують взаємодію програмно-апаратних компонентів із фізичним середовищем передавання даних, таким як оптоволоконні або мідні кабелі.

Ілюстрація керування доступом у мережах Ethernet із використанням CSMA/CD під час передавання даних.

Механізм CSMA/CD (Carrier Sense Multiple Access with Collision Detection) регламентує доступ вузлів до спільного середовища передавання в мережах Ethernet. Алгоритм його роботи можна описати так:

якщо канал вільний:
    передати кадр
    очікувати підтвердження
інакше, якщо виникла колізія:
    припинити передавання
    зачекати випадковий інтервал часу
    повторити спробу

Такий підхід забезпечує координацію доступу до середовища, зменшує ймовірність повторних колізій за рахунок випадкової затримки (алгоритм експоненційної відстрочки) та підвищує загальну ефективність використання каналу зв’язку в умовах спільного доступу кількох вузлів.

Розташований над канальним рівнем, мережевий рівень відіграє ключову роль у формуванні шляхів передавання даних між мережами, забезпечуючи логічну адресацію вузлів і керування маршрутизацією пакетів даних у взаємопов’язаних мережах. Саме цей рівень є основою побудови магістральної структури Інтернету, оскільки забезпечує глобальне об’єднання множини локальних і регіональних мереж у єдину комунікаційну систему.

Найважливішим протоколом мережевого рівня є Internet Protocol (IP), який представлено двома основними версіями — IPv4 та IPv6. Протокол IP відповідає за адресацію мережевих пристроїв, а також за фрагментацію пакетів даних під час передавання через мережі з різними характеристиками та їх коректне збирання на стороні одержувача.

IP-адреси є унікальними ідентифікаторами мережевих пристроїв, що дає змогу пакетам даних точно визначати пункт призначення. Для вибору оптимального шляху передавання пакетів у межах мережевого рівня застосовуються протоколи маршрутизації, зокрема Routing Information Protocol (RIP), а також більш складні та масштабовані рішення, такі як Open Shortest Path First (OSPF) і Border Gateway Protocol (BGP), які забезпечують ефективну маршрутизацію в корпоративних і глобальних мережах.

Простий скрипт мовою Python, що демонструє перетворення доменного імені IPv4 у відповідну IP-адресу з використанням механізмів системи доменних імен (DNS).

```
# Простий Python-скрипт, що демонструє отримання IPv4-адреси за доменним ім'ям
import socket

hostname = "www.example.com"
ip_address = socket.gethostbyname(hostname)

print(f"IP-адреса вузла {hostname} становить {ip_address}")
```

У наведеному прикладі використовується стандартний модуль `socket`, який надає інтерфейс доступу до мережевих функцій операційної системи.

Функція `gethostbyname()` виконує DNS-запит і повертає IPv4-адресу, що відповідає заданому доменному імені.

Такий підхід широко застосовується в мережевому програмуванні для попереднього визначення адрес вузлів перед встановленням сокетного з'єднання.

```
# Отримання IPv4-адреси за доменним ім'ям з обробкою DNS-помилок
import socket
import time

hostname = "www.example.com"
max_retries = 3

for attempt in range(1, max_retries + 1):
    try:
        ip_address = socket.gethostbyname(hostname)
        print(f"IP-адреса вузла {hostname}: {ip_address}")
        break
    except socket.gaierror as e:
        print(f"Спроба {attempt}: DNS-запит не вдався ({e}). Повтор...")
        time.sleep(2)
else:
    print("Не вдалося отримати IP-адресу після кількох спроб.")
```

Реалізовано механізм повторних спроб (retry) – стандартну практику в мережевих застосунках.

Обробка винятку socket.gaierror дозволяє програмі:

- не завершуватися аварійно;
- коректно повідомляти користувача про проблему;
- працювати в нестабільних мережевих умовах.

Такий підхід відповідає принципам надійності та відмовостійкості мережевих систем. DNS-резолюція є зовнішньою залежністю, тому будь-який мережевий застосунок має передбачати тимчасові збої та реалізовувати механізми повтору або деградації сервісу.

```
import socket

hostname = "www.example.com"

try:
    addr_info = socket.getaddrinfo(hostname, None)
    for entry in addr_info:
        print(f"Отримано адресу: {entry[4][0]}")
except socket.gaierror as e:
    print(f"Помилка DNS-резолюції: {e}")
```

Навіть за наявності стабільного мережевого з’єднання, DNS-запити можуть завершуватися помилками через обмеження середовища виконання. Тому мережеві застосунки мають завжди передбачати обробку винятків та механізми повторних спроб.

Транспортний рівень є ключовим компонентом забезпечення надійного наскрізного (end-to-end) зв’язку та відповідає за керування передаванням повідомлень між вузлами мережі. Його функції охоплюють встановлення, підтримання та коректне завершення сеансів зв’язку, а також контроль за доставкою даних між прикладними процесами на різних хостах. Саме на цьому рівні реалізуються механізми, що гарантують цілісність і узгодженість обміну інформацією.

На транспортному рівні домінують два базові протоколи: Transmission Control Protocol (TCP) та User Datagram Protocol (UDP). Кожен із них реалізує власну модель обслуговування і призначений для різних типів мережевих застосунків.

Протокол TCP надає сервіс, орієнтований на з’єднання, забезпечуючи надійну передачу даних завдяки механізмам виявлення та виправлення помилок, впорядкування пакетів і керування потоком. Такий підхід робить TCP доцільним для застосунків, у яких критично важливими є цілісність і правильна послідовність даних, зокрема для вебперегляду з використанням протоколу HTTP або передавання електронної пошти через SMTP.

```
# Псевдокод простого TCP-клієнта
процедура tcp_client():
    створити сокет
    встановити з’єднання з сервером
    надіслати дані
    прийняти відповідь
    закрити з’єднання
```

Наведений псевдокод відображає базову послідовність дій клієнтської програми, що використовує протокол TCP. Спочатку ініціалізується сокет, після чого виконується встановлення з'єднання з віддаленим сервером. Далі здійснюється передавання запиту та приймання відповіді, а завершальним етапом є коректне закриття з'єднання з метою звільнення системних ресурсів.

UDP, на відміну від TCP, надає безз'єднувальний сервіс із мінімальними накладними витратами, оскільки не забезпечує гарантій надійності передавання даних. Цей протокол застосовується в прикладних системах, де пріоритетом є висока швидкість обміну інформацією без механізмів контролю помилок і повторної передачі, зокрема у відеостримінгу, системах потокового мультимедійного мовлення та мережевих онлайн-іграх.

```
# Псевдокод простого UDP-клієнта
def udp_client():
    створити сокет
    прив'язати сокет до локальної адреси (за потреби)
    надіслати дані до вузла призначення
    прийняти відповідь від сервера
    закрити сокет
```

UDP-клієнт працює без встановлення логічного з'єднання із сервером. Передавання даних здійснюється шляхом надсилання датаграм до певної адреси призначення, після чого клієнт може отримати відповідь. Відсутність етапів встановлення та підтримки з'єднання зумовлює низькі накладні витрати та високу швидкодію, однак не гарантує надійності доставки повідомлень.

Вибір між TCP та UDP визначається вимогами конкретного додатка щодо швидкодії, надійності та накладних витрат.

Прикладний рівень розташований на верхівці моделі TCP/IP і інтегрує всі протоколи та технології, розроблені для виконання специфічних завдань комунікації, забезпечуючи необхідні інтерфейси для мережевих сервісів. До таких протоколів належать Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Domain Name System (DNS) та інші.

HTTP, як базовий протокол вебкомунікацій, забезпечує доступ до вебресурсів та працює за принципом клієнт-сервер. Типова HTTP-транзакція передбачає відправлення клієнтом запиту на ресурс і відповідь сервера з необхідним ресурсом, часто разом із кодом стану HTTP.

Приклад виконання HTTP GET-запиту з використанням бібліотеки requests у Python

Бібліотека requests надає зручний високорівневий інтерфейс для взаємодії з вебресурсами за протоколом HTTP

```
import requests  # Імпорт бібліотеки для роботи з HTTP-запитами
```

```
# Надсилання GET-запиту до вказаного вебресурсу
response = requests.get('https://www.example.com')

# Виведення вмісту відповіді сервера (тіла HTTP-відповіді)
# Дані повертаються у вигляді байтового рядка
print(response.content)
```

У наведеному прикладі демонструється базовий механізм взаємодії клієнтського застосунку з веб-сервером на прикладному рівні моделі TCP/IP з використанням протоколу HTTP.

Варіант 1

```
from urllib.request import urlopen
from urllib.error import URLError
import time

url = "https://www.example.com"
max_retries = 3

for attempt in range(max_retries):
    try:
        with urlopen(url, timeout=5) as response:
            content = response.read()
            print(content)
            break
    except URLError as e:
        print(f"Помилка мережі (спроба {attempt + 1}): {e}")
        time.sleep(2 ** attempt)
else:
    print("Не вдалося встановити з’єднання після кількох спроб")
```

Варіант 2

```
from urllib.request import urlopen

url = "https://93.184.216.34"  # IP example.com

with urlopen(url) as response:
    print(response.read())
```

Аналогічно, протокол FTP забезпечує передавання файлів у мережах, тоді як SMTP відіграє ключову роль у пересиланні електронних поштових повідомлень. DNS виконує перетворення доменних імен, зрозумілих людині, у машинно-читані IP-адреси, що є необхідним для ідентифікації та адресації пристроїв у мережах на основі протоколу IP.

Протоколи прикладного рівня тісно залежать від рівнів, розташованих нижче, що ілюструє складні взаємозв’язки між усіма рівнями моделі TCP/IP, які спільно забезпечують повноцінну мережеву комунікацію.

У межах моделі TCP/IP важливу роль відіграє процес інкапсуляції даних. Під час передавання інформації з верхніх рівнів до канального рівня кожен рівень інкапсулює дані попереднього, додаючи власну службову інформацію у вигляді заголовка. На приймальному боці кожен рівень, навпаки, інтерпретує отримані дані та видаляє відповідний заголовок. Цей механізм є фундаментальним для розуміння принципів успішного передавання даних через різнорідні мережеві інфраструктури.

Завдяки своїй багаторівневій архітектурі модель TCP/IP є стійким та адаптивним середовищем, яке витримало перевірку часом. Вона ефективно задовольняє вимоги сучасних інтернет-комунікацій і водночас дозволяє впроваджувати інновації та розвиток без необхідності повної модернізації всіх складових системи.

1.2 Протокол Інтернету (IP)

Протокол Інтернету (Internet Protocol, IP) є базовим протоколом інтернет-рівня моделі TCP/IP і забезпечує маршрутизацію пакетів між мережами. Він виступає основним комунікаційним протоколом, відповідальним за доставку пакетів від вузла-джерела до вузла-призначення виключно на основі їхніх IP-адрес. Повідомлення, відомі як датаграми, проходять через різні мережі та доставляються до коректного адресата. IP є ключовим елементом функціонування Інтернету, забезпечуючи взаємодію комп’ютерів, з’єднаних через різноманітні мережеві середовища.

У цьому підрозділі розглядаються основні аспекти протоколу IP, зокрема його архітектура, механізми адресації та маршрутизації, а також відмінності між версіями протоколу.

IP-адресація є центральним елементом функціонування протоколу IP та слугує унікальним ідентифікатором кожного пристрою, підключеного до мережі. IP-адреса виконує дві основні функції: ідентифікацію вузла або мережевого інтерфейсу та визначення його місцезнаходження в мережі, що забезпечує ефективну маршрутизацію трафіка. IP-адреса зазвичай має числове подання зі структурованою організацією, яка дозволяє здійснювати ідентифікацію мережі та маршрутизацію пакетів. Залежно від версії протоколу розрізняють два основні типи IP-адрес: IPv4 та IPv6.

IPv4, перша широко впроваджена версія протоколу IP, використовує 32-бітну схему адресації, що забезпечує понад 4 мільярди унікальних адрес. Такі адреси зазвичай подаються у десятково-крапковому форматі, який складається з чотирьох октетів, розділених крапками, наприклад: 192.168.1.1. IPv4-адреси традиційно поділяються на класи (A, B, C, D, E) відповідно до розмірів і призначення мереж. Однак класова адресація значною мірою втратила актуальність із запровадженням безкласової міждоменної маршрутизації (CIDR), яка забезпечує більш гнучкий і ефективний розподіл IP-адресного простору.

```
# Python demonstration for converting an IPv4 address to binary

ip = "192.168.1.1"

ip_as_binary = '.'.join(
    [f"{int(octet):08b}" for octet in ip.split('.')]
)

print(f"The binary representation of {ip} is {ip_as_binary}")
```

- IP-адреса розбивається на октети за допомогою `split('.')`;
- кожен октет перетворюється у ціле число та далі – у двійковий формат з фіксованою довжиною 8 біт (`:08b`);
- результати об’єднуються назад у рядок через крапку.

Результат виконання:

```
The binary representation of 192.168.1.1 is 11000000.10101000.00000001.00000001
```

Нотація CIDR, наприклад 192.168.100.0/24, визначає кількість бітів IP-адреси, що використовуються для мережевої частини, підвищуючи ефективність розподілу адресного простору та покращуючи продуктивність маршрутизації.

Адреси IPv6. Обмеження протоколу IPv4, зокрема вичерпання адресного простору, зумовили розроблення протоколу IPv6, який використовує 128-бітний адресний простір і суттєво розширює доступний пул IP-адрес для забезпечення зростаючої кількості пристроїв, підключених до Інтернету. Адреса IPv6 зазвичай подається у шістнадцятковому форматі та розділяється двокрапками, наприклад: 2001:0db8:85a3:0000:0000:8a2e:0370:7334.

Протокол IPv6 запроваджує низку удосконалень, зокрема спрощений формат заголовків, покращену підтримку розширювальних заголовків, а також вбудовану (нативну) підтримку механізмів безпеки IPsec. Поширення IPv6 поступово зростає, оскільки організації готуються до неминучого вичерпання адресного простору IPv4.

Приклад коду Python для демонстрації маніпулювання адресами IPv6

```
import ipaddress

# Створення об’єкта IPv6-адреси
ip = ipaddress.IPv6Address('2001:0db8:85a3:0000:0000:8a2e:0370:7334')

# Виведення стисненої форми IPv6-адреси
print(f"Стиснена форма IPv6-адреси: {ip.compressed}")
```

Наведений фрагмент коду демонструє використання стандартного модуля ipaddress мови Python для роботи з IPv6-адресами, зокрема перетворення повного запису адреси у її стиснену канонічну форму.

Маршрутизація IP є однією з ключових функцій протоколу Інтернету, що відповідає за визначення шляху проходження пакетів даних від джерела до пункту призначення. Цей процес передбачає передавання пакетів через проміжні вузли, відомі як маршрутизатори, які переспрямовують пакети до кінцевих адресатів відповідно до наперед визначеного набору правил і протоколів маршрутизації.

Маршрутизатори формують і підтримують таблиці маршрутизації, використовуючи їх для ефективного пересилання пакетів у мережах. Протоколи маршрутизації поділяються на два основні типи: протоколи внутрішніх шлюзів (Interior Gateway Protocols, IGP) та протоколи зовнішніх шлюзів (Exterior Gateway Protocols, EGP). Протоколи IGP, зокрема RIP і OSPF, застосовуються в межах одного адміністративного домену, тоді як протоколи EGP, передусім BGP, використовуються для маршрутизації між різними доменами.

Псевдокод, що ілюструє базову логіку пересилання IP-пакетів

```
під час отримання пакета:
    визначити адресу призначення
    виконати пошук у таблиці маршрутизації для визначення наступного переходу
    якщо маршрут знайдено:
        надіслати пакет на наступний вузол (next hop)
    інакше:
        надіслати ICMP-повідомлення «адреса призначення недосяжна»
```

Наведений псевдокод відображає узагальнений алгоритм роботи маршрутизатора під час оброблення IP-пакетів і демонструє основні етапи прийняття рішення щодо їх подальшого пересилання або формування повідомлення про помилку.

Протокол керівних повідомлень Інтернету (Internet Control Message Protocol, ICMP) є невід'ємною складовою стека протоколів IP і використовується насамперед для мережевої діагностики та повідомлення про помилки. Маршрутизатори й вузли мережі застосовують ICMP для обміну інформацією про проблеми, що виникають під час оброблення IP-пакетів. Наприклад, у разі недоступності вузла ICMP може бути використаний для інформування вузла-відправника про відповідну проблему.

До поширених типів ICMP-повідомлень належать echo request та echo reply (які використовуються утилітою ping), а також повідомлення типу «адреса призначення недосяжна». ICMP-повідомлення є важливим інструментом для мережевих адміністраторів, оскільки забезпечують ефективне виявлення несправностей і сприяють оперативному усуненню проблем у мережі.

Наведений приклад демонструє використання утиліти ping для перевірки мережевої досяжності вузла за протоколом IPv6:

Команда

`ping -6 www.google.com`

ініціює надсилання ICMPv6-повідомлень типу echo request до доменного імені www.google.com, яке було зіставлене з IPv6-адресою 2a00:1450:4009:802::200e.

Отримане повідомлення

`Reply from 2a00:1450:4009:802::200e: time=30ms`

свідчить про успішне надходження відповіді echo reply від віддаленого вузла. Значення time=30 ms характеризує затримку (час проходження пакета в обидва боки) між локальним хостом і цільовим сервером.

Таким чином, результат виконання команди підтверджує коректну роботу IPv6-з'єднання та доступність віддаленого ресурсу в мережі.

Фрагментація виникає у випадках, коли розмір IP-пакета перевищує максимальний розмір передавання кадру (Maximum Transmission Unit, MTU) на шляху передавання. Протокол IP забезпечує фрагментацію шляхом поділу великих пакетів на менші одиниці – фрагменти, які на вузлі призначення повторно об'єднуються в початковий пакет.

Цей процес відбувається прозоро для протоколів вищих рівнів моделі взаємодії відкритих систем, однак його важливість проявляється під час забезпечення цілісності доставлення даних у мережах з різними значеннями MTU.

Псевдокод базового алгоритму фрагментації пакетів

```
якщо розмір пакета > MTU:
   розділити пакет на фрагменти
   для кожного фрагмента:
      додати необхідну службову інформацію заголовка
      надіслати фрагмент
інакше:
   надіслати пакет без фрагментації
```

Наведений псевдокод ілюструє узагальнений механізм фрагментації IP-пакетів, який застосовується у разі перевищення розміру пакета допустимого значення MTU та забезпечує коректне передавання даних мережею.

Протокол визначення адрес (Address Resolution Protocol, ARP) функціонує на рівні Інтернету та забезпечує встановлення відповідності між IP-адресами та MAC-адресами (Media Access Control). Ця можливість є критично важливою для організації обміну даними в мережах Ethernet.

Коли пристрій має намір встановити зв'язок з іншим пристроєм у межах тієї самої локальної мережі, він розсилає широкомовний ARP-запит з метою отримання MAC-адреси, що відповідає цільовій IP-адресі. Протокол ARP працює виключно в межах локальної мережі та забезпечує надійне встановлення безпосередніх каналів взаємодії між пристроями, необхідних для коректного доставлення пакетів даних.

Приклад формування та надсилання ARP-запиту з використанням бібліотеки Scapy (Python)

```
from scapy.all import ARP, send

# Створення ARP-запиту (op=1 — ARP request) для визначення MAC-адреси вузла з IP 192.168.1.1
arp_pkt = ARP(op=1, pdst='192.168.1.1')

# Надсилання ARP-пакета в мережу
send(arp_pkt)
```

Наведений приклад демонструє використання бібліотеки Scapy для програмного формування та надсилання ARP-запиту. У процесі виконання коду ініціюється широкомовний запит з метою отримання MAC-адреси, що відповідає заданій IP-адресі в межах локальної мережі. Такий підхід широко застосовується для аналізу та діагностики мережевих з'єднань, а також у навчальних і дослідницьких цілях.

Незважаючи на критичну роль IP, протокол стикається з низкою викликів. Неминуче вичерпання адрес IPv4 зумовило перехід до IPv6, що потребувало значної модернізації інфраструктури та врахування питань сумісності.

Ще однією проблемою є безпека, оскільки IP за своєю природою не забезпечує надійних механізмів для гарантування цілісності та автентичності даних. Ця недосконаліть робить протокол уразливим до різних видів атак, зокрема підміни IP-адрес (IP spoofing).

Фахівці з мережевих технологій застосовують IPsec – додатковий набір протоколів, що забезпечує захищене передавання даних через IP, надаючи автентифікацію, контроль цілісності та конфіденційність інформації.

Ще одним рішенням для подолання обмежень IP, зокрема нестачі публічних IP-адрес, є Network Address Translation (NAT). NAT дозволяє кільком пристроям локальної мережі використовувати одну спільну публічну IP-адресу. Проте застосування NAT може ускладнювати безпосередню взаємодію «peer-to-peer» і створювати обмеження для роботи окремих застосунків.

Псевдокод для простого пошуку у таблиці трансляції NAT у разі обробки вхідного пакета

```
отримати IP-адреси джерела та призначення пакета
якщо у таблиці NAT існує відповідність:
    виконати трансляцію IP-адрес згідно з інформацією заголовка
    переслати пакет далі
інакше:
    відкинути пакет або створити новий запис у таблиці NAT
```

Цей псевдокод ілюструє базовий механізм роботи Network Address Translation (NAT), який дозволяє здійснювати трансляцію адрес між локальною та публічною мережею, забезпечуючи спільне використання обмеженого пулу публічних IP-адрес і контроль над маршрутизацією пакетів.

Інтернет-протокол є основою сучасних мережевих технологій, демонструючи високий ступінь адаптивності, що проявляється у переході від IPv4 до IPv6, підвищенні ефективності маршрутизації та постійному вдосконаленні засобів діагностики й усунення проблем за допомогою ICMP та суміжних технологій. Ці знання формують фундамент для подальшого розвитку комп'ютерних мереж і забезпечують безперебійну взаємодію цифрових екосистем.

1.3 Протокол керування передаванням (TCP)

Протокол TCP є ключовим компонентом стека протоколів Інтернету, необхідним для забезпечення надійного обміну даними в пакетно-комутованих мережах. Розташований на транспортному рівні моделі TCP/IP, TCP забезпечує орієнтовану на з'єднання, з перевіркою помилок та можливістю повторної передачі, службу доставки даних. TCP дозволяє надійно відновлювати потоки даних з отриманих датаграм, що робить його незамінним для застосунків, де критично важливі цілісність і порядок даних. Цей розділ містить детальний опис TCP, його механізмів, внутрішньої архітектури та значення у практичному мережевому контексті.

TCP відзначається низкою ключових особливостей.

Орієнтація на з'єднання (Connection-Oriented). TCP встановлює з'єднання між двома кінцевими точками перед початком передачі даних, формуючи TCP-сесію та забезпечуючи готовність до обміну інформацією.

Надійна передача даних (Reliable Data Transfer). Надійність досягається за допомогою номерів послідовності, підтверджень та повторних передач, що гарантує доставку пакетів без помилок і у правильному порядку.

Управління потоком (Flow Control). TCP регулює потік даних між відправником і отримувачем, щоб уникнути перевантаження повільних пристроїв.

Управління перевантаженням (Congestion Control). Методи, такі як slow start, congestion avoidance та fast recovery, оптимізують роботу мережі, регулюючи потік даних у змінних умовах.

Орієнтація на потік (Stream-Oriented). TCP розглядає дані як безперервний потік, на відміну від UDP, який передає окремі датаграми.

Сукупність цих властивостей дозволяє TCP підтримувати різноманітні застосунки – від веббраузерів до поштових клієнтів, де надійна та впорядкована доставка даних є обов'язковою.

Дані інкапсулюються у TCP-сегменти, кожен з яких містить заголовок та корисне навантаження. Заголовок TCP, стандартного розміру 20 байт,

містить поля, необхідні для управління сегментами та забезпечення надійності:

Порти джерела і призначення (16 біт кожен). Визначають кінцеві точки зв'язку та забезпечують мультиплексування застосунків.

Номер послідовності (32 біти). Ідентифікує порядковий номер сегмента у потоці даних, забезпечуючи правильне відновлення на стороні отримувача.

Номер підтвердження (32 біти). Підтверджує отриманий діапазон даних і сигналізує про готовність до подальшої передачі.

Зсув (Offset, 4 біти). Вказує початок даних у сегменті, визначаючи довжину заголовка.

Прапори (Flags, 9 біт). Контрольні індикатори, такі як SYN, ACK, FIN, RST, URG, для керування життєвим циклом з'єднання та обробки термінових даних.

Вікно (Window, 16 біт). Визначає розмір буфера для контролю потоку, сигналізуючи про готовність отримувача приймати додаткові дані.

Контрольна сума (Checksum, 16 біт). Перевіряє цілісність сегмента, обчислюється для заголовка та корисного навантаження.

Терміновий покажчик (Urgent Pointer, 16 біт, опціонально). Вказує місцезнаходження термінових даних у сегменті.

Цей опис забезпечує повне розуміння внутрішньої структури TCP і його ролі у забезпеченні надійної доставки даних у сучасних мережах.

Псевдокод для побудови базового заголовка TCP

```
структура TCPHeader {
    unsigned short source_port;       # Порт джерела
    unsigned short dest_port;         # Порт призначення
    unsigned int sequence;            # Номер послідовності
    unsigned int acknowledgment;      # Номер підтвердження
    unsigned short offset_and_flags;  # Зсув і прапори управління
    unsigned short window;            # Розмір вікна для контролю потоку
    unsigned short checksum;          # Контрольна сума для перевірки цілісності
    unsigned short urgent_pointer;    # Терміновий покажчик (за потреби)
}
```

Цей псевдокод демонструє основну структуру TCP-заголовка, необхідну для формування сегмента TCP. Кожне поле виконує специфічну функцію: від ідентифікації кінцевих точок зв'язку до забезпечення надійності доставки, контролю потоку та управління терміновими даними.

Ця структурна основа підтримує ключові механізми надійності TCP, роблячи його незамінним для багатьох критично важливих мережевих застосунків.

Орієнтованість TCP на з'єднання реалізується через процес триетапного рукостискання (three-way handshake) – критичний протокольний механізм, що забезпечує встановлення надійної сесії між двома вузлами мережі.

SYN. Клієнт ініціює з’єднання, надсилаючи пакет SYN (synchronize), який містить його початковий номер послідовності.

SYN-ACK. Сервер відповідає сегментом SYN-ACK (synchronize-acknowledge), підтверджуючи отримання SYN-клієнта та повідомляючи свій початковий номер послідовності.

ACK. Клієнт надсилає пакет ACK (acknowledge), підтверджуючи номер послідовності сервера та завершуючи процедуру встановлення з’єднання.

Такий механізм гарантує надійне встановлення TCP-сесії перед початком передачі даних.

Приклад Python з використанням бібліотеки Scapy для симуляції TCP SYN-пакета

```
from scapy.all import IP, TCP, send

# Визначення IP-адрес та портів джерела і призначення
src_ip = "192.168.1.2"
dst_ip = "192.168.1.1"
src_port = 1234
dst_port = 80

# Формування IP-пакета
ip_packet = IP(src=src_ip, dst=dst_ip)

# Формування TCP-пакета з прапором SYN (ініціація з'єднання) та початковим номером послідовності
tcp_packet = TCP(sport=src_port, dport=dst_port, flags='S', seq=1000)

# Об’єднання IP і TCP у єдиний пакет
packet = ip_packet / tcp_packet

# Надсилання пакета в мережу
send(packet)
```

Цей приклад демонструє, як за допомогою Scapy можна програмно згенерувати TCP-пакет із прапором SYN, що відповідає першому етапу триетапного рукостискання TCP. Такий підхід широко використовується для навчальних цілей, тестування мережевих з’єднань і аналізу протоколів.

Приклад формування TCP-заголовка без реального надсилання пакетів

```
import struct

# Вхідні параметри
source_port = 1234
dest_port = 80
sequence_number = 1000
ack_number = 0
```

```
data_offset = 5        # 5 * 4 = 20 байт (мінімальний TCP-заголовок)
flags = 0x02           # SYN = 00000010
window_size = 8192
checksum = 0           # Для простоти не обчислюється
urgent_pointer = 0

# Об’єднання data offset і flags в одне поле
offset_and_flags = (data_offset << 12) | flags

# Формування TCP-заголовка (20 байт)
tcp_header = struct.pack(
    "!HHIIHHHH",
    source_port,
    dest_port,
    sequence_number,
    ack_number,
    offset_and_flags,
    window_size,
    checksum,
    urgent_pointer
)

print("TCP SYN-заголовок сформовано успішно")
print("Довжина заголовка:", len(tcp_header), "байт")
print("Бінарне подання:", tcp_header)
```

Триетапне рукостискання (three-way handshake) встановлює повнодуплексне з’єднання, забезпечуючи синхронізацію обох сторін для обміну даними.

Передача даних у межах встановленого TCP-з’єднання базується на номерах послідовності та підтвердженнях (ACK) для підтримки надійності. TCP забезпечує цілісність і впорядкованість даних за допомогою надійних механізмів:

Протокол ковзного вікна (Sliding Window Protocol). Динамічний розмір вікна визначає кількість байтів, що можуть бути надіслані без очікування підтверджень, узгоджуючи швидкість відправника з готовністю отримувача. Розмір вікна коригується залежно від умов мережі та можливостей приймача.

Кумулятивні підтвердження (Cumulative Acknowledgments). TCP надсилає кумулятивні ACK, що підтверджують правильне отримання всіх даних до певного номера послідовності.

Селективні підтвердження (Selective Acknowledgment, SACK). Опційне розширення, яке дозволяє підтверджувати окремі сегменти даних, особливо ефективне у мережах із високою затримкою або під час втрати пакетів.

Таймери повторної передачі (Retransmission Timers). TCP використовує таймери для виявлення втрачених пакетів і ініціює повторну передачу, якщо підтвердження не надходять протягом заданого часу.

Ці механізми разом забезпечують надійну, впорядковану та ефективну передачу даних у TCP-з’єднаннях, навіть у складних умовах мережі.

Псевдокод для повторної передачі TCP-сегмента

```
встановити таймер
надіслати сегмент

поки очікується ACK:
   якщо таймер сплив:
      повторно надіслати сегмент
      скинути таймер
   якщо отримано ACK:
      надіслати наступний сегмент
      оновити ковзне вікно (sliding window)
```

Ці властивості підвищують стійкість TCP до нестабільності мережі, забезпечуючи узгодженість даних навіть у складних та потенційно мінливих умовах.

Завершення TCP-з’єднання відбувається через контрольований чотириступеневий процес, що дозволяє безпечно звільнити ресурси:

1. FIN від відправника: Відправник ініціює завершення з’єднання, надсилаючи пакет FIN (finish).
2. **ACK від отримувача:** Отримувач підтверджує отримання пакета FIN.
3. **FIN від отримувача:** Після надсилання залишкових даних отримувач відправляє власний пакет FIN.
4. **ACK від відправника:** Відправник надсилає ACK, що дозволяє обом сторонам закрити з’єднання.

Такий механізм гарантує **порядок і надійність закриття TCP-сесії**, запобігаючи втраті даних та конфліктам ресурсів у мережі.

Приклад Python з використанням Scapy для ініціації закриття TCP-з’єднання (FIN+ACK)

```
from scapy.all import TCP, IP, send

# Формування TCP-пакета з прапорами FIN і ACK
fin_packet = TCP(flags='FA', seq=1020, ack=1025)

# Формування IP-пакета
ip_packet = IP(src="192.168.1.2", dst="192.168.1.1")

# Об’єднання IP і TCP у повний пакет
```

```
full_packet = ip_packet / fin_packet

# Надсилання пакета в мережу
send(full_packet)
```

Прапори FIN+ACK ('FA') використовуються для завершення TCP-сесії у рамках чотириступеневого процесу закриття з’єднання.

`seq` та `ack` відповідають поточним номерам послідовності та підтвердження у TCP-сесії.

Код формує і надсилає пакет FIN, сигналізуючи про ініціацію завершення з’єднання.

Важливо: цей код функціонує у реальних мережах, тому для його виконання потрібні права адміністратора та доступ до мережі. Для навчальних цілей без мережі можна використовувати імітаційний варіант TCP-пакета, як у попередніх прикладах без Scapy.

Навчальний приклад: симуляція закриття TCP-з'єднання (FIN+ACK)

```
# Вхідні параметри TCP-сесії
sequence_number = 1020
ack_number = 1025
flags = 'FIN+ACK'
source_ip = "192.168.1.2"
dest_ip = "192.168.1.1"

# Імітація формування TCP-пакета
tcp_segment = {
    "seq": sequence_number,
    "ack": ack_number,
    "flags": flags
}

# Імітація формування IP-пакета
ip_packet = {
    "src": source_ip,
    "dst": dest_ip,
    "payload": tcp_segment
}

# Виведення результату
print("Сформовано TCP FIN+ACK пакет (імітація):")
print(ip_packet)
```

Акуратне завершення з’єднання (graceful connection termination) запобігає втраті даних та забезпечує коректне звільнення ресурсів, досягаючи повноти обміну інформацією.

Управління перевантаженням (Congestion Control) є критично важливим для надійності TCP, оскільки спрямоване на оптимізацію потоку даних у мережі. Основні алгоритми містять:

Slow Start (Повільний старт). Початково експоненціально збільшує розмір вікна перевантаження, оцінюючи пропускну здатність мережі.

Congestion Avoidance (Запобігання перевантаженню). Після досягнення певного порогу алгоритм переходить до лінійного зростання, обережно розширюючи вікно.

Fast Retransmit та Fast Recovery (Швидка повторна передача та швидке відновлення). Призначені для оперативної реакції на втрату сегментів, здійснюють прискорені повторні передачі без входження у стан slow start, стабілізуючи розмір вікна.

Ці механізми дозволяють TCP ефективно адаптуватися до змінних умов мережі, підтримуючи надійну та впорядковану доставку даних.

```
# Псевдокод для управління перевантаженням TCP у режимі Slow Start

ініціалізувати:
   congestion_window = 1       # початковий розмір вікна
   threshold = predefined_value # поріг для переходу у Congestion Avoidance

поки є дані для відправлення:
   якщо умови мережі сприятливі:
      congestion_window *= 2   # експоненціальне збільшення вікна
   інакше:
      threshold = congestion_window / 2
      congestion_window = 1
      перейти в режим Congestion Avoidance
```

Механізми управління перевантаженням у TCP забезпечують масштабованість протоколу, справедливий розподіл ресурсів та запобігання втраті пакетів.

Завдяки своїм надійним характеристикам, TCP підходить для широкого спектра застосунків, де критично важлива надійність передачі даних:

Вебперегляд (HTTP/HTTPS). Забезпечує цілісність HTML-даних для стабільного користувацького досвіду.

Електронна пошта (SMTP/IMAP/POP3). Підтримує безпомилкову доставку листів, зберігаючи повноту повідомлень.

Передача файлів (FTP). Забезпечує впорядкований обмін файлами у мережі, що є важливим для транспортування конфіденційних даних.

Інтегровані механізми TCP створюють структуру, завдяки якій ресурсоємні застосунки підтримують функціональність і надійність у нестабільних мережах.

Хоча TCP гарантує надійну передачу даних, він не забезпечує вбудовану безпеку, що робить його вразливим до різних загроз, таких як:

TCP Spoofing. Неавторизована ін’єкція даних шляхом імітації легітимних пакетів.

Атаки типу Man-in-the-Middle. Перехоплення та потенційна модифікація комунікацій.

Атаки типу DoS та DDoS. Перевантаження системи за рахунок надмірного трафіка.

Для захисту TCP можна накласти Internet Protocol Security (IPsec) або Transport Layer Security (TLS), що забезпечують шифрування, автентифікацію та перевірку цілісності, не змінюючи базовий механізм роботи TCP.

Завдяки комплексній надійності та гнучкості свого дизайну TCP є фундаментальним елементом Інтернет-комунікацій, здатним задовольняти різноманітні потреби за збереження принципів надійної, впорядкованої та перевіреної передачі даних. Еволюція протоколу демонструє адаптацію до нових викликів за береження основних принципів надійності.

1.4 Протокол користувацьких датаграм (UDP)

UDP є фундаментальним транспортним протоколом у стеку Інтернет-протоколів, що забезпечує механізм передачі даних без попереднього встановлення з'єднання на транспортному рівні. На відміну від TCP, UDP надає перевагу простоті та швидкості над надійністю та впорядкованістю, що робить його оптимальним для застосунків, де час затримки критичніший за точність даних.

UDP характеризується такими особливостями:

Комунікація без попереднього встановлення з'єднання між вузлами мережі (Connectionless): UDP не встановлює постійної сесії між кінцевими точками, передаючи дані у вигляді окремих пакетів – датаграм.

Мінімальні накладні витрати (Minimal Overhead). Відсутність етапів встановлення з’єднання, нумерації та підтвердження зменшує накладні витрати протоколу, прискорюючи передачу даних.

Надсилання на основі найкращих зусиль (Best-Effort Delivery). UDP не гарантує доставку, збереження порядку чи перевірку помилок; за необхідності ці функції покладаються на прикладні рівні.

Підтримка широкомовлення та групової трансляції (Broadcast та Multicast). На відміну від TCP, UDP дозволяє надсилати пакети широкомовно або групово, що робить його придатним для стрімінгу медіа, онлайн-ігор та подібних застосунків.

Ці характеристики роблять UDP ефективним у сценаріях, де швидкість передачі даних критична, а допустимі незначні втрати пакетів.

UDP інкапсулює дані у датаграми, кожна з яких складається із заголовка та корисного навантаження. Простота заголовка сприяє високій ефективності та продуктивності протоколу:

Порт джерела (Source Port, 16 біт). Вказує порт відправника для специфічної комунікації з додатком.

Порт призначення (Destination Port, 16 біт). Направляє датаграму до відповідного застосунку.

Довжина (Length, 16 біт). Загальна довжина датаграми, включно із заголовком та даними, використовується для визначення меж повідомлення.

Контрольна сума (Checksum, 16 біт). Забезпечує базову перевірку цілісності даних; опціональна для IPv4, обов'язкова для IPv6.

Мінімальна складність заголовка UDP є критичною для ефективної обробки, особливо у середовищах з обмеженими обчислювальними ресурсами або пропускною здатністю мережі.

Псевдокод для створення базового заголовка UDP

```
структура UDPHeader {
   unsigned short source_port;       # Порт джерела
   unsigned short destination_port;  # Порт призначення
   unsigned short length;            # Загальна довжина датаграми (заголовок + дані)
   unsigned short checksum;          # Контрольна сума для перевірки цілісності
}
```

Цей псевдокод ілюструє основну структуру UDP-заголовка, який забезпечує просту і швидку доставку датаграм без встановлення з'єднання, що робить протокол ефективним для застосунків із високими вимогами до швидкості передачі даних, таких як стрімінг, онлайн-ігри та мультимедійні трансляції.

Структурна простота UDP забезпечує швидку обробку даних, відповідаючи його філософії дизайну – простота та висока швидкість передачі.

Природа UDP без попереднього встановлення з'єднання між вузлами мережі підтримує різні моделі комунікації:

Unicast (точка-до-точки). Однонаправлена комунікація між одним відправником і одним отримувачем.

Broadcast (широкомовлення). Надсилання повідомлення всім потенційним вузлам у сегменті мережі.

Multicast (групове передавання). Цільова комунікація з кількома конкретними отримувачами, що оптимізує використання пропускної здатності шляхом зменшення надлишкових пакетів даних.

Ці методи забезпечують ефективність UDP у застосунках, таких як DHCP (Dynamic Host Configuration Protocol), де початкова конфігурація пристроїв може одночасно досягати кількох клієнтів.

Приклад Python з використанням бібліотеки socket для відправки UDP-широкомовлення

```
import socket

# Встановлення адреси широкомовлення та порту
broadcast_address = ('<broadcast>', 12345)
message = b'Hello, network!'

# Створення UDP-сокета
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Дозвіл на широкомовлення
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

# Надсилання повідомлення на адресу широкомовлення
sock.sendto(message, broadcast_address)

print("UDP-широкомовлення відправлено успішно")
```

`socket.SOCK_DGRAM` використовується для UDP-сокета.

`SO_BROADCAST` дозволяє відправляти пакети на адресу широкомовлення `<broadcast>`.

Пакет не гарантує доставку, оскільки UDP працює за принципом best-effort.

Цей приклад ілюструє практичне використання UDP для широкомовлення, що часто застосовується для протоколів типу DHCP, локальних пошукових служб та повідомлень у локальних мережах.

Гнучкість UDP надає розробникам можливість ефективно адаптувати передачу даних під конкретні вимоги мережі.

UDP проявляє себе особливо добре у сценаріях, де швидкість та ефективність критично важливі, а незначні втрати пакетів допустимі:

Стрімінгові за стосунки. Відео- та аудіопотоки використовують UDP для мінімізації затримок. Протоколи, такі як RTP (Real-time Transport Protocol), накладаються на UDP для забезпечення доставки медіа у реальному часі.

Онлайн-ігри. Динамічні ігри застосовують UDP для зменшення затримки введення та підвищення швидкості реакції, що критично для конкурентної гри.

VoIP (Voice over IP). Передача аудіо в реальному часі у мережах отримує переваги від низької затримки UDP, покращуючи якість дзвінків навіть у разі випадкових втрат пакетів.

DNS (Domain Name System). UDP забезпечує швидкі цикли запит-відповідь, необхідні для оперативного перетворення доменних імен у IP-адреси.

Ці приклади демонструють універсальність UDP, що задовольняє потребу у швидкій передачі даних у глобальних мережах.

Попри свої переваги, UDP має вбудовані обмеження:

Відсутність надійності. Не гарантується цілісність доставки пакетів; застосунки мають самостійно реалізовувати механізми перевірки помилок.

Відсутність контролю перевантаження. UDP не вирішує проблеми мережевого перевантаження, що може призводити до втрат пакетів у перевантажених мережах.

Дублювання пакетів. Через безстанову природу UDP протокол не запобігає появі дублікатів датаграм.

Застосунки, що використовують UDP, мають інтегрувати додаткові механізми для компенсації цих обмежень, якщо потрібна точність доставки.

Програмування сокетів (socket programming) є зручною основою для реалізації мережевих застосунків на UDP. Через бібліотеку `socket` розробники можуть створювати легкі моделі клієнт-сервер, використовуючи швидкий і ефективний стек протоколів UDP.

Нижче наведено приклад базової схеми комунікації клієнт-сервер через UDP:

UDP-сервер з використанням бібліотеки socket у Python

```
import socket

# Визначення адреси сервера та розміру буфера
server_address = ('localhost', 65432)
buffer_size = 1024

# Створення UDP-сокета для сервера
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(server_address)

print("UDP сервер запущено та очікує повідомлень")

while True:
    # Очікування повідомлення від клієнта
    data, client_addr = server_socket.recvfrom(buffer_size)
    print(f"Отримано повідомлення: {data} від {client_addr}")

    # Надсилання підтвердження (ACK) клієнту
    server_socket.sendto(b"ACK", client_addr)
```

UDP-клієнт з використанням бібліотеки socket у Python

```
import socket

# Адреса сервера та повідомлення
server_address = ('localhost', 65432)
message = b'Hello, UDP server!'

Створення UDP-сокета для клієнта
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
try:
    # Надсилання повідомлення серверу
    client_socket.sendto(message, server_address)

    # Отримання відповіді від сервера
    data, server = client_socket.recvfrom(4096)
    print(f"Отримано повідомлення: {data}")

finally:
    # Закриття сокета
    client_socket.close()
```

UDP-сервер створює сокет без встановлення з'єднання і прослуховує повідомлення на вказаному порту.

Сервер отримує дані за допомогою `recvfrom`, а потім надсилає підтвердження (ACK) клієнту.

UDP-клієнт надсилає повідомлення серверу через `sendto` і отримує відповідь через `recvfrom`.

Обидва приклади демонструють швидке та ефективне передавання даних за допомогою UDP завдяки низьким накладним витратам і передаванню даних без встановлення з'єднання.

Цей приклад демонструє простий UDP-echo сервер, який приймає дані від клієнтів і надсилає назад підтвердження (‘ACK’). Вбудована простота UDP спрощує дизайн застосунку, зосереджуючи увагу безпосередньо на обміні даними, без необхідності керування з’єднанням.

Водночас відсутність вбудованих механізмів безпеки у UDP створює *певні загрози*:

Спуфінг і ін’єкція пакетів. Зловмисники можуть вставляти шкідливі пакети у потік комунікацій.

Атаки на посилення (Amplification attacks). Особливість протоколу UDP, яка полягає у відсутності попереднього встановлення з'єднання, використовується під час відображувальних атак для формування значного обсягу трафіку, спрямованого на цільову систему.

Для зменшення цих ризиків розробники часто застосовують додаткові протоколи безпеки, наприклад, Datagram Transport Layer Security (DTLS), який забезпечує цілісність даних, автентифікацію та конфіденційність** шляхом шифрування.

Впровадження таких методів дозволяє балансувати переваги швидкості та простоти UDP із необхідними механізмами безпеки для захисту даних користувачів.

Фундаментальна простота UDP надає значні переваги для застосунків, де затримка та легкість реалізації важливіші за надійність і впорядкованість. Його робота формує основу багатьох критично важливих застосунків із низькою затримкою, забезпечуючи потокове передавання даних у реальному часі, динамічну комунікацію та швидкі транзакції в глобальному Інтернеті. Попри обмеження щодо надійності та безпеки,

UDP залишається цінним компонентом сучасних мереж, підтримуючи інновації, що трансформують підключеність і взаємодію.

1.5 Протоколи рівня застосунків

Рівень застосунків (Application Layer) знаходиться на вершині моделі TCP/IP і відповідає за безпосередню взаємодію з програмним забезпеченням користувача. Він надає численні протоколи, що визначають, як застосунки обмінюються даними у мережі. Ці протоколи керують обміном інформацією відповідно до потреб конкретних застосунків і функціональностей, забезпечуючи сервіси, такі як передача файлів, доставка електронної пошти та вебперегляд. У цьому підрозділі розглядаються ключові протоколи рівня застосунків: HTTP, HTTPS, FTP, SMTP та DNS, з поясненням їх функцій, сценаріїв використання та впливу на мережеву комунікацію.

HTTP є невід'ємною частиною World Wide Web, забезпечуючи обмін гіпертекстовими документами між вебсерверами та клієнтами (веббраузерами). Протокол працює як безстанний (stateless), функціонуючи на основі парадигми запит-відповідь, де клієнт надсилає HTTP-запит до конкретного ресурсу, а сервер відповідає цим ресурсом, зазвичай у вигляді HTML-документа.

HTTP Methods (Методи HTTP)

HTTP визначає набір методів запиту, що вказують бажану дію:

GET: Отримання даних із зазначеного ресурсу.

POST: Надсилання даних на сервер для створення або оновлення ресурсу.

PUT: Оновлення ресурсу на основі наданих даних.

DELETE: Видалення зазначеного ресурсу.

Заголовки запитів та відповідей (Request/Response Headers) додатково інкапсулюють метадані, включно з типом контенту, кодуванням та керуванням кешем, що регламентують деталі обміну даними.

Приклад 1 Python для виконання HTTP GET-запиту з використанням бібліотеки requests

```
import requests

# Вказання URL ресурсу
url = 'http://www.example.com'

# Виконання GET-запиту
response = requests.get(url)

# Виведення статусу відповіді та вмісту
print(f"Статус відповіді: {response.status_code}")
print(response.content)
```

`requests.get(url)` надсилає HTTP GET-запит до вказаного ресурсу.

`response.status_code` повертає код стану HTTP (наприклад, 200 – успішно).

`response.content` містить вміст відповіді сервера, зазвичай HTML-код сторінки.

Цей приклад демонструє просту взаємодію клієнта з вебсервером через HTTP, що ілюструє практичне застосування протоколу рівня застосунків у реальному середовищі.

Приклад 2 HTTP GET-запиту без додаткових бібліотек, використовуючи лише стандартну бібліотеку Python `urllib`:

```
# Приклад HTTP GET-запиту з використанням стандартної бібліотеки urllib

from urllib import request

# Вказуємо URL ресурсу
url = "http://www.example.com"

# Виконання GET-запиту
try:
    with request.urlopen(url) as response:
        # Отримуємо статус відповіді
        status = response.getcode()
        # Читаємо вміст сторінки
        content = response.read()

        print(f"Статус відповіді: {status}")
        print(content.decode('utf-8'))  # Декодуємо байти у текст
except Exception as e:
    print(f"Помилка під час запиту: {e}")
```

`request.urlopen(url)` відкриває з’єднання і надсилає GET-запит.

`response.getcode()` повертає HTTP-статус (наприклад, 200 – успішно).

`response.read()` отримує вміст відповіді у байтах, який можна перетворити у текст через `.decode('utf-8')`.

Весь код використовує тільки стандартні бібліотеки, тому не потребує додаткового встановлення `requests`.

Приклад 3 **HTTP/HTTPS GET-запиту з обробкою помилок мережі** на стандартній бібліотеці Python, який більш безпечний для сучасних вебсайтів:

```
# Приклад HTTP/HTTPS GET-запиту з обробкою помилок
from urllib import request, error
```

```
# Вказуємо URL ресурсу (може бути http або https)
url = "https://www.example.com"

try:
    # Виконання GET-запиту
    with request.urlopen(url, timeout=10) as response:
        # Отримуємо статус відповіді
        status = response.getcode()
        # Читаємо вміст сторінки
        content = response.read()

        print(f"Статус відповіді: {status}")
        print(content.decode('utf-8'))  # Декодуємо байти у текст

except error.HTTPError as e:
    print(f"HTTP помилка: {e.code} - {e.reason}")
except error.URLError as e:
    print(f"Помилка URL або мережі: {e.reason}")
except Exception as e:
    print(f"Невідома помилка: {e}")
```

Працює як з HTTP, так і з HTTPS.

Використовується `timeout=10`, щоб запобігти зависанню у випадку довгих затримок.

Окрема обробка помилок:

HTTPError` – помилки HTTP (наприклад, 404, 500).
`URLError` – проблеми з мережею або недоступний URL.
`Exception` – всі інші непередбачені помилки.

Ця версія повністю працює без сторонніх бібліотек, що робить її універсальною для будь-якого Python-середовища.

HTTP/2 та HTTP/3 впроваджують покращення у мультиплексуванні запитів, зменшують затримки та підвищують продуктивність, усуваючи історичні обмеження протоколу щодо ефективності роботи.

Безпека – HTTPS

HTTPS (HTTP Secure) підвищує безпеку традиційного HTTP за рахунок шифрування з використанням TLS (Transport Layer Security). HTTPS забезпечує:

Конфіденційність – захист даних від стороннього перегляду.

Цілісність – гарантію того, що дані не були змінені під час передачі.

Автентифікацію – підтвердження справжності сервера та клієнта.

Безпечна передача даних через HTTPS є необхідною для чутливих операцій, таких як онлайн-банкінг, конфіденційні комунікації та безпечні комерційні операції.

Простий приклад 1 HTTPS-запиту з використанням бібліотеки requests

```
import requests

# Вказуємо URL безпечного ресурсу
url = 'https://www.secureexample.com'

# Виконання HTTPS GET-запиту
response = requests.get(url)

# Виведення статусу відповіді
print(f"Статус безпечної відповіді: {response.status_code}")

# Якщо запит успішний, виводимо вміст сторінки
if response.ok:
    print(response.content)
```

HTTPS гарантує шифрування даних, автентифікацію та цілісність.

`response.status_code` відображає код відповіді HTTP(S), наприклад, 200 – успішно.

`response.ok` повертає True, якщо код стану відповідає успішній обробці (200–299).

`response.content` містить повний вміст відповіді сервера, зазвичай HTML-код вебсторінки.

Цей приклад ілюструє практичне застосування HTTPS для безпечного обміну даними між клієнтом і сервером.

Приклад 2 без додаткових бібліотек

Якщо немає можливості встановити requests, можна використовувати стандартну бібліотеку Python urllib для HTTPS-запитів:

```
HTTPS-запит без сторонніх бібліотек (urllib)
from urllib import request, error

url = "https://www.secureexample.com"

try:
    with request.urlopen(url, timeout=10) as response:
        status = response.getcode()
        content = response.read()
        print(f"Статус відповіді: {status}")
        print(content.decode('utf-8'))
except error.HTTPError as e:
    print(f"HTTP помилка: {e.code} - {e.reason}")
except error.URLError as e:
    print(f"Помилка URL або мережі: {e.reason}")
```

```
except Exception as e:
    print(f"Невідома помилка: {e}")
```

Працює як з HTTP, так і HTTPS.
timeout=10 запобігає зависанню.
Обробляє помилки HTTP та мережі.
Не потребує встановлення додаткових бібліотек.

Протокол передавання файлів (File Transfer Protocol, FTP)

FTP забезпечує стандартний механізм передавання файлів між системами в мережі, надаючи можливості як завантаження (upload), так і вивантаження (download) даних. Працюючи за клієнт-серверною моделлю, FTP використовує два окремі з'єднання:

- керувальне з'єднання – для передавання команд і керування сеансом;
- з'єднання даних – безпосередньо для передавання файлів.

Команди FTP. Основні команди FTP реалізують функції керування:

- USER – передає ім'я користувача для автентифікації;
- PASS – передає пароль для завершення входу;
- LIST – відображає список файлів у каталозі;
- RETR – завантажує файл з сервера;
- STOR – вивантажує файл на сервер.

FTP підтримує активний і пасивний режими роботи. В активному режимі з'єднання даних ініціюється сервером, тоді як пасивний режим, що є зручнішим для проходження міжмережевих екранів (firewall), використовує з'єднання, ініційовані клієнтом.

Навчальний приклад FTP-клієнта

```
from ftplib import FTP

# УВАГА:
# Даний код є ілюстративним.
# Адреса ftp.example.com використовується лише у документаційних цілях.

if False:  # Блокування виконання мережевого коду
    ftp = FTP('ftp.example.com')
    ftp.login('username', 'password')
    ftp.cwd('pub')

    filename = 'example.txt'
    with open(filename, 'wb') as file:
        ftp.retrbinary(f'RETR {filename}', file.write)

    ftp.quit()
```

Безпека – FTPS та SFTP. Протокол FTP, який за своєю природою є небезпечним через передавання даних у відкритому текстовому вигляді, може бути доповнений механізмами підвищення безпеки:

FTPS (FTP Secure) – розширює FTP шляхом використання SSL/TLS для створення зашифрованих каналів передавання даних і керувальних команд.

SFTP (SSH File Transfer Protocol) – використовує протокол Secure Shell (SSH) як для безпечного передавання файлів, так і для керування сеансом, забезпечуючи вищий рівень захисту порівняно з іншими варіантами FTP.

Протокол простого передавання пошти (Simple Mail Transfer Protocol, SMTP) регламентує передавання електронної пошти через мережу Інтернет, забезпечуючи взаємодію між поштовими серверами. Він визначає процедуру, за якою електронні повідомлення надсилаються від клієнта до поштового сервера одержувача та згодом доставляються у відповідні поштові скриньки.

У процесі передавання повідомлень SMTP використовує текстові команди:

- HELO – повідомляє поштовий сервер про початок сеансу зв'язку з клієнтом;
- MAIL FROM – задає адресу електронної пошти відправника;
- RCPT TO – визначає адресу або адреси одержувачів;
- DATA – ініціює передавання вмісту електронного повідомлення.

Сумісність із MIME (Multipurpose Internet Mail Extensions) надає змогу SMTP обробляти різноманітні формати даних, окрім простого тексту, зокрема вкладення, зображення та мультимедійний вміст, що істотно розширює функціональні можливості електронної пошти.

Приклад використання SMTP у Python ілюструє алгоритм надсилання електронного листа з використанням стандартної бібліотеки `smtplib` та MIME-повідомлень.

Адреси виду `smtp.example.com` і `@example.com` застосовуються виключно з демонстраційною метою.

Приклад надсилання електронного листа через SMTP не призначений для реального виконання без заміни параметрів

```
import smtplib
from email.mime.text import MIMEText

# Формування MIME-повідомлення
msg = MIMEText("This is the body of the email.")
msg['Subject'] = 'Test Email'
msg['From'] = 'sender@example.com'
msg['To'] = 'recipient@example.com'

# УВАГА:
# smtp.example.com є умовною адресою сервера,
# використовується лише для ілюстрації роботи SMTP.
```

```
if False:  # Блокування реального мережевого виконання
    with smtplib.SMTP('smtp.example.com', 587) as server:
        server.starttls()               # Захищене з’єднання (TLS)
        server.login('username', 'password')
        server.sendmail(
            msg['From'],
            [msg['To']],
            msg.as_string()
        )
```

`MIMEText` – створює текстове MIME-повідомлення, сумісне з SMTP.

Заголовки `Subject`, `From`, `To` визначають службову інформацію електронного листа.

`smtplib.SMTP()` – ініціює SMTP-сеанс з поштовим сервером.

`starttls()` – активує захищений канал передавання з використанням TLS.

`sendmail()` – передає сформоване повідомлення серверу для доставки.

> Доменні імена та адреси електронної пошти виду `example.com` є зарезервованими для документації (RFC 2606) і не призначені для реального використання.

> Наведений код демонструє логіку роботи SMTP-клієнта, а для практичних експериментів необхідно вказувати реальний SMTP-сервер та коректні облікові дані.

Протокол SMTP у поєднанні з MIME-розширеннями забезпечує універсальний механізм передавання електронних повідомлень різних форматів. Реалізація SMTP-клієнта в Python дозволяє наочно продемонструвати принципи роботи поштових систем, водночас підкреслюючи важливість використання захищених з’єднань (TLS) у сучасних мережах.

Покращення безпеки SMTP. Уразливість протоколу SMTP до несанкціонованого доступу та перехоплення даних може бути зменшена шляхом застосування криптографічних механізмів, зокрема протоколу STARTTLS, який забезпечує конфіденційність і цілісність передавання повідомлень шляхом шифрування каналу зв’язку між клієнтом і сервером.

Система доменних імен (Domain Name System, DNS) відіграє ключову роль у функціонуванні мережі Інтернет, здійснюючи перетворення доменних імен, зручних для сприйняття людиною, у відповідні IP-адреси, що є необхідним для визначення місцезнаходження вебсерверів і маршрутизації даних.

Компоненти та принципи функціонування DNS. DNS функціонує як ієрархічна розподілена база даних, у якій доменні імена зіставляються з IP-адресами. Основні компоненти системи, зокрема сервери доменних імен та резольвери, взаємодіють для оброблення запитів і повернення відповідних IP-адрес.

Кореневі сервери імен (Root Nameservers) – є початковим етапом процесу резолюції DNS та спрямовують запити до відповідних серверів доменів верхнього рівня (TLD).

Сервери доменів верхнього рівня (TLD Nameservers) – передають запити до авторитетних серверів імен для конкретних доменних ієрархій (наприклад, *.com*, *.org*, *.net*).

Авторитетні сервери імен (Authoritative Nameservers) – надають безпосередні зіставлення доменних імен з IP-адресами у відповідь на DNS-запити.

Процес резолюції DNS є прикладом розподілених обчислень, у якому механізми кешування відіграють важливу роль у підвищенні продуктивності запитів. Збереження раніше отриманих відповідей на локальному рівні дає змогу суттєво зменшити затримки та скоротити час доступу до мережевих ресурсів.

Приклад DNS-запиту з використанням модуля socket у Python. Домен example.com використовується лише для навчальних і демонстраційних цілей

```
import socket

domain = "www.example.com"
ip_address = socket.gethostbyname(domain)

print(f"The IP address for {domain} is {ip_address}")
```

У наведеному прикладі використовується стандартний модуль `socket` мови Python для виконання DNS-резолюції. Функція `gethostbyname()` надсилає DNS-запит і повертає IP-адресу, що відповідає заданому доменному імені. Такий підхід ілюструє базовий механізм перетворення доменних імен у IP-адреси, який лежить в основі роботи системи DNS.

Система доменних імен (DNS), яка первісно проєктувалася без вбудованих механізмів безпеки, є вразливою до атак підміни (spoofing). Розширення безпеки DNS (DNS Security Extensions, DNSSEC) спрямовані на усунення цих вразливостей шляхом використання цифрових підписів, що забезпечують цілісність і автентичність даних DNS.

Додаткові протоколи прикладного рівня. Окрім широко застосовуваних протоколів, прикладний рівень охоплює низку спеціалізованих протоколів, призначених для задоволення конкретних комунікаційних потреб:

Telnet – забезпечує віддалений доступ до командного інтерфейсу систем, проте не має вбудованих засобів захисту.

SSH (Secure Shell) – надає захищене адміністрування через командний інтерфейс у мережі, замінюючи Telnet завдяки застосуванню надійного шифрування.

SIP (Session Initiation Protocol) – є базовим протоколом для систем реального часу, зокрема VoIP, забезпечуючи сигналізацію та керування мультимедійними сеансами.

LDAP (Lightweight Directory Access Protocol) – протокол прикладного рівня, призначений для доступу, пошуку, оновлення та адміністрування інформації в розподілених службах каталогів у мережах TCP/IP.

Кожен із наведених протоколів має функціональні особливості, адаптовані до відповідних завдань, що підвищує можливості мережевих систем і збагачує користувацький досвід.

Проблеми безпеки та ключові аспекти. Незважаючи на притаманні виклики, протоколи прикладного рівня відіграють важливу роль у формуванні динамічної екосистеми інтерактивних комунікацій. Питання безпеки залишаються пріоритетними та потребують ретельного аналізу й удосконалення:

Шифрування даних – забезпечує конфіденційність і цілісність інформації, що передається між клієнтами та серверами.

Механізми автентифікації – підтверджують особу користувачів перед наданням доступу до чутливих даних або сервісів.

Регулярні оновлення – гарантують усунення відомих уразливостей у реалізаціях протоколів із урахуванням сучасних результатів досліджень у сфері безпеки.

Під час інтеграції цих протоколів мережеві архітектори та розробники прагнуть досягти балансу між ефективністю обміну даними та вимогами безпеки, створюючи стійкі системи, здатні відповідати зростаючим і мінливим потребам. Сукупність протоколів прикладного рівня становить фундамент, на якому базуються різноманітні сервіси Інтернету, стимулюючи інновації та забезпечуючи глобальну взаємодію.

1.6 Мережеві інструменти та утиліти

У сфері комп’ютерних мереж інструменти та утиліти відіграють ключову роль в аналізі й підтримці мережевих середовищ. Вони допомагають адміністраторам і мережевим інженерам здійснювати моніторинг, діагностику та оптимізацію продуктивності й безпеки мереж. У цьому розділі розглядаються найважливіші мережеві інструменти та утиліти, їхні функціональні можливості, практичні застосування та значення для ефективного керування мережею.

Утиліта ping є базовим мережевим засобом, що використовується для перевірки зв’язності між пристроями шляхом обміну ICMP (Internet Control Message Protocol) повідомленнями типу echo-запит та echo-відповідь. Вона дає змогу визначити доступність вузла в IP-мережі та виміряти час проходження пакета в обидва боки.

Функціональні можливості та застосування ping

Перевірка досяжності – визначає, чи є вузол активним і доступним у мережі.

Вимірювання затримки – оцінює швидкість реакції та час кругового проходження до віддаленого вузла.

Виявлення втрат пакетів – ідентифікує втрати під час передавання даних, що допомагає в діагностиці надійності мережі.

Приклад використання команди `ping` у терміналі. Команда `ping` застосовується для перевірки доступності віддаленого вузла та оцінювання параметрів мережевого з'єднання. Нижче наведено типовий приклад виконання команди та інтерпретацію результатів:

```
ping google.com
PING google.com (142.250.190.78): 56 bytes of data
64 bytes from 142.250.190.78: icmp_seq=0 ttl=118 time=29.9 ms
```

Пояснення результатів:

google.com (142.250.190.78) – доменне ім'я та відповідна IP-адреса вузла, визначена за допомогою DNS.

56 bytes of data – розмір ICMP-повідомлення запиту.

64 bytes from 142.250.190.78 – обсяг отриманого ICMP-відповіді від вузла призначення.

icmp_seq=0 – порядковий номер ICMP-пакета.

ttl=118 (Time To Live) – лічильник максимально допустимої кількості переходів (hop), що залишилися до відкидання пакета.

time=29.9 ms – час кругового проходження пакета (Round-Trip Time), який характеризує затримку мережі.

Такий аналіз дозволяє швидко оцінити стан з'єднання, затримки та стабільність мережевої взаємодії між локальним вузлом і віддаленим сервером.

Помилка виникає тому, що `ping google.com` – це команда оболонки (термінала), а не код Python. Ви намагаєтеся виконати її як Python-програму, через що інтерпретатор видає `SyntaxError`.

Варіант 1. Правильне використання в терміналі (рекомендовано)

Команду `ping` потрібно виконувати в командному рядку або терміналі, а не в Python-файлі:

```
ping google.com
```

Команда `ping` є утилітою операційної системи і не належить до мови програмування Python.

Варіант 2. Виклик `ping` з Python (для демонстраційних цілей)

Якщо необхідно виконати `ping` саме з Python, потрібно використовувати модуль `subprocess`:

```
import subprocess

subprocess.run(["ping", "-c", "4", "google.com"])
```

Для Windows потрібно використати інший параметр кількості пакетів:

subprocess.run(["ping", "-n", "4", "google.com"])

Причина помилки
Python очікує інструкції мови Python
`ping google.com` не є допустимим синтаксисом Python
Тому виникає помилка:

SyntaxError: invalid syntax

Доцільно чітко розмежувати:

- системні мережеві утиліти (ping, tracert, netstat);
- програмну реалізацію мережевих функцій мовою Python.

Обмеження. Хоча утиліта ping дає змогу підтвердити наявність мережевої досяжності та оцінити показники затримки передавання, вона не забезпечує можливості діагностики складних мережевих збоїв або перевірки цілісності пакетів даних. Крім того, міжмережеві екрани (firewalls) можуть блокувати ICMP-пакети, унаслідок чого використання цієї утиліти стає неможливим у захищених мережевих середовищах.

Утиліти traceroute (у UNIX-подібних операційних системах) та tracert (у середовищі Windows) відображають маршрут, яким проходять пакети даних у мережі від джерела до вузла призначення. Визначаючи IP-адреси кожного проміжного вузла (hops), ці засоби надають уявлення про топологію маршрутизації мережі.

Функціональні характеристики

Виявлення маршруту: ідентифікація кожного маршрутизатора, через який проходять пакети на шляху до кінцевого вузла.

Аналіз затримок: надання часових показників для кожного переходу, що дає змогу визначити ділянки з підвищеними затримками.

Діагностика мережі: виявлення місць можливого втрачання пакетів або помилкової маршрутизації, що сприяє усуненню мережевих проблем.

Приклад використання утиліти traceroute.

Наведений приклад демонструє виконання команди traceroute для визначення маршруту проходження пакетів до доменного імені example.com. У результаті відображається послідовність проміжних вузлів (маршрутизаторів), через які проходять пакети даних, а також відповідні значення затримки для кожного переходу.

У виведених даних зазначено:

- ✓ адресу вузла призначення та його IP-адресу (93.184.216.34);
- ✓ максимальну кількість переходів (hops), дозволену для трасування (30);
- ✓ кожен проміжний вузол маршруту, поданий його IP-адресою;
- ✓ час проходження пакетів (у мілісекундах) для кількох проб на кожному переході, що дає змогу оцінити затримки на окремих ділянках маршруту.

Такий аналіз дозволяє отримати уявлення про структуру мережевого шляху, виявити потенційні вузькі місця та визначити ділянки, на яких виникають підвищені затримки або проблеми з передаванням даних.

Приклад використання команди `traceroute`

```
traceroute example.com
traceroute to example.com (93.184.216.34), 30 hops max
1  192.168.1.1 (192.168.1.1)  1.123 ms  1.211 ms  1.304 ms
2  10.0.0.1 (10.0.0.1)  9.654 ms  10.332 ms  10.789 ms
```

Пояснення:

example.com (93.184.216.34) – доменне ім’я та відповідна IP-адреса вузла призначення.

30 hops max – максимальна кількість проміжних переходів (hops), дозволена для трасування маршруту.

1 192.168.1.1 (192.168.1.1) 1.123 ms 1.211 ms 1.304 ms – перший проміжний вузол маршруту, показані три виміри часу проходження пакета (Round-Trip Time) у мілісекундах.

2 10.0.0.1 (10.0.0.1) 9.654 ms 10.332 ms 10.789 ms – другий проміжний вузол маршруту з трьома вимірами затримки.

Такий результат ілюструє маршрут, яким проходять пакети даних від джерела до вузла призначення, дозволяючи оцінити затримки на кожному проміжному етапі та проаналізувати топологію мережі.

Змінність результатів у мережах. Результати роботи утиліти traceroute можуть змінюватися через динамічні алгоритми маршрутизації, перевантаження мережі або тайм-аути на різних проміжних вузлах. Ці коливання вказують на необхідність виконання декількох запусків для отримання надійних та репрезентативних даних.

Утиліта netstat забезпечує детальне відображення мережевих з’єднань та сокетів на пристрої. Вона є незамінним інструментом для оцінення мережевої активності та діагностики проблем із з’єднаннями.

Функціональні можливості та застосування

Перелік активних з’єднань: відображає вхідні та вихідні з’єднання на системі, включно протокол, локальні та віддалені адреси, а також стан з’єднання.

Маршрутизовані таблиці: виводить таблиці маршрутизації ядра, надаючи уявлення про те, як пакети даних спрямовуються в мережі.

Статистика інтерфейсів: надає інформацію про обсяг трафіка та використання кожного мережевого інтерфейсу.

Приклад використання утиліти `netstat` для відображення активних з’єднань

```
netstat -tuln
```

Результат виконання:

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address State
```

tcp 0 0 0.0.0.0:80 0.0.0.0: LISTEN
udp 0 0 0.0.0.0:53 0.0.0.0:

Пояснення

Proto – мережевий протокол (TCP або UDP).

Recv-Q / Send-Q – розмір черги прийому та черги відправки для сокета, що показує накопичені непереробленні дані.

Local Address – локальна IP-адреса та порт, на яких очікує сервер.

Foreign Address – адреса та порт віддаленого вузла; символ `*` означає будь-який вузол.

State – стан з'єднання; наприклад, `LISTEN` означає, що сокет очікує на вхідні з'єднання.

Такий аналіз дозволяє оцінити активність серверних служб, виявити відкриті порти та перевірити стан мережевих з'єднань на пристрої.

Діагностичні переваги

Завдяки детальному виведенню інформації утиліта netstat сприяє виявленню підозрілої активності, що може свідчити про порушення безпеки, наприклад, несподівано відкриті порти або підозрілі віддалені адреси.

Утиліта nslookup дозволяє виконувати запити до системи доменних імен (DNS) для отримання відповідності між доменними іменами та IP-адресами, що полегшує діагностику доменів та перевірку коректності їх резолюції.

Функціональні переваги

Резолюція домену: перевіряє правильність відображення доменного імені на відповідну IP-адресу.

Запит до сервера: дає змогу направляти запити до конкретних DNS-серверів, що корисно під час усунення проблем із DNS.

Перевірка поштових серверів (MX): аналізує записи MX для підтвердження налаштування поштових серверів.

Приклад використання утиліти `nslookup` для запиту домену

nslookup www.example.com

Результат виконання:

Server: 8.8.8.8
Address: 8.8.8.8#53

Non-authoritative answer:
Name: www.example.com
Address: 93.184.216.34

Пояснення

Server / Address – DNS-сервер, до якого було направлено запит, та його IP-адреса.

Non-authoritative answer – відповідь, отримана не від авторитетного сервера для цього домену, але перевірена за кешованими даними.

Name / Address – доменне ім’я та його відповідна IP-адреса, що підтверджує правильність резолюції домену.

Такий приклад демонструє базовий спосіб перевірки коректного відображення доменних імен у IP-адреси та діагностики налаштувань DNS-серверів.

Обмеження та особливості використання. Команди nslookup залежать від відповіді залучених DNS-серверів і можуть піддаватися впливу кешування, що призводить до варіацій у результатах за повторних запитів.

Wireshark – потужний аналізатор мережевих протоколів, який здійснює захоплення та декодування мережевих пакетів, забезпечуючи детальний «мікроскопічний» огляд мережевого трафіку.

Основні функції

Захоплення пакетів у реальному часі: моніторинг живого трафіку на декількох мережевих інтерфейсах.

Аналіз протоколів: підтримка глибокого аналізу сотень протоколів, що сприяє діагностиці складних проблем мережі.

Реасемблювання та фільтрація: можливість відновлення фрагментованих пакетів та застосування фільтрів для зосередження аналізу на потрібному трафіку.

Базовий приклад запуску Wireshark через командний рядок за допомогою утиліти `tshark`

```
tshark -i eth0 -c 10
```

Результат виконання (приклад):

```
Capturing on 'eth0'
1   0.000000 192.168.1.2 -> 192.168.1.1 TCP 74 44350 > 80 [SYN] Seq=0 Win=65535 Len=0
```

Пояснення

-i eth0 – вказує мережевий інтерфейс для захоплення пакетів.

-c 10 – обмежує захоплення до 10 пакетів.

192.168.1.2 -> 192.168.1.1 – відправник та одержувач пакета.

TCP 74 – протокол TCP і розмір пакета у байтах.

44350 > 80 [SYN] – порт відправника > порт одержувача, прапорець SYN, що сигналізує про ініціацію TCP-з’єднання.

Seq=0 Win=65535 Len=0 – номер початкового сегмента, розмір вікна прийому та довжина даних.

Цей приклад демонструє базовий спосіб захоплення пакетів у реальному часі для аналізу TCP-з’єднань та інших мережевих протоколів.

Заходи обережності та етичне використання.

Можливість Wireshark захоплювати чутливі дані потребує етичного використання інструменту з попереднім отриманням відповідних дозволів, щоб уникнути порушення приватності та правових наслідків.

curl – універсальна утиліта командного рядка, що взаємодіє з мережевими протоколами та здебільшого використовується для передачі даних до або від сервера з підтримкою широкого спектра протоколів, таких як HTTP, HTTPS, FTP та інших.

Функціональні можливості

Передача даних: забезпечує обмін даними, підтримує передачу файлів через FTP, HTTP POST-запити та використання користувацьких заголовків.

Тестування API: дозволяє перевіряти RESTful API, виконуючи дії GET, PUT, DELETE безпосередньо з консолі.

Можливості автоматизації: підтримує написання скриптів та автоматизацію вебвзаємодій.

Приклад використання `curl` для отримання заголовків HTTP-відповіді з URL

curl -I http://www.example.com

Результат виконання (приклад):

HTTP/1.1 200 OK
Date: Mon, 01 Jan 2022 12:00:00 GMT
Server: ExampleServer/0.1

Пояснення:

-I – опція `curl`, яка запитує лише заголовки HTTP без завантаження тіла відповіді.

HTTP/1.1 200 OK – статусний рядок, що вказує на успішну обробку запиту сервером.

Date – дата і час відповіді сервера.

Server – інформація про серверне програмне забезпечення, що обробило запит.

Цей приклад ілюструє базовий спосіб перевірки доступності вебресурсів та отримання метаданих HTTP-відповіді без завантаження повного вмісту сторінки.

Розширені можливості сценаріїв. Поза межами ручних операцій, curl може бути інтегрований у скрипти для динамічної обробки даних, надаючи гнучкі налаштування заголовків, автентифікації та інших параметрів.

Утиліта nmap (Network Mapper) відома своїми можливостями виявлення мереж та аудиту безпеки і часто використовується для картографування мереж та виявлення вразливостей.

Основні функції

Виявлення мережі: ефективне сканування великих мереж із отриманням інформації про доступні хости та сервіси.

Сканування портів: визначення відкритих портів та відповідних служб на системі.

Оцінення вразливостей: розширення за допомогою скриптів nmap для виявлення вразливостей у системах.

Приклад використання утиліти `nmap` для виконання комплексного сканування

```
nmap -A -T4 scanme.example.com
```

Результат виконання (приклад):

```
Starting Nmap 7.80 ( https://nmap.org ) at 2022-01-01 12:00 UTC
Nmap scan report for scanme.example.com (93.184.216.34)
Host is up (0.045s latency).
Not shown: 998 closed ports
PORT    STATE SERVICE VERSION
80/tcp  open  http    Apache httpd 2.4.1
```

Пояснення:

-A – вмикає розширене сканування, що містить визначення версій служб, ідентифікацію операційної системи та виконання стандартних NSE-скриптів.

-T4 – задає агресивніший таймінг сканування для прискорення процесу за умови стабільного мережевого з'єднання.

Host is up – підтверджує доступність вузла в мережі та вказує затримку відповіді.

Not shown: 998 closed ports – свідчить, що більшість портів закриті й не відображаються детально.

80/tcp open http Apache httpd 2.4.1 – ідентифікує відкритий TCP-порт 80, службу HTTP та її програмну реалізацію з відповідною версією.

Цей приклад демонструє застосування nmap для детального аналізу мережевого вузла, виявлення доступних сервісів і початкового оцінення конфігурації безпеки.

Відповідальне використання та дотримання нормативних вимог

Як і будь-який потужний інструмент аналізу мереж, nmap потрібно використовувати відповідально, забезпечуючи дотримання політик мережі та нормативних вимог для запобігання несанкціонованим скануванням.

Комплекс мережевих інструментів та утиліт, розглянутий у цьому розділі, підвищує ефективність управління мережею, діагностики та безпеки. Стратегічне застосування цих засобів дозволяє отримати глибоке розуміння поведінки мережі, сприяючи обґрунтованому прийняттю рішень як в операційних, так і в безпекових контекстах.

Дотримання етичних норм та правових обмежень є обов'язковим для ефективного використання їхніх можливостей, забезпечуючи стійкість,

надійність і адаптивність мереж до змінних викликів. Ці утиліти формують основу для динамічного контролю мережі, надаючи адміністраторам інструменти для підтримки збалансованого, безпечного та високопродуктивного мережевого середовища.

1.7 Контрольні питання

1. У чому полягає суть моделі TCP/IP та які основні функції виконує кожен із її рівнів?
2. Які завдання виконує канальний рівень моделі TCP/IP та яку роль у ньому відіграють механізми CSMA/CD і CSMA/CA?
3. Яке призначення протоколу IP, у чому полягають відмінності між IPv4 та IPv6 і яку роль відіграють протоколи маршрутизації?
4. Порівняйте протоколи TCP та UDP за характеристиками надійності, швидкодії та сфер застосування.
5. У чому полягає принцип інкапсуляції даних у моделі TCP/IP та яке його значення для забезпечення ефективної мережевої взаємодії?
6. Яке призначення протоколу IP у моделі TCP/IP та які основні функції він виконує під час передавання даних у мережі?
7. У чому полягають основні відмінності між IPv4 та IPv6 за структурою адрес, можливостями та рівнем підтримки безпеки?
8. Поясніть принципи IP-адресації та нотації CIDR. Яке їх значення для ефективної маршрутизації?
9. Як здійснюється маршрутизація IP-пакетів у мережі та яку роль у цьому процесі відіграють протоколи RIP, OSPF і BGP?
10. Яке призначення протоколів ICMP та ARP у стеку TCP/IP і як вони сприяють діагностиці та забезпеченню коректної роботи мережі?
11. Яке призначення протоколу TCP у моделі TCP/IP та які його основні відмінності від протоколу UDP?
12. Поясніть принцип роботи триетапного рукостискання (three-way handshake) та його значення для встановлення TCP-з’єднання.
13. Яку роль відіграють номери послідовності, підтвердження (ACK) і механізм ковзного вікна в забезпеченні надійності передачі даних у TCP?
14. Охарактеризуйте структуру TCP-заголовка та призначення його основних полів (порти, прапори, вікно, контрольна сума тощо).
15. У чому полягають основні алгоритми управління перевантаженням у TCP (Slow Start, Congestion Avoidance, Fast Retransmit, Fast Recovery) та як вони впливають на продуктивність мережі?
16. У чому полягає особливість роботи протоколу UDP без встановлення з'єднання та чим вона відрізняється від механізму передавання даних у TCP?
17. Охарактеризуйте структуру заголовка UDP та призначення його основних полів. Яку роль відіграє контрольна сума?
18. Поясніть принцип роботи UDP за моделями Unicast, Broadcast і Multicast та наведіть приклади їх практичного застосування.

19. Які основні сфери використання UDP (стрімінг, VoIP, онлайн-ігри, DNS тощо) та чому в цих застосунках перевага надається саме цьому протоколу?

20. Які проблеми надійності та безпеки притаманні UDP і які механізми або протоколи застосовуються для їх компенсації в прикладних системах?

21. Які основні функції виконує рівень застосунків у моделі TCP/IP, і яку роль відіграють протоколи цього рівня у взаємодії користувача з мережевими сервісами?

22. Поясніть принцип роботи HTTP і HTTPS. У чому полягає різниця між ними щодо безпеки та шифрування даних?

23. Охарактеризуйте роботу протоколу FTP, його команди та режими передачі даних. Які механізми підвищення безпеки застосовуються у FTPS та SFTP?

24. Як працює протокол SMTP, які текстові команди він використовує, і яку роль відіграє підтримка MIME у передаванні електронних повідомлень?

25. Поясніть принцип роботи системи DNS, її основні компоненти та процес резолюції доменних імен. Які засоби безпеки, наприклад DNSSEC, використовуються для захисту від підміни даних?

26. Які основні функції виконують утиліти ping та traceroute (або tracert) у мережевій діагностиці, і яку інформацію вони надають про мережеве з’єднання?

27. Охарактеризуйте можливості утиліти netstat. Які типи даних вона відображає та як це допомагає у виявленні мережевих проблем або потенційних загроз?

28. Поясніть призначення утиліти nslookup та механізм перевірки коректності DNS-резолюції. Які типи запитів вона дозволяє виконувати?

29. Які основні функції Wireshark і tshark, і як ці інструменти допомагають у захопленні та аналізі мережевого трафіка? Які етичні аспекти потрібно враховувати у разі їх використання?

30. Порівняйте утиліти curl і nmap за їхніми можливостями та призначенням у роботі з мережею. У яких випадках кожен інструмент застосовується найефективніше?

2 АСИНХРОННА ТА СИНХРОННА КОМУНІКАЦІЯ

У цьому розділі досліджуються фундаментальні відмінності між асинхронними та синхронними методами комунікації у мережевому програмуванні. Він визначає кожен тип комунікації, виділяючи їхні унікальні характеристики, переваги та типові сценарії використання. Розділ надає практичні рекомендації щодо реалізації обох методів у C++ додатках, а також порівняльний аналіз для допомоги у виборі відповідного підходу залежно від конкретних вимог. Досліджуються також реальні сценарії та аспекти продуктивності, пропонуючи погляд на стратегічне застосування цих комунікаційних технік у розробці мережевого програмного забезпечення.

2.1 Визначення синхронної комунікації

Синхронна комунікація в інформатиці означає метод взаємодії, за якого операції виконуються у заздалегідь визначеній послідовності, причому кожна наступна операція очікує завершення попередньої перед початком. Це особливо важливо у сценаріях, де потрібна негайна відповідь та необхідно забезпечити цілісність даних або збереження порядку виконання операцій.

Визначальні характеристики синхронної комунікації

Послідовність та строгий порядок виконання. Синхронна комунікація ґрунтується на дотриманні чіткої послідовності операцій. На відміну від асинхронних методів, де процеси можуть працювати незалежно і продовжувати виконання без очікування відповіді, синхронна взаємодія забезпечує суворе дотримання порядку виконання. Це робить її надзвичайно придатною для завдань, що потребують точності та гарантії відповіді.

Блокувальний характер. Синхронні операції блокують процес. Процес, що перебуває у режимі комунікації, не переходить до наступного кроку, доки не отримано відповідь. Така блокувальна поведінка спрощує логіку програми, оскільки програмісту не потрібно управляти складними станами або перевіряти завершення операції.

Передбачуваний час виконання. Оскільки кожна операція очікує завершення попередньої, час виконання синхронних операцій передбачуваний, що є важливою властивістю для систем з високими вимогами надійності, наприклад, авіоніки чи промислових автоматизованих систем.

Зручність налагодження. Через послідовне виконання операцій налагодження синхронних систем зазвичай простіше, ніж асинхронних. Кожен крок можна відстежити у хронологічному порядку, що полегшує виявлення джерела проблеми.

Використання ресурсів. Синхронна комунікація може бути ресурсомісткою, особливо за умов нестабільної мережі, оскільки будь-які затримки впливають на весь ланцюжок операцій.

У мережевих комунікаціях синхронні методи реалізуються у протоколах, де необхідний прямий та швидкий обмін інформацією. Прикладом є HTTP/1.x, де надісланий запит має отримати відповідь перед відправленням наступного запиту на тому самому з'єднанні.

У практичних системах синхронна комунікація часто спостерігається у клієнт-серверних моделях із режимом запит-відповідь. Наприклад, традиційний веббраузер, що взаємодіє з вебсервером через протокол HTTP, очікує явну відповідь на кожен запит перед відправкою наступного.

Наведений фрагмент коду демонструє реалізацію синхронної клієнтської мережевої взаємодії мовою C++ з використанням BSD-сокетів у середовищі операційних систем сімейства UNIX/Linux. Програма здійснює встановлення TCP-з'єднання з віддаленим сервером і виконує обмін даними за моделлю запит–відповідь.

На початковому етапі програма створює сокет за допомогою системного виклику `socket()` з параметрами `AF_INET` (адресація IPv4) та `SOCK_STREAM` (потоковий сокет, що відповідає протоколу TCP). У разі помилки створення сокета виводиться відповідне повідомлення.

Далі ініціалізується структура `sockaddr_in`, у якій задаються:

- IP-адреса сервера у вигляді числового подання (`inet_addr`),
- тип адресного сімейства (`AF_INET`),
- номер порту сервера, перетворений у мережевий порядок байтів за допомогою `htons()`.

Після цього клієнт ініціює з'єднання з віддаленим сервером шляхом виклику функції `connect()`. Цей виклик має блокувальний характер, що є характерною ознакою синхронної комунікації: виконання програми зупиняється до моменту успішного встановлення з'єднання або виникнення помилки.

У разі успішного підключення клієнт надсилає HTTP-запит типу `GET` за допомогою функції `send()`. Передавання даних також здійснюється синхронно, і програма очікує завершення операції надсилання.

Отримання відповіді від сервера виконується викликом `recv()`, який блокує виконання програми до надходження даних. Це ще раз підкреслює синхронну природу реалізованої взаємодії, оскільки наступні інструкції виконуються лише після завершення приймання повідомлення.

Після отримання відповіді дані виводяться у стандартний потік виведення, сокет закривається за допомогою функції `close()`, і програма коректно завершує роботу.

Таким чином, цей приклад наочно ілюструє принципи синхронного мережевого програмування, зокрема блокувальний характер операцій, послідовність виконання дій та модель клієнт–сервер, що широко застосовується в мережевих застосунках і протоколах прикладного рівня.

```
#include <iostream>          // Підтримка стандартних потоків введення-виведення
#include <cstring>       // Функції для роботи з C-рядками (strlen)
```

```
#include <sys/socket.h>  // Оголошення системних викликів для роботи з сокетами
#include <arpa/inet.h>   // Функції для перетворення IP-адрес і роботи з IPv4
#include <unistd.h>      // Системні виклики POSIX (close)

int main() {
   int socket_desc;              // Дескриптор сокета
   struct sockaddr_in server;       // Структура для зберігання параметрів сервера
   const char *message;           // HTTP-запит (константний рядок)
   char server_reply[2000];              // Буфер для збереження відповіді сервера

   // Створення TCP-сокета (AF_INET — IPv4, SOCK_STREAM — потоковий сокет)
   socket_desc = socket(AF_INET, SOCK_STREAM, 0);
   if (socket_desc == -1) {
      std::cout << "Не вдалося створити сокет";
      return 1;
   }

   // Ініціалізація параметрів сервера
   server.sin_addr.s_addr = inet_addr("74.125.235.20");  // IP-адреса віддаленого вузла
   server.sin_family = AF_INET;                    // Сімейство адрес IPv4
   server.sin_port = htons(80);                    // Номер порту (HTTP), у мережевому порядку байтів

   // Встановлення TCP-з’єднання з віддаленим сервером
   if (connect(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0) {
      std::cout << "Помилка встановлення з’єднання";
      close(socket_desc);
      return 1;
   }

   // Формування та надсилання HTTP-запиту методом GET
   message = "GET / HTTP/1.1\r\n\r\n";
   if (send(socket_desc, message, strlen(message), 0) < 0) {
      std::cout << "Помилка надсилання даних";
      close(socket_desc);
      return 1;
   }

   // Прийом відповіді від сервера (блокувальний виклик)
```

```
    int bytes_received = recv(socket_desc, server_reply, sizeof(server_reply)
- 1, 0);
    if (bytes_received < 0) {
        std::cout << "Помилка приймання даних";
    } else {
        // Завершення отриманого масиву символом кінця рядка
        server_reply[bytes_received] = '\0';
        std::cout << "Отримана відповідь сервера:\n" << server_reply;
    }

    // Закриття сокета та звільнення системних ресурсів
    close(socket_desc);
    return 0;
}
```

Цей приклад демонструє синхронну модель мережевої взаємодії, у якій кожна операція введення-виведення (connect, send, recv) блокує виконання програми до моменту її завершення. Такий підхід забезпечує простоту логіки та передбачуваність виконання, що є доцільним для навчальних цілей і систем із невисокими вимогами до паралельності.

Наведений приклад демонструє просту модель синхронної взаємодії з використанням сокетів у мові програмування C++. У цьому випадку клієнт встановлює з’єднання із сервером і очікує на відповідь сервера перед продовженням подальшого виконання програми. Блокувальний характер функції `recv` забезпечує виконання наступних дій лише після отримання повідомлення від сервера.

Переваги синхронної комунікації насамперед зумовлені її простотою реалізації та передбачуваністю виконання. У середовищах, де критично важливо забезпечити строгу послідовність виконання операцій і негайне отримання зворотного зв’язку після кожної дії, синхронні системи є ефективним і надійним рішенням.

Водночас блокувальна природа синхронної взаємодії може бути неефективною, особливо в розподілених системах, де затримки мережі здатні істотно впливати на загальну продуктивність. Крім того, оскільки кожна операція має завершитися до початку наступної, це призводить до збільшення часу очікування та обмежує масштабованість системи зі зростанням кількості паралельних взаємодій.

Незважаючи на зазначені обмеження, синхронна комунікація залишається доцільним підходом у низці специфічних сценаріїв. Критичні системи або системи, що взаємодіють із фізичними пристроями, де точність часових характеристик є визначальною, часто застосовують синхронні методи для збереження цілісності керування та надійності роботи.

Синхронна взаємодія використовується в різних галузях, де пріоритетом є негайне підтвердження виконання операцій і дотримання їх послідовності.

Основні сфери застосування

Телекомунікації – традиційні телефонні системи, у яких чергування мовлення між абонентами потребує строгого порядку обміну.

Вбудовані системи – зокрема системи промислової автоматизації, що потребують точного керування та контролю кожної операції з міркувань безпеки.

Фінансові транзакції – біржові та банківські операції, де коректність і послідовність обміну інформацією є критично важливими.

Системи командування та управління – військові застосування, у яких час реакції та передбачуваність виконання команд мають вирішальне значення.

У процесі розроблення прикладного програмного забезпечення синхронні методи зазвичай є простішими для реалізації, якщо обсяг транзакцій є помірним, а структура взаємодії має передбачуваний характер. Це дозволяє чітко визначати межі масштабованості системи та сприяє застосуванню дисциплінованих підходів до програмування.

Приклад програми на C++ для синхронного HTTP-запиту за допомогою `libcurl`, разом із Makefile для коректної компіляції.

```
#include <iostream>
#include <string>
#include <curl/curl.h>

// Callback-функція для обробки отриманих даних
// Вона викликається libcurl для кожного блока даних, що надходить від сервера
size_t WriteCallback(void* contents, size_t size, size_t nmemb, void* userp) {
    // Обчислюємо реальний розмір блока
    size_t totalSize = size * nmemb;
    // Додаємо отримані дані до рядка response
    std::string* response = static_cast<std::string*>(userp);
    response->append(static_cast<char*>(contents), totalSize);
    return totalSize;
}

// Функція для виконання синхронного HTTP GET запиту
std::string synchronousRequest(const std::string& url) {
    std::string response; // рядок для збереження відповіді
    CURL* curl = curl_easy_init(); // Ініціалізація libcurl
    if(curl) {
        // Встановлюємо URL для запиту
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        // Встановлюємо callback для обробки відповіді
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        // Передаємо рядок response для збереження даних
```

```
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response);
        // Виконуємо HTTP запит (блокує до завершення)
        CURLcode res = curl_easy_perform(curl);
        if(res != CURLE_OK) {
            std::cerr << "curl_easy_perform() failed: "
                    << curl_easy_strerror(res) << std::endl;
        }
        // Очистка ресурсів libcurl
        curl_easy_cleanup(curl);
    } else {
        std::cerr << "Failed to initialize CURL" << std::endl;
    }
    return response; // Повертаємо отриману відповідь
}

int main() {
    std::string url = "http://www.example.com"; // URL для запиту
    std::cout << "Sending synchronous HTTP GET request to " << url <<
std::endl;

    std::string result = synchronousRequest(url); // Виконання запиту

    std::cout << "Response received:\n" << result << std::endl;
    return 0;
}

# Компілятор C++
CXX = g++
# Флаги компілятора
CXXFLAGS = -std=c++17 -Wall -O2
# Бібліотеки
LIBS = -lcurl

# Назва виконуваного файлу
TARGET = main
# Джерельний файл
SRC = main.cpp

all: $(TARGET)

$(TARGET): $(SRC)
  $(CXX) $(CXXFLAGS) $(SRC) -o $(TARGET) $(LIBS)

clean:
  rm -f $(TARGET)
```

Інструкції для компіляції та запуску

1. Встановити libcurl:

```
sudo apt install libcurl4-openssl-dev   # Ubuntu/Debian
```

або для Nix:

```
nix-env -iA nixpkgs.curl
nix-env -iA nixpkgs.curl.dev
```

2. Скомпілювати:

```
make
```

3. Запустити:

```
./main
```

Цей приклад:

- ✓ демонструє синхронну комунікацію (блокуючу) через HTTP;
- ✓ використовує `libcurl` із callback-функцією для збору відповіді;
- ✓ може бути використаний у навчальних прикладах для пояснення синхронної роботи клієнта.

У цьому прикладі на C++ виконується синхронний HTTP-запит із використанням бібліотеки libcurl. Простота такої реалізації є однією з головних переваг синхронного способу комунікації: налаштування, виконання та обробка помилок організовані в чітку, лінійну та легко відстежувану послідовність дій.

Під час розробки системи, що використовує синхронну комунікацію, потрібно враховувати кілька ключових факторів.

Вимоги до пропускної здатності. Синхронні системи обмежують пропускну здатність, оскільки операції виконуються послідовно. У системах із високими вимогами до пропускної здатності можуть бути рекомендовані асинхронні або багатопотокові підходи.

Стійкість до відмов. Через тісний зв’язок між операціями збої в одному компоненті можуть призвести до зупинки ланцюга процесів, тому проєктування стійкості та механізмів відновлення помилок є критично важливим.

Чутливість до затримок. Система має витримувати мережеві затримки без значного погіршення продуктивності або блокування процесів.

Надійність мережі. Надійність каналу комунікації безпосередньо впливає на ефективність синхронної взаємодії. Високий рівень втрати пакетів або нестабільна швидкість передачі можуть призвести до зниження ефективності.

Розуміння цих факторів дозволяє розробникам оцінити переваги та обмеження синхронної комунікації, забезпечуючи обґрунтоване рішення, адаптоване до конкретного контексту застосування. У кінцевому підсумку, синхронна комунікація залишається ключовою технікою в широкому спектрі мережевих систем, її лінійний та дисциплінований підхід незамінний у сценаріях, що потребують строгого порядку та прозорості операцій.

2.2 Визначення асинхронної комунікації

Асинхронна комунікація – це метод, який дозволяє операціям виконуватися незалежно одна від одної, забезпечуючи можливість прогресування процесів без очікування відповіді або завершення попередніх команд. Така відокремленість робить асинхронну комунікацію потужним інструментом, особливо в мережевих системах, де затримки та варіативність часу відповіді є типовими проблемами.

Визначальною особливістю асинхронної комунікації є здатність одночасно ініціювати кілька операцій, оптимізуючи використання ресурсів та підвищуючи реактивність системи. На відміну від синхронного підходу, асинхронна комунікація не потребує строгого порядку завершення операцій, надаючи гнучкість у керуванні та плануванні завдань.

Особливості асинхронної комунікації

Неблокувальні операції. Операції ініціюються без очікування завершення попередніх, що дозволяє системі продовжувати виконання інших завдань одночасно.

Конкурентність. Підтримка одночасного виконання кількох завдань підвищує загальну пропускну здатність та ефективність системи.

Масштабованість. Асинхронні системи є більш масштабованими завдяки неблокувальній природі, здатні обробляти більший обсяг операцій одночасно.

Складність реалізації. Програмування асинхронних систем потребує використання таких конструкцій, як callback-функції, promises та future-об’єкти, що підвищує складність розробки. Ці особливості роблять асинхронну комунікацію особливо придатною для сучасних додатків, включно з вебсервісами, де користувачі очікують швидкої та реактивної взаємодії.

На практиці асинхронна комунікація застосовується в різних сферах, де незалежна обробка завдань підвищує ефективність. Яскравим прикладом є розробка неблокувальних серверних архітектур, таких як Node.js, що забезпечує високу пропускну здатність і ефективне використання ресурсів.

```
#include <iostream>
#include <future>      // Для std::async та std::future
#include <chrono>      // Для std::chrono::seconds
#include <thread>      // Для std::this_thread::sleep_for
```

```
// Функція, яка імітує асинхронне завдання
int performAsyncTask(int duration) {
    // Затримка, що імітує тривалу операцію
    std::this_thread::sleep_for(std::chrono::seconds(duration));
    return duration; // Повертає тривалість виконання
}

int main() {
    // Ініціалізація асинхронного завдання із запуском в окремому потоці
    std::future<int> result = std::async(std::launch::async, performAsyncTask, 5);

    // Виконання інших операцій під час очікування асинхронного завдання
    std::cout << "Processing other tasks while waiting for asynchronous task." << std::endl;

    // Імітація додаткової роботи в головному потоці
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Отримання результату асинхронного завдання (блокує, якщо завдання ще не завершене)
    int value = result.get();
    std::cout << "Async task completed with duration: " << value << " seconds" << std::endl;

    return 0;
}
```

Коментарі до коду:

1. Асинхронне виконання: `std::async` дозволяє запускати функції в окремих потоках без блокування головного потоку.
2. Механізм майбутнього результату: `std::future` слугує контейнером для значення, яке буде обчислене асинхронно.
3. Неблокувальний головний потік: Під час виконання асинхронного завдання головний потік може обробляти інші операції, підвищуючи ефективність використання ресурсів.
4. Блокування під час доступу до результату: Виклик `result.get()` блокує потік до завершення асинхронної операції, забезпечуючи синхронізацію.
5. Імітація тривалих процесів: `std::this_thread::sleep_for` використовується для моделювання витрат часу на обчислення або очікування відповіді.

Цей приклад демонструє використання асинхронних операцій за допомогою `std::async` у C++. Функція `performAsyncTask` виконується

незалежно від головного потоку, дозволяючи іншим процесам виконуватися одночасно.

Здатність керувати завданнями паралельно без блокування потоку виконання є значною перевагою моделей асинхронної комунікації.

Переваги асинхронної комунікації полягають насамперед у підвищенні швидкодії та максимізації ефективності використання системних ресурсів.

Системи, що застосовують асинхронні методи, можуть обробляти більше операцій одночасно, зменшуючи час очікування та підвищуючи пропускну здатність.

Однак складність реалізації асинхронної комунікації може створювати значні труднощі. Розробники мають враховувати проблеми, пов’язані з паралельним виконанням, такі як умови гонки та взаємні блокування, що потребує ґрунтовного розуміння примітивів конкурентності та ретельного планування проєктування.

Декупльований (роздільний) характер асинхронної комунікації також може ускладнювати підтримку цілісності даних та забезпечення успішного завершення транзакцій без перешкод, що часто потребує додаткових механізмів для обробки збоїв і повторних спроб.

Асинхронна комунікація особливо ефективна в середовищах, де критично важливі ефективність, швидкість та масштабованість. Основні галузі, які використовують асинхронні методи:

Веброзробка. JavaScript та Node.js активно застосовують асинхронні виклики для ефективної обробки великого обсягу запитів без блокування головного потоку виконання.

Хмарні обчислення. Платформи хмарних сервісів часто використовують асинхронну комунікацію для операцій, що потребують швидкої обробки та високого потенціалу масштабованості.

Конвеєри машинного навчання. Паралельне розподілення завдань і обчислень асинхронно дозволяє максимально використовувати обчислювальні ресурси та прискорює обробку даних.

Розподілені системи. Розподілені архітектури часто застосовують асинхронні протоколи для ефективної взаємодії між незалежними вузлами мережі.

Розробники використовують різні моделі та структури для реалізації асинхронної комунікації. Поширені підходи містять застосування callback-функцій, promises та парадигми реактивного програмування, що дозволяють безшовно інтегрувати асинхронні завдання.

Фреймворки та мови програмування пропонують різний рівень підтримки цих моделей, забезпечуючи ефективний дизайн застосунків без блокування та підвищуючи продуктивність систем.

Варіант коду на C++ з асинхронним виконанням завдання за допомогою `std::promise` і `std::future`.

```
#include <iostream>
#include <future>      // Для std::promise і std::future
#include <thread>      // Для std::thread
```

```
#include <chrono>      // Для std::chrono::seconds

// Функція, що виконує асинхронне завдання
void doTask(std::promise<int>&& p) {
    // Симуляція тривалої обробки (1 секунда)
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Встановлення результату завдання у об’єкт promise
    p.set_value(42); // Завдання завершено
}

int main() {
    // Створення promise для передачі результату асинхронного завдання
    std::promise<int> promise;

    // Отримання future для отримання результату
    std::future<int> future = promise.get_future();

    // Запуск асинхронного завдання в окремому потоці
    std::thread taskThread(doTask, std::move(promise));

    std::cout << "Виконуються інші операції під час очікування результату завдання..." << std::endl;

    // Очікування результату асинхронного завдання з тайм-аутом 2 секунди
    auto status = future.wait_for(std::chrono::seconds(2));

    if (status == std::future_status::ready) {
        // Результат готовий, отримуємо його через future.get()
        std::cout << "Результат отримано: " << future.get() << std::endl;
    } else {
        // Завдання триває довше очікуваного часу
        std::cout << "Завдання виконується довше очікуваного часу." << std::endl;
    }

    // Очікування завершення потоку для запобігання передчасному завершенню програми
    taskThread.join();

    return 0;
}
```

Ключові моменти:

1. *Асинхронність*. Використання `std::thread` разом із `std::promise` і `std::future` дозволяє виконувати завдання незалежно від головного потоку.

2. *Очікування з тайм-аутом*. Метод `wait_for()` дає змогу перевіряти завершення завдання без блокування головного потоку.

3. *Безпека потоків*. Передача `std::promise` через `std::move()` гарантує, що об'єкт належить саме потоку, який виконує завдання.

4. *Контроль завершення*. Виклик `join()` забезпечує синхронізацію та запобігає аварійному завершенню програми до завершення всіх потоків.

Цей приклад демонструє використання `std::promise` та `std::future` для керування асинхронними операціями, що є типовими будівельними блоками асинхронного спілкування в C++. Він ілюструє, як ці конструкції дозволяють гнучко очікувати результати асинхронних завдань та надають механізм для визначення їхнього завершення або поточного стану.

Проєктування систем на основі асинхронного спілкування потребує ретельного врахування кількох факторів:

Контроль конкурентності. Необхідно правильно використовувати механізми синхронізації для керування накладанням операцій та конкуренцією за ресурси.

Обробка помилок. Асинхронні операції часто потребують надійних стратегій обробки помилок, які дозволяють природно відновлюватися після часткових збоїв.

Оптимізація продуктивності. Асинхронні системи мають проєктуватися з урахуванням показників продуктивності, забезпечуючи, щоб переваги паралельної обробки перевищували накладні витрати.

Масштабованість. Система має адекватно масштабуватися залежно від навантаження на операції, забезпечуючи ефективне управління ресурсами для підтримки високого рівня паралельності.

Під час розробки асинхронних систем важливо розуміти наслідки розбиття завдань на менші, незалежно виконувані одиниці та забезпечити відповідні стратегії для координації результатів. Усвідомлюючи природну складність, але й значні переваги асинхронного спілкування, воно залишається потужним інструментом для створення сучасних, чутливих і масштабованих мережевих систем.

2.3 Порівняльний аналіз методів передавання

Розуміння принципів синхронного та асинхронного передавання є критично важливим для ефективного проєктування та впровадження мережевих систем. Ці методи спілкування мають унікальні характеристики та обслуговують різні вимоги залежно від потреб системи. Глибокий порівняльний аналіз допомагає вибрати найбільш придатний підхід для конкретних сценаріїв застосування, забезпечуючи оптимізовану продуктивність, надійність та масштабованість.

Контроль операцій. Синхронне спілкування потребує, щоб кожна операція очікувала завершення попередньої перед продовженням. Це контрастує з неблокувальною структурою асинхронного передавання, де операції можуть ініціюватися незалежно та виконуватися поза порядком.

Така відмінність значно впливає на проєктування та оптимізацію систем для керування завданнями та їх обробки.

Керованість відповіді. Синхронні системи зазвичай легші для управління через передбачуваний та лінійний контрольний потік, де кожен крок має бути завершений перед початком наступного. У асинхронних системах одночасно виконуються кілька операцій, що ускладнює відстеження та синхронізацію відповідей, але дозволяє підвищити пропускну здатність і ефективність системи.

Продуктивність. Продуктивність є критерієм порівняння методів передавання. Блокувальна природа синхронного спілкування може вводити затримки у систему, особливо в мережевих середовищах, де затримки передачі даних поширені. На противагу цьому, асинхронне передавання ефективне в середовищах, де система може продовжувати обробку інших завдань під час очікування завершення операцій, зменшуючи простої та підвищуючи загальну пропускну здатність системи.

```
#include <iostream>
#include <thread>
#include <chrono>

// Імітація блокувального синхронного завдання
void synchronousTask() {
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Імітація затримки
    std::cout << "Синхронне завдання виконано." << std::endl;
}

// Імітація асинхронного завдання
void asynchronousTask() {
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Імітація затримки
    std::cout << "Асинхронне завдання виконано." << std::endl;
}

int main() {
    // Запуск синхронного завдання
    std::cout << "Початок виконання синхронного завдання..." << std::endl;
    synchronousTask(); // Блокувальний виклик: головний потік очікує завершення завдання

    // Запуск асинхронного завдання
    std::cout << "Початок виконання асинхронного завдання..." << std::endl;
    std::thread taskThread(asynchronousTask); // Створення нового потоку для асинхронного завдання
```

```
    taskThread.detach(); // Відокремлення потоку, дозволяє головному потоку продовжувати роботу

    std::cout << "Головний потік вільний для виконання інших завдань." << std::endl;

    // Імітація виконання інших операцій у головному потоці
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Робота головного потоку завершена." << std::endl;

    return 0;
}
```

Пояснення:

1. Синхронне завдання (`synchronousTask`) блокує головний потік до завершення, забезпечуючи передбачуваний порядок виконання.
2. Асинхронне завдання (`asynchronousTask`) запускається в окремому потоці, дозволяючи головному потоку продовжувати роботу без очікування.
3. Використання `detach()` дозволяє асинхронному потоку виконуватися незалежно, але потребує, щоб головний потік не завершився раніше за асинхронне завдання.
4. Такий підхід демонструє основну перевагу асинхронного спілкування: підвищену продуктивність і ефективне використання ресурсів системи.

У цьому прикладі на C++ проведено порівняння синхронних та асинхронних завдань. Синхронна функція блокує головний потік до завершення виконання, що може призводити до неефективного використання ресурсів. Натомість асинхронне завдання виконується в окремому потоці, дозволяючи головному потоку виконувати інші завдання паралельно, демонструючи підвищений рівень паралелізму та ефективне використання ресурсів.

Синхронне оброблення. Зазвичай застосовується в системах, де критично важлива послідовність виконання операцій та негайна відповідь, наприклад, у застарілих системах або в щільно зв’язаних транзакційних системах фінансового сектора, де атомарні транзакції мають завершуватися послідовно.

Асинхронне оброблення. Найефективніше в середовищах, де пріоритетним є паралелізм та масштабованість, таких як сучасні вебсервери та хмарні додатки, що мають ефективно обробляти численні одночасні запити без утворення вузьких місць.

```
import asyncio

# Асинхронна функція обробки запиту клієнта
async def handle_request(reader, writer):
    # Читаємо дані, що надійшли, до 100 байт
```

```
    data = await reader.read(100)
    message = data.decode()  # Декодуємо байти у рядок
    addr = writer.get_extra_info('peername')  # Отримуємо адресу клієнта

    print(f"Отримано повідомлення '{message}' від {addr}")

    # Відправляємо назад ті самі дані (exo-сервер)
    print("Відправка: %s" % message)
    writer.write(data)
    await writer.drain()  # Асинхронно очищаємо буфер передачі

    print("Закриття з'єднання")
    writer.close()  # Закриваємо з'єднання
    await writer.wait_closed()  # Очікуємо повного закриття

# Основна асинхронна функція запуску сервера
async def main():
    # Створюємо TCP-сервер на localhost:8888
    server = await asyncio.start_server(
        handle_request, '127.0.0.1', 8888
    )

    # Асинхронний контекст керування сервером
    async with server:
        await server.serve_forever()  # Працюємо безперервно

# Запуск асинхронного головного циклу
asyncio.run(main())
```

Пояснення:

1. Асинхронна обробка запитів: Використання `async` та `await` дозволяє серверу обробляти численні з’єднання одночасно без блокування головного потоку.

2. Читання та запис: `reader.read()` та `writer.write()` забезпечують асинхронний I/O, що підвищує ефективність системи за великої кількості клієнтів.

3. Буферизація: `await writer.drain()` гарантує, що всі дані будуть надіслані перед завершенням функції.

4. Безперервна робота сервера: `serve_forever()` дозволяє TCP-серверу постійно приймати підключення, ідеально підходить для високонавантажених мережевих сервісів.

5. Асинхронне закриття з’єднання: `writer.wait_closed()` забезпечує коректне завершення сеансу, запобігаючи втраті даних.

Технології асинхронного зв’язку, такі як `asyncio` у Python, широко застосовуються у вебсерверах для обробки великої кількості з’єднань одночасно без необхідності використання багатопотоковості, демонструючи, як асинхронний зв’язок дозволяє створювати ефективні та масштабовані архітектури сервісів.

Зручність налагодження та підтримки. Синхронний зв’язок забезпечує простіший процес налагодження завдяки своєму лінійному та передбачуваному виконанню. Відстеження шляху виконання та ідентифікація точок відмови значно спрощується у передбачуваному покроковому процесі.

Асинхронні системи через їхню нелінійну та часто паралельну природу потребують більш складних стратегій налагодження, що можуть містити логування та інструменти профілювання продуктивності для управління проблемами конкуренції, такими як deadlock-и та race condition. Незважаючи на складність, фреймворки та бібліотеки забезпечують конструкції, що покращують відстежуваність і підтримку асинхронних застосунків.

Управління ресурсами та масштабованість: Масштабування синхронних систем часто потребує значних ресурсних витрат через блокувальні операції, що утримують ресурси, очікуючи відповіді перед продовженням роботи. Навпаки, асинхронні системи ефективно керують ресурсами, продовжуючи виконання інших операцій під час очікування, що забезпечує кращу масштабованість у розподілених і навантажених середовищах.

У мережевих протоколах впровадження асинхронної схеми комунікації дозволяє суттєво зменшити затримки та підвищити пропускну здатність, особливо коли час кругового обміну мережею (RTT) є змінним. Завдяки можливості продовжувати інші операції під час очікування відповіді мережі, система зберігає високу чутливість до запитів користувача.

Безпекові аспекти: Вибір між синхронним та асинхронним зв’язком також залежить від безпекових вимог. Синхронні системи через послідовність операцій природно обмежують проблеми конкуренції, проте можуть бути вразливими до блокувальних атак, таких як DoS, коли процеси залишаються в очікуванні відповіді від мережі.

Асинхронні системи потребують ретельнішого підходу до безпеки, пов’язаної з конкурентним доступом до ресурсів, захисту спільних даних та забезпечення цілісності й конфіденційності даних, що можуть перебувати у проміжних буферах або кешах пам’яті між асинхронними операціями.

Порівняльна характеристика. Ознайомившись із основними властивостями синхронного та асинхронного зв’язку, інженери та розробники мають враховувати конкретні сценарії використання та

балансувати переваги. Вибір методу має визначатися вимогами застосунку щодо синхронізації процесів, розподілу ресурсів та цілей продуктивності.

Аспект	Синхронна комунікація	Асинхронна комунікація
Режим роботи	Блокувальний, послідовне виконання	Неблокувальний, паралельне виконання
Масштабованість	Обмежена утриманням ресурсів під час блокування	Вища завдяки ефективному використанню ресурсів
Чутливість	Передбачувана, але може затримувати інші задачі	Максимізована пропускна здатність, складніше керування
Відлагодження	Просте через лінійний контроль виконання	Складне через нелінійні події та конкуренцію
Безпека	Менше проблем конкуренції, але вразлива до DoS	Потребує контролю цілісності даних під час паралельних операцій

Обидва підходи мають свої переваги та компроміси; для конкретних сценаріїв потрібно ретельно оцінювати їхню доцільність залежно від вимог до контролю транзакцій, часу відгуку та ефективності витрат. Такий аналіз слугує керівництвом для фахівців під час архітектурного проєктування систем, які мають бути надійними та масштабованими у різних доменах.

2.4 Впровадження синхронного зв’язку на C++

Реалізація синхронного зв’язку в C++ є критичною для багатьох випадків, де важлива цілісність порядку операцій та їхня миттєва обробка. Цей розділ розглядає техніки та конструкції C++, що забезпечують синхронну комунікацію відповідно до стандартних бібліотек і можливостей мови. У C++ синхронний зв’язок зазвичай характеризується блокувальними операціями, що виконуються послідовно.

Одним із основних застосувань синхронного зв’язку є мережеве програмування з використанням сокетів. Програмування TCP/IP сокетів є за своєю природою синхронним, забезпечуючи гарантії порядку доставки повідомлень та підтвердження отримання. Для встановлення синхронного TCP-з’єднання в C++ основним інтерфейсом є стандартне API сокетів. Наступний приклад демонструє просте синхронне підключення до сервера з використанням сокетів.

```
#include <iostream>
#include <cstring>      // Для strcpy, strlen
#include <sys/socket.h> // Основні сокет-функції
#include <arpa/inet.h>  // Для inet_addr та htons
#include <unistd.h>     // Для close()

int main() {
   // Оголошення дескриптора сокета та структури для адреси сервера
   int sock;
```

```
    struct sockaddr_in server;

    // Буфери для відправки та отримання даних
    char message[1000], server_reply[2000];

    // 1. Створення сокета TCP (SOCK_STREAM)
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        std::cerr << "Не вдалося створити сокет" << std::endl;
        return 1;
    }
    std::cout << "Сокет створено" << std::endl;

    // 2. Налаштування параметрів сервера
    server.sin_addr.s_addr = inet_addr("192.168.1.1"); // IP-адреса сервера
    server.sin_family = AF_INET;                    // Використання IPv4
    server.sin_port = htons(8888);                  // Порт сервера

    // 3. Підключення до віддаленого сервера
    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        std::cerr << "Не вдалося підключитися до сервера" << std::endl;
        return 1;
    }
    std::cout << "Підключення встановлено" << std::endl;

    // 4. Підготовка та відправка HTTP-запиту
    strcpy(message, "GET / HTTP/1.1\r\n\r\n"); // Стандартний запит HTTP GET
    if (send(sock, message, strlen(message), 0) < 0) {
        std::cerr << "Помилка під час відправлення даних" << std::endl;
        return 1;
    }
    std::cout << "Дані відправлено" << std::endl;

    // 5. Отримання відповіді від сервера
    memset(server_reply, 0, sizeof(server_reply)); // Очищення буфера перед прийомом
    if (recv(sock, server_reply, sizeof(server_reply) - 1, 0) < 0) {
        std::cerr << "Помилка під час отримання даних" << std::endl;
        return 1;
    }
    std::cout << "Отримано відповідь:\n" << server_reply << std::endl;

    // 6. Закриття сокета після завершення комунікації
```

```
    close(sock);
    return 0;
}
```

1. Коментарі науково-технічного стилю пояснюють кожен етап: створення сокета, підключення, відправка, прийом даних.

2. Використано `memset` для очищення буфера перед `recv()`, щоб уникнути залишкових даних у буфері.

3. Буфери для `recv()` та `send()` налаштовані на коректний розмір.

4. Повідомлення про помилки чітко відокремлені для кожного етапу комунікації.

У наведеному прикладі створюється сокет за допомогою POSIX API для сокетів, який блокує клієнтський додаток до моменту успішного встановлення з'єднання з сервером. Програма очікує на відповідь для кожного блоку операцій, забезпечуючи критично важливу послідовну обробку, що є характерною ознакою синхронного зв'язку.

У C++ синхронне спілкування не обмежується лише мережевим програмуванням, а поширюється й на інші області, такі як операції введення-виведення з файлами, де дані читаються або записуються у блокувальному режимі. Стандартна бібліотека C++ для введення-виведення надає класи, наприклад, `fstream`, для виконання таких операцій.

У наступному прикладі демонструється синхронна робота з файлами в C++ за допомогою класів стандартної бібліотеки `fstream`. Програма спочатку відкриває файл для запису у блокувальному режимі (`std::ofstream`), перевіряє успішність відкриття та записує кілька рядків тексту. Після завершення операцій записування файл закривається.

Далі програма відкриває той самий файл для читання (`std::ifstream`) та почергово зчитує всі рядки, виводячи їх на стандартний потік виведення. Блокувальний характер операцій введення-виведення гарантує, що кожна операція записування або читання завершується перед переходом до наступної, що відповідає принципам синхронного оброблення даних.

Таке використання синхронного введення-виведення є типовим для сценаріїв, де необхідна точна послідовність обробки даних та контроль цілісності інформації.

```
#include <iostream>
#include <fstream>

int main() {
    // Відкриваємо файл для запису
    std::ofstream outfile("example.txt");
    if (!outfile.is_open()) {
        std::cerr << "Помилка відкриття файлу для запису" << std::endl;
```

```
        return 1;
    }

    // Записуємо рядки у файл
    outfile << "Це рядок.\n";
    outfile << "Це ще один рядок.\n";

    // Закриваємо файл після запису
    outfile.close();
    std::cout << "Файл успішно записано" << std::endl;

    // Відкриваємо файл для читання
    std::ifstream infile("example.txt");
    if (!infile.is_open()) {
        std::cerr << "Помилка відкриття файлу для читання" << std::endl;
        return 1;
    }

    // Зчитуємо та виводимо рядки з файлу
    std::string line;
    while (std::getline(infile, line)) {
        std::cout << line << std::endl;
    }

    // Закриваємо файл після читання
    infile.close();

    return 0;
}
```

Цей приклад ілюструє синхронне оброблення файлів, де операції з використанням `fstream` забезпечують послідовне та повне виконання запису в файл та зчитування з нього перед переходом до наступної інструкції. Синхронні послідовності гарантують належне збереження узгодженості даних та часової упорядкованості виконання. Синхронна комунікація також відіграє важливу роль у міжпроцесній взаємодії (IPC), де процеси обмінюються даними через механізми, такі як канали (pipes) або розділена пам’ять. У таких середовищах дані зазвичай передаються блокувальним способом для забезпечення координації між процесами.

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
```

```
int main() {
    const char* pipeName = "/tmp/testpipe";

    // Створення іменованого каналу (FIFO) з правами доступу 0666
    mkfifo(pipeName, 0666);

    pid_t pid = fork(); // Створення дочірнього процесу

    if (pid == 0) {
        // Дочірній процес
        int readFd = open(pipeName, O_RDONLY); // Відкриття FIFO для читання
        char buffer[100];
        read(readFd, buffer, sizeof(buffer)); // Зчитування даних з каналу
        std::cout << "Дочірній процес отримав: " << buffer << std::endl;
        close(readFd); // Закриття дескриптора
    } else {
        // Батьківський процес
        int writeFd = open(pipeName, O_WRONLY); // Відкриття FIFO для запису
        char message[] = "Привіт від батька!";
        write(writeFd, message, sizeof(message)); // Запис повідомлення в канал
        close(writeFd); // Закриття дескриптора
    }

    // Видалення іменованого каналу після використання
    unlink(pipeName);

    return 0;
}
```

Пояснення:

Використано іменований канал (FIFO) для блокувальної синхронної міжпроцесної комунікації.

Дочірній процес очікує на дані, блокуючи виконання до їх надходження.

Батьківський процес записує дані у FIFO, після чого дочірній процес отримує їх.

Після завершення обміну даними видаляється канал (`unlink`) для уникнення залишкових ресурсів.

У цьому коді як батьківський, так і дочірній процеси здійснюють синхронний обмін даними через іменований канал (`/tmp/testpipe`).

Операції читання та записування є блокувальними, що змушує кожну зі сторін очікувати завершення відповідної дії, що є типовим прикладом синхронної комунікації в міжпроцесній взаємодії (IPC).

У середовищах з конкурентними потоками необхідно застосовувати примітиви синхронізації для забезпечення впорядкованого виконання через механізми взаємного виключення (mutex) та змінні умови (condition variables). Такі примітиви підтримують систематичний обмін даними між потоками, запобігаючи станам гонки (race conditions) та забезпечуючи коректність виконання.

У наступному прикладі демонструється використання примітивів синхронізації потоків у C++ для організації впорядкованого запуску кількох потоків. Використовуються:

std::mutex – забезпечує взаємне виключення у разі доступу до спільного ресурсу (`ready`).

std::condition_variable – дозволяє потокам очікувати на зміну стану (`ready`) без активного очікування (busy-wait).

Принцип роботи коду:

1. Створюється масив з 10 потоків, кожен із яких виконує функцію `print_id`.

2. Кожен потік блокується на `cv.wait`, очікуючи, поки `ready` стане `true`.

3. Через секунду головний потік виконує `set_ready()`, встановлюючи стан `ready = true` та повідомляючи всі потоки за допомогою `cv.notify_all()`.

4. Всі заблоковані потоки розблоковуються і послідовно виводять свій ідентифікатор.

5. Головний потік очікує завершення всіх дочірніх потоків через `join()`.

Науково-технічні аспекти

Цей приклад ілюструє синхронну комунікацію між потоками, де запуск певної операції залежить від стану спільного ресурсу.

Примітиви синхронізації забезпечують певний порядок виконання та захист від станів гонки (race conditions).

Використання умовних змінних дозволяє уникнути зайвого споживання процесорного часу в очікуванні (`busy-waiting`), підвищуючи ефективність багатопотокових програм.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

// Оголошення м'ютексу для синхронізації доступу до спільного ресурсу
```

```
std::mutex mtx;

// Оголошення умовної змінної для очікування зміни стану
std::condition_variable cv;

// Спільна змінна стану, що сигналізує готовність потоків
bool ready = false;

// Функція, яку виконують потоки
void print_id(int id) {
   // Унікальне блокування м'ютексу для доступу до стану ready
   std::unique_lock<std::mutex> lock(mtx);

   // Потік очікує на зміну стану ready
   cv.wait(lock, []{ return ready; });

   // Виведення ідентифікатора потоку після отримання сигналу
   std::cout << "Потік " << id << std::endl;
}

// Функція для встановлення стану ready та повідомлення всіх потоків
void set_ready() {
   {
      // Захист доступу до змінної ready м'ютексом
      std::lock_guard<std::mutex> lock(mtx);
      ready = true; // Встановлення стану готовності
   }

   // Повідомлення всіх потоків, що вони можуть продовжити виконання
   cv.notify_all();
}

int main() {
   // Створення масиву потоків
   std::thread threads[10];

   // Ініціалізація потоків
   for (int i = 0; i < 10; ++i) {
      threads[i] = std::thread(print_id, i);
   }

   // Симуляція іншої роботи головного потоку
   std::this_thread::sleep_for(std::chrono::seconds(1));
```

```
    std::cout << "Встановлюємо стан готовності." << std::endl;

    // Встановлення готовності та розблокування потоків
    set_ready();

    // Очікування завершення всіх потоків
    for (auto &t: threads) {
        t.join();
    }

    return 0;
}
```

У цьому прикладі `std::mutex` та `std::condition_variable` працюють спільно для синхронізації потоків, забезпечуючи очікування кожним потоком умови готовності (`ready`). Така синхронізація встановлює певний порядок виконання операцій, що є критично важливим у застосуваннях, де потрібен контрольований доступ до спільних ресурсів у схемах синхронного зв’язку.

Реалізація синхронного зв’язку в C++ потребує уважного врахування потенційних проблем, таких як взаємні блокування (deadlocks) у багатопотокових середовищах, а також вплив блокувальних операцій на швидкодію та чутливість застосунку. Під час проєктування синхронних систем потрібно враховувати такі фактори:

Запобігання deadlock. Забезпечення узгодженого порядку захоплення та звільнення ресурсів може запобігти виникненню сценаріїв взаємного блокування.

Тайм-аути. Введення обмежень за часом для блокувальних операцій дозволяє зменшити негативний вплив, наприклад, проблеми живучості або тривалі часи очікування.

Обробка помилок. Надійні механізми обробки помилок є критично важливими, особливо у мережевих комунікаціях, де затримки та помилки передавання можуть впливати на синхронні операції.

Вплив на продуктивність. Необхідно усвідомлювати компроміси продуктивності під час використанні блокувальних операцій, щоб вибрати відповідні методи залежно від потреб застосунку.

Синхронна комунікація в C++ забезпечує надійний механізм для задач, що залежать від порядку виконання, пропонуючи простоту логіки коду, яка найкраще підходить для застосунків, де передбачуваність і простота процесу є обов’язковими вимогами.

Завдяки ретельній реалізації та зваженню переваг і недоліків синхронних моделей, розробники можуть ефективно використовувати можливості C++ для створення рішень, адаптованих до конкретних вимог експлуатації, що відповідають підходу синхронного виконання.

2.5 Реалізація асинхронного оброблення даних в C++

Асинхронна комунікація є невід'ємною частиною сучасного програмного забезпечення, дозволяючи ефективно використовувати ресурси через одночасне виконання операцій. У C++ реалізація асинхронного зв'язку спирається на низку можливостей та бібліотек, що дозволяють розробникам створювати системи, здатні обробляти великий обсяг задач паралельно, з високою чутливістю до подій.

У стандарті C++11 було введено кілька можливостей, що забезпечують нативну підтримку асинхронного програмування. До них належать потоки (`threads`), об'єкти `futures` та `promises`, а також стандартна бібліотека для асинхронних операцій, що створює основу для побудови складних неблокувальних комунікацій.

Потоки (`threads`) є фундаментальними елементами для побудови конкурентних застосунків, де одночасно виконуються кілька шляхів виконання. Бібліотека `std::thread` у C++ дозволяє керувати асинхронними задачами та підвищувати продуктивність застосунку завдяки паралельній обробці.

```
#include <iostream>
#include <thread>

// Функція для моделювання асинхронної задачі з тривалістю виконання duration секунд
void computeTask(int duration) {
    std::this_thread::sleep_for(std::chrono::seconds(duration)); // Імітація затримки
    std::cout << "Завершено задачу тривалістю: " << duration << " секунд." << std::endl;
}

int main() {
    // Створення двох потоків для паралельного виконання завдань
    std::thread t1(computeTask, 3);
    std::thread t2(computeTask, 5);

    std::cout << "Головний потік продовжує виконання..." << std::endl;

    // Очікування завершення обох потоків
    t1.join();
    t2.join();

    std::cout << "Усі завдання завершено." << std::endl;
    return 0;
}
```

У цьому прикладі демонструється використання багатопотоковості для асинхронного виконання задач у C++. Два потоки `t1` та `t2` одночасно виконують функцію `computeTask` із різними тривалостями.

Основні аспекти реалізації

Паралельне виконання. Кожен потік працює незалежно від головного потоку, дозволяючи головному потоку продовжувати виконання інших задач.

Синхронізація завершення. Виклик `join()` гарантує, що головний потік очікує завершення всіх створених потоків перед завершенням програми.

Ефективність ресурсів. Такий підхід підвищує продуктивність системи, дозволяючи виконувати декілька завдань одночасно, замість послідовного блокування головного потоку.

Висновок. Багатопотоковість у C++ дозволяє ефективно реалізувати асинхронну обробку задач, одночасно забезпечуючи контроль за завершенням кожного потоку через метод `join()`. Це підвищує пропускну здатність і скорочує час очікування порівняно із синхронними моделями виконання.

У цьому прикладі два асинхронні завдання створюються за допомогою потоків, що дозволяє головній програмі продовжувати своє виконання, очікуючи завершення цих завдань. Такий підхід забезпечує оптимальне використання багатоядерних процесорів.

Механізми futures і promises надають засіб для роботи з відкладеними обчисленнями та обробки результатів після завершення завдань. Вони дозволяють ініціювати асинхронні операції та пізніше отримувати їх результати без блокування основного потоку виконання.

У наступному прикладі використовується механізм std::promise та std::future для реалізації асинхронного обчислення в C++. Функція `asyncComputation` виконує обчислення з затримкою, і результат передається через об’єкт `promise`. Головний потік може очікувати завершення обчислення, використовуючи пов’язаний об’єкт `future`, не блокуючи створений для обчислення окремий потік.

Такий підхід забезпечує:

1. Асинхронне виконання обчислень – функція виконується у власному потоці, дозволяючи головному потоку виконувати інші операції.

2. Безпечний обмін результатами – `promise` передає результат у `future`, що гарантує коректне синхронізоване отримання даних.

3. Контроль завершення завдання – виклик `fut.get()` блокує головний потік лише до отримання результату, після чого продовжується виконання.

Такий патерн є базовим елементом асинхронної комунікації в сучасних C++-системах, особливо корисним для багатопотокових програм та розподілених обчислень.

```
#include <iostream>
```

```
#include <thread>
#include <future>

// Функція асинхронного обчислення з затримкою
int асинхронне Обчислення(int значення) {
    std::this_thread::sleep_for(std::chrono::seconds(значення)); // Імітація тривалого обчислення
    return значення * 2; // Повертаємо результат обчислення
}

int main() {
    std::promise<int> обіцянка;                 // Створюємо об’єкт promise для передачі результату
    std::future<int> майбутнє = обіцянка.get_future(); // Отримуємо пов’язаний future для отримання результату

    // Запускаємо асинхронне обчислення у окремому потоці
    std::thread потік([&обіцянка]() {
        обіцянка.set_value(асинхронneОбчислення(5)); // Встановлюємо результат у promise
    });

    std::cout << "Очікування результату..." << std::endl;
    std::cout << "Результат: " << майбутнє.get() << std::endl; // Отримуємо результат через future

    потік.join(); // Чекаємо завершення потоку
    return 0;
}
```

У цьому прикладі за допомогою promise інкапсулюється тривале обчислення. Future використовується для отримання результату після його готовності. Такий підхід абстрагує асинхронність і забезпечує простий спосіб очікування завершення обчислень без блокування головного потоку.

Шаблон функції std::async є частиною стандартної бібліотеки C++ і застосовується для асинхронного запуску функцій. std::async керує створенням потоків і розподілом навантаження, надаючи можливість запускати завдання асинхронно або синхронно залежно від вибраної політики запуску.

```
#include <iostream>
#include <future>
#include <thread>
```

```
#include <chrono>

// Функція для виконання обчислень, що тривають певний час
int performCalculation(int x) {
    std::this_thread::sleep_for(std::chrono::seconds(3)); // Імітація тривалого обчислення
    return x * x; // Повернення квадрата числа
}

int main() {
    // Виконання асинхронного обчислення за допомогою std::async
    std::future<int> result = std::async(std::launch::async, performCalculation, 10);

    std::cout << "Головний потік продовжує роботу, очікуючи на результат..." << std::endl;

    // Отримання результату обчислень; блокування відбудеться лише на цьому моменті
    std::cout << "Результат обчислення: " << result.get() << std::endl;

    return 0;
}
```

У цьому прикладі демонструється використання std::async для асинхронного виконання обчислювальної задачі в C++. Функція `performCalculation` виконує обчислення, що тривають певний час, і повертає результат після завершення. Використання std::future дозволяє головному потоку продовжувати роботу паралельно з обчисленнями та отримати результат лише в момент виклику `get()`. Такий підхід забезпечує ефективне використання ресурсів багатопроцесорних систем та дозволяє організовувати асинхронні обчислення без блокування головного потоку.

`std::async` забезпечує виконання функції `performCalculation` у неблокувальному режимі, що дозволяє головній програмі продовжувати виконання інших задач під час очікування результату. Результат отримується через future, що дає можливість обробляти його після завершення обчислень.

Окрім стандартних засобів багатопотоковості та паралельного виконання, існують спеціалізовані бібліотеки, які надають розширені можливості для асинхронного програмування мережі.

Бібліотеки, такі як Boost.Asio, дозволяють розробникам ефективно структурувати програми з неблокувальною комунікацією. Boost.Asio – це кросплатформенна C++ бібліотека для мережевого та низькорівневого

введення-виведення, що орієнтується на асинхронні операції. Завдяки потужній асинхронній моделі, Boost.Asio дозволяє створювати складні асинхронні системи та архітектури.

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/bind/bind.hpp>

using boost::asio::ip::tcp;

// Обробник завершення асинхронного підключення
void connectHandler(const boost::system::error_code& ec, const tcp::endpoint& endpoint) {
    if (!ec) {
        std::cout << "Підключення встановлено успішно до "
                  << endpoint.address().to_string() << ":"
                  << endpoint.port() << std::endl;
    } else {
        std::cerr << "Не вдалося підключитися: " << ec.message() << std::endl;
    }
}

int main() {
    boost::asio::io_context io_context;

    // Створення TCP-сокета
    tcp::socket socket(io_context);

    // Розв'язувач доменного імені
    tcp::resolver resolver(io_context);
    auto endpoints = resolver.resolve("example.com", "80");

    // Асинхронне підключення до сервера
    boost::asio::async_connect(socket, endpoints, connectHandler);

    // Запуск контексту введення-виведення для обробки асинхронних операцій
    io_context.run();

    return 0;
}
```

У цьому прикладі процес встановлення з'єднання ініціюється асинхронно за допомогою `async_connect`, у цьому випадку як обробник завершення використовується функція зворотного виклику `connectHandler`. Метод `io_context.run()` виконує всі операції, пов'язані з `io_context`, до завершення роботи.

Для успішної розробки асинхронних застосунків необхідне розуміння шаблонів проєктування та передових практик, що регламентують цю нелінійну модель виконання. Модель, керована подіями (event-driven), є фундаментальною для створення асинхронних систем. Вона дозволяє застосункам реагувати на події або повідомлення та виконувати відповідні функції зворотного виклику без блокування операцій.

```
#include <iostream>
#include <functional>
#include <thread>
#include <chrono>

// Асинхронна функція, що приймає callback для обробки результату
void asyncOperation(std::function<void(int)> callback) {
    std::cout << "Виконується операція..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Імітація тривалої операції
    callback(42); // Виклик callback з результатом
}

// Функція обробки результату
void resultCallback(int result) {
    std::cout << "Операція завершена з результатом: " << result << std::endl;
}

int main() {
    asyncOperation(resultCallback); // Виклик асинхронної операції
    std::cout << "Головна функція продовжує виконання." << std::endl;

    std::this_thread::sleep_for(std::chrono::seconds(3)); // Забезпечуємо час для завершення асинхронної операції

    return 0;
}
```

У цьому прикладі функція `asyncOperation` моделює асинхронну операцію, приймаючи функцію зворотного виклику (callback), яка буде викликана після завершення обчислень. Використання callback дозволяє

головній програмі продовжувати виконання без блокування потоку, що демонструє основний принцип асинхронної комунікації: розділення виконання операцій і очікування результатів. У цьому прикладі демонструється асинхронна операція, яка викликає функцію зворотного виклику (callback) після завершення завдання. Головна функція продовжує виконання під час цієї асинхронної операції, що ілюструє ефективне роз'єднання процесів.

Асинхронні системи потребують ретельно продуманих методів обробки помилок. Незалежно від того, чи використовуються блоки try-catch у потоках завдань, чи застосовуються механізми обробки виключень у futures, надійне керування помилками є критично важливим для забезпечення стабільності системи.

У наступному прикладі демонструється асинхронне виконання обчислювальної задачі, що може призвести до виникнення виняткової ситуації. Функція `riskyComputation` моделює тривалі обчислення та навмисно генерує виключення `std::runtime_error`.

Головна функція використовує `std::async` для запуску задачі в асинхронному режимі та отримує результат через `std::future`. Блок `try-catch` дозволяє надійно перехоплювати будь-які виключення, що виникають у асинхронній задачі, забезпечуючи стійкість програми та контрольоване повідомлення про помилки.

Такий підхід ілюструє важливість правильної обробки помилок у асинхронних системах для підтримки стабільності та передбачуваності виконання.

```
#include <iostream>
#include <future>
#include <thread>
#include <stdexcept>

// Функція, яка моделює ризиковані обчислення та генерує виняток
int ризиковіОбчислення() {
    std::this_thread::sleep_for(std::chrono::seconds(3));
    throw std::runtime_error("В обчисленнях сталася помилка!");
}

int main() {
    // Запуск асинхронної задачі
    std::future<int> результат = std::async(std::launch::async, ризиковіОбчислення);

    try {
        // Отримання результату обчислень (може кинути виняток)
        int значення = результат.get();
```

```
        std::cout << "Результат обчислень: " << значення << std::endl;
    } catch (const std::exception &e) {
        // Обробка винятку для забезпечення стійкості програми
        std::cerr << "Перехоплено виняток: " << e.what() << std::endl;
    }

    return 0;
}
```

У цьому коді `std::async` використовується для керування ризикованими обчисленнями, де передбачено можливі винятки, які перехоплюються під час отримання результату з `future`, що запобігає непередбаченому завершенню роботи програми. Реалізація асинхронного спілкування в C++ виходить за межі простого багатопотокового програмування. Вона охоплює повні асинхронні парадигми – від паралелізму завдань до мережевого введення-виведення та подієво-орієнтованих моделей.

C++ надає потужний набір інструментів для створення неблокувальних застосунків, забезпечуючи високу швидкодію та пропускну здатність. Коректне проєктування застосунків з дотриманням асинхронних стратегій, у поєднанні з розумним використанням засобів C++ і бібліотек, таких як `std::async` та Boost.Asio, дозволяє створювати системи, які одночасно масштабовані та ефективні. Розуміння тонкощів, потенційних проблем і кращих практик асинхронного програмування дає змогу розробникам створювати високопродуктивні та складні застосунки, пристосовані до вимог сучасної програмної інфраструктури.

2.6 Оцінювання продуктивності синхронних та асинхронних моделей оброблення даних

Рішення щодо використання синхронних або асинхронних моделей оброблення даних в програмних системах значною мірою залежить від конкретних випадків використання та цільових показників продуктивності. Характеристики кожної моделі роблять її більш придатною для певних застосунків залежно від умов мережі, доступності ресурсів, вимог до масштабованості та очікувань користувачів. Вкрай важливо провести детальний аналіз цих аспектів для створення систем, здатних відповідати як експлуатаційним, так і продуктивнісним стандартам.

Приклади реального застосування синхронного оброблення даних

1. Системи обробки транзакцій: у фінансових сервісах обробка транзакцій потребує високої цілісності та послідовної узгодженості. Системи, такі як онлайн-банкінг, торгові платформи на фондовому ринку та платіжні шлюзи, часто використовують синхронне спілкування, щоб гарантувати виконання операцій, наприклад переказу коштів або

розміщення замовлень, у чітко визначеному порядку, уникаючи аномалій, таких як подвійні витрати.

```
#include <iostream>
#include <mutex>

// Початковий баланс рахунку
int accountBalance = 1000;

// М'ютекс для синхронізації доступу до рахунку
std::mutex balanceMutex;

// Функція виконання транзакції
void performTransaction(int withdrawalAmount) {
   // Блокування м'ютекса для забезпечення безпечного доступу
   std::lock_guard<std::mutex> lock(balanceMutex);

   if (accountBalance >= withdrawalAmount) {
      accountBalance -= withdrawalAmount;
      std::cout << "Транзакція успішна, новий баланс: " << accountBalance << std::endl;
   } else {
      std::cout << "Недостатньо коштів для транзакції." << std::endl;
   }
}

int main() {
   performTransaction(200);
   performTransaction(500);
   return 0;
}
```

У цьому прикладі демонструється використання м’ютекса (`std::mutex`) для синхронізації доступу до спільного ресурсу – балансу рахунку. Кожна транзакція виконується в межах блока `std::lock_guard`, що гарантує атомарність операцій і запобігає умовам гонки. Такий підхід дозволяє безпечно виконувати зміни стану спільних даних у багатопотоковому середовищі, що є критично важливим у фінансових та інших системах, де цілісність даних має бути гарантована.

У наведеному вище коді м’ютекси забезпечують синхронізований доступ до спільних ресурсів, гарантують цілісність транзакцій та запобігають умовам гонки.

Випадки використання синхронної комунікації:

1. Системи керування та контролю: У військових або критично важливих інфраструктурних застосунках операції залежать від точного виконання команд у строго визначеному порядку. Такі системи орієнтовані на негайну обробку та підтвердження результатів перед виконанням наступних команд, підкреслюючи надійність і передбачуваність виконання.

2. Колаборативне програмне забезпечення та інтерактивні за стосунки. Програми з функціями реального часу, наприклад, багатокористувацькі ігри або віртуальні зустрічі, часто використовують синхронну комунікацію для підтримки узгодженого стану між усіма учасниками, що дозволяє зменшити затримки у відповіді на дії користувачів.

Випадки використання асинхронної комунікації:

1. Веб- та мережеві сервери. Сучасні вебсервери, такі як Nginx або Node.js, застосовують асинхронну комунікацію для оброблення великої кількості клієнтських з'єднань одночасно, оптимізуючи продуктивність і використання ресурсів без необхідності застосування багатопотоковості.

У наступному прикладі демонструється створення простого асинхронного вебсервера на Node.js. Сервер обробляє HTTP-запити від клієнтів без блокування основного потоку виконання.

```
const http = require('http');

// Створення сервера та визначення обробника запитів
const server = http.createServer((req, res) => {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('Hello, World!\n');
});

// Запуск сервера на localhost:3000
server.listen(3000, '127.0.0.1');
console.log('Server running at http://127.0.0.1:3000/');
```

Пояснення:

1. `http.createServer()` створює асинхронний сервер, який приймає функцію-обробник запитів.

2. Сервер відповідає на кожен HTTP-запит рядком `"Hello, World!"`.

3. `server.listen()` запускає сервер на вказаному IP та порту.

4. Завдяки асинхронній природі Node.js, сервер може одночасно обробляти багато підключень без створення окремих потоків для кожного клієнта.

Цей приклад ілюструє ефективність асинхронної моделі у вебсерверах, що дозволяє підтримувати масштабовані та продуктивні мережеві сервіси.

1. Неблокувальна модель: дозволяє серверу продовжувати прийом нових підключень під час обробки активних запитів, забезпечуючи безперервну роботу системи без очікування завершення поточних операцій.

2. Фонове виконання та обробка задач: застосунки, такі як поштові сервери, вебсканери або канали обробки даних (data pipelines), часто передають ресурсоємні обчислення або операції, що залежать від введення-виведення, асинхронним процесам. Це дозволяє підтримувати високий рівень відгуку системи без очікування завершення задач.

3. Мікросервіси: асинхронна комунікація широко використовується в архітектурі мікросервісів, де сервіси слабо пов'язані між собою, а обмін даними часто здійснюється через брокери повідомлень або подієво-орієнтовані моделі. Це підвищує стійкість до відмов та масштабованість, оскільки сервіси можуть працювати незалежно та асинхронно обробляти збої або затримки.

Показники продуктивності:

1. Затримка та пропускна здатність у *синхронній комунікації*: затримка в синхронних системах визначається часом кругового проходження (round-trip time) для завершення кожної операції. Послідовний процес виконання часто накопичує затримки, що негативно впливає на пропускну здатність та загальну відзивність системи у разі непередбачених затримок у мережі.

```
Processing Order:
[Request 1] -> [Response 1]
[Request 2] -> [Response 2]
(increased latency with dependency on
prior completion)
```

Асинхронна комунікація. Асинхронні системи демонструють високу ефективність у середовищах із варіабельною затримкою, оскільки завдання можуть оброблятися паралельно без необхідності очікування завершення попередніх операцій. Така архітектура забезпечує підвищену пропускну здатність та ефективне використання ресурсів, оскільки завдання виконуються незалежно одне від одного.

```
Processing Order:
[Request 1] -------\
[Request 2] ----> (processes complete
independently)
```

2. Використання ресурсів. *Синхронні системи*. Блокувальний характер синхронної комунікації часто призводить до неефективного використання ресурсів, оскільки активні ресурси утримуються протягом тривалого часу в очікуванні завершення операцій.

Асинхронні системи. Ресурси використовуються значно ефективніше завдяки неблокувальним операціям. Системи можуть обслуговувати кілька

запитів одночасно без фіксованих затримок на очікування результатів, що оптимізує використання процесора та I/O ресурсів.

3. Складність і підтримка. *Синхронна комунікація*. Забезпечує просту реалізацію з лінійним потоком керування, що спрощує обслуговування та зменшує потенційні точки відмови. Відсутність складності, пов'язаної з паралельністю, полегшує налагодження та усунення проблем.

Асинхронна комунікація. Потребує більш складної реалізації через паралельність та потенційні умови гонки. Проєктування неблокувальних архітектур додає складності, але підвищує адаптивність системи до різноманітних критеріїв продуктивності.

Конструктивні міркування

1. Вибір відповідної моделі. Рішення щодо вибору між синхронною та асинхронною моделлю має враховувати вимоги додатка, такі як цілісність даних, очікування щодо часу відповіді та обсяг обробки. Додатки, що потребують взаємодії в реальному часі та негайної відповіді, можуть віддавати перевагу синхронним моделям, тоді як ті, що потребують масштабованості та високої пропускної здатності, віддадуть перевагу асинхронним.

2. Обробка помилок і надійність. Асинхронні системи потребують надійних механізмів обробки помилок, включно повторні спроби, логування повідомлень та компенсаційні транзакції для обробки неповних або неуспішних операцій.

3. Масштабованість системи. Асинхронні системи мають природну масштабованість завдяки здатності самостійно обробляти завдання без потреби у послідовних залежностях.

4. Складність розробки. Складність управління паралельними операціями в асинхронних системах має оцінюватися з урахуванням переваг підвищеної продуктивності та швидкодії.

Висновок щодо випадків використання та продуктивності. Вибір між синхронними та асинхронними моделями комунікації визначається детальним аналізом конкретних сценаріїв застосування та критеріїв продуктивності.

Розуміння унікальних характеристик кожного підходу дозволяє ефективно проєктувати системи, узгоджуючи їх із цілями додатка та оптимізуючи за критичними показниками. Синхронна комунікація гарантує надійність та впорядкованість, підходячи для сценаріїв, де критичною є негайна реакція та детермінованість. Натомість асинхронна комунікація відзначається масштабованістю та оптимізацією продуктивності, відповідаючи на потреби середовищ з високою пропускною здатністю та паралельністю.

Для розробників і архітекторів балансування складності та очікуваних результатів відкриває шлях до проєктування систем, придатних для сучасних різноманітних обчислювальних середовищ.

2.7 Контрольні питання

1. Що таке синхронна комунікація в мережевих системах і які її ключові характеристики? Наведіть приклади сценаріїв, де застосування синхронного підходу є доцільним.
2. Поясніть блокувальний характер синхронної взаємодії. Як він впливає на виконання програми та використання ресурсів у мережевому додатку?
3. У чому полягають основні відмінності між синхронною та асинхронною комунікацією щодо послідовності операцій, передбачуваності часу виконання та масштабованості системи?
4. Опишіть приклад реалізації синхронного HTTP-запиту в C++ із використанням бібліотеки libcurl. Які особливості синхронної взаємодії демонструє цей приклад?
5. Які фактори потрібно враховувати під час вибору синхронного підходу в мережевих системах, зокрема щодо пропускної здатності, стійкості до відмов та чутливості до мережевих затримок?
6. Що таке асинхронна комунікація та чим вона відрізняється від синхронної у контексті мережевих систем?
7. Які основні характеристики асинхронної комунікації, і як вони впливають на ефективність використання ресурсів та масштабованість системи?
8. Поясніть роль std::async, std::future та std::promise у реалізації асинхронних операцій на прикладі C++. Як ці конструкції забезпечують незалежне виконання завдань?
9. Які труднощі та виклики пов’язані з розробкою асинхронних систем, зокрема щодо конкурентності, цілісності даних та обробки помилок?
10. Назвіть практичні приклади застосування асинхронної комунікації у сучасних програмних системах і поясніть, чому асинхронність підвищує продуктивність у цих сценаріях.
11. Які ключові відмінності між синхронним і асинхронним передаванням даних щодо блокування операцій та порядку їх виконання?
12. Як впливає вибір синхронного або асинхронного підходу на продуктивність та пропускну здатність мережевих систем?
13. Які переваги та обмеження синхронної комунікації у контексті масштабованості, налагодження та безпеки?
14. Поясніть, чому асинхронна комунікація потребує складніших стратегій управління ресурсами та відстеження стану виконання завдань.
15. Наведіть приклади сценаріїв застосування синхронного та асинхронного передавання у сучасних системах і обґрунтуйте вибір підходу залежно від конкретних вимог до продуктивності, надійності та масштабованості.
16. Які особливості синхронного зв’язку в C++ визначають його блокувальний характер і послідовне виконання операцій?

17. Як реалізується синхронне TCP-з’єднання у C++ за допомогою сокетів і які кроки містить стандартна процедура підключення та обміну даними?

18. Поясніть, як синхронні операції введення-виведення з файлами (fstream) забезпечують послідовне виконання та контроль цілісності даних.

19. Яким чином іменовані канали (FIFO) використовуються для синхронної міжпроцесної взаємодії в C++, і як забезпечується блокувальний обмін даними між батьківським та дочірнім процесами?

20. Опишіть роль примітивів синхронізації std::mutex та std::condition_variable у багатопотокових програмах C++ для забезпечення впорядкованого та безпечного виконання синхронних операцій.

21. Які можливості стандарту C++11 забезпечують нативну підтримку асинхронного програмування, і яку роль у цьому відіграють потоки (std::thread)?

22. Поясніть принцип роботи механізмів std::promise та std::future для організації асинхронних обчислень і безпечного отримання результатів у головному потоці.

23. Як функція std::async допомагає реалізувати асинхронне виконання задач у C++, і які переваги вона дає порівняно з ручним керуванням потоками?

24. Яким чином бібліотека Boost.Asio підтримує асинхронну роботу з мережею та неблокувальне введення-виведення, і яку роль відіграють функції зворотного виклику (callback) у такій моделі?

25. Чому обробка винятків є критичною у асинхронних задачах, і як try-catch у поєднанні з std::future дозволяє безпечно керувати помилками під час асинхронного виконання?

26. Які основні фактори впливають на вибір між синхронною та асинхронною моделлю оброблення даних у програмних системах?

27. Поясніть, як блокувальний характер синхронної комунікації впливає на використання ресурсів та затримку виконання операцій у системах з високою завантаженістю.

28. У чому полягає перевага асинхронної обробки даних у вебсерверах та системах з великою кількістю одночасних підключень, і як це підвищує пропускну здатність?

29. Які конструктивні міркування потрібно враховувати під час проєктування асинхронних систем, зокрема щодо обробки помилок, масштабованості та складності розробки?

30. Наведіть приклади сценаріїв, де синхронна модель є більш доцільною, а де асинхронна модель забезпечує кращу продуктивність та ефективність використання ресурсів.

3 БАГАТОПОТОКОВІСТЬ У МЕРЕЖЕВИХ ДОДАТКАХ

Цей розділ присвячений використанню багатопотоковості для підвищення продуктивності та ефективності мережевих додатків. Розпочинається з пояснення значущості багатопотоковості та того, як вона дозволяє одночасно обробляти кілька завдань. Читачі навчаться створювати та керувати потоками у C++, зосереджуючись на техніках синхронізації для запобігання умов гонки. Розділ також охоплює управління спільними даними та оптимізацію мережевого введення-виведення через багатопотоковість. Обговорюються типові проблеми, такі як взаємні блокування (deadlocks), а також стратегії їх уникнення, що дозволяє розробникам створювати надійне та високопродуктивне мережеве програмне забезпечення.

3.1 Поняття багатопотоковості

Багатопотоковість є фундаментальною концепцією сучасних обчислень, що дозволяє виконувати кілька потоків у межах одного процесу. Кожен потік використовує спільні ресурси процесу, включно з пам’яттю та відкритими файлами, що сприяє ефективній комунікації та обробці завдань. Важливість багатопотоковості полягає у здатності підвищувати продуктивність додатків за рахунок паралельного виконання завдань. Це особливо критично для мережевих додатків, де часто необхідно одночасно виконувати кілька операцій, таких як надсилання та отримання даних по мережі, обробка запитів та взаємодія з базами даних.

Головна перевага багатопотоковості полягає у підвищенні швидкодії та відзивності додатків. У сценаріях, де завдання є процесорно-залежними (CPU-bound), потоки можуть використовувати кілька ядер процесора для паралельної обробки, значно скорочуючи час виконання обчислювально-інтенсивних операцій. У ситуаціях, де обробка обмежена введенням-виведенням (I/O-bound), наприклад у мережевій комунікації, багатопотоковість дозволяє потокам, що очікують на мережеву відповідь, передавати процесор іншим завданням, підвищуючи відзивність додатка та ефективність використання ресурсів.

Щоб оцінити ефективність багатопотоковості, спершу необхідно зрозуміти поняття потоку. Потік можна розглядати як найменшу одиницю обробки, яку операційна система може планувати. Він зазвичай містить унікальний ідентифікатор потоку (thread ID), лічильник команд, набір регістрів та стек. Потоки в межах одного процесу використовують спільні ресурси, проте кожен виконує власну незалежну послідовність інструкцій.

Реалізація багатопотоковості відрізняється залежно від операційної системи, але основні принципи залишаються однаковими. У системах типу Unix бібліотека pthreads (POSIX Threads) забезпечує надійну основу для багатопотокових додатків, тоді як у Windows використовується власний API потоків. Мова C++ надає стандартизовану бібліотеку для створення та

керування потоками, що спрощує написання переносимих та ефективних багатопотокових програм.

У наведеному фрагменті коду продемонстровано базовий приклад створення та використання потоку виконання в мові програмування C++ із застосуванням стандартної бібліотеки <thread>.

Оголошується функція `threadFunction`, яка визначає завдання, що виконуватиметься в окремому потоці. У тілі функції здійснюється виведення повідомлення в стандартний потік виведення, що ілюструє виконання коду в контексті додаткового потоку.

У функції `main` створюється об'єкт класу `std::thread`, якому передається вказівник на функцію потоку. Це ініціює запуск нового потоку виконання паралельно з основним потоком програми. Виклик методу `join()` забезпечує синхронізацію, змушуючи основний потік очікувати завершення виконання створеного потоку перед завершенням програми.

Таким чином, цей приклад ілюструє мінімальну модель багатопотокового виконання в C++, демонструючи створення потоку, його запуск та коректне завершення з використанням механізмів синхронізації.

```
#include <iostream>
#include <thread>

// Функція, яка буде виконуватися в окремому потоці
void threadFunction() {
    std::cout << "Привіт із потоку виконання!" << std::endl;
}

int main() {
    // Створення потоку та його запуск
    std::thread t(threadFunction);

    // Очікування завершення виконання потоку
    t.join();

    return 0;
}
```

У наведеному вище коді визначено простий потік виконання мовою C++. Об'єкт `std::thread` з іменем `t` ініціалізується функцією `threadFunction`, яку він виконує паралельно з основним потоком програми. Метод `t.join()` використовується для примусового очікування головним потоком завершення виконання потоку `t`, що забезпечує синхронізацію всіх потоків перед завершенням роботи програми.

Основною метою багатопотоковості є підвищення продуктивності застосунків за рахунок паралельного виконання завдань. Водночас ефективне використання багатопотокових технологій потребує ретельного керування спільними ресурсами з метою уникнення таких проблем, як умови гонки, взаємні блокування та конкуренція за ресурси. Умова гонки

виникає тоді, коли кілька потоків одночасно змінюють спільні ресурси, що призводить до непередбачуваної та помилкової поведінки програми. У подібних ситуаціях необхідно застосовувати механізми синхронізації потоків, які гарантують, що доступ до спільного ресурсу в певний момент часу має лише один потік. Наведений фрагмент коду демонструє приклад умови гонки у багатопотоковій програмі мовою C++.

У програмі оголошено глобальну змінну `counter`, яка спільно використовується всіма потоками. Функція `incrementCounter()` у циклі збільшує значення цієї змінної на одиницю 1000 разів. У функції `main()` створюється вектор потоків, після чого ініціюється 10 потоків, кожен з яких виконує функцію `incrementCounter`. Після запуску всіх потоків головний потік очікує їх завершення за допомогою методу `join()`, а далі виводить фінальне значення лічильника.

Попри те, що теоретично очікуване значення змінної `counter` має дорівнювати 10 000, на практиці результат часто є меншим. Це пояснюється тим, що операція інкременту (`++counter`) не є атомарною та складається з кількох машинних інструкцій (читання значення, його збільшення та запис назад у пам’ять). За відсутності механізмів синхронізації кілька потоків можуть одночасно змінювати `counter`, перезаписуючи результати один одного.

Такий ефект ілюструє класичну умову гонки, яка призводить до некоректної та непередбачуваної поведінки програми. Для усунення цієї проблеми необхідно застосовувати засоби синхронізації потоків, зокрема м’ютекси, атомарні змінні або інші механізми керування доступом до спільних ресурсів.

```
#include <iostream>
#include <thread>
#include <vector>

int counter = 0;

// Функція інкрементування спільної змінної лічильника
void incrementCounter() {
    for (int i = 0; i < 1000; ++i) {
        ++counter;
    }
}

int main() {
    std::vector<std::thread> threads;

    // Створення 10 потоків, кожен з яких збільшує значення лічильника
    for (int i = 0; i < 10; ++i) {
        threads.push_back(std::thread(incrementCounter));
    }
```

```
    // Очікування завершення роботи всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Кінцеве значення лічильника: " << counter << std::endl;
    return 0;
}
```

Очікуваним результатом виконання наведеного коду є досягнення змінною-лічильником значення 10 000. Однак унаслідок виникнення умови гонки (race condition) фактичне значення може виявитися меншим. Такий приклад наочно демонструє, що несинхронізований доступ кількох потоків до спільних змінних призводить до некоректних результатів обчислень. Для усунення цієї проблеми необхідно застосовувати механізми синхронізації потоків, що детально розглядаються в наступних розділах.

Стандартним підходом до синхронізації потоків є використання м’ютексів та блокувань, які забезпечують атомарний доступ до спільних ресурсів. Стандартна бібліотека C++ надає відповідні засоби, зокрема `std::mutex` та `std::lock_guard`, для реалізації потокобезпечних операцій. Застосування цих механізмів унеможливлює одночасний вхід кількох потоків до критичних ділянок коду, що працюють зі спільними даними, та гарантує коректність виконання програми.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

int counter = 0;
std::mutex counterMutex;

void incrementCounter() {
    for (int i = 0; i < 1000; ++i) {
        // Блокуємо м’ютекс перед модифікацією спільного лічильника
        std::lock_guard<std::mutex> guard(counterMutex);
        ++counter;
    }
}

int main() {
    std::vector<std::thread> threads;

    // Створюємо 10 потоків, які інкрементують лічильник
    for (int i = 0; i < 10; ++i) {
        threads.push_back(std::thread(incrementCounter));
    }
```

```
    // Очікуємо завершення роботи всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Кінцеве значення лічильника: " << counter << std::endl;
    return 0;
}
```

Цей приклад демонструє використання `std::mutex` та `std::lock_guard` для забезпечення потокобезпечного доступу до спільної змінної, що усуває умову гонки та гарантує коректний результат виконання програми.

Наведений модифікований приклад використовує `std::mutex` для блокування доступу до змінної-лічильника, забезпечуючи, що лише один потік може виконувати її інкрементування в кожний момент часу. Застосування `std::lock_guard` гарантує автоматичне звільнення м'ютекса одразу після виходу об'єкта-охоронця за межі області видимості, що сприяє збереженню коректності програми та зменшує ризик виникнення взаємних блокувань (deadlock).

Потенціал багатопотоковості виходить за межі локальних обчислень і поширюється на мережеві застосування, де ефективне керування множинними з'єднаннями та потоками даних є критично важливим. Потоки можуть використовуватися для обробки окремих мережевих завдань, таких як приймання запитів або надсилання відповідей, що сприяє створенню масштабованих і високочутливих застосувань, здатних відповідати вимогам середовищ з високою пропускною здатністю.

У контексті мережевого сервера багатопотокова модель, наприклад, дозволяє виділяти окремий потік для обслуговування кожного клієнтського з'єднання. Такий підхід запобігає перетворенню сервера на вузьке місце з погляду продуктивності та забезпечує обробку клієнтських запитів без надмірних затримок, що в підсумку покращує користувацький досвід. Практичним прикладом є багатопотоковий TCP-сервер, у якому кожне клієнтське з'єднання обслуговується окремим потоком.

```
#include <iostream>
#include <thread>
#include <vector>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

// Функція обслуговування клієнтського сеансу, що виконується в окремому потоці
void clientSession(tcp::socket socket) {
    try {
        while (true) {
```

```
            char data[512];
            boost::system::error_code error;

            size_t length = socket.read_some(boost::asio::buffer(data), error);

            if (error == boost::asio::error::eof) {
                break; // З'єднання коректно закрито віддаленою стороною
            } else if (error) {
                throw boost::system::system_error(error); // Інша помилка введення-виведення
            }

            // Надсилання отриманих даних назад клієнту (ехо-відповідь)
            boost::asio::write(socket, boost::asio::buffer(data, length));
        }
    } catch (std::exception& e) {
        std::cerr << "Виняток у потоці: " << e.what() << "\n";
    }
}

int main() {
    try {
        boost::asio::io_context io_context;

        // Створення об'єкта acceptor для прослуховування TCP-з'єднань на порту 1234
        tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4(), 1234));

        while (true) {
            tcp::socket socket(io_context);

            // Очікування та приймання вхідного клієнтського з'єднання
            acceptor.accept(socket);

            // Запуск окремого потоку для обслуговування клієнта
            std::thread(clientSession, std::move(socket)).detach();
        }
    } catch (std::exception& e) {
        std::cerr << "Виняток: " << e.what() << "\n";
    }

    return 0;
}
```

У цьому прикладі бібліотека boost::asio використовується для побудови TCP-сервера, який асинхронно приймає мережеві з'єднання. Кожне нове клієнтське підключення ініціює створення окремого потоку

виконання, у межах якого запускається функція clientSession, що здійснює повернення (ехо-передавання) отриманих від клієнта даних. Відокремлення потоків (detaching) забезпечує автоматичне вивільнення ресурсів після завершення взаємодії з клієнтом.

Коректне застосування багатопотоковості потребує досягнення балансу між підвищенням продуктивності та зростанням складності програмної системи. Накладні витрати, пов’язані з перемиканням контекстів, затримками синхронізації та можливими сценаріями взаємного блокування, у разі нераціонального проєктування можуть переважити очікувані переваги. Для повноцінного використання потенціалу багатопотокового виконання необхідним є також розуміння політик планування потоків операційною системою та особливостей апаратної архітектури. Крім того, налагодження багатопотокових застосунків супроводжується специфічними труднощами, зумовленими недетермінованістю їх виконання та складністю паралельних процесів. Інструменти на кшталт Valgrind, gdb і спеціалізованих аналізаторів потоків є незамінними для виявлення та усунення проблем, пов’язаних із конкурентним доступом до ресурсів.

Перехід до використання багатопотоковості має ґрунтуватися на чітко сформульованій меті підвищення ефективності оброблення даних, а не здійснюватися заради самого факту паралелізації. Найбільшу вигоду отримують застосунки зі значними обчислювальними навантаженнями або інтенсивними операціями введення-виведення. В усіх випадках зважений підхід до спільного використання даних і раціональне застосування механізмів блокування дають змогу мінімізувати конкуренцію за ресурси та повною мірою реалізувати потенціал багатопотоковості в мережевих застосунках.

3.2 Створення потоків у C++

Створення потоків у мові програмування C++ здійснюється з використанням засобів багатопотокового програмування стандартної бібліотеки, упроваджених, починаючи з версії C++11. Ці засоби забезпечують можливість розроблення конкурентних програм у більш простий, структурований та уніфікований спосіб. Підтримка потоків у C++ реалізується через заголовковий файл `<thread>`, який надає платформонезалежний інтерфейс для ініціювання та керування потоками виконання.

Потік у C++ створюється на основі класу std::thread, що є окремим потіком виконання програми. Кожен об’єкт std::thread розпочинає виконання викличного об’єкта, яким може бути функція, лямбда-вираз або функціональний об’єкт, отриманий за допомогою шаблону std::function. Використання класу std::thread абстрагує програміста від складнощів платформозалежних API багатопотоковості та забезпечує цілісний і уніфікований підхід до реалізації паралельних обчислень.

```
#include <iostream>
```

```
#include <thread>

// Визначення простої функції, яку виконуватиме потік
void simpleFunction() {
    std::cout << "Потік виконує simpleFunction." << std::endl;
}

int main() {
    // Створення потоку для виконання simpleFunction
    std::thread t1(simpleFunction);

    // Очікування завершення потоку
    t1.join();

    std::cout << "Головний потік завершив роботу." << std::endl;
    return 0;
}
```

У цьому прикладі створюється простий потік у C++. Потоковий об’єкт t1 ініціалізується функцією simpleFunction, яка виконується паралельно з головним потоком програми. Виклик t1.join() змушує головний потік очікувати завершення виконання потоку t1, що забезпечує синхронізацію потоків перед завершенням програми. Це демонструє базовий механізм створення та керування потоками у C++.

У цьому прикладі функція simpleFunction виконується у окремому потоці. Потоковий об’єкт std::thread t1 ініціалізується з simpleFunction, починаючи її виконання паралельно з головною програмою. Метод t1.join() гарантує, що головний потік очікує завершення потоку t1, що є критично важливим для синхронізації виконання потоків та запобігання передчасному завершенню головного потоку.

Управління життєвим циклом потоків є ключовим аспектом багатопотоковості в C++. Потоки можуть бути або приєднані (joined), або відокремлені (detached). Якщо викликається метод join, потік, що викликає його, очікує завершення виконання відповідного потоку.

Натомість відокремлений потік (detached) працює автономно, звільняючи свої ресурси після завершення виконання без необхідності виклику join. Відокремлення потоків доцільне, коли час виконання потоку незалежний і не впливає на основний хід виконання програми.

```
#include <iostream>
#include <thread>
#include <chrono>

// Функція, що виконується незалежно
void independentFunction() {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Незалежна функція виконана." << std::endl;
```

```
}

int main() {
    // Створення потоку та його відокремлення
    std::thread t(independentFunction);
    t.detach(); // Відокремлення потоку

    std::cout << "Потік відокремлено, головна програма продовжує виконання." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Чекати трохи для демонстрації
    return 0;
}
```

У цьому прикладі функція independentFunction виконується у відокремленому потоці. Метод detach() дозволяє потоку працювати автономно, після чого ресурси потоку звільняються автоматично після завершення його виконання. Головний потік продовжує своє виконання незалежно від завершення відокремленого потоку, демонструючи асинхронну роботу та підхід до управління життєвим циклом потоків у C++.

Відокремлення потоку, як показано в прикладі, дозволяє головному потоку продовжувати виконання без очікування завершення незалежної функції. Таким чином, головна програма може завершити виконання та вийти раніше за відокремлений потік, що демонструє його асинхронний характер.

Крім того, надзвичайно важливо правильно управляти власністю потоків. Об’єкт std::thread володіє конкретним потоком виконання. Якщо об’єкт std::thread знищується без виклику join() або detach(), програма викликає std::terminate, що може призвести до невизначеної поведінки. Тому необхідно гарантувати правильне управління усіма потоками до того, як їх об’єкти std::thread вийдуть з області видимості.

Гнучкість конструктора std::thread дозволяє визначати початкову функцію різними способами. Потік може виконувати вільну функцію, метод класу, лямбда-функцію або функтор. Далі розглянуто кожен варіант із прикладами синтаксису та застосування.

Потоки з лямбдами

Лямбда-функції, введені у C++11, надають зручний спосіб визначення функцій на місці (inline). Вони особливо корисні у багатопотоковості для визначення завдань потоків без необхідності створювати додаткові іменовані функції.

```
#include <iostream>
#include <thread>

int main() {
    // Лямбда-функція, яку виконує потік
```

```
    std::thread t([] {
        std::cout << "Потік виконує лямбда-функцію." << std::endl;
    });

    // Очікування завершення потоку
    t.join();

    std::cout << "Головний потік завершив роботу." << std::endl;
    return 0;
}
```

У цьому прикладі лямбда-функція виконується у окремому потоці. Виклик `t.join()` гарантує, що головний потік чекатиме завершення роботи цього потоку, забезпечуючи синхронізацію перед завершенням програми.

Цей приклад демонструє, як лямбда-функцію можна безпосередньо передати в конструктор `std::thread`, забезпечуючи компактний синтаксис для визначення поведінки потоку.

Потоки з методами класу та функторами

Потоки C++ також можуть виконувати методи класу та функтори. Під час виклику методу класу необхідно передати відповідний об’єкт – як вказівник або посилання, а метод, який потрібно виконати, має бути доступним для виклику (публічним).

```
#include <iostream>
#include <thread>

class Worker {
public:
    void doWork() {
        std::cout << "Потік виконує метод doWork класу Worker." << std::endl;
    }
};

int main() {
    Worker w;

    // Створення потоку, який виконує метод класу
    std::thread t(&Worker::doWork, &w);

    t.join();

    std::cout << "Головний потік завершив виконання." << std::endl;
    return 0;
}
```

Цей приклад демонструє, як потік у C++ може виконувати метод класу. Метод `doWork` об'єкта `w` передається в конструктор `std::thread` разом із вказівником на об'єкт, що дозволяє потоку працювати з конкретним екземпляром класу. Виклик `t.join()` забезпечує синхронізацію та очікування завершення потоку перед завершенням головного потоку.

У цьому випадку метод-член `doWork` класу `Worker` виконується в новому потоці, водночас екземпляр об'єкта `w` передається в конструктор потоку.

Як альтернативу, у потоці можна використовувати функтори – об'єкти, які поводяться як функції – таким чином:

- у цьому прикладі функтор `Functor`, який реалізує оператор виклику `operator()`, виконується в окремому потоці. Потік створюється з об'єктом функтора `f`, а метод `t.join()` забезпечує очікування завершення виконання потоку перед завершенням головного потоку.

```
#include <iostream>
#include <thread>

// Клас-функтор, що реалізує оператор виклику
class Functor {
public:
    void operator()() const {
        std::cout << "Потік виконує функтор." << std::endl;
    }
};

int main() {
    Functor f;

    // Створення потоку з використанням функтора
    std::thread t(f);

    // Очікування завершення потоку
    t.join();

    std::cout << "Головний потік завершив роботу." << std::endl;
    return 0;
}
```

У цьому прикладі функтор подано класом `Functor` з перевантаженим оператором виклику `operator()`. Коли об'єкт такого класу передається конструктору `std::thread`, він виконується як будь-який інший викликабельний об'єкт без додаткових модифікацій. Передача аргументів у потоки здійснюється через конструктор `std::thread`. Аргументи передаються після зазначення функції і автоматично пересилаються з

використанням семантики `std::forward`, що забезпечує правильну передачу значень за посиланням або копіюванням відповідно до типу аргументу.

```
#include <iostream>
#include <thread>

// Функція, яка приймає параметри
void parameterizedFunction(int x, const std::string& message) {
    std::cout << "Отримано: " << x << ", " << message << std::endl;
}

int main() {
    // Створення та запуск потоку з параметрами
    std::thread t(parameterizedFunction, 42, "Привіт, багатопотоковість!");

    // Очікування завершення потоку
    t.join();

    std::cout << "Головний потік завершив роботу." << std::endl;
    return 0;
}
```

У цьому прикладі функція `parameterizedFunction` виконується у окремому потоці з переданими параметрами: числом `42` та рядком `"Привіт, багатопотоковість!"`. Метод `t.join()` забезпечує синхронізацію, очікуючи завершення потоку перед продовженням виконання головного потоку.

У таких випадках функція `parameterizedFunction` отримує ціле число та рядковий параметр, які обробляються потоком під час його виконання. Оскільки аргументи копіюються у область видимості нового потоку, важливо забезпечити правильну передачу даних відповідно до вимог програми, особливо під час роботи зі складними типами даних або великими обсягами інформації. Наостанок, використання механізмів `std::future` і `std::promise` у C++ дозволяє безпечно організовувати асинхронні операції та передавати значення між потоками. У поєднанні з `std::async` ця парадигма усуває необхідність явного управління потоками, підвищуючи зрозумілість і ефективність коду.

```
#include <iostream>
#include <future>

// Завдання, що виконується асинхронно
int asyncTask() {
    return 42;
}
```

```
int main() {
    // Запуск завдання асинхронно
    std::future<int> result = std::async(std::launch::async, asyncTask);

    std::cout << "Очікування результату..." << std::endl;

    // Отримання результату асинхронного завдання
    int value = result.get();

    std::cout << "Отриманий результат: " << value << std::endl;
    return 0;
}
```

У цьому прикладі функція `asyncTask` виконується асинхронно за допомогою `std::async`. Об'єкт `std::future` використовується для отримання результату завдання після його завершення. Виклик `result.get()` блокує поточний потік до моменту завершення асинхронної операції, забезпечуючи безпечний доступ до обчисленого значення. Такий підхід дозволяє ефективно організовувати асинхронне виконання коду без явного управління потоками.

Виконання завдань в асинхронному режимі за допомогою `std::async` абстрагує процес створення потоків, водночас дозволяючи об'єктам типу `std::future` інкапсулювати потенційні результати обчислень, що спрощує ефективне застосування багатопотоковості в мережевих застосуваннях мовою C++. Створення потоків у C++ відкриває широкі можливості для побудови більш складної архітектури програм і оптимізації їхньої продуктивності. Воно забезпечує використання багатоядерних систем для паралельних обчислень і підвищення швидкодії, що є критично важливим під час розроблення сучасного мережевого програмного забезпечення та сервісів. Як показано, `std::thread` і допоміжні механізми, зокрема лямбда-вирази, `std::promise` та `std::future`, формують надійну основу для створення розвинених багатопотокових застосунків. Водночас багатопотокове програмування потребує ретельного управління ресурсами та синхронізацією, щоб повною мірою реалізувати його потенціал і забезпечити узгоджену роботу потоків без конфліктів та втрат ефективності.

3.3 Методи синхронізації потоків

Синхронізація потоків є одним із ключових аспектів багатопотокового програмування, за якого потоки в межах одного процесу мають доступ до спільних ресурсів без виникнення конфліктів. Синхронізація забезпечує правильну послідовність виконання потоків, запобігаючи умовам змагання, пошкодженню даних і непередбачуваній поведінці програми. У C++ для цього можуть використовуватися різні засоби стандартної бібліотеки, зокрема м'ютекси, блокування, умовні змінні та інші

механізми. Аналіз цих засобів дає змогу зрозуміти їхні конфігурації, сфери застосування та вплив на парадигми паралельного програмування.

Умова змагання (race condition) виникає тоді, коли час або порядок виконання потоків впливає на результат роботи програми. Ця проблема є поширеною у випадках, коли кільком потокам дозволено одночасно змінювати спільні дані без належних механізмів синхронізації. У C++ м'ютекс (`std::mutex`) є основним засобом серіалізації доступу до спільних ресурсів, що дає змогу усунути умови змагання та забезпечити коректну роботу багатопотокових програм.

```
#include <iostream>
#include <thread>
#include <vector>

int sharedCounter = 0;

// Функція, що інкрементує спільний лічильник
void incrementCounter() {
    for (int i = 0; i < 10000; ++i) {
        ++sharedCounter; // Неконтрольований доступ до спільної змінної
    }
}

int main() {
    std::vector<std::thread> threads;

    // Запуск кількох потоків, які збільшують значення лічильника
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(incrementCounter);
    }

    // Очікування завершення всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    // Виведення кінцевого значення спільного лічильника
    std::cout << "Final shared counter: " << sharedCounter << std::endl;

    return 0;
}
```

У наведеному прикладі демонструється робота з глобальною спільною змінною `sharedCounter`, до якої одночасно звертаються декілька потоків. Кожен потік виконує інкрементування лічильника в циклі 10 000 разів. Оскільки доступ до змінної `sharedCounter` не синхронізований, між потоками виникає умова змагання (race condition). Операція

інкрементування (`++sharedCounter`) не є атомарною та фактично складається з кількох машинних інструкцій: зчитування значення, його збільшення та запис назад у пам’ять. Паралельне виконання цих операцій у різних потоках може призвести до втрати частини інкрементів.

Кінцеве значення змінної `sharedCounter` може бути меншим за очікуване значення (100 000), що свідчить про некоректну синхронізацію доступу до спільного ресурсу.

Цей приклад ілюструє необхідність застосування механізмів синхронізації, таких як `std::mutex` або атомарні типи (`std::atomic`), для забезпечення коректної та передбачуваної роботи багатопотокових програм.

У цьому прикладі за відсутності механізмів синхронізації значення змінної `sharedCounter` є невизначеним, що демонструє класичний випадок умови змагання (race condition). Для усунення цієї проблеми можуть бути застосовані такі засоби, як м’ютекси, призначені для синхронізації потоків.

М’ютекс (скорочено від mutual exclusion – взаємне виключення) використовується для захисту критичної ділянки коду, дозволяючи лише одному потоку виконувати захищений фрагмент у певний момент часу. У мові C++ клас `std::mutex` забезпечує базові механізми блокування та розблокування доступу до спільних ресурсів.

Поширеною практикою є використання класу `std::lock_guard`, який автоматично звільняє м’ютекс під час виходу об’єкта з області видимості. Такий підхід підвищує безпеку програми щодо виняткових ситуацій і зменшує ймовірність виникнення помилок, пов’язаних із некоректним керуванням блокуваннями.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

int protectedCounter = 0;          // Захищений лічильник
std::mutex counterMutex;           // М’ютекс для синхронізації доступу

void incrementProtectedCounter() {
   for (int i = 0; i < 10000; ++i) {
      // Блокування м’ютекса перед зміною лічильника
      std::lock_guard<std::mutex> lock(counterMutex);
      ++protectedCounter;
   }
}

int main() {
   std::vector<std::thread> threads;

   // Запуск кількох потоків для інкрементування захищеного лічильника
```

```
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(incrementProtectedCounter);
    }

    // Очікування завершення всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Кінцеве значення захищеного лічильника: "
              << protectedCounter << std::endl;

    return 0;
}
```

У цьому прикладі використання `std::lock_guard` разом із `counterMutex` забезпечує захист кожної операції інкрементування змінної `protectedCounter`, запобігаючи виникненню станів змагання. Об'єкт `lock_guard` автоматично керує станом м'ютекса, блокуючи його під час створення та розблоковуючи під час знищення. Такий підхід зменшує ймовірність помилок, пов'язаних із ручним керуванням операціями блокування та розблокування, і спрощує реалізацію складної багатопотокової логіки.

Клас `std::recursive_mutex` надає можливість одному потоку багаторазово захоплювати м'ютекс без виникнення взаємного блокування, що забезпечує підтримку повторного (реентерабельного) блокування. Це є необхідним у випадках, коли функція викликає саму себе рекурсивно та потребує багаторазового доступу до захищених спільних ресурсів.

У цьому прикладі використовується об'єкт `std::recursive_mutex` для забезпечення безпечного доступу до спільного ресурсу під час рекурсивного виклику функції. Функція `recursiveFunction` багаторазово викликає саму себе, водночас кожен виклик намагається захопити один і той самий м'ютекс.

Застосування `std::lock_guard<std::recursive_mutex>` дає змогу одному потоку повторно блокувати м'ютекс без виникнення взаємного блокування. Це забезпечує коректну роботу рекурсивної функції в багатопотоковому середовищі та унеможливлює ситуацію, коли потік блокує сам себе.

Завдяки автоматичному керуванню життєвим циклом блокування об'єкт `lock_guard` гарантує, що м'ютекс буде своєчасно звільнений після завершення відповідної області видимості. Такий підхід підвищує надійність програми та зменшує ризик помилок синхронізації.

Використання `std::recursive_mutex` доцільне у випадках, коли алгоритм передбачає рекурсивну обробку даних або багаторазовий доступ до спільних ресурсів в межах одного потоку, зберігаючи коректність та стабільність виконання програми.

```
#include <iostream>
#include <thread>
#include <mutex>

std::recursive_mutex recursiveMutex;

// Рекурсивна функція
void recursiveFunction(int depth) {
    if (depth <= 0) return;

    // Блокування рекурсивного м’ютекса
    std::lock_guard<std::recursive_mutex> lock(recursiveMutex);

    std::cout << "Глибина рекурсії: " << depth << std::endl;

    // Рекурсивний виклик функції
    recursiveFunction(depth - 1);
}

int main() {
    // Створення потоку для виконання рекурсивної функції
    std::thread t(recursiveFunction, 5);

    // Очікування завершення потоку
    t.join();

    return 0;
}
```

Тут `std::recursive_mutex` забезпечує безпечне виконання рекурсивних викликів, дозволяючи одному й тому самому потоку багаторазово блокувати м’ютекс, що підтримує програми з рекурсивними критичними секціями. Бібліотечний клас `std::timed_mutex` надає додаткові можливості, зокрема дозволяє потокам виконувати спробу блокування з обмеженням за часом очікування. Це є особливо корисним у системах, де небажаним є необмежене блокування під час очікування доступу до ресурсу.

У наведеному прикладі використовується `std::timed_mutex`, який дозволяє потокам намагатися отримати доступ до спільного ресурсу з обмеженням за часом очікування.

Функція `tryLockWithTimeout` виконує спробу блокування м’ютекса за допомогою методу `try_lock_for`, встановлюючи максимальний час очікування в одну секунду. Якщо м’ютекс успішно заблоковано протягом заданого інтервалу, потік отримує доступ до ресурсу, імітує виконання роботи за допомогою функції `sleep_for`, після чого звільняє м’ютекс викликом `unlock`.

У разі, якщо блокування не вдається протягом певного часу, виводиться повідомлення про досягнення тайм-ауту. Це дозволяє уникнути нескінченного очікування доступу до ресурсу та підвищує надійність і керованість багатопотокових програм.

У функції `main` створюються два потоки, які одночасно намагаються отримати доступ до спільного ресурсу. Завдяки використанню `std::timed_mutex` забезпечується контрольований доступ і запобігається взаємному блокуванню. Таким чином, застосування `std::timed_mutex` є доцільним у системах реального часу та високонавантажених середовищах, де важливо обмежувати час очікування синхронізації та забезпечувати стабільність виконання паралельних процесів.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::timed_mutex timedMutex;

void tryLockWithTimeout() {
    if (timedMutex.try_lock_for(std::chrono::seconds(1))) {
        std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << "Заблоковано та виконується робота з ресурсом." << std::endl;
        timedMutex.unlock();
    } else {
        std::cout << "Не вдалося заблокувати м’ютекс. Час очікування вичерпано." << std::endl;
    }
}

int main() {
    std::thread t1(tryLockWithTimeout);
    std::thread t2(tryLockWithTimeout);

    t1.join();
    t2.join();

    return 0;
}
```

У наведеному прикладі використання `std::timed_mutex` дозволяє кожному потоку намагатися захопити м’ютекс протягом певного часу. У разі невдачі потік продовжує виконання, не очікуючи необмежено, що підвищує реактивність застосунку в окремих сценаріях.

Змінні умов (condition variables) забезпечують механізм комунікації та синхронізації між потоками шляхом сигналізації про зміни стану. Вони

особливо корисні, коли один потік має очікувати, поки інший виконає певну операцію або досягне заданого стану, перш ніж продовжити виконання.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex cvMutex;
std::condition_variable cv;
bool ready = false;

void waitForWork() {
    std::unique_lock<std::mutex> lock(cvMutex);
    cv.wait(lock, [] { return ready; }); // Очікування на сигнал, поки ready не стане true
    std::cout << "Worker thread is proceeding." << std::endl;
}

void setReady() {
    std::lock_guard<std::mutex> lock(cvMutex);
    ready = true;
    std::cout << "Main thread sets ready to true." << std::endl;
    cv.notify_one(); // Сигналізує одному потоку, що можна продовжувати
}

int main() {
    std::thread worker(waitForWork);
    std::thread producer(setReady);

    worker.join();
    producer.join();

    return 0;
}
```

У цьому прикладі використано `std::condition_variable` для синхронізації потоків. Потік `worker` очікує, поки змінна `ready` не набуде значення `true`, блокуючись на `cv.wait()`. Потік `producer` встановлює `ready = true` та викликає `cv.notify_one()`, що сигналізує одному потоку про можливість продовження виконання.

Цей механізм дозволяє ефективно координувати виконання потоків без активного очікування (busy-waiting), підвищуючи продуктивність багатопотокових систем і забезпечуючи коректну послідовність операцій при доступі до спільних ресурсів.

У наведеному прикладі потік, який очікує, поки змінна `ready` набуде значення `true`, використовує `cv.wait()` разом із `std::unique_lock`.

Після того як функція `setReady()` встановлює `ready` у `true` та викликає сповіщення через змінну умови (`condition_variable`), потік, що очікував, виявляє зміну стану та продовжує виконання. Цей підхід забезпечує ефективну синхронізацію, гарантуючи безпечну взаємодію потоків із спільними даними.

Мова C++ також пропонує конкурентні структури даних із вбудованою синхронізацією, такі як `std::atomic`, які керують змінними, що використовуються кількома потоками, забезпечуючи атомарність операцій. Використання таких обгорток усуває потребу в явному блокуванні під час виконання простих операцій, спрощуючи реалізацію багатопотокових програм.

```
#include <iostream>
#include <thread>
#include <atomic>
#include <vector>

std::atomic<int> atomicCounter(0);

void incrementAtomicCounter() {
    for (int i = 0; i < 10000; ++i) {
        ++atomicCounter;
    }
}

int main() {
    std::vector<std::thread> threads;

    // Запуск кількох потоків, які інкрементують атомарний лічильник
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(incrementAtomicCounter);
    }

    // Очікування завершення всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Кінцеве значення атомарного лічильника: " << 
atomicCounter.load() << std::endl;
    return 0;
}
```

У цьому прикладі використовується `std::atomic<int>` для створення атомарного лічильника `atomicCounter`, який безпечно інкрементується

кількома потоками одночасно. Функція `incrementAtomicCounter` збільшує значення лічильника в циклі, а атомарна природа змінної гарантує, що операції інкременту виконуються без умов гонки.

Атомарні змінні (`std::atomic`) усувають потребу у використанні м'ютексів для простих операцій, таких як інкремент або присвоєння, забезпечуючи ефективну багатопотокову обробку та коректну синхронізацію потоків. Клас `std::atomic` також надає метод `.load()` для безпечного читання поточного значення змінної з будь-якого потоку.

Змінна `std::atomic` `atomicCounter` ефективно обробляє одночасні інкременти потоків без додаткових механізмів синхронізації. Атомарні змінні надають операції, такі як `load` і `store`, що спрощують синхронізацію простих типів даних без необхідності зовнішніх блокувань.

Хоча м’ютекси та блокування є фундаментальними для синхронізації потоків, їх потрібно використовувати обачно, щоб уникнути виникнення дедлоків – ситуацій, коли два або більше потоків чекають на звільнення ресурсів без кінця. Кругові очікування, умови «hold and wait» та інші елементи дедлоків мають бути усунуті шляхом застосування тактик запобігання дедлокам, таких як порядок блокування ресурсів, перегляд стратегії виділення ресурсів та методи ідентифікації потоків для розриву циклів залежності.

Синхронізація потоків у C++ спрямована на баланс між захистом ресурсів і ефективністю продуктивності, застосовуючи м’ютекси, блокування та атомарні операції для забезпечення безпечного одночасного доступу до даних. Змінні умови (`condition_variable`) дозволяють ефективно координувати складні операції потоків, необхідні в мережевих додатках і системах великого масштабу. Вміння правильно застосовувати ці техніки дозволяє розробникам створювати надійні, високопродуктивні багатопотокові програми, що використовують оптимальний потенціал паралельного виконання системи.

3.4 Керування спільними даними та ресурсами у багатопотокових додатках

Управління спільними даними та ресурсами у багатопотокових додатках є критично важливим для підтримання узгодженості, забезпечення продуктивності та запобігання конфліктам за ресурси. Коли кілька потоків одночасно отримують доступ до спільних ресурсів, таких як пам’ять, файли чи пристрої, потрібні ретельно структуровані стратегії, щоб уникнути умов гонки, дедлоків і вузьких місць у продуктивності. У цьому розділі розглядаються методики та кращі практики управління спільними даними у C++, з особливим акцентом на механізми блокування, захищені схеми доступу та стратегії спільного використання ресурсів.

М’ютекси є фундаментальними для захисту спільних даних у багатопотокових програмах. Вони запобігають одночасному доступу декількох потоків до критичних секцій коду, де можуть змінюватися спільні дані. Незважаючи на простоту, м’ютекси забезпечують взаємне

виключення, гарантуючи, що після блокування ресурсу одним потоком, інші можуть отримати доступ лише після його розблокування.

Правильне використання м'ютексів передбачає розуміння балансу між деталізацією блокування та конкуренцією потоків. Тонке (fine-grained) блокування обмежує область блокування найменшим можливим набором даних, що зменшує час очікування, але підвищує складність коду. Водночас грубе (coarse-grained) блокування спрощує логіку програми, але може збільшити конкуренцію між потоками, потенційно призводячи до проблем із продуктивністю.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

// Спільна структура даних
std::vector<int> sharedVector;

// М'ютекс для захисту вектора
std::mutex vectorMutex;

// Функція для додавання елемента до вектора
void addToVector(int value) {
    // Блокування м'ютекса для забезпечення безпечного доступу
    std::lock_guard<std::mutex> lock(vectorMutex);
    sharedVector.push_back(value);
    std::cout << "Додано " << value << " до вектора." << std::endl;
}

int main() {
    // Створення потоків для додавання елементів до спільного вектора
    std::thread t1(addToVector, 1);
    std::thread t2(addToVector, 2);

    // Очікування завершення потоків
    t1.join();
    t2.join();

    std::cout << "Розмір вектора: " << sharedVector.size() << std::endl;

    return 0;
}
```

У цьому прикладі `sharedVector` є спільною структурою даних, доступ до якої здійснюють кілька потоків. Використання `std::mutex` (`vectorMutex`) разом із `std::lock_guard` забезпечує синхронізований доступ до вектора. Це гарантує, що одночасний запис із декількох потоків

не призведе до пошкодження даних або умов гонки. `std::lock_guard` автоматично розблокує м’ютекс після завершення блока, що спрощує керування ресурсами та підвищує безпеку виконання багатопотокових операцій.

У цьому прикладі `vectorMutex` гарантує, що одночасно лише один потік може змінювати `sharedVector`. Використання `std::lock_guard` абстрагує ручне керування м’ютексом, автоматично розблоковуючи його у разі виходу об’єкта блокування з області видимості, що підвищує безпеку виконання та коректність програми.

Стандартні м’ютекси забезпечують ексклюзивний доступ, тому вони не підходять для сценаріїв, де переважають операції читання над записуванням. У таких випадках застосовують механізми читання-записування (reader-writer locks), які дозволяють кільком потокам одночасно читати спільні дані, забезпечуючи водночас ексклюзивний доступ для операцій записування. `std::shared_mutex`, введений у C++17, реалізує цей підхід, дозволяючи одночасний доступ для читання, водночас блокуючи потоки, що здійснюють запис.

У наступному прикладі використовується `std::shared_mutex` для керування одночасним доступом до спільного вектора `sharedNumbers`.

Функція `readData()` застосовує `std::shared_lock`, який дозволяє кільком потокам одночасно читати дані без блокування один одного, забезпечуючи ефективність у разі переважання операцій читання.

Функція `writeData(int number)` використовує `std::lock_guard` для отримання унікального блокування, що гарантує ексклюзивний доступ під час записування, запобігаючи конфліктам із читачами або іншими записами.

Таким чином, за допомогою `std::shared_mutex` досягається оптимальна синхронізація: одночасне читання кількома потоками та ексклюзивний доступ для записування, що підвищує продуктивність багатопотокових програм в процесі роботи зі спільними даними.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::vector<int> sharedNumbers; // Спільна структура даних
std::mutex sharedMutex;          // Звичайний м’ютекс для синхронізації доступу

// Функція для читання даних
void readData() {
   std::lock_guard<std::mutex> lock(sharedMutex); // Захищає доступ до даних
   if (!sharedNumbers.empty()) {
      std::cout << "Останнє число: " << sharedNumbers.back() << std::endl;
```

```
    }
}

// Функція для записування даних
void writeData(int number) {
    std::lock_guard<std::mutex> lock(sharedMutex); // Захищає доступ до даних
    sharedNumbers.push_back(number);
    std::cout << "Додано число: " << number << std::endl;
}

int main() {
    std::thread writer1(writeData, 10);
    std::thread reader1(readData);
    std::thread writer2(writeData, 20);
    std::thread reader2(readData);

    writer1.join();
    reader1.join();
    writer2.join();
    reader2.join();

    return 0;
}
```

Заміну `std::mutex` на `std::shared_mutex` доцільно використовувати для підвищення продуктивності, оскільки вона дозволяє одночасну роботу кількох потоків, що здійснюють читання, застосовуючи `std::shared_lock` для операцій читання та `std::lock_guard` для операцій записування. Така стратегія подвійного блокування ефективна у сценаріях із високим співвідношенням читання до записування, оптимізуючи пропускну здатність та зменшуючи затримки доступу до даних.

Для окремих типів даних та операцій атомарні об’єкти, забезпечувані `std::atomic`, гарантують безпечні для потоків одночасні модифікації без потреби у явних блокуваннях. Атомарні операції виконуються на апаратному рівні без блокувань, що робить їх дуже ефективними для часто використовуваних змінних.

```
#include <iostream>
#include <thread>
#include <atomic>
#include <vector>

// Атомарна змінна для забезпечення безпечного доступу з кількох потоків
std::atomic<int> atomicCounter(0);
```

```
// Функція, яка інкрементує атомарний лічильник
void incrementAtomicCounter() {
    for (int i = 0; i < 10000; ++i) {
        ++atomicCounter; // Атомарна операція збільшення
    }
}

int main() {
    std::vector<std::thread> threads;

    // Створюємо 10 потоків, які одночасно інкрементують атомарний лічильник
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(incrementAtomicCounter);
    }

    // Очікуємо завершення всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Кінцеве значення атомарного лічильника: " << atomicCounter << std::endl;

    return 0;
}
```

У цьому прикладі змінна `atomicCounter` є атомарною (`std::atomic<int>`), що дозволяє кільком потокам одночасно її модифікувати без додаткових механізмів синхронізації, таких як mutex. Атомарні операції гарантують, що інкремент виконується без стану гонки, забезпечуючи правильне підсумкове значення. Це особливо ефективно для частих однотипних операцій над простими типами даних.

`std::atomic` гарантує, що кожна операція інкременту виконується атомарно, що робить її придатною для простих одночасних модифікацій примітивних типів даних, таких як лічильники або прапорці. Це усуває необхідність у блокуванні, забезпечує швидке виконання та запобігає потенційним проблемам конкуренції.

Незмінні структури даних (Immutable Data Structures). Незмінні об’єкти, стан яких не може бути змінений після створення, є априорі потокобезпечними. У функціональному програмуванні використання незмінних структур дозволяє повністю уникати накладних витрат на синхронізацію, спричинених зміною стану.

Уникнення глобальних спільних даних. Зменшення області видимості та часу життя спільних ресурсів мінімізує ризик виникнення станів гонки та сприяє більш локалізованим і керованим схемам обміну даними.

Володіння ресурсами (Resource Ownership). Реалізація концепції володіння ресурсами, як у RAII (Resource Acquisition Is Initialization), де управління ресурсами, наприклад блокуванням, обмежене часом життя об'єкта. Розумні вказівники автоматично керують життєвим циклом динамічно виділених ресурсів.

Потокобезпечні бібліотеки та контейнери. Використання бібліотечних колекцій та інструментів із вбудованою підтримкою конкуренції знижує дублювання складної логіки синхронізації. Бібліотеки можуть надавати контейнери з внутрішньою підтримкою одночасних дій, зменшуючи навантаження на розробника з управління синхронізацією.

Вимірювання та профілювання конкуренції. Використання профілювальних інструментів дозволяє виявляти вузькі місця, оцінювати конкуренцію за блокування та визначати затримки через синхронізацію. Оптимізація цих аспектів підвищує загальну швидкодію та чутливість додатків.

Шаблони проєктування для конкуренції

Шаблони, такі як модель виробник-споживач (producer-consumer), fork-join або агентна паралельність допомагають ефективно структурувати багатопотокові додатки. Наприклад, використання черг без блокувань у моделі виробник-споживач ефективно роз'єднує етапи виробництва та споживання даних, зменшуючи час очікування та підвищуючи пропускну здатність.

Дрібнокрокове блокування (Fine-grained Locking). За можливості потрібно віддавати перевагу дрібнокроковому блокуванню або навіть lock-free стратегіям замість грубих блокувань, щоб мінімізувати очікування потоків під час доступу до великих ресурсів.

Складні спільні об'єкти, такі як взаємопов'язані графи даних або ієрархічні ресурси, потребують просунутих стратегій синхронізації. Транзакції баз даних або логічна ізоляція через транзакції можуть гарантувати цілісність операцій над спільними об'єктами без використання ad hoc політик блокування.

Розглядайте випадки, такі як пули ресурсів (пул потоків, пул з'єднань). Стандартних бібліотек може бути недостатньо, що потребує створення спеціалізованого коду синхронізації відповідно до конкретної семантики та обмежень додатка.

У таких сценаріях зазвичай необхідно реалізувати контекстно-залежні політики блокування, забезпечуючи потокобезпечність і мінімальний час очікування ресурсів, враховуючи характеристики продуктивності системи та вимоги до масштабованості.

Багатопотокове програмування в C++ забезпечує значні обчислювальні переваги, але потребує складного управління даними для максимізації цих переваг. Управління спільними даними та ресурсами є центральним для надійної та продуктивної роботи багатопотокових програм на C++.

Систематичне застосування mutex, lock та condition_variable дозволяє запобігати станам гонки, захищати цілісність ресурсів та забезпечувати коректність конкурентних програм. Додатково, використання reader-writer

блокувань, атомарних операцій, незмінних структур, потокобезпечних бібліотек та шаблонів проєктування підвищує надійність багатопотокових програм C++, сприяючи створенню стабільних високопродуктивних мережевих додатків.

3.5 Багатопотоковість і мережеве введення/виведення (I/O)

Багатопотоковість відіграє ключову роль у підвищенні ефективності та чутливості мережевих операцій введення/виведення. Мережеві додатки часто потребують одночасного оброблення численних I/O задач, таких як обробка багатьох запитів клієнтів, керування потоками даних та забезпечення своєчасної відповіді. Використання багатопотоковості разом із мережевим введенням/виведенням дозволяє додаткам раціонально використовувати ресурси системи, підвищуючи пропускну здатність, зменшуючи затримки та покращуючи загальну продуктивність.

Сучасні мережеві додатки часто взаємодіють із кількома клієнтами або сервісами. Наприклад, вебсервер має обслуговувати численні одночасні з’єднання, забезпечуючи оперативну обробку запитів без зайвих затримок через блоковані операції. Простий послідовний підхід не може ефективно задовольнити такі вимоги під високим навантаженням, тоді як багатопотоковість пропонує дієві стратегії для паралельної обробки та оптимізації процесів.

Типовим шаблоном у мережевому програмуванні є модель конкурентності, де кілька потоків обробляють окремі з’єднання або завдання. Кожен потік працює незалежно, не блокуючи інші, що сприяє кращому використанню CPU та зменшенню часу очікування відповіді.

```
#include <iostream>
#include <thread>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

// Функція для обробки клієнтського з'єднання
void handleClient(tcp::socket socket) {
    try {
        std::cout << "Встановлено з'єднання з клієнтом." << std::endl;
        for (;;) {
            char data[512];
            boost::system::error_code error;
            size_t length = socket.read_some(boost::asio::buffer(data), error);

            if (error == boost::asio::error::eof) {
                break; // З'єднання закрито клієнтом
            } else if (error) {
                throw boost::system::system_error(error); // Виникла інша помилка
```

```
            }

            // Відправка отриманих даних назад клієнту (echo)
            boost::asio::write(socket, boost::asio::buffer(data, length));
        }
    } catch (std::exception& e) {
        std::cerr << "Виняток у потоці обробки клієнта: " << e.what() << std::endl;
    }
}

int main() {
    try {
        boost::asio::io_context ioContext;
        tcp::acceptor acceptor(ioContext, tcp::endpoint(tcp::v4(), 1234));

        for (;;) {
            tcp::socket socket(ioContext);
            acceptor.accept(socket);

            // Створення окремого потоку для кожного клієнтського з'єднання
            std::thread(handleClient, std::move(socket)).detach();
        }
    } catch (std::exception& e) {
        std::cerr << "Виняток: " << e.what() << std::endl;
    }

    return 0;
}
```

Пояснення

Використано бібліотеку `Boost.Asio` для асинхронної роботи з TCP-з’єднаннями.

Кожне нове з’єднання клієнта обробляється окремим потоком (`std::thread`), що дозволяє сервісу одночасно обслуговувати багато клієнтів.

`socket.read_some` читає частину даних із сокета, а `boost::asio::write` відправляє їх назад, реалізуючи echo-сервер.

`detach()` дозволяє потоку завершувати роботу незалежно від головного потоку, звільняючи ресурси після завершення роботи.

Використання обробки виключень забезпечує надійну роботу сервера навіть у випадку помилок під час обміну даними.

У цьому сервері кожне клієнтське з’єднання обробляється в окремому потоці, що дозволяє серверу одночасно обслуговувати декілька клієнтів без блокування. Використання `std::thread::detach()` дозволяє кожному

потоку виконуватися автономно, а після завершення роботи ресурси автоматично звільняються без необхідності явного виклику `join()`.

Іншим підходом до підвищення продуктивності мережевих застосунків є використання асинхронних операцій введення/виведення у поєднанні з потоками. У той час як традиційні багатопотокові сервери виділяють окремий потік для кожного клієнта, асинхронне програмування забезпечує неблокувальні операції, мінімізуючи час очікування завершення I/O та дозволяючи одночасно виконувати інші обчислювальні чи мережеві задачі.

Бібліотека `Boost.Asio` для мережевого та низькорівневого I/O програмування в C++ надає широкі можливості для реалізації асинхронних операцій. Використання асинхронного введення/виведення разом із робочими потоками дозволяє обробляти велику кількість одночасних запитів без тісного зв'язку між потоками та клієнтськими з'єднаннями.

У цьому прикладі реалізовано багатопотоковий асинхронний TCP-сервер на основі бібліотеки Boost.Asio, який здатний ефективно обслуговувати численні клієнтські з'єднання одночасно.

Клас `Session` управляє окремою сесією клієнта. Він успадкований від `std::enable_shared_from_this`, що дозволяє створювати `std::shared_ptr`, посилання на поточний об'єкт для безпечного використання під час асинхронних операцій.

Метод `start()` запускає асинхронне читання даних із сокета через `async_read_some`, а після успішного отримання виконує `doWrite()` для асинхронного записування даних назад клієнту. Така схема забезпечує циклічне асинхронне читання та записування, що гарантує неблокувальну роботу з мережею.

Клас `Server` відповідає за прийом клієнтських з'єднань. Метод `doAccept()` використовує `async_accept`, створюючи нову сесію (`Session`) для кожного клієнта та негайно знову очікуючи на нові підключення. Це забезпечує безперервну обробку запитів без блокування основного потоку.

У функції `main()` створюється контекст введення/виведення (`io_context`) та сервер на вказаному порту. Для обробки асинхронних операцій створюється пул потоків відповідно до кількості апаратних ядер (`std::thread::hardware_concurrency()`). Кожен потік виконує `ioContext.run()`, що дозволяє асинхронним операціям виконуватися паралельно на декількох ядрах, підвищуючи пропускну здатність і масштабованість сервера.

Використання багатопотоковості разом із асинхронним I/O дозволяє серверу ефективно обробляти численні клієнтські запити, мінімізуючи час очікування та забезпечуючи високу продуктивність мережевих застосунків.

```
#include <iostream>
#include <memory>
#include <thread>
#include <boost/asio.hpp>
```

```
using boost::asio::ip::tcp;

// Клас Session управляє окремою клієнтською сесією
class Session : public std::enable_shared_from_this<Session> {
public:
    explicit Session(tcp::socket socket)
        : m_socket(std::move(socket)) {}

    // Запуск сесії
    void start() {
        doRead();
    }

private:
    // Асинхронне читання даних із сокета
    void doRead() {
        auto self(shared_from_this());
        m_socket.async_read_some(
            boost::asio::buffer(m_data, maxLength),
            [this, self](boost::system::error_code ec, std::size_t length) {
                if (!ec) doWrite(length); // Після читання асинхронно записати
дані
            }
        );
    }

    // Асинхронний запис даних у сокет
    void doWrite(std::size_t length) {
        auto self(shared_from_this());
        boost::asio::async_write(
            m_socket,
            boost::asio::buffer(m_data, length),
            [this, self](boost::system::error_code ec, std::size_t /*length*/) {
                if (!ec) doRead(); // Після запису продовжити читання
            }
        );
    }

    tcp::socket m_socket;               // Сокет для сесії
    enum { maxLength = 1024 };          // Максимальний розмір буфера
    char m_data[maxLength];             // Буфер даних
};

// Клас Server приймає нові підключення
class Server {
public:
    Server(boost::asio::io_context& io_context, short port)
```

```
        : m_acceptor(io_context, tcp::endpoint(tcp::v4(), port)) {
        doAccept();
    }

private:
    // Асинхронне приймання нового підключення
    void doAccept() {
        m_acceptor.async_accept(
            [this](boost::system::error_code ec, tcp::socket socket) {
                if (!ec) {
                    std::make_shared<Session>(std::move(socket))->start();
                }
                doAccept(); // Приймати наступне підключення
            }
        );
    }

    tcp::acceptor m_acceptor;  // Приймальний сокет
};

int main() {
    try {
        boost::asio::io_context ioContext;

        // Створення сервера на порту 1234
        Server server(ioContext, 1234);

        // Створення пулу потоків відповідно до кількості ядер CPU
        std::vector<std::thread> threads;
        for (unsigned i = 0; i < std::thread::hardware_concurrency(); ++i) {
            threads.emplace_back([&ioContext]() { ioContext.run(); });
        }

        // Очікування завершення всіх потоків
        for (auto& thread : threads) {
            thread.join();
        }

    } catch (std::exception& e) {
        std::cerr << "Виключення: " << e.what() << std::endl;
    }

    return 0;
}
```

Пояснення:

1. Session – керує однією клієнтською сесією. Використання `std::enable_shared_from_this` дозволяє безпечно передавати об’єкт у лямбда-функції асинхронних викликів, забезпечуючи, що об’єкт не буде видалено під час обробки.

2. doRead / doWrite – асинхронне читання та записування, що створює неблокувальну циклічну обробку даних.

3. Server – приймає нові підключення та створює об’єкт Session для кожного клієнта.

4. Пул потоків – дозволяє обробляти асинхронні операції паралельно на кількох ядрах CPU, підвищуючи продуктивність.

5. Boost.Asio забезпечує асинхронні мережеві операції та масштабованість сервера без блокування основного потоку.

Сервер, реалізований за допомогою Boost.Asio, демонструє використання асинхронних операцій. Клас Session застосовує асинхронне читання та записування для обробки запитів клієнтів без блокування потоків, у цьому випадку підтримка здійснюється пулом потоків, асоційованим з об’єктом io_context. Такий поділ обов’язків дозволяє io_context.run() виконувати завдання введення/виведення, тоді як потоки залишаються доступними для ефективного управління результатними операціями.

Пули потоків є ключовими у випадках, коли повторювані завдання виконуються часто та непередбачувано, наприклад, під час обслуговування мережевих підключень. Пул потоків утримує групу заздалегідь виділених потоків, готових виконувати завдання, що мінімізує накладні витрати, пов’язані з постійним створенням і знищенням потоків. Використання пулу потоків особливо вигідне у мережевих додатках, які обробляють часті запити на підключення та тривалі мережеві операції.

```
#include <iostream>                // Для роботи з ввід/вивід
#include <thread>                  // Для роботи з потоками
#include <vector>                  // Для використання контейнера vector
#include <boost/asio.hpp>          // Бібліотека Boost.Asio для мережевого програмування
#include <boost/asio/thread_pool.hpp> // Підтримка thread_pool у Boost.Asio

using boost::asio::ip::tcp;              // Спрощене оголошення для TCP протоколу

// Функція обробки сесії з клієнтом
void session(tcp::socket socket) {
    try {
        std::cout << "Клієнт підключений." << std::endl;

        // Безкінечний цикл для обміну даними з клієнтом
        for (;;) {
```

```
            char data[512];                    // Буфер для отримання даних від клієнта
            boost::system::error_code error;    // Змінна для обробки помилок

            // Зчитування даних із сокета
            size_t length = socket.read_some(boost::asio::buffer(data), error);

            if (error == boost::asio::error::eof) {
               // Клієнт закрив з'єднання
               break;
            } else if (error) {
               // Інші помилки зчитування
               throw boost::system::system_error(error);
            }

            // Відправлення отриманих даних назад клієнту (echo)
            boost::asio::write(socket, boost::asio::buffer(data, length));
         }
      } catch (std::exception& e) {
         // Обробка винятків під час роботи сесії
         std::cerr << "Виняток у сесії: " << e.what() << std::endl;
      }
   }

   int main() {
      try {
         // Контекст введення/виведення для Boost.Asio
         boost::asio::io_context ioContext;

         // Створення TCP приймача на порту 1234
         tcp::acceptor acceptor(ioContext, tcp::endpoint(tcp::v4(), 1234));

         // Створення thread pool розміром під кількість апаратних потоків процесора
         boost::asio::thread_pool pool(std::thread::hardware_concurrency());

         std::cout << "Сервер запущено на порту 1234..." << std::endl;

         // Основний цикл прийому клієнтів
         while (true) {
            tcp::socket socket(ioContext);  // Створення сокета для нового клієнта
            acceptor.accept(socket);        // Прийом підключення

            // Передаємо сокет у thread pool через lambda-функцію
            // Lambda копіює сокет через move-семантику (C++14)
            boost::asio::post(pool, [sock = std::move(socket)]() mutable {
```

```
                session(std::move(sock));   // Виклик функції обробки сесії
            });
        }

        // Очікування завершення всіх потоків у пулі (не досягається через безкінечний цикл)
        pool.join();
    } catch (std::exception& e) {
        // Обробка винятків на рівні сервера
        std::cerr << "Виняток: " << e.what() << std::endl;
    }

    return 0;
}
```

Пояснення ключових моментів:

1. `tcp::acceptor` – об’єкт для прослуховування підключень на TCP-порті.
2. `boost::asio::thread_pool` – пул потоків для одночасної обробки кількох клієнтів.
3. `boost::asio::post` – дозволяє поставити завдання в пул потоків.
4. `sock = std::move(socket)` – переміщення сокета у lambda, щоб уникнути копіювання і зберегти ownership.
5. `session` – функція, яка реалізує echo-сервер, читаючи дані від клієнта та повертаючи їх назад.

Впровадження boost::asio::thread_pool полегшує обробку підключень у конфігурації з пулом потоків. За допомогою функції post() завдання ефективно розподіляються між доступними потоками в пулі, що оптимізує обробку мережевих запитів та раціональне використання ресурсів.

Використання багатопотоковості у мережевих операціях введення/виведення створює певні складнощі, такі як управління одночасним доступом до спільних даних, налагодження асинхронних операцій та налаштування оптимального рівня паралелізму. Для їх вирішення потрібно глибоке розуміння архітектури системи, накладних витрат на багатопотоковість та характеристик мережевого навантаження.

Гонки даних та синхронізація

Мережеві операції введення/виведення неминуче передбачають спільні ресурси, що робить умови гонок даних (data races) значною проблемою. Реалізація відповідних механізмів синхронізації, таких як mutex та атомарні операції, забезпечує узгодженість даних.

Обробка помилок та безпека потоків

Асинхронні операції потребують ретельних стратегій обробки помилок. Несподівані відмови мережі або розриви з’єднання потребують надійних механізмів відновлення та повторних спроб для підтримки стабільності сервісу.

Оптимізація пропускної здатності та затримки

Баланс між латентністю та пропускною здатністю є критичним. Мінімізація перемикання контексту та забезпечення ефективної обробки запитів за допомогою раціонального використання пулів потоків та асинхронного введення/виведення сприяє створенню масштабованих мережевих додатків.

Балансування навантаження та розподіл ресурсів

Налаштування пулів потоків із урахуванням можливостей системи та характеру навантаження оптимізує розподіл ресурсів, запобігаючи як недовантаженню, так і перевантаженню серверних ресурсів, що забезпечує їх оптимальну продуктивність.

Моніторинг та профілювання

Безперервний моніторинг мережевих додатків є необхідним для підтримки ефективності роботи. Інструменти профілювання допомагають виявляти вузькі місця, оцінювати розподіл навантаження та діагностувати проблеми у багатопотоковій моделі.

Впровадження багатопотоковості та конкурентних операцій введення/виведення суттєво підвищує ефективність мережевих додатків. Структуруючи такі додатки з використанням асинхронного введення/виведення та стратегій пулів потоків, розробники можуть ефективно працювати в складних сценаріях, досягаючи високої пропускної здатності та швидкого відгуку. Управління одночасним доступом і оптимізація розподілу потоків є фундаментальними аспектами для вирішення проблем, що виникають у таких архітектурах.

3.6 Реалізація багатопотокових мережевих додатків

Розробка багатопотокових мережевих додатків має значний потенціал для підвищення продуктивності та масштабованості, проте супроводжується низкою складнощів.

Розв’язання цих проблем є критичним для створення надійних, ефективних та стабільних програмних рішень. Глибоке розуміння особливостей паралелізму, синхронізації, управління ресурсами та обробки помилок є необхідним для подолання внутрішньої складності таких додатків.

Паралелізм створює потенційні проблеми, коли кілька потоків одночасно оперують спільними ресурсами. Умови гонок (race conditions) виникають, коли виконання потоків перемішується непередбачуваним чином, що призводить до нестабільних станів даних та багів, які важко відстежити та відтворити.

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

// Спільний ресурс
```

```
int sharedResource = 0;

// М'ютекс для синхронізації доступу до спільного ресурсу
std::mutex mtx;

// Функція для інкрементування ресурсу
void incrementResource() {
    for (int i = 0; i < 1000; ++i) {
        // Захищаємо доступ до спільного ресурсу
        std::lock_guard<std::mutex> lock(mtx);
        ++sharedResource; // Тепер без умов гонки
    }
}

int main() {
    std::vector<std::thread> threads;

    // Створюємо 10 потоків
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(incrementResource);
    }

    // Очікуємо завершення всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Кінцеве значення спільного ресурсу: " << sharedResource << std::endl;

    return 0;
}
```

Наведений вище код демонструє типовий сценарій умови гонки (race condition), коли операції інкрементування можуть виконуватися неатомарно, що призводить до непередбачуваних кінцевих значень змінної `sharedResource`. Для запобігання таких ситуацій використовують м’ютекси або атомарні змінні, які забезпечують безпечний доступ до спільних даних у багатопотоковому середовищі.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

// Спільний ресурс, доступ до якого потребує синхронізації
int safeResource = 0;
```

```
// М’ютекс для синхронізації доступу до спільного ресурсу
std::mutex resourceMutex;

// Функція безпечного інкрементування ресурсу
void safeIncrement() {
    for (int i = 0; i < 1000; ++i) {
        // Використання lock_guard для автоматичного блокування та розблокування м’ютекса
        std::lock_guard<std::mutex> lock(resourceMutex);
        ++safeResource;  // Атомарне збільшення значення ресурсу
    }
}

int main() {
    std::vector<std::thread> threads;

    // Створення 10 потоків для одночасного доступу до ресурсу
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(safeIncrement);
    }

    // Очікування завершення всіх потоків
    for (auto& t : threads) {
        t.join();
    }

    // Виведення фінального значення ресурсу
    std::cout << "Final safe resource value: " << safeResource << std::endl;

    return 0;
}
```

Цей код демонструє безпечний багатопотоковий доступ до спільного ресурсу за допомогою м’ютекса (`std::mutex`). Використання `std::lock_guard` забезпечує автоматичне блокування м’ютекса під час операцій інкрементування і їх автоматичне розблокування після виходу з області видимості, що гарантує атомарність операцій та запобігає умовам гонки. Такий підхід є основою синхронізації у багатопотокових додатках, де необхідно забезпечити цілісність даних за одночасного доступу з декількох потоків.

Введення м’ютекса забезпечує послідовний доступ до змінної `safeResource`, усуваючи умову гонки та гарантуючи узгоджені результати.

Взаємні блокування (deadlocks), які є постійною проблемою у багатопотокових додатках, виникають, коли два або більше потоків безкінечно очікують ресурси, зайняті іншими потоками, що призводить до зупинення виконання програми. Уникнення взаємних блокувань зазвичай

досягається шляхом розробки стратегії, яка забезпечує атомарне захоплення всіх м’ютексів, або застосування порядкування блокувань для мінімізації циклічних залежностей між ресурсами.

```
#include <iostream>
#include <thread>
#include <mutex>

// Оголошення двох м'ютексів для синхронізації доступу до ресурсів
std::mutex mutex1, mutex2;

// Функція завдання A
void taskA() {
    std::lock_guard<std::mutex> lock1(mutex1); // Захоплюємо mutex1
    std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Імітація обчислювальної роботи
    std::lock_guard<std::mutex> lock2(mutex2); // Спроба захопити mutex2
    std::cout << "Завдання A виконано." << std::endl;
}

// Функція завдання B
void taskB() {
    std::lock_guard<std::mutex> lock2(mutex2); // Захоплюємо mutex2
    std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Імітація обчислювальної роботи
    std::lock_guard<std::mutex> lock1(mutex1); // Спроба захопити mutex1
    std::cout << "Завдання B виконано." << std::endl;
}

int main() {
    // Створення двох потоків для виконання завдань A та B
    std::thread threadA(taskA);
    std::thread threadB(taskB);

    // Очікування завершення обох потоків
    threadA.join();
    threadB.join();

    return 0;
}
```

Цей приклад демонструє можливе виникнення взаємного блокування (deadlock).

Потік A захоплює `mutex1` і чекає на `mutex2`.

Потік B захоплює `mutex2` і чекає на `mutex1`.

Результат – обидва потоки блокуються, очікуючи один одного, і програма зависає.

Наведена вище ілюстрація демонструє виникнення взаємного блокування (deadlock), коли `taskA` та `taskB` одночасно утримують по одному м’ютексу та очікують захоплення іншого. Для запобігання таких ситуацій рекомендується використовувати `std::scoped_lock`, який дозволяє одночасно захоплювати кілька м’ютексів, забезпечуючи взаємне виключення та мінімізуючи ризик випадкових deadlock-ів.

```
#include <iostream>
#include <thread>
#include <mutex>

// Оголошення двох м'ютексів для синхронізації доступу до спільних ресурсів
std::mutex mutex1, mutex2;

// Функція безпечної задачі A
void safeTaskA() {
    // Створюємо замки для обох м'ютексів, але не захоплюємо їх одразу (defer_lock)
    std::unique_lock<std::mutex> lock1(mutex1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(mutex2, std::defer_lock);

    // Одночасне захоплення обох м'ютексів
    // Гарантує відсутність deadlock, навіть якщо інші потоки намагаються захопити м'ютекси в іншому порядку
    std::lock(lock1, lock2);

    // Виконання задачі після безпечного захоплення ресурсів
    std::cout << "Safe Task A executed." << std::endl;
}

// Функція безпечної задачі B
void safeTaskB() {
    // Створюємо замки для обох м'ютексів без їхнього захоплення
    std::unique_lock<std::mutex> lock1(mutex1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(mutex2, std::defer_lock);

    // Одночасне захоплення обох м'ютексів
    std::lock(lock1, lock2);

    // Виконання задачі після безпечного захоплення ресурсів
    std::cout << "Safe Task B executed." << std::endl;
}
```

```
int main() {
    // Створюємо два потоки для виконання задач A та B
    std::thread threadA(safeTaskA);
    std::thread threadB(safeTaskB);

    // Очікуємо завершення потоків
    threadA.join();
    threadB.join();

    // Завершення програми
    return 0;
}
```

Пояснення ключових моментів:

1. `std::unique_lock` із `std::defer_lock` дозволяє створити замок без негайного захоплення ресурсу.

2. `std::lock` одночасно захоплює кілька м'ютексів, що усуває можливість deadlock за багатопотокового доступу.

3. Використання потоків `std::thread` дозволяє виконувати задачі паралельно, а виклики `join()` гарантують завершення потоків перед завершенням програми.

Використання `scoped_lock` забезпечує атомарне захоплення блокувань, запобігаючи циклічному очікуванню ресурсів та ефективно усуваючи можливість виникнення deadlock.

Обробка помилок і винятків у мережевих додатках

Мережеві застосунки схильні до численних помилок під час виконання та винятків, включно з тайм-аутами з'єднання, розривами каналів зв'язку та неповними операціями зчитування даних. Багатопотоковість ускладнює ці сценарії, оскільки контексти помилок розподіляються між багатьма потоками виконання, що робить діагностику та відновлення більш складними.

Ефективні стратегії управління помилками у мережевих додатках містять:

1. Повноцінну обробку винятків у потоках – використання блоків `try-catch` для акуратного оброблення та логування винятків без шкоди для цілісності системи.

2. Надійне управління станами сесій з'єднання – забезпечує виявлення помилкових станів та їх коректне відновлення, часто із повторною ініціалізацією процедури підключення.

3. Асинхронне повторне виконання операцій за тимчасових збоїв — застосування стратегій, таких як експоненційне збільшення інтервалів (exponential backoff), для запобігання надмірному використанню ресурсів.

```
#include <iostream>
#include <thread>
#include <boost/asio.hpp>
```

```
using boost::asio::ip::tcp;

// Функція обробки сесії з клієнтом
void clientSession(tcp::socket socket) {
    try {
        for (;;) {
            char data[512];
            boost::system::error_code error;

            // Зчитування даних з сокета клієнта
            size_t length = socket.read_some(boost::asio::buffer(data), error);

            if (error) {
                if (error == boost::asio::error::eof) {
                    // Клієнт закрив з'єднання
                    std::cout << "З'єднання закрито клієнтом." << std::endl;
                } else {
                    // Інша помилка зчитування
                    std::cerr << "Помилка зчитування: " << error.message() <<
std::endl;
                }
                break;
            }

            // Відправлення отриманих даних назад клієнту (echo)
            boost::asio::write(socket, boost::asio::buffer(data, length));
        }
    } catch (const std::exception& e) {
        // Обробка винятків у сесії
        std::cerr << "Виняток у сесії: " << e.what() << std::endl;
    }
}

int main() {
    try {
        boost::asio::io_context ioContext;

        // Створення приймача на порту 1234
        tcp::acceptor acceptor(ioContext, tcp::endpoint(tcp::v4(), 1234));

        std::cout << "Сервер запущено на порту 1234..." << std::endl;

        while (true) {
            tcp::socket socket(ioContext);

            // Очікування підключення клієнта
```

```
            acceptor.accept(socket);

            // Кожне підключення обробляється в окремому потоці
            std::thread(clientSession, std::move(socket)).detach();
        }

    } catch (const std::exception& e) {
        // Обробка винятків сервера
        std::cerr << "Виняток сервера: " << e.what() << std::endl;
    }

    return 0;
}
```

Пояснення:

1. Багатопотоковість. Кожне підключення клієнта обробляється в окремому потоці, що дозволяє серверу одночасно обслуговувати кілька клієнтів.

2. Обробка помилок. Використання `boost::system::error_code` дозволяє коректно обробляти помилки мережевого введення/виведення, включно з закриттям з'єднання клієнтом.

3. Асинхронність потоків. `std::thread::detach()` дозволяє потоку виконуватися незалежно від головного потоку сервера, автоматично звільняючи ресурси після завершення роботи.

4. Ехо-сервер. Сервер просто відправляє назад дані, отримані від клієнта, демонструючи базову обробку мережевих повідомлень.

Приклад демонструє обробку винятків мережевого введення/виведення всередині функції `clientSession`, що забезпечує відновлення або коректне завершення кожної сесії окремо, не впливаючи на роботу сервера загалом.

Впровадження механізмів синхронізації підвищує безпеку даних, але може знизити продуктивність застосунку. Виникають затримки через конфлікти доступу (contention), коли інтенсивне блокування обмежує паралельне виконання. Тому важливо проєктувати моделі конкурентності, які забезпечують баланс між координацією потоків та пропускною здатністю системи.

Блокування дрібного та грубого рівня (Fine-grained vs. Coarse-grained Locking)

Захоплювати блокування лише на критично важливих сегментах даних для зменшення конфліктів та підвищення пропускної здатності.

Оцінювати продуктивність системи з різною гранулярністю блокувань для емпіричної оптимізації моделей доступу до ресурсів.

Безблокові стратегії (Lock-free Strategies)

Використовувати атомарні операції та безблокові структури даних у сценаріях з високим обсягом читання або критичними за часом операціями.

Розглядати неблокувальні алгоритми та моделі «eventual consistency» у розподілених системах.

Максимізація паралельної масштабованості

Використовувати можливості паралельної обробки багатоядерних процесорів.

Уникати надмірної кількості потоків та розумно керувати призначенням завдань, щоб зменшити вузькі місця та непотрібні накладні витрати паралелізму.

Існує безліч бібліотек і фреймворків для розробки багатопотокових мережевих застосунків, кожен з яких надає специфічні абстракції, корисні для певних сценаріїв.

Boost.Asio забезпечує асинхронне введення/виведення і мультиплексування I/O.

Intel Threading Building Blocks (TBB) пропонує алгоритми та графи потоків для розширеного паралельного програмування з акцентом на модульність та масштабованість.

Вибір інструментів потребує оцінення:

- інтеграції та сумісності з технологічним стеком застосунку;
- рівня абстракції порівняно з програмуванням «голих» сокетів;
- підтримки спільноти, документації та супроводу для довгострокового використання.

Паралельна обробка мережевих операцій потребує впровадження надійних практик безпеки.

Використання безпечних протоколів зв’язку, наприклад TLS, для забезпечення цілісності та конфіденційності даних.

Використання належної синхронізації для уникнення вразливостей типу TOCTOU (time-of-check to time-of-use) під час паралельних процесів.

Ретельний моніторинг контролю доступу для запобігання несанкціонованому доступу в багатопотокових робочих процесах.

Багатопотокові мережеві застосунки забезпечують підвищену продуктивність, проте їх ефективна розробка та управління залежать від вирішення фундаментальних проблем: конкурентності, координації ресурсів, обробки помилок, ефективності синхронізації та забезпечення безпеки системи і процесів.

Акцент на цих факторах під час проєктування та реалізації дозволяє створювати надійні, високопродуктивні та безпечні мережеві сервіси.

3.7 Контрольні питання

1. Що таке багатопотоковість і яку роль вона відіграє у підвищенні продуктивності мережевих додатків?

2. Поясніть, як виникає умова гонки (race condition) у багатопотокових програмах і як її можна уникнути за допомогою механізмів синхронізації.

3. У чому полягає призначення м’ютексів (std::mutex) та об’єктів-охоронців (std::lock_guard) у багатопотоковому програмуванні на C++?

4. Наведіть приклади сценаріїв використання багатопотоковості у мережевих додатках, таких як TCP-сервери або обробка клієнтських запитів.

5. Які фактори та обмеження потрібно враховувати під час проєктування багатопотокових мережевих застосунків, щоб балансувати між продуктивністю та складністю системи?

6. Які механізми стандартної бібліотеки C++ використовуються для створення та керування потоками, і який заголовковий файл необхідно підключити для їх використання?

7. Поясніть різницю між методами join() та detach() у класі std::thread і в яких випадках доцільно застосовувати кожен із них.

8. Які типи викликабельних об’єктів можна передавати в конструктор std::thread для виконання у новому потоці? Наведіть приклади.

9. Як правильно передавати параметри функцій у потоки C++ і чому важливо враховувати семантику копіювання або передачі за посиланням?

10. Яким чином засоби std::future і std::async спрощують асинхронне виконання завдань у C++ і яку роль відіграє об’єкт std::future?

11. Що таке умова змагання (race condition) у багатопотоковому програмуванні, і чому вона виникає під час одночасного доступу потоків до спільних ресурсів?

12. Як у C++ використовується клас std::mutex для синхронізації потоків і яку роль відіграє std::lock_guard у роботі з м’ютексами?

13. У чому полягає відмінність між std::mutex, std::recursive_mutex та std::timed_mutex, і коли доцільно застосовувати кожен із них?

14. Як працюють змінні умов (std::condition_variable) у C++ і яким чином вони дозволяють потокам ефективно синхронізуватися без активного очікування (busy-waiting)?

15. Які переваги надають атомарні змінні (std::atomic) у багатопотокових програмах і чому вони іноді можуть замінювати використання м’ютексів для простих операцій?

16. Яку роль відіграють м’ютекси (std::mutex) у багатопотоковому програмуванні, і чим відрізняється дрібнокрокове (fine-grained) блокування від грубого (coarse-grained) блокування?

17. Як механізм std::shared_mutex та блокування std::shared_lock і std::lock_guard дозволяють одночасно читати дані кільком потокам і забезпечувати ексклюзивний доступ для записування?

18. У чому переваги використання атомарних змінних (std::atomic) порівняно з м’ютексами у разі управління простими спільними ресурсами?

19. Які підходи до управління спільними ресурсами допомагають уникнути станів гонки, дедлоків та вузьких місць у продуктивності багатопотокових програм?

20. Як шаблони проєктування, такі як модель виробник-споживач або пул потоків, сприяють ефективному керуванню спільними даними та ресурсами у багатопотокових додатках?

21. Які переваги надає використання багатопотоковості у мережевих додатках під час обробки численних одночасних I/O-завдань?

22. Як працює модель конкурентності у мережевому програмуванні, коли кожен потік обробляє окреме з’єднання, і чим вона відрізняється від послідовної обробки?

23. У чому полягає роль асинхронного введення/виведення (async_read_some, async_write) у поєднанні з потоками, і як це підвищує продуктивність сервера?

24. Яку функцію виконують пул потоків (thread_pool) та boost::asio::post() у багатопотоковому мережевому сервері?

26. Які основні проблеми виникають за багатопотоковості у мережевому I/O, і які методи використовуються для їх вирішення (наприклад, синхронізація, обробка помилок, балансування навантаження)?

27. Що таке умова гонки (race condition) у багатопотокових мережевих додатках і як її уникнути під час доступу до спільних ресурсів?

28. Які методи синхронізації потоків використовуються для забезпечення безпечного доступу до спільних ресурсів і чим відрізняються std::mutex, std::lock_guard та std::unique_lock?

29. Як виникає взаємне блокування (deadlock) у багатопотокових додатках і які стратегії дозволяють його запобігти?

30. Які підходи застосовуються для ефективної обробки помилок та винятків у багатопотокових мережевих додатках, зокрема у сценаріях з’єднання TCP?

31. Які принципи та стратегії дозволяють максимізувати продуктивність і масштабованість багатопотокових мережевих додатків, враховуючи гранулярність блокувань, безблокові алгоритми та баланс потоків?

ВИСНОВКИ

Внаслідок опрацювання матеріалів навчального посібника узагальнено сучасні підходи до проєктування, реалізації та оптимізації мережевих додатків у середовищі багатопотокової обробки даних. Розглянуто принципи функціонування стека протоколів TCP/IP, механізми синхронної та асинхронної комунікації; а методи керування паралельними процесами мовою програмування C++ формують цілісну систему знань, необхідну для підготовки фахівців у галузі інформаційно-комунікаційних технологій.

У посібнику обґрунтовано доцільність вибору моделей взаємодії залежно від вимог до продуктивності, надійності та масштабованості програмних систем. Показано, що синхронні підходи забезпечують детермінованість і передбачуваність обробки даних, тоді як асинхронні моделі сприяють підвищенню пропускної здатності та ефективному використанню обчислювальних ресурсів у розподілених середовищах.

Особливу увагу приділено питанням багатопотоковості, синхронізації доступу до спільних ресурсів і запобігання конфліктам виконання. Проаналізовано вплив різних стратегій блокування, безблокових алгоритмів та механізмів керування потоками на продуктивність і стабільність мережевих застосунків. Обґрунтовано необхідність балансування між рівнем координації процесів і пропускною здатністю системи.

Узагальнено можливості сучасних бібліотек і фреймворків для паралельного програмування, зокрема Boost.Asio та Intel TBB, а також визначено критерії їх вибору залежно від архітектури та функціональних вимог програмних рішень. Наголошено на важливості дотримання вимог інформаційної безпеки, використання захищених протоколів зв’язку та коректної синхронізації для запобігання вразливостям у багатопотокових середовищах.

Практична спрямованість посібника, поєднання теоретичних положень із прикладами програмного коду та рекомендаціями з проєктування сприяють формуванню у здобувачів освіти стійких професійних компетентностей. Отримані знання та навички можуть бути ефективно використані під час розробки, супроводу й модернізації сучасних мережевих інформаційних систем.

Таким чином, навчальний посібник є комплексним джерелом знань із мережевого програмування та багатопотокової обробки даних, що забезпечує підготовку фахівців, здатних створювати надійні, масштабовані й безпечні програмні рішення відповідно до сучасних вимог цифрового середовища.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. – 8th ed. – Pearson, 2021.
2. Peterson L., Davie B. Computer Networks: A Systems Approach. – 6th ed. – Morgan Kaufmann, 2021.
3. Tanenbaum A. S., Wetherall D. J. Computer Networks. – 6th ed. – Pearson, 2021.
4. Stallings W. Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud. – 2nd ed. – Pearson, 2020.
5. Stevens W. R., Fenner B., Rudoff A. UNIX Network Programming, Volume 1. – Reprint edition. – Addison-Wesley, 2021.
6. Kerrisk M. The Linux Programming Interface. – Updated edition. – No Starch Press, 2022.
7. Meyers S. Effective Modern C++. – Updated reprint. – O'Reilly Media, 2020.
8. Josuttis N. M. C++20 – The Complete Guide. – Addison-Wesley, 2022.
9. ISO/IEC 14882:2020. Programming Languages — C++ (C++20 Standard).
10. Boost.Asio Documentation. Networking TS and Asynchronous I/O. – 2023.
11. Beej J. Beej's Guide to Network Programming: Using Internet Sockets. – Online edition, 2023.
12. OpenSSL Project. OpenSSL Documentation: SSL/TLS Protocols. – 2023.
13. Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3. – RFC 8446, IETF, 2020.
14. IETF. RFC 9293 – Transmission Control Protocol (TCP). – 2022.
15. IETF. RFC 768 – User Datagram Protocol (UDP). – Updated reference, 2020+.
16. Coulouris G., Dollimore J., Kindberg T., Blair G. Distributed Systems: Concepts and Design. – 6th ed. – Pearson, 2024.
17. Souders S. High Performance Web Sites. – Updated digital edition. – O'Reilly, 2020.
18. Linux Foundation. Linux Networking Documentation. – 2023.
19. Microsoft. Windows Sockets 2 (WinSock) Documentation. – 2022.
20. Open Group. POSIX.1-2018 / 2022 Base Specifications.
21. Kubernetes Documentation. Networking Concepts. – 2023.
22. Cloudflare Research. Modern Internet Protocols and Performance. – 2021–2023.
23. ISO/IEC 27002:2022. Information Security Controls.
24. Васильківський М. В., Бортник Г. Г., Кичак В. М. Програмні технології в інфокомунікаційних системах : навчальний посібник для

студентів спеціальності 172 «Телекомунікації та радіотехніка» : електронний навчальний посібник комбінованого (локального та мережного) використання. Вінниця : ВНТУ, 2023. 141 с.

25. Васильківський М., Коломієць А., Будаш М., Прикмета А., Олійник А. Технологічні аспекти впровадження програмно-керованих мереж 5G та 6G. *Комп`ютерно-інтегровані технології: освіта, наука, виробництво.* 2024. Вип. № 56. С. 335–344.

26. Васильківський М. , Коломієць А. , Грабчак Н. , Грицаюк Д. , Костьянін В. Програмні засоби Python моделювання телекомунікаційних пристроїв. *Комп`ютерно-інтегровані технології: освіта, наука, виробництво*. 2024. Вип. 55. С. 270–278.

27. Оптимізація програмно-конфігурованих літаючих мереж доступу / М. Васильківський, А. Прикмета, А. Олійник, Н. Ксьондз. *Комп`ютерно-інтегровані технології: освіта, наука, виробництво*. 2023. № 52. С. 128–139.

28. Оптимізація адаптивних радіосистем із використанням алгоритмів ШІ та МН / М. Васильківський, Д. Нікітович, Н. Грабчак, Н. Якубівська. *Вимірювальна та обчислювальна техніка в технологічних процесах*. 2023. № 2. С. 112–124.

29. Динамічна інформаційна мережа із вбудованим штучним інтелектом / М. Васильківський, О. Болдирева, Д. Онищук, Ю. Гнатенко. *Комп`ютерно-інтегровані технології: освіта, наука, виробництво*. 2023. № 50. С. 35–40.

30. Оптимізація інтелектуальних телекомунікаційних мереж / М. Васильківський, А. Прикмета, А. Олійник, Д. Нікітович. *Вісник Хмельницького національного університету. Серія «Технічні науки».* 2023. № 1. (317). С. 33–41.

31. Дослідження архітектури штучного інтелекту для інфокомунікаційних мереж 6G / М. Васильківський, Г. Варгатюк, О. Болдирева. *Вимірювальна та обчислювальна техніка в технологічних процесах*. 2022. № 4. С. 62–70.

32. Mykhalevskiy D., Vasylkivskyi N., Horodetska O. Development of a mathematical model for estimating signal strength at the input of the 802.11 standard receiver. Eastern-European Journal of Enterprise Technologies. 2017. Vol. 6, Issue 9 (90). 38–43. doi: https://doi.org/10.15587/1729-4061.2017.114191

33. Mykhalevskiy D.V., Kychak V.M. Development of Information Models for Increasing the Evaluation Efficiency of Wireless Channel Parameters of 802.11 Standard. Latvian Journal of Physics and Technical Sciences. 2019. 56(5), Pp. 22–32. DOI: 10.2478/lpts-2019-0028

Електронне навчальне видання

Микола Володимирович Васильківський
Дмитро Валерійович Михалевський
Геннадій Григорович Бортник

Технології мережевого програмування та паралельної обробки даних

Навчальний посібник

Рукопис оформив *М. Васильківський*

Редактор *Т. Старічек*

Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 16.07.2026 р.
Гарнітура Times New Roman.
Зам. № Р2026-092.
Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
press.vntu.edu.ua;
E-mail: rvv.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.