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

МЕРЕЖЕВЕ ПРОГРАМУВАННЯ:

Проєктування ефективних систем зв'язку

Міністерство освіти і науки України
Вінницький національний технічний університет

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

Мережеве програмування: проєктування ефективних систем зв’язку

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 Мережеве програмування: проєктування ефективних систем зв’язку : навчальний посібник [Електронний ресурс] / Васильківський М. В., Михалевський Д. В., Бортник Г. Г. – Вінниця : ВНТУ, 2026. – 125 с.

ISBN 978-617-8927-00-4 (PDF)

Навчальний посібник присвячений основам мережевого програмування на C++, зокрема створенню та налаштуванню сокетів, організації клієнт-серверної взаємодії та надійної передачі даних у сучасних інфокомунікаційних мережах. Розглянуто ключові мережеві протоколи, моделі OSI та TCP/IP, а також проблеми їх практичної реалізації. Значна увага приділена обробці помилок, винятковим ситуаціям і забезпеченню стійкості мережевих додатків. Посібник поєднує теоретичні знання з практичними прикладами і призначений для студентів та інженерів, які прагнуть створювати ефективні та надійні мережеві системи.

УДК 621.0

ISBN 978-617-8927-00-4(PDF)

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API (Application Programming Interface) – прикладний програмний інтерфейс

ARP (Address Resolution Protocol) – протокол визначення MAC-адреси за IP-адресою

BSD (Berkeley Software Distribution) – сімейство UNIX-подібних операційних систем

CLI (Command Line Interface) – інтерфейс командного рядка

CPU (Central Processing Unit) – центральний процесор

DNS (Domain Name System) – система доменних імен

DoS (Denial of Service) – атака типу «відмова в обслуговуванні»

FD (File Descriptor) – дескриптор файлу

FTP (File Transfer Protocol) – протокол передавання файлів

HTTP (HyperText Transfer Protocol) – протокол передавання гіпертексту

HTTPS (HyperText Transfer Protocol Secure) – захищена версія протоколу HTTP

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

IP (Internet Protocol) – мережевий протокол Інтернету

IoT (Internet of Things) – Інтернет речей

LAN (Local Area Network) – локальна обчислювальна мережа

MAC (Media Access Control) – керування доступом до середовища

MTU (Maximum Transmission Unit) – максимальний розмір пакета передавання

NAT (Network Address Translation) – перетворення мережевих адрес

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

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

QoS (Quality of Service) – якість обслуговування

RFC (Request for Comments) – технічні специфікації IETF

SDN (Software-Defined Networking) – програмно-керовані мережі

Socket – програмний інтерфейс для мережевої взаємодії

TCP (Transmission Control Protocol) – протокол керування передаванням

TLS (Transport Layer Security) – протокол захищеного передавання даних

UDP (User Datagram Protocol) – дейтаграмний протокол користувача

URL (Uniform Resource Locator) – уніфікований локатор ресурсів

WAN (Wide Area Network) – глобальна обчислювальна мережа

WinSock (Windows Sockets API) – реалізація сокетів у ОС Windows

XML (eXtensible Markup Language) – розширювана мова розмітки

ВСТУП

Мережеве програмування є одним із ключових напрямів сучасної комп'ютерної науки, оскільки воно становить основу інфраструктури, що забезпечує обмін даними між інформаційними системами на значних географічних відстанях. У міру розвитку цифрового середовища зростає потреба в ефективних і надійних системах зв'язку, що виводить мережеве програмування на передові позиції технологічного прогресу.

Навчальний посібник «Програмування мереж із використанням C++: побудова ефективних систем зв'язку» має на меті надати комплексний виклад матеріалу для здобувачів освіти та фахівців, які прагнуть опанувати методи мережевого програмування з використанням мови C++. Мова C++ відома своєю високою продуктивністю та ефективністю, що робить її доцільним вибором для розроблення мережевих застосунків, які потребують швидкої й точної обробки даних.

В посібнику розглядаються базові поняття мережевого програмування, детально аналізуються ключові мережеві протоколи та формуються практичні навички програмування з використанням сокетів. Структура видання побудована за принципом послідовного ускладнення матеріалу: від огляду фундаментальних принципів комп'ютерних мереж до вивчення складніших аспектів, зокрема синхронної та асинхронної взаємодії, багатопотоковості та керування потоками даних.

Особливу увагу приділено стеку протоколів TCP/IP, який є основою функціонування мережі Інтернет. Розуміння архітектури та принципів роботи TCP/IP є необхідною умовою підготовки фахівця з мережевого програмування, тому в посібнику докладно розкрито складові цього стека та особливості їх інтеграції в мережеві застосунки.

Важливим аспектом мережевого програмування є питання безпеки. У тексті розглядаються методи та найкращі практики захисту передавання даних, автентифікації учасників мережевої взаємодії та протидії потенційним загрозам. Опанування цих підходів є необхідним для розроблення надійних застосунків, здатних функціонувати в умовах постійно зростаючих викликів у сфері кібербезпеки.

Окремий розділ присвячено оптимізації продуктивності мережевих застосунків. Аналізуються стратегії підвищення швидкодії та ефективності систем зв'язку, зокрема методи виявлення вузьких місць, оптимального використання обчислювальних і мережевих ресурсів та зменшення затримок передавання даних.

Завершальні розділи орієнтовані на практичну розробку мережевих застосунків і містять покрокові рекомендації щодо формування концепції, проєктування та реалізації програмних рішень. Використання прикладів і сценаріїв, наближених до реальних умов експлуатації, сприяє закріпленню

теоретичного матеріалу та підготовці здобувача вищої освіти до практичного застосування набутих знань.

Внаслідок опрацювання навчального посібника здобувачі вищої освіти отримають ґрунтовну підготовку з мережевого програмування мовою C++. Вони набудуть знань і практичних навичок, необхідних для проєктування та реалізації ефективних систем зв’язку, здатних відповідати сучасним вимогам розроблення мережевого програмного забезпечення. Поєднання теоретичних засад і практичної спрямованості робить цей посібник корисним як для початківців, так і для досвідчених розробників, які прагнуть поглибити свою професійну компетентність у галузі мережевого програмування.

1 ВСТУП ДО МЕРЕЖЕВОГО ПРОГРАМУВАННЯ

Мережеве програмування становить основу сучасного взаємопов'язаного цифрового середовища, забезпечуючи ефективну взаємодію різнорідних інформаційних систем у межах комп'ютерних мереж. У цьому розділі закладаються базові засади для розуміння ключових понять комп'ютерних мереж із зосередженням уваги на фундаментальних принципах, що забезпечують обмін даними між вузлами мережі.

Особливу увагу приділено клієнт-серверній моделі взаємодії, яка є однією з основних парадигм побудови мережевих застосунків. Також вводиться базова термінологія мережевого програмування та розглядається роль мови програмування C++ у розробленні ефективних і високопродуктивних мережевих програмних систем.

Окремим аспектом розділу є формування відповідного середовища розроблення, необхідного для практичної реалізації розглянутих підходів і методів. Це створює надійне підґрунтя для подальшого поглибленого вивчення тем мережевого програмування та опанування складніших концепцій і технологій.

1.1 Огляд інфокомунікаційних мереж

Комп'ютерні мережі є основою сучасних систем зв'язку та цифрової інфраструктури й призначені для забезпечення спільного використання ресурсів і обміну даними між множиною пристроїв. Комплексне розуміння комп'ютерних мереж передбачає аналіз їх базових компонентів, призначення, різновидів і принципів функціонування.

Комп'ютерна мережа являє собою сукупність обчислювальних пристроїв, які називають вузлами, з'єднаних між собою з метою передавання даних і спільного використання ресурсів. Такі мережі охоплюють не лише персональні обчислювальні пристрої, а й складніші конфігурації, зокрема центри оброблення даних і хмарні інфраструктури. Основною метою побудови комп'ютерної мережі є використання взаємодії сучасних обчислювальних систем для підвищення ефективності, рівня співпраці та якості комунікації як у локальному, так і в глобальному масштабах.

Типова комп'ютерна мережа складається з кількох ключових компонентів: фізичних середовищ передавання даних, апаратних засобів (маршрутизаторів, комутаторів тощо), що керують потоками інформації, програмних протоколів, які регламентують правила передавання та форматування даних, а також мережевих сервісів, що забезпечують доступ до ресурсів і реалізацію комунікаційних процесів.

Комп'ютерні мережі класифікують за масштабом, функціональним призначенням і топологією. До найпоширеніших типів належать:

1. Локальна обчислювальна мережа (Local Area Network, LAN). Локальні мережі охоплюють відносно невелику географічну територію,

наприклад одну будівлю або кампус. Для них характерні високі швидкості передавання даних і малі затримки. Пристрої в межах LAN можуть безпосередньо взаємодіяти між собою, зазвичай із використанням технологій Ethernet або бездротового доступу. Основною перевагою локальних мереж є можливість ефективного спільного використання ресурсів, таких як принтери чи локальні сховища даних.

2. Глобальна обчислювальна мережа (Wide Area Network, WAN). Глобальні мережі охоплюють значні географічні простори та можуть використовувати супутникові канали зв'язку, орендовані телекомунікаційні лінії або міжнародні мережі. WAN забезпечують з'єднання між містами, країнами й навіть континентами. Класичним прикладом глобальної мережі є мережа Інтернет, яка об'єднує розрізнені мережі в усьому світі.

3. Міська обчислювальна мережа (Metropolitan Area Network, MAN). Міські мережі охоплюють територію міста або великого кампусу й зазвичай базуються на волоконно-оптичних каналах зв'язку для надання високошвидкісних сервісів доступу до Інтернету. За масштабом MAN займають проміжне положення між LAN і WAN та широко застосовуються для організації широкосмугового доступу в межах населених пунктів.

4. Персональна обчислювальна мережа (Personal Area Network, PAN). Персональні мережі призначені для обміну даними на дуже малих відстанях, зокрема для з'єднання мобільних і персональних пристроїв. Зазвичай вони реалізуються з використанням технологій Bluetooth або USB і забезпечують швидкий та зручний обмін інформацією між смартфонами, планшетами та ноутбуками.

5. Бездротові мережі (Wireless LAN і Wireless WAN). Бездротові мережі використовують радіотехнології зв'язку, зокрема Wi-Fi та стільникові мережі передачі даних. Вони можуть відповідати різним масштабам мереж – від локальних до глобальних – і забезпечують гнучкість та мобільність, що є критично важливими характеристиками в умовах сучасного орієнтованого на мобільність цифрового середовища.

Функціональна комп'ютерна мережа складається з низки критично важливих компонентів, кожен з яких відіграє визначальну роль у забезпеченні ефективної комунікації та обміну даними. До основних компонентів належать:

Мережеві інтерфейсні карти (Network Interface Cards, NIC). Ці апаратні компоненти забезпечують підключення пристроїв до мережі. Мережева інтерфейсна карта слугує інтерфейсом між материнською платою пристрою та мережевим середовищем і зазвичай підтримує як дротові (Ethernet), так і бездротові (Wi-Fi) способи з'єднання.

Комутатори та концентратори (Switches and Hubs). Ці пристрої забезпечують передавання даних у межах мережі шляхом з'єднання кількох вузлів. Концентратори транслюють отримані дані на всі підключені пристрої незалежно від адреси призначення, тоді як комутатори здійснюють інтелектуальне спрямування трафіка до конкретних пристроїв на основі їх MAC-адрес.

Маршрутизатори (Routers) забезпечують з'єднання між кількома мережами та керують передаванням пакетів даних між ними, вибираючи оптимальні маршрути. Вони функціонують на мережевому рівні моделі OSI та використовують IP-адреси для визначення шляхів передавання даних і керування трафіком у мережі Інтернет.

Середовища передавання даних (Communication Media). До середовищ передавання належать усі типи фізичних і бездротових носіїв, якими передаються дані. Дротовий зв'язок реалізується з використанням мідних кабелів, волоконно-оптичних ліній або коаксіальних кабелів. Бездротовий зв'язок базується на використанні радіохвиль, мікрохвиль і інфрачервоного випромінювання.

Протоколи обміну даними (Communication Protocols) являють собою формалізовані набори правил, що регламентують обмін даними між мережевими пристроями. До найважливіших належить стек протоколів Transmission Control Protocol / Internet Protocol (TCP/IP), який визначає принципи пакетування, адресації, передавання, маршрутизації та приймання даних у мережі Інтернет.

Топологія мережі визначає просторову структуру та спосіб з'єднання її вузлів, відображаючи, яким чином пристрої взаємопов'язані між собою. Найпоширенішими топологіями є такі.

Шинна топологія (Bus Topology). Усі мережеві пристрої під'єднані до одного центрального кабелю. Недоліком цієї топології є висока ймовірність колізій даних, що призводить до зниження продуктивності зі збільшенням кількості вузлів.

Зіркова топологія (Star Topology). Кожен пристрій безпосередньо під'єднаний до центрального вузла, зазвичай комутатора або концентратора. Вихід з ладу одного з'єднання не впливає на роботу інших, що робить таку топологію надійною та масштабованою.

Кільцева топологія (Ring Topology). Пристрої з'єднані в замкнене кільце, де кожен вузол має зв'язок із двома сусідніми. Така топологія дозволяє ефективно керувати колізіями, однак є вразливою до відмов, оскільки пошкодження одного елемента може призвести до зупинки всієї мережі.

Сіткова топологія (Mesh Topology). Кожен мережевий пристрій з'єднаний із кількома іншими вузлами. Така структура забезпечує високу надійність і відмовостійкість завдяки надлишковості шляхів передавання даних.

Гібридна топологія (Hybrid Topology). Гібридні топології поєднують елементи кількох базових топологій, що дозволяє адаптувати мережеву структуру до конкретних вимог і поєднати переваги різних конфігурацій.

Протоколи відіграють ключову роль у комп'ютерних мережах, оскільки забезпечують стандартизацію процесів обміну даними, визначаючи формати повідомлень, процедури передавання, механізми виявлення та виправлення помилок. Вони функціонують на різних рівнях моделей OSI та TCP/IP, забезпечуючи сумісність і ефективну взаємодію між різними застосунками та мережами.

Модель OSI містить такі рівні.

Фізичний рівень – відповідає за передавання та приймання необроблених бітових потоків через фізичне середовище.

Канальний рівень – забезпечує передавання даних між безпосередньо з'єднаними вузлами та виконує виявлення й корекцію помилок фізичного рівня.

Мережевий рівень – відповідає за визначення маршрутів, маршрутизацію та пересилання пакетів.

Транспортний рівень – гарантує надійне передавання даних, сегментацію та контроль помилок.

Сеансовий рівень – встановлює, підтримує та завершує сеанси зв'язку між застосунками.

Рівень подання – виконує перетворення форматів даних, шифрування та стиснення.

Прикладний рівень – забезпечує безпосередню взаємодію з прикладними програмами та надання мережевих сервісів користувачеві.

Модель TCP/IP є спрощеним і практично орієнтованим набором протоколів, що лежить в основі функціонування сучасних цифрових мереж і мережі Інтернет зокрема. Вона складається з таких рівнів:

Рівень доступу до мережі (Link Layer) – охоплює функції фізичного та канального рівнів моделі OSI.

Мережевий рівень (Internet Layer) – відповідає мережевому рівню OSI та забезпечує маршрутизацію й адресацію.

Транспортний рівень (Transport Layer) – реалізує основні функції передавання даних, аналогічні транспортному рівню OSI.

Прикладний рівень (Application Layer) – об'єднує функції прикладного, сеансового та рівня подання моделі OSI.

У межах цих моделей протоколи TCP, IP, HTTP, HTTPS, FTP і DNS демонструють різноманітні підходи до забезпечення надійної та ефективної мережевої взаємодії.

Протокол TCP реалізує орієнтований на з'єднання механізм передавання даних. Далі буде розглянуто приклад реалізації базового TCP-клієнта мовою C++ з використанням інтерфейсу сокетного програмування.

```
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>

int main() {
    // Створення TCP-сокета
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket < 0) {
        std::cerr << "Помилка створення сокета" << std::endl;
        return -1;
    }
```

```
        // Опис адреси сервера
        sockaddr_in serverAddress;
        serverAddress.sin_family = AF_INET;         // Використання IPv4
        serverAddress.sin_port = htons(8080);       // Номер порту сервера

        // Перетворення IP-адреси з текстового у бінарний формат
        if (inet_pton(AF_INET, "192.168.1.1", &serverAddress.sin_addr) <= 0)
{
            std::cerr << "Невірна IP-адреса або адреса не підтримується" <<
std::endl;
            return -1;
        }

        // Встановлення з’єднання з сервером
        if (connect(clientSocket,
                (sockaddr*)&serverAddress,
                sizeof(serverAddress)) < 0) {
            std::cerr << "Не вдалося встановити з’єднання" << std::endl;
            return -1;
        }

        // Передавання повідомлення серверу
        const char *message = "Hello Server";
        send(clientSocket, message, strlen(message), 0);

        // Закриття сокета
        close(clientSocket);
        return 0;
    }
```

Наведений програмний код демонструє базову реалізацію TCP-клієнта, який встановлює з’єднання із сервером за заданою IP-адресою та портом і передає тестове повідомлення. Підключаються стандартні бібліотеки введення-виведення, а також системні бібліотеки для роботи з сокетами, мережевими адресами та системними викликами.

За допомогою функції socket() створюється сокет типу SOCK_STREAM, що відповідає протоколу TCP. Структура sockaddr_in використовується для задання параметрів сервера: типу адресації (IPv4), номера порту та IP-адреси. Функція inet_pton() перетворює текстове подання IP-адреси у бінарний формат, придатний для мережевих операцій. З’єднання з сервером встановлюється за допомогою функції connect(). Після успішного встановлення з’єднання клієнт надсилає серверу текстове повідомлення за допомогою функції send(). Після завершення передавання даних сокет коректно закривається викликом close().

Наведений приклад демонструє процес встановлення TCP-з’єднання, перетворення IP-адреси та передавання даних на сервер, акцентуючи увагу на практичних аспектах мережевого програмування мовою C++.

Комп’ютерні мережі суттєво розширюють можливості обміну даними та спільного використання ресурсів, що робить їх невід’ємною складовою сучасних організацій і повсякденної діяльності користувачів. Забезпечуючи електронну комунікацію, підтримку віддаленої роботи та функціонування платформ онлайн-навчання, мережі становлять основу цифрового суспільства, поєднуючи різноманітні системи й користувачів у глобальному масштабі з безпрецедентним рівнем ефективності.

1.2 Основи мережевої комунікації

Мережева комунікація є ключовим чинником об’єднання різнорідних інформаційних систем і забезпечує ефективний обмін даними між ними в межах різних типів мереж. Принципи мережевої комунікації ґрунтуються на узгодженій взаємодії протоколів, апаратних засобів і програмних механізмів, які спільно забезпечують передавання інформації від одного вузла до іншого. Ґрунтовне розуміння цих основ є необхідною передумовою для розроблення високоефективних мережевих застосунків і підтримання безперервного та надійного потоку даних у мережах.

У центрі мережевої комунікації перебуває процес передавання даних між вузлами-відправниками та вузлами-одержувачами. Для формалізації та реалізації цього процесу використовуються моделі комунікації, зокрема модель OSI та модель TCP/IP, які пропонують структурований підхід до аналізу й упровадження мережевих взаємодій. Зазначені моделі поділяють функціональні можливості мережі на окремі рівні, кожен з яких відповідає за виконання певних операцій, сукупність яких забезпечує передавання даних.

1. Модель взаємодії відкритих систем (Open Systems Interconnection, OSI) є концептуальною основою для опису мережевих процесів і складається із семи рівнів.

Фізичний рівень – забезпечує фізичне з’єднання між пристроями та відповідає за передавання й приймання необроблених бінарних даних через фізичне середовище. До цього рівня належать специфікації апаратних компонентів, зокрема кабелів, комутаторів і мережевих інтерфейсних карт.

Канальний рівень – забезпечує надійне передавання даних між безпосередньо з’єднаними вузлами, а також виконує функції контролю помилок і синхронізації кадрів. На цьому рівні функціонують протоколи Ethernet і PPP.

Мережевий рівень – відповідає за пересилання пакетів шляхом маршрутизації, адресації та керування перевантаженнями мережі. До цього рівня належить, зокрема, протокол Internet Protocol (IP).

Транспортний рівень – забезпечує надійність кінцевого передавання даних і цілісність інформації завдяки механізмам керування потоком, сегментації та виправлення помилок. На цьому рівні працюють протоколи TCP і UDP.

Сеансовий рівень – керує встановленням, підтриманням і завершенням сеансів взаємодії між прикладними програмами, а також забезпечує синхронізацію діалогу між ними.

Рівень подання – виконує перетворення форматів даних, їх шифрування та стиснення, забезпечуючи узгодженість між мережевими та прикладними форматами.

Прикладний рівень – безпосередньо взаємодіє з прикладним програмним забезпеченням, надаючи мережеві сервіси, зокрема передавання файлів, електронну пошту та доступ до вебресурсів.

2. Модель Transmission Control Protocol / Internet Protocol (TCP/IP) є спрощеною та практично орієнтованою моделлю, що набула значного поширення в мережах Інтернет. У цій моделі функції мережевої взаємодії згруповані в чотири рівні.

Рівень доступу до мережі (Link Layer) – об'єднує функції фізичного та канального рівнів моделі OSI й відповідає за передавання даних між мережевими інтерфейсами.

Мережевий рівень (Internet Layer) – аналогічний мережевому рівню OSI та забезпечує адресацію і маршрутизацію пакетів між мережами.

Транспортний рівень (Transport Layer) – надає сервіси передавання даних із підтримкою надійності, керування помилками та потоком, відповідно до функцій транспортного рівня OSI.

Прикладний рівень (Application Layer) – охоплює всі протоколи верхнього рівня, що забезпечують функціонування інтернет-застосунків, і відповідає прикладному рівню моделі OSI.

Мережева комунікація охоплює низку ключових компонентів, необхідних для встановлення та підтримання ефективного обміну даними:

Кінцеві системи, до яких зазвичай належать користувацькі пристрої (комп'ютери, смартфони, планшети), слугують точками мережевої взаємодії. Термін «хост» зазвичай використовується для позначення пристроїв, що надають мережеві сервіси, зокрема серверів. Поняття вузол є узагальненим і застосовується до будь-якого пристрою, підключеного до мережі.

Комутатори функціонують на канальному рівні моделі OSI та забезпечують локальну мережеву взаємодію, приймаючи вхідні пакети даних і спрямовуючи їх безпосередньо до вузла призначення. Маршрутизатори працюють на мережевому рівні, здійснюючи передавання пакетів між різними мережами на основі IP-адрес та використовуючи алгоритми й протоколи маршрутизації для ефективного керування мережевим трафіком.

Мережева комунікація підтримується різними середовищами передавання, які поділяються на дротові та бездротові. До дротових належать виті пари, коаксіальні кабелі та волоконно-оптичні лінії зв'язку.

Бездротові середовища реалізуються з використанням радіохвиль і мікрохвиль, що забезпечують мобільність і гнучкість мережевих з'єднань.

Протоколи визначають правила та домовленості, відповідно до яких здійснюється обмін даними між мережевими пристроями. Ключовими протоколами транспортного рівня є TCP і UDP. Протокол TCP забезпечує надійний, орієнтований на з'єднання, обмін даними, гарантуючи доставлення пакетів без помилок і в правильній послідовності. Протокол UDP, навпаки, реалізує швидший, неорієнтований на з'єднання механізм передавання, який доцільно використовувати в застосунках, де пріоритет надається швидкодії, а не надійності, наприклад під час потокової трансляції мультимедійних даних.

Клієнт-серверна модель є базовою архітектурою мережевої взаємодії, у межах якої клієнти ініціюють запити на отримання ресурсів або сервісів, а сервери обробляють ці запити та надають відповідні відповіді. Саме ця модель лежить в основі більшості сучасних мережевих застосунків, зокрема вебсистем, електронної пошти та баз даних.

Клієнт являє собою обчислювальний пристрій або програмний застосунок, який ініціює запит на отримання мережевих сервісів чи ресурсів. Клієнт виступає активною стороною взаємодії, отримуючи дані або виконуючи операції на основі відповідей сервера.

Сервери – це потужні обчислювальні системи або програмні застосунки, призначені для надання сервісів у мережі. Вони здатні одночасно обслуговувати велику кількість клієнтів, зберігаючи ресурси та забезпечуючи віддалений доступ до даних, програм і функціональних можливостей.

З огляду на те, що клієнт-серверна архітектура визначає виконання багатьох критично важливих операцій у сучасних мережах, важливо розуміти її практичні реалізації, зокрема використання протоколу HTTP у вебзастосунках або FTP для передавання файлів. Нижче буде розглянуто приклад програмного коду мовою C++, який ілюструє базову конфігурацію TCP-сервера з використанням сокетного програмування.

```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>

int main() {
    // Створення TCP-сокета
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0) {
        std::cerr << "Помилка створення сокета" << std::endl;
        return -1;
    }

    // Налаштування адреси сервера
```

```
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;          // IPv4
    serverAddress.sin_addr.s_addr = INADDR_ANY;   // Прийом з'єднань
з усіх інтерфейсів
    serverAddress.sin_port = htons(8080);        // Номер порту

    // Прив’язка сокета до адреси та порту
    if (bind(serverSocket,
        (sockaddr*)&serverAddress,
        sizeof(serverAddress)) < 0) {
      std::cerr << "Помилка прив’язки сокета" << std::endl;
      return -1;
    }

    // Перехід у режим очікування клієнтських з’єднань
    if (listen(serverSocket, 3) < 0) {
      std::cerr << "Помилка прослуховування" << std::endl;
      return -1;
    }

    std::cout << "Сервер очікує на підключення клієнтів..." << std::endl;

    // Прийом з’єднання від клієнта
    int clientSocket;
    sockaddr_in clientAddress;
    socklen_t clientLength = sizeof(clientAddress);

    clientSocket = accept(serverSocket,
                  (sockaddr*)&clientAddress,
                  &clientLength);
    if (clientSocket < 0) {
      std::cerr << "Помилка приймання клієнта" << std::endl;
      return -1;
    }

    // Приймання повідомлення від клієнта
    char buffer[1024] = {0};
    read(clientSocket, buffer, 1024);
    std::cout << "Отримано повідомлення: " << buffer << std::endl;

    // Надсилання відповіді клієнту
    const char* response = "Hello from server";
    send(clientSocket, response, strlen(response), 0);

    // Закриття сокетів
    close(clientSocket);
    close(serverSocket);
```

```
    return 0;
}
```

Наведений приклад демонструє типову послідовність дій під час створення TCP-сервера, який приймає вхідні з'єднання від клієнтів, отримує дані та надсилає відповідь.

За допомогою функції socket() створюється серверний сокет, орієнтований на встановлення надійного з'єднання за протоколом TCP. Структура sockaddr_in використовується для задання параметрів сервера, зокрема типу адресації (IPv4), IP-адреси та номера порту. Функція bind() прив'язує створений сокет до конкретної адреси й порту, що дозволяє серверу приймати з'єднання на певному мережевому інтерфейсі. Виклик listen() переводить сокет у режим очікування вхідних з'єднань, формуючи чергу запитів від клієнтів. Функція accept() використовується для приймання клієнтського з'єднання та створення окремого сокета для взаємодії з конкретним клієнтом.

Після встановлення з'єднання сервер зчитує дані, надіслані клієнтом, та виводить отримане повідомлення. У відповідь сервер надсилає клієнтові текстове повідомлення. Після завершення обміну даними клієнтський і серверний сокети коректно закриваються.

Розглянутий приклад ілюструє класичну реалізацію клієнт-серверної взаємодії на основі протоколу TCP/IP. Він демонструє ключові етапи серверного сокетного програмування: створення та конфігурування сокета, організацію прослуховування порту, приймання клієнтських з'єднань, обмін даними та коректне завершення сеансу зв'язку. Наведений код є базовою моделлю TCP-сервера і може бути використаний як основа для розроблення більш складних мережевих застосунків із підтримкою багатокористувацької взаємодії та розширених механізмів оброблення запитів.

На відміну від клієнт-серверної моделі, однорангові мережі (P2P) передбачають рівноправний розподіл мережевих ролей між усіма учасниками. Кожен вузол, або peer, у такій мережі має однакові можливості та обов'язки, що забезпечує безпосередній обмін ресурсами й інформацією без використання централізованого сервера.

Ця модель широко застосовується в блокчейн-системах і застосунках для обміну файлами (наприклад, BitTorrent), де зберігання та поширення даних здійснюється децентралізовано. Однорангові мережі підвищують стійкість і доступність систем за рахунок розподілу функцій і використання сукупних ресурсів усіх вузлів. Водночас вони створюють специфічні виклики, пов'язані з забезпеченням цілісності даних і рівня інформаційної безпеки.

Процеси мережевої комунікації супроводжуються низкою проблем, які необхідно враховувати для забезпечення ефективності, надійності та безпеки обміну даними:

Затримка (Latency) – характеризує часові затримки під час оброблення або передавання даних. Підвищена затримка негативно

впливає на продуктивність застосунків, особливо в системах реального часу.

Пропускна здатність (Bandwidth) – визначає максимальний обсяг даних, який може бути переданий через мережеве з'єднання за одиницю часу. Обмежена пропускна здатність призводить до перевантаження мережі та зниження швидкості передавання даних.

Надійність (Reliability) – відображає стабільність і передбачуваність процесів передавання даних у мережі. Забезпечення надійності передбачає мінімізацію втрат пакетів, усунення розривів з'єднання та запобігання пошкодженню даних.

Безпека (Security) – полягає в захисті інформації під час її передавання мережею. Для цього застосовуються методи шифрування, автентифікації та контролю доступу, які запобігають перехопленню, модифікації або несанкціонованому доступу до даних.

Суть мережевої комунікації полягає у здатності швидко та надійно передавати інформацію між різними об'єктами та через різні середовища передавання. Будучи основою сучасних цифрових екосистем, механізми мережевої комунікації формують підґрунтя для синхронізації роботи різнорідних систем і забезпечують ефективну взаємодію користувачів на значних відстанях. Розуміння цих фундаментальних принципів сприяє створенню програмних застосунків, здатних повною мірою використовувати можливості мереж, одночасно враховуючи проблеми затримок, пропускної здатності та безпеки в умовах зростаючої глобальної взаємопов'язаності.

1.3 Основна мережева термінологія

Чітке розуміння мережевої термінології є необхідною умовою для усвідомлення принципів взаємодії мережевих систем та їх узгодженого функціонування. Мережева термінологія охоплює базові поняття, механізми й компоненти, що забезпечують обмін даними в комп'ютерних мережах. У цьому підрозділі розглядаються ключові мережеві терміни та концепції, що надають пояснення як простих, так і складних процесів мережевої взаємодії.

IP-адреси – це числові ідентифікатори, які призначаються кожному пристрою, підключеному до комп'ютерної мережі, що використовує протокол Internet Protocol для обміну даними. Вони виконують дві основні функції: ідентифікацію вузлів або мережевих інтерфейсів та визначення їх місцезнаходження в мережі. Розрізняють два основні стандарти IP-адресації:

1. IPv4 – найбільш поширена версія протоколу, що використовує 32-бітну схему адресації та забезпечує приблизно 4,3 мільярда унікальних адрес. Такі адреси зазвичай подаються у десятково-крапковому форматі, поділеному на чотири октети, наприклад: `192.168.1.1`.

2. IPv6 – розроблений у відповідь на стрімке зростання кількості підключених до Інтернету пристроїв і вичерпання адресного простору IPv4. Цей стандарт застосовує 128-бітну схему адресації, що забезпечує

експоненційно більшу кількість унікальних адрес. IPv6-адреси записуються у шістнадцятковому форматі та розділяються двокрапками, наприклад: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`.

IP-адреси також використовують механізм маскування підмереж, який застосовується для поділу мережі на менші підмережі з метою підвищення ефективності маршрутизації та оптимізації розподілу адресного простору.

У мережевих технологіях порти та сокети є ключовими елементами, що забезпечують можливість одночасного встановлення декількох сеансів зв'язку між різними застосунками та пристроями.

Порти – це числові ідентифікатори в межах IP-адреси, які визначають конкретні процеси або служби. Вони дозволяють одному вузлу надавати декілька сервісів одночасно, кожен з яких прослуховує власний номер порту. Стандартні порти, такі як порт 80 для протоколу HTTP або порт 443 для HTTPS, забезпечують уніфікований доступ до відповідних мережевих служб.

Сокети – мережевий сокет є кінцевою точкою для надсилання або приймання даних у комп'ютерній мережі. Сокети використовують комбінацію IP-адреси та номера порту для встановлення двостороннього каналу зв'язку. Вони є базовим елементом сокетного програмування, забезпечуючи керування мережевими операціями, зокрема надсиланням, прийманням даних та розривом з'єднання.

Наведений нижче фрагмент коду мовою C++ демонструє приклад налаштування простого клієнтського сокета з використанням протоколу TCP.

```
#include <iostream>          // Забезпечує введення та виведення потоків (std::cout, std::cerr)
#include <sys/socket.h>     // Містить визначення функцій і структур для роботи з сокетами
#include <arpa/inet.h>       // Забезпечує функції для перетворення IP-адрес (inet_pton)
#include <unistd.h>         // Надає доступ до системних викликів POSIX (close, read)
#include <cstring>          // Містить функції для роботи з рядками (strlen)

int main() {

    // Створення клієнтського сокета:
    // AF_INET — використання IPv4,
    // SOCK_STREAM — потоковий сокет (TCP),
    // 0 — автоматичний вибір протоколу
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Перевірка успішності створення сокета
    if (clientSocket < 0) {
        std::cerr << "Socket creation failed" << std::endl;
        return -1;
```

```
    }

    // Опис структури мережевої адреси сервера
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;          // Вказується сімейство адрес IPv4
    serverAddress.sin_port = htons(8080);          // Номер порту у мережевому порядку байтів

    // Перетворення IP-адреси з текстового у бінарний формат
    // 127.0.0.1 — локальний хост (loopback)
    if (inet_pton(AF_INET, "127.0.0.1", &serverAddress.sin_addr) <= 0) {
       std::cerr << "Invalid IP address" << std::endl;
       return -1;
    }

    // Встановлення TCP-з'єднання з сервером
    if (connect(clientSocket,
            (struct sockaddr*)&serverAddress,
            sizeof(serverAddress)) < 0) {
       std::cerr << "Connection error" << std::endl;
       return -1;
    }

    // Буфер для приймання даних від сервера
    char buffer[1024] = {0};

    // Повідомлення, яке надсилається серверу
    const char* message = "Hello Server";

    // Надсилання даних серверу через встановлене TCP-з'єднання
    send(clientSocket, message, strlen(message), 0);

    // Приймання відповіді від сервера
    read(clientSocket, buffer, 1024);

    // Виведення отриманого повідомлення
    std::cout << "Server response: " << buffer << std::endl;

    // Закриття сокета та звільнення системних ресурсів
    close(clientSocket);

    return 0;
}
```

Тут клієнт встановлює з'єднання з локальним сервером на порту 8080, демонструючи використання сокетів для забезпечення мережевої

взаємодії. На початковому етапі виконується створення сокета за допомогою функції `socket()`, де вказується використання адресного сімейства IPv4 (AF_INET) та потокового типу сокета SOCK_STREAM, що відповідає протоколу TCP. У разі помилки створення сокета програма коректно завершує виконання з виведенням діагностичного повідомлення.

Далі ініціалізується структура `sockaddr_in`, яка містить параметри мережевої адреси сервера: тип адреси, номер порту (перетворений у мережевий порядок байтів за допомогою `htons`) та IP-адресу. Функція `inet_pton()` виконує перетворення IP-адреси з текстового подання у бінарний формат, придатний для мережевої взаємодії.

Після коректної підготовки адреси відбувається встановлення з'єднання з сервером за допомогою функції `connect()`. Успішне виконання цієї операції означає створення надійного двостороннього каналу передавання даних між клієнтом і сервером.

У процесі обміну даними клієнт надсилає серверу текстове повідомлення за допомогою функції `send()`, після чого очікує відповідь, яка зчитується у буфер викликом `read()`. Отримане повідомлення виводиться на екран, що підтверджує коректність встановленого з'єднання та обміну інформацією.

Завершальним етапом є виклик функції `close()`, яка забезпечує закриття сокета та звільнення системних ресурсів. Таким чином, наведений приклад наочно ілюструє базові принципи клієнт-серверної взаємодії, використання IP-адрес, портів і сокетів, а також практичну реалізацію транспортного рівня моделі TCP/IP мовою C++.

Такий анотований код демонструє базову реалізацію TCP-клієнта, що встановлює з'єднання з сервером, здійснює двосторонній обмін даними та коректно завершує сеанс зв'язку. Приклад є типовим для вивчення сокетного програмування, принципів клієнт-серверної архітектури та практичної реалізації транспортного рівня моделі TCP/IP у навчальних і лабораторних роботах.

Протоколи являють собою формалізовані набори правил, що забезпечують ефективну взаємодію між мережевими пристроями. До основних мережевих протоколів належать:

TCP/IP (Transmission Control Protocol / Internet Protocol). Цей набір протоколів є базовим для мережевих комунікацій в Інтернеті. Протокол TCP відповідає за надійну та впорядковану доставку даних, тоді як IP забезпечує маршрутизацію пакетів і адресацію. У сукупності вони формують універсальну основу передавання даних у різнорідних мережах.

UDP (User Datagram Protocol) забезпечує передачу даних без встановлення з'єднання між вузлами мережі. Він доцільний для застосунків, у яких пріоритетом є швидкість передавання, а не гарантована доставка, зокрема для потокового мовлення та голосових сервісів VoIP.

HTTP/HTTPS (Hypertext Transfer Protocol / Secure) регламентує спосіб передавання інформації у вебсередовищі. HTTPS є розширенням HTTP і забезпечує захищений обмін даними шляхом використання криптографічних механізмів (SSL/TLS), що гарантують цілісність і конфіденційність інформації.

FTP (File Transfer Protocol) призначений для передавання файлів у мережі, забезпечуючи їх завантаження та вивантаження. Протокол використовує окремі керувальні та інформаційні з'єднання між клієнтом і сервером і, як правило, потребує автентифікації користувачів.

SMTP (Simple Mail Transfer Protocol) є ключовим протоколом електронної пошти, який відповідає за передавання повідомлень від відправника до поштових серверів отримувачів. Для підвищення безпеки обміну часто застосовуються механізми шифрування.

Мережевий інтерфейс може бути апаратним (наприклад, мережева карта) або програмним компонентом, що забезпечує взаємодію комп'ютера з мережею. Він керує вхідним і вихідним мережевим трафіком, використовуючи фізичні та логічні адреси.

MAC-адреси (Media Access Control) є унікальними ідентифікаторами мережевих інтерфейсів, що використовуються на канальному рівні моделі OSI. Вони мають формат шести пар шістнадцяткових чисел, наприклад 00:1A:2B:3C:4D:5E, і є апаратними ідентифікаторами, призначеними виробником мережевого обладнання.

Система доменних імен призначена для перетворення зрозумілих людині доменних імен (наприклад, www.example.com) у числові IP-адреси, що використовуються мережевою інфраструктурою для маршрутизації та передавання даних.

DNS має ієрархічну структуру, яка починається з кореневих серверів, далі містить домени верхнього рівня (.com, .org, .net), домени другого рівня та конкретні імена вузлів. Процес DNS-резолюції полягає у перетворенні доменного імені на відповідну IP-адресу. Він може залучати проміжні елементи, зокрема рекурсивні резолвери та авторитетні сервери імен. Трансляція мережевих адрес дозволяє кільком пристроям локальної мережі використовувати одну публічну IP-адресу для доступу до Інтернету. NAT сприяє економії адресного простору та підвищує рівень безпеки, приховуючи внутрішні IP-адреси від зовнішніх мереж. Типовим прикладом використання NAT є домашні маршрутизатори, які транслюють трафік з приватних IP-адрес локальної мережі у публічну адресу провайдера та перенаправляють вхідні пакети до відповідних внутрішніх вузлів.

Міжмережеві екрани (firewalls) є системами мережевої безпеки, що запобігають несанкціонованому доступу, водночас дозволяючи легітимний обмін даними. Вони можуть бути апаратними або програмними та фільтрують трафік відповідно до заданих правил безпеки.

До основних протоколів безпеки належать:

SSL/TLS (Secure Sockets Layer / Transport Layer Security). Протоколи, що забезпечують захищене й зашифроване з'єднання між мережевими вузлами. Вони широко застосовуються у HTTPS для гарантування конфіденційності та цілісності даних.

VPN (Virtual Private Network). Технологія створення захищеного, зашифрованого каналу зв'язку поверх незахищених мереж, зокрема

Інтернету. VPN забезпечує безпечний віддалений доступ до приватних мереж і захист інформації під час передавання.

Розуміння наведених термінів і концепцій розкриває складність мережевої архітектури та формує базу знань, необхідну для опанування більш складних питань комп'ютерних мереж, а також для вирішення практичних завдань сумісності систем і управління мережевою безпекою.

1.4 Роль мови C++ у мережевому програмуванні

Мова програмування C++ відіграє важливу та визначальну роль у сфері мережевого програмування, поєднуючи високу продуктивність, універсальність і детальний контроль над системними ресурсами. Завдяки здатності забезпечувати як низькорівневий доступ до системи, так і розвинені засоби високорівневого програмування, C++ є пріоритетним вибором для розроблення надійних і ресурсоефективних мережевих застосунків. Її застосування охоплює широкий спектр галузей – від системного програмного забезпечення до масштабних мережевих сервісів і телекомунікаційної інфраструктури.

C++ відома своєю високою продуктивністю, що є критично важливим чинником у мережевому програмуванні, де ключову роль відіграють затримки та пропускна здатність. Оптимізації компілятора та можливість низькорівневого доступу до апаратних і системних ресурсів дають змогу розробникам тонко налаштовувати застосунки для досягнення максимальної швидкодії та мінімального споживання ресурсів.

Стандартна бібліотека C++ і Стандартна шаблонна бібліотека (STL) надають багатий набір інструментів і структур даних, що забезпечують ефективне управління ресурсами та виконання операцій у мережевому програмуванні. Контейнерні класи, алгоритми та ітератори STL підтримують складні операції оброблення даних, характерні для мережевих процесів. C++ реалізує потужні засоби об'єктно-орієнтованого програмування, які забезпечують модульність, повторне використання та масштабованість мережевих застосунків. Інкапсуляція, успадкування та поліморфізм дозволяють проєктувати складні мережеві архітектури з використанням впорядкованих класів та інтерфейсів. За допомогою C++ розробники можуть безпосередньо взаємодіяти з ресурсами операційної системи через системні виклики, керуючи сокетами, потоками виконання та файловими дескрипторами. Це є критично важливим для таких завдань, як сокетне програмування, де потрібне точне керування пакетами даних і мережевими з'єднаннями. C++ підтримує кросплатформну розробку, що є надзвичайно цінною властивістю для створення мережевих застосунків, призначених для роботи в різних апаратних і програмних середовищах. Бібліотеки, зокрема Boost і Qt, розширюють можливості мови C++, забезпечуючи портативність коду на таких платформах, як Windows, Linux і macOS. C++ має велику кількість сторонніх бібліотек, які суттєво спрощують мережеве програмування. Вони надають вищі рівні абстракції, засоби безпеки та механізми підключення, що сприяє швидкому розробленню мережевих застосунків. Бібліотеки Boost.Asio, Poco та cpp-

netlib підтримують асинхронні операції введення/виведення, реалізацію HTTP-клієнтів і серверів, а також роботу з додатковими мережевими протоколами.

Сокетне програмування є фундаментом мережевого програмування в C++ і передбачає створення, налаштування та керування мережевими сокетами для забезпечення обміну даними між кінцевими точками. Використовуючи API сокетів Berkeley, мова C++ забезпечує реалізацію мережевої взаємодії на основі протоколів TCP та UDP, що дозволяє створювати як надійні з’єднання з контролем доставки, так і високопродуктивні сервіси без встановлення з'єднання.

```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>

int createServerSocket(int port) {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0) {
        std::cerr << "Failed to create socket" << std::endl;
        exit(EXIT_FAILURE);
    }

    sockaddr_in serverAddress{};
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(port);

    if (bind(serverSocket, (struct sockaddr*)&serverAddress,
sizeof(serverAddress)) < 0) {
        std::cerr << "Bind failed" << std::endl;
        close(serverSocket);
        exit(EXIT_FAILURE);
    }

    return serverSocket;
}

void startListening(int serverSocket, int maxClients) {
    if (listen(serverSocket, maxClients) < 0) {
        std::cerr << "Listen error" << std::endl;
        close(serverSocket);
        exit(EXIT_FAILURE);
    }
    std::cout << "Server is listening..." << std::endl;
}
```

```
int main() {
    int port = 8080;
    int serverSocket = createServerSocket(port);
    startListening(serverSocket, 5); // черга до 5 клієнтів

    while (true) {
        sockaddr_in clientAddress{};
        socklen_t clientLen = sizeof(clientAddress);

        // Прийом підключення клієнта
        int clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddress, &clientLen);
        if (clientSocket < 0) {
            std::cerr << "Failed to accept client" << std::endl;
            continue;
        }

        std::cout << "Client connected!" << std::endl;

        // Надсилання вітального повідомлення клієнту
        const char* message = "Hello from server!";
        send(clientSocket, message, strlen(message), 0);

        // Закриття підключення (мінімальна обробка)
        close(clientSocket);
    }

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

Сервер створює TCP-сокет (socket), прив’язує його до IP і порту (bind) та переводить у режим прослуховування (listen). У нескінченному циклі сервер приймає підключення клієнтів (accept), надсилає привітальне повідомлення (send) і закриває клієнтський сокет (close). Після завершення роботи закривається і серверний сокет. Цей фрагмент демонструє створення та прив’язку TCP-сокета сервера, готового до прийому вхідних підключень клієнтів.

Мультитредінг. Використання конструкцій мультитредінгу в C++, таких як бібліотека `std::thread` у стандарті C++11, підвищує продуктивність мережевих додатків за рахунок ефективної обробки кількох одночасних підключень. Мультитредінг дозволяє мережевим додаткам виконувати окремі завдання паралельно, покращуючи швидкодію та використання ресурсів.

```
#include <iostream>        // std::cout, std::cerr
#include <thread>          // std::thread
#include <sys/socket.h>    // socket(), bind(), listen(), accept()
#include <netinet/in.h>    // sockaddr_in, INADDR_ANY
#include <unistd.h>        // close()
#include <cstring>         // memset()
#include <cstdlib>         // exit(), EXIT_FAILURE

/**
 * Створення серверного TCP-сокета та прив’язка до заданого порту.
 */
int createServerSocket(int port) {
    // Створення TCP-сокета (IPv4, потоковий)
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0) {
        std::cerr << "Failed to create socket" << std::endl;
        exit(EXIT_FAILURE);
    }

    // Ініціалізація структури адреси сервера
    sockaddr_in serverAddress{};
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(port);

    // Прив’язка сокета до IP-адреси та порту
    if (bind(serverSocket,
             reinterpret_cast<sockaddr*>(&serverAddress),
             sizeof(serverAddress)) < 0) {
        std::cerr << "Bind failed" << std::endl;
        close(serverSocket);
        exit(EXIT_FAILURE);
    }

    return serverSocket;
}

/**
 * Переведення серверного сокета у режим прослуховування.
 */
void startListening(int serverSocket, int maxClients) {
    if (listen(serverSocket, maxClients) < 0) {
        std::cerr << "Listen error" << std::endl;
        close(serverSocket);
        exit(EXIT_FAILURE);
    }
```

```
    std::cout << "Server is listening..." << std::endl;
}

/**
 * Обробка клієнтського підключення (виконується у окремому потоці).
 */
void handleClient(int clientSocket) {
    const char* message = "Hello from multithreaded server!\n";
    send(clientSocket, message, std::strlen(message), 0);

    // Завершення роботи з клієнтом
    close(clientSocket);
}

/**
 * Головна функція сервера.
 */
int main() {

    // Ініціалізація серверного сокета
    int serverSocket = createServerSocket(8080);

    // Перехід у режим очікування клієнтів
    startListening(serverSocket, 5);

    // Основний цикл обробки підключень
    while (true) {

        // Прийом клієнтського з’єднання
        int clientSocket = accept(serverSocket, nullptr, nullptr);

        if (clientSocket >= 0) {
            // Створення окремого потоку для клієнта
            std::thread clientThread(handleClient, clientSocket);

            // Відокремлення потоку для незалежного виконання
            clientThread.detach();
        } else {
            std::cerr << "Failed to accept connection" << std::endl;
        }
    }

    // Закриття серверного сокета (логічно недосяжне)
    close(serverSocket);
    return 0;
}
```

На початку програми підключено стандартні та системні бібліотеки, необхідні для реалізації мережевого сервера:

`<iostream>` – забезпечує механізми стандартного введення/виведення для формування діагностичних та інформаційних повідомлень.

`<thread>` – надає засоби багатопоточності стандарту C++11, зокрема клас `std::thread` для паралельної обробки клієнтів.

`<sys/socket.h>` – містить оголошення функцій низькорівневого сокетного API (створення, прив'язка, прослуховування та прийом з'єднань).

`<netinet/in.h>` – визначає структури та константи для роботи з IPv4-адресацією.

`<unistd.h>` – надає доступ до системних викликів POSIX, зокрема функції `close()` для звільнення ресурсів.

`<cstring>` – використовується для роботи з C-рядками та їх довжиною.

`<cstdlib>` – містить механізми аварійного завершення програми у разі критичних помилок.

Функція `createServerSocket()` реалізує початковий етап розгортання мережевого сервера. У її межах:

- створюється TCP-сокет із використанням протоколу IPv4;
- перевіряється коректність створення сокета та обробляється можливий збій;
- формується структура адреси сервера, яка визначає сімейство адрес, порт і допустимі мережеві інтерфейси;
- здійснюється прив'язка сокета до локальної IP-адреси та порту.

Таким чином, ця функція інкапсулює низькорівневу логіку ініціалізації серверного сокета і повертає його дескриптор для подальшого використання.

Функція `startListening()` відповідає за підготовку сервера до прийому вхідних підключень. Вона:

- переводить створений і прив'язаний сокет у режим прослуховування;
- задає максимальну кількість клієнтів, які можуть одночасно перебувати в черзі очікування;
- забезпечує базову обробку помилок на етапі запуску сервера.

Успішне виконання цієї функції означає, що сервер готовий до встановлення TCP-з'єднань.

Функція `handleClient()` реалізує мінімальний приклад взаємодії з клієнтом. Вона:

- надсилає клієнту тестове повідомлення, що підтверджує успішне підключення;
- завершує роботу з клієнтським сокетом, звільняючи системні ресурси.

Розміщення цієї логіки в окремій функції дозволяє легко масштабувати сервер, доповнюючи її складнішими алгоритмами обміну даними.

Функція `main()` координує роботу всіх складових серверного застосунку:

- ініціалізує серверний сокет на заданому порту;
- переводить сервер у режим очікування клієнтських підключень;
- організовує безперервний цикл прийому з'єднань.

Для кожного успішного підключення створюється окремий потік виконання, у якому здійснюється обробка клієнта. Використання відокремлених потоків (`detach`) дозволяє серверу обслуговувати кілька клієнтів паралельно, не блокуючи основний цикл прийому з'єднань.

Загалом програма реалізує багатопотоковий TCP-сервер, побудований на основі сокетного API та стандартних засобів багатопоточності C++. Такий підхід забезпечує:

- конкурентну обробку клієнтів;
- ефективне використання ресурсів процесора;
- масштабованість серверного застосунку.

Поданий код є базовим прикладом серверної архітектури та може слугувати відправною точкою для створення повнофункціональних мережевих сервісів. Тут сервер створює окремий потік виконання для кожного вхідного клієнтського підключення, що забезпечує паралельну обробку декількох клієнтів одночасно та підвищує масштабованість і відгукливість мережевого застосунку.

Мова програмування C++ завдяки своїй гнучкості та високій продуктивності дозволяє розробляти власні обробники протоколів і мережеві сервіси. У C++ можуть бути реалізовані механізми аналізу протокольних повідомлень (парсингу), серіалізації та десеріалізації даних, а також системи обробки помилок із точним контролем на рівні операційної системи.

Бібліотеки, наприклад Boost.Asio, забезпечують підтримку асинхронних операцій введення/виведення, що є критично важливим для неблокувальної мережевої взаємодії. У такому підході операції I/O не призупиняють виконання програми, що дозволяє зберігати високу пропускну здатність і ефективно використовувати обчислювальні ресурси системи.

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

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

/**
 * Асинхронний TCP-сервер на базі Boost.Asio
 */
void asyncHandler() {

    // У сучасних версіях Boost.Asio використовується io_context,
    // а не застарілий io_service
    boost::asio::io_context ioContext;
```

```
    // TCP-акцептор, що прослуховує порт 8080 на всіх інтерфейсах
    tcp::acceptor acceptor(
        ioContext,
        tcp::endpoint(tcp::v4(), 8080)
    );

    // Сокет для прийому клієнтського підключення
    tcp::socket socket(ioContext);

    /**
     * Асинхронний обробник прийому з'єднання.
     * Для async_accept(socket, handler) обробник
     * ПРИЙМАЄ ЛИШЕ error_code.
     */
    auto handleAccept =
        [&](const boost::system::error_code& errorCode)
        {
            if (!errorCode) {
                std::cout << "Accepted connection from: "
                          << socket.remote_endpoint().address()
                          << std::endl;
            } else {
                std::cerr << "Accept error: "
                          << errorCode.message()
                          << std::endl;
            }
        };

    // Ініціація асинхронного прийому клієнта
    acceptor.async_accept(socket, handleAccept);

    // Запуск циклу обробки асинхронних подій
    ioContext.run();
}

int main() {
    asyncHandler();
    return 0;
}
```

Наведений приклад демонструє ініціалізацію асинхронного TCP-сервера з використанням бібліотеки Boost.Asio, який забезпечує неблокувальне приймання клієнтських з'єднань у асинхронному режимі. Підключається бібліотека Boost.Asio для реалізації асинхронних мережевих операцій та стандартна бібліотека введення/виведення. Простір

імен `boost::asio::ip::tcp` використовується для зручного доступу до TCP-компонентів.

Об’єкт `io_context` є центральним елементом асинхронної моделі Boost.Asio та відповідає за керування чергою неблокувальних операцій і їх виконання.

Акцептор прив’язується до локального порту 8080 і переходить у стан очікування вхідних клієнтських з’єднань, виконуючи роль точки входу для нових TCP-сеансів.

Об’єкт `tcp::socket` використовується як кінцева точка для встановлення з’єднання з клієнтом у межах асинхронного прийому.

Лямбда-функція `handleAccept` реалізує зворотний виклик, який виконується після завершення операції прийому з’єднання. Вона аналізує код помилки та виводить інформацію про успішне підключення або повідомлення про помилку.

Виклик `async_accept` ініціює неблокувальне очікування клієнта, передаючи керування обробнику після встановлення TCP-з’єднання.

Метод `ioContext.run()` активує механізм виконання асинхронних операцій і забезпечує обробку всіх зареєстрованих подій.

У функції `main()` викликається процедура ініціалізації асинхронного сервера, після чого програма коректно завершує виконання.

Poco – набір C++-бібліотек, орієнтованих на розроблення мережево-центрованих та інтернет-застосунків, що містить реалізацію мережевих алгоритмів, операцій введення/виведення та протоколів прикладного рівня.

cpp-netlib широко застосовується для створення високопродуктивних мережевих серверів і клієнтів, забезпечуючи підтримку протоколів та надаючи високорівневі абстракції для мережевої взаємодії.

Модуль Qt Network – складова інструментарію Qt, що містить класи для роботи з TCP/IP, UDP, HTTP, SSL/TLS та іншими поширеними мережевими задачами.

Мова програмування C++ довела свою ефективність у широкому спектрі прикладних сценаріїв, зокрема в системах реального часу, ігрових серверах, пристроях Інтернету речей (IoT) та платформах високочастотної торгівлі, де критично важливою є стабільна та передбачувана продуктивність. Поєднання високої надійності та розвиненої екосистеми бібліотек забезпечує C++ статус базової технології мережевого програмування. Оволодіння C++ для мережевих застосувань надає довгострокові переваги, дозволяючи створювати масштабовані, ефективні та кросплатформні мережеві рішення.

1.5 Налаштування середовища розробки

Створення правильно налаштованого середовища розробки є фундаментальним кроком у мережевому програмуванні. Воно формує умови для ефективного написання коду, налагодження та розгортання мережевих застосунків. Надійне середовище забезпечує безшовну інтеграцію інструментів розробки, бібліотек і систем контролю версій, що є необхідним для колективної роботи та масштабованого процесу розробки

програмного забезпечення. Налаштування середовища для C++ у мережевому програмуванні передбачає ретельний підбір і конфігурацію компіляторів, інтегрованих середовищ розробки (IDE), бібліотек та тестових фреймворків.

Компілятор є основним компонентом середовища розробки C++, який перетворює високорівневий код на машинний, виконуваний системою. До найбільш поширених компіляторів, які використовуються в індустрії та навчальних закладах, належать:

GCC (GNU Compiler Collection): універсальний компілятор, що підтримує широкий спектр мов програмування, включно з C++. Є стандартним компілятором для багатьох Unix-подібних операційних систем та забезпечує високу відповідність стандартам C++. На системах Ubuntu установка виконується через пакетний менеджер командою:

```
sudo apt update
sudo apt install build-essential
```

Clang: розроблений проєктом LLVM, відомий швидкою компіляцією, детальними повідомленнями про помилки та сумісністю з GCC. Оптимальний для проєктів із потребою у додатковому аналізі коду або інтеграції компіляції з IDE, такими як Xcode. Установлення Clang на Ubuntu здійснюють командою:

```
sudo apt install clang
```

MSVC (Microsoft Visual C++): популярний для платформ Windows. Глибоко інтегрований з Visual Studio, забезпечує широкий набір бібліотек і засобів налагодження, а також підтримує управління проєктами та розгортання застосунків.

Intel C++ Compiler: відомий оптимізацією високопродуктивних застосунків, що підвищує швидкість виконання на процесорах Intel за рахунок спеціалізованих оптимізацій.

Інтегроване середовище розробки (IDE) поєднує редагування, компіляцію та налагодження в одному інтерфейсі, підвищуючи продуктивність за рахунок автоматизації повторюваних завдань і надання інструментів для управління кодом.

Visual Studio Code: легкий кросплатформений редактор з розширеннями для C++, наприклад, від Microsoft, що надає IntelliSense, налагодження та перегляд коду. Для мережевого програмування рекомендується інтегрувати розширення CMake і Git для управління версіями.

CLion: IDE від JetBrains, орієнтована на кросплатформну розробку C++. Забезпечує глибокий аналіз коду, розумне рефакторування та інструменти для всього проєкту. Інтеграція з CMake робить її зручною для роботи з різноманітними проєктами та залежностями.

Eclipse CDT: розширення Eclipse для розробки на C/C++, підтримує редагування, налагодження та інтеграцію з системами контролю версій.

Qt Creator: оптимальний для розробки застосунків на Qt, надає зручні інструменти для створення GUI, мережевого програмування та підтримує широкий спектр модулів і бібліотек Qt.

Для прискорення розробки мережевих застосунків рекомендується інтегрувати C++-бібліотеки, які забезпечують готовий функціонал для протоколів, асинхронних операцій та обробки даних.

Boost.Asio: призначена для створення масштабованих мережевих та низькорівневих I/O-застосунків. Підтримує синхронну та асинхронну взаємодію, легко комбінується з іншими бібліотеками Boost. Для установлення на Ubuntu:

```
sudo apt install libboost-all-dev
```

Poco C++ Libraries: надають можливості для мережевої взаємодії та доступу до даних, зручні для IoT та мережевих застосунків.

cpp-netlib: забезпечує реалізацію HTTP та інших протоколів, ефективна для веборієнтованих мережевих застосунків, підтримує обробку JSON та RESTful API.

Системи контролю версій є необхідними у мережевих проєктах, особливо у разі командної роботи та підтримки застосунків у часі.

Git: розподілена система контролю версій з розгалуженими можливостями та підтримкою спільної роботи. Платформи GitHub, GitLab, Bitbucket пропонують хмарні рішення для командної роботи та управління проєктами. Типові операції Git у проєкті містять ініціалізацію репозиторію, коміти, гілкування, злиття та синхронізацію з віддаленими серверами.

```
git init
git add .
git commit -m "Initial commit"
git remote add origin <repository-url>
git push -u origin master
```

Інструменти автоматизації збірки керують компіляцією вихідного коду та складанням застосунку. Вони автоматизують трудомісткі та схильні до помилок операції, забезпечуючи послідовність складання та спрощуючи управління проєктом.

CMake: призначений для управління процесами виконання проєктів незалежно від компілятора, забезпечуючи більший контроль і гнучкість у C++-проєктах. CMake дозволяє визначати параметри програми, керувати залежностями та створювати файли програм.

Установлення на Ubuntu можна виконати командою:

```
sudo apt install cmake
```

Приклад базового `CMakeLists.txt` для простого проєкту:

```
cmake_minimum_required(VERSION 3.10)
project(NetworkApp)
set(CMAKE_CXX_STANDARD 17)
add_executable(NetworkApp main.cpp)
```

Makefiles: використовуються в Unix-подібних середовищах і визначають правила компіляції та лінкування програми, стандартизуючи процеси складання. Makefiles описують перевірку залежностей, команди компіляції файлів та цілі (targets) для застосунків.

Тестування в C++ мережевому програмуванні передбачає перевірку надійності застосунку, виявлення помилок і забезпечення відповідності протоколам та вимогам безпеки. Інструменти налагодження та тестові фреймворки доповнюють розробку, надаючи реальний час інспекції та автоматизовану перевірку.

GDB (GNU Debugger): потужний інструмент для налагодження програм на C++. Дозволяє перевіряти значення змінних, стан програми та хід виконання. GDB працює через командний рядок, що робить його ефективним для складних застосунків, що потребують глибокого аналізу.

Приклад базового робочого процесу:

```
g++ -g -o application main.cpp
gdb ./application
```

Valgrind: виконує динамічний аналіз для виявлення помилок роботи з пам'яттю та проблем з багатопоточністю, що часто виникають у мережевих застосунках.

Google Test (GTest): фреймворк для юніт-тестування C++-застосунків, що дозволяє автоматизовано створювати та виконувати тести. Підтримує перевірки (assertions), набори тестів (test suites) та фікстури (fixtures) для контролю кожного компоненту програми.

Приклад тесту:

```
TEST(SocketTest, ConnectionSuccess) {
   ASSERT_EQ(connectToServer(), 0);
}
```

Налаштування середовища розробки, яке враховує потреби мережевого програмування на C++, виходить за рамки простих установлень. Воно потребує продуманої інтеграції інструментів, спеціальних конфігурацій та стратегічного підбору засобів, що відповідають цілям застосунку, динаміці роботи команди та цільовим платформам. Така ретельна підготовка створює умови для розробки мережевого програмного забезпечення з високою точністю, ефективністю та адаптивністю в сучасній технологічно розвиненій екосистемі.

1.6 Контрольні питання

1. У чому полягає суть мережевого програмування та яку роль воно відіграє в сучасному цифровому середовищі?
2. Охарактеризуйте основні типи комп'ютерних мереж (LAN, MAN, WAN, PAN) та наведіть приклади їх практичного застосування.
3. Які ключові апаратні й програмні компоненти входять до складу комп'ютерної мережі та які функції вони виконують?
4. Поясніть відмінності між моделями OSI та TCP/IP і призначення рівнів кожної з них у процесі обміну даними.
5. Опишіть принцип роботи TCP-клієнта на основі сокетного програмування мовою C++ та призначення основних системних викликів, використаних у наведеному прикладі.
6. У чому полягає суть мережевої комунікації та які основні компоненти забезпечують процес обміну даними між вузлами мережі?
7. Порівняйте модель OSI та модель TCP/IP, охарактеризувавши призначення їх рівнів і роль у реалізації мережевої взаємодії.
8. Які функції виконують комутатори та маршрутизатори в процесі мережевої комунікації та на яких рівнях моделей OSI і TCP/IP вони працюють?
9. Поясніть відмінності між протоколами TCP і UDP та наведіть приклади застосувань, у яких доцільно використовувати кожен із них.
10. Опишіть принцип роботи клієнт-серверної моделі мережевої взаємодії та основні етапи створення TCP-сервера засобами сокетного програмування мовою C++.
11. Що таке IP-адреса та які основні відмінності між стандартами IPv4 і IPv6 з погляду адресації та призначення?
12. Поясніть призначення портів і сокетів у мережевій взаємодії та їх роль у реалізації клієнт-серверної архітектури.
13. Які функції виконують основні мережеві протоколи (TCP, UDP, HTTP/HTTPS, FTP, SMTP) та в яких випадках доцільно їх використовувати?
14. У чому полягає різниця між IP-адресами та MAC-адресами і на яких рівнях моделі OSI вони застосовуються?
15. Опишіть призначення та принципи роботи DNS, NAT і міжмережевих екранів (firewalls) у забезпеченні доступності, безпеки та ефективності мережевих з'єднань.
16. У чому полягають основні переваги мови C++ для розроблення мережевих застосунків порівняно з мовами вищого рівня абстракції?
17. Яку роль відіграє сокетне програмування в C++ та які основні системні виклики використовуються для створення TCP-сервера?
18. Як засоби об'єктно-орієнтованого програмування та стандартна бібліотека C++ (STL) сприяють модульності, масштабованості та ефективності мережевих застосунків?

19. Поясніть призначення багатопоточності в мережевому програмуванні C++ та переваги використання std::thread для обробки паралельних клієнтських з’єднань.

20. У чому полягає відмінність між синхронним і асинхронним мережевим програмуванням у C++ та яку роль відіграють бібліотеки Boost.Asio, Poco і Qt Network у реалізації неблокувальних мережевих операцій?

21. Яку роль відіграє правильно налаштоване середовище розробки у процесі створення мережевих застосунків мовою C++ та які основні компоненти воно містить?

22. Порівняйте основні компілятори C++ (GCC, Clang, MSVC, Intel C++ Compiler) та поясніть, у яких випадках доцільно використовувати кожен із них.

23. Які переваги надає використання інтегрованих середовищ розробки (IDE) під час мережевого програмування на C++ і які IDE є найбільш поширеними?

24. Яке призначення інструментів автоматизації складання (CMake, Makefiles) у C++-проєктах і як вони спрощують управління залежностями та процесом компіляції?

25. Які інструменти налагодження та тестування застосовуються в мережевому програмуванні на C++ і яку роль вони відіграють у забезпеченні надійності та якості програмного забезпечення?

2 ОСОБЛИВОСТІ МЕРЕЖЕВИХ ПРОТОКОЛІВ

У розділі розглядається ключова роль мережевих протоколів у забезпеченні безперебійної взаємодії між пристроями. Досліджуючи структуру та функціонування моделей, таких як OSI та TCP/IP, у розділі підкреслюється, як ці протоколи регулюють обмін даними в мережах. Аналізуються широко використовувані інтернет-протоколи, такі як HTTP, FTP та DNS, що дозволяє отримати комплексне уявлення про їх призначення та сфери застосування. Крім того, у розділі розглядаються проблеми впровадження протоколів, готуючи здобувачів до потенційних труднощів, які можуть виникати у практичних сценаріях мережевої розробки.

2.1 Поняття мережевих протоколів

Поняття мережевих протоколів займає фундаментальне місце в галузі комп'ютерних мереж. Мережеві протоколи визначають правила та конвенції взаємодії між мережевими пристроями, забезпечуючи надійну передачу та прийом даних через різноманітні типи мереж. Ці протоколи можна порівняти з мовами, якими «спілкуються» комп'ютери, дозволяючи встановлювати чіткі та структуровані канали комунікації. Розуміння протоколів у мережевій інженерії передбачає усвідомлення їхньої ключової ролі у гармонізації процесів обміну даними. Кожен протокол спеціально розроблений для виконання певної функції в мережі, охоплюючи різні аспекти комунікації, такі як синхронізація, обробка помилок, пакування даних та маршрутизація. Специфікації протоколу містять комплексний набір правил, що охоплюють синтаксис, семантику та час передачі даних. Синтаксис визначає структуру або формат даних, семантика – значення кожної частини бітів, а час передачі узгоджує координацію відправлення та прийому даних. Різноманіття мережевих протоколів формує кілька рівнів комунікації, які можна візуалізувати через структуровані моделі, такі як OSI та TCP/IP. Ці моделі поділяють функції мережі на рівні, забезпечуючи основу для стандартизації протоколів та сумісності між ними. Незважаючи на різні підходи до побудови моделей, їх спільною метою є підвищення ефективності та надійності мережевої взаємодії.

У практичному контексті мережеві протоколи можна класифікувати за кількома основними категоріями:

Протоколи комунікації встановлюють процедури передачі даних між мережевими пристроями. Прикладами є Transmission Control Protocol (TCP) та User Datagram Protocol (UDP). Їхня головна функція полягає у забезпеченні надійної відправки та прийому пакетів даних.

Маршрутизувальні протоколи забезпечують коректну маршрутизацію даних у мережах, визначаючи шляхи доставки пакетів. До цієї категорії належать Border Gateway Protocol (BGP) та Open Shortest Path First (OSPF).

Протоколи управління використовуються для адміністрування мережевих пристроїв та усунення проблем у мережі. Типовий приклад – Simple Network Management Protocol (SNMP).

Протоколи безпеки забезпечують аутентифікацію та авторизацію користувачів, а також шифрування даних. Зразками є Secure Sockets Layer (SSL) та Transport Layer Security (TLS).

Кожен протокол надає власний набір можливостей, що сприяє ефективній роботі мереж. Наприклад, TCP забезпечує надійну, орієнтовану на з'єднання службу, яка гарантує доставку даних з перевіркою помилок та підтвердженням відправки й отримання на вузлах джерела та призначення. UDP, завдяки своїй простоті й ефективності дозволяє відправляти дані без попередньої організації спеціальних каналів або маршрутів передачі, що робить його придатним для застосувань із вимогами до часу передачі, наприклад, потокового відео або аудіо. Для подальшого розуміння значущості мережевих протоколів доцільно розглянути практичний приклад коду, який ілюструє базову реалізацію моделі TCP «клієнт-сервер». У цьому сценарії відбувається ініціація з'єднання, обмін даними та завершення комунікації між клієнтом та сервером.

```
import socket
import time

HOST = '127.0.0.1'  # Адреса сервера
PORT = 65432        # Порт сервера
RETRY_COUNT = 5     # Кількість спроб підключення
RETRY_DELAY = 1     # Затримка між спробами (с)

def tcp_client():
    """
    TCP-клієнт:
    - Підключається до сервера з повторними спробами
    - Отримує дані від сервера
    """
    for attempt in range(RETRY_COUNT):
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
                client_socket.connect((HOST, PORT))
                print(f"[INFO] Connected to server on {HOST}:{PORT}")

                # Отримання даних від сервера
                data = client_socket.recv(1024)
                print("[INFO] Received:", data.decode())
                return  # Після успішного підключення виходимо з циклу
        except ConnectionRefusedError:
            print(f"[WARN] Server not ready, retrying {attempt+1}/{RETRY_COUNT}...")
```

```
            time.sleep(RETRY_DELAY)
        print("[ERROR] Could not connect to server after multiple attempts.")

if __name__ == "__main__":
    tcp_client()
```

Програма реалізує TCP-клієнт, який підключається до сервера за заданою адресою та портом. Клієнт приймає дані від сервера і виводить їх на екран. У разі недоступності сервера здійснюються повторні спроби підключення з затримкою. Після успішного підключення або вичерпання спроб програма завершуються.

Імпорт бібліотек: socket – робота з TCP/IP-сокетами. time – організація затримок між спробами підключення.

Конфігурація параметрів: HOST, PORT – адреса та порт сервера. RETRY_COUNT, RETRY_DELAY – кількість спроб і затримка між ними.

Функція tcp_client(): Створює TCP-сокет і намагається підключитися до сервера з повторними спробами. Успішне підключення: прийом даних від сервера (recv) і виведення повідомлення. Невдале підключення: повторна спроба після затримки (sleep) до вичерпання ліміту.

Обробка помилок: ConnectionRefusedError – сервер недоступний, виведення попередження. Після кількох невдалих спроб виведення повідомлення про помилку [ERROR].

Запуск програми: Блок if __name__ == "__main__": виконує функцію tcp_client() під час запуску скрипта.

Скрипт клієнта встановлює підключення до TCP-сервера за заданою IP-адресою та номером порту. Після встановлення з’єднання демонструється передача повідомлень за допомогою `s.sendall()` та отримання відповіді сервера через `s.recv()`, завершуючи базовий цикл обміну даними.

Протоколи виконують роль стандартів для побудови взаємодійних мережевих середовищ, дозволяючи апаратним і програмним засобам різних виробників взаємодіяти та сприяючи розвитку інновацій у сфері мережевих технологій. Дотримання стандартних протоколів забезпечує мінімальні налаштування для пристроїв і гарантує безперебійну роботу користувача, що особливо важливо у середовищах із обладнанням від різних постачальників.

Мережеві протоколи також регулюють передачу даних і запобігають перевантаженню каналів за допомогою механізмів керування потоком, таких як віконні алгоритми TCP. Крім того, протоколи забезпечують контроль помилок і корекцію даних за допомогою контрольних сум, циклічних надлишкових перевірок (CRC) та підтверджень доставки, що гарантує коректне надходження інформації до отримувача.

Особливе значення мають протоколи безпеки, які захищають комунікації, наприклад, застосування TLS для забезпечення безпечних вебтранзакцій.

```
# Python-код простого сервера з підтримкою SSL
import ssl
```

```
import socket

def ssl_server():
    # Створення SSL-контексту для автентифікації клієнтів
    context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)

    # Завантаження сертифіката сервера та закритого ключа
    context.load_cert_chain(certfile="server.pem", keyfile="server.key")

    # Створення TCP-сокета та прив’язка до локальної адреси і порту
    bindsocket = socket.socket()
    bindsocket.bind(('127.0.0.1', 10023))
    bindsocket.listen(5)

    # Основний цикл обробки клієнтських підключень
    while True:
        newsocket, fromaddr = bindsocket.accept()

        # Обгортання TCP-з’єднання у захищене SSL-з’єднання
        conn = context.wrap_socket(newsocket, server_side=True)
        try:
            data = conn.recv(1024)
            if data:
                # Надсилання відповіді клієнту через захищений канал
                conn.sendall(b"Вітаємо! Захищене SSL-з'єднання
встановлено.")
        finally:
            # Закриття з’єднання з клієнтом
            conn.close()

if __name__ == '__main__':
    ssl_server()
```

SSL-сервер ініціалізує захищений мережевий контекст, використовуючи криптографічний сертифікат і закритий ключ для автентифікації та шифрування з’єднання. Після прив’язки до заданої IP-адреси та порту сервер переходить у режим очікування клієнтських підключень. Кожне прийняте TCP-з’єднання обгортається в SSL-канал, що забезпечує конфіденційність і цілісність передаваних даних. Обмін інформацією з клієнтом здійснюється через захищений канал, після чого з’єднання коректно завершується.

Сервер використовує протокол SSL для формування захищеного каналу зв’язку з клієнтом, забезпечуючи конфіденційність і цілісність даних, що передаються через потенційно небезпечні мережеві середовища. Такий рівень захисту є критично важливим для сучасних прикладних систем, у яких питання збереження та захисту інформації мають першочергове значення.

Безперервна еволюція мережевих протоколів відображає здатність мережевих технологій адаптуватися до нових вимог. Від простих механізмів обміну даними в епоху ARPANET до сучасного високошвидкісного Інтернету протоколи постійно вдосконалювалися з метою ефективного керування зростаючими обсягами трафіку, протидії більш складним атакам і задоволення безпрецедентного попиту на обробку даних у реальному часі. Це зумовлює регулярне оновлення наявних стандартів і розроблення нових протоколів для вирішення спеціалізованих завдань.

Підсумовуючи, знання мережевих протоколів надає мережевим адміністраторам, інженерам і розробникам можливість проєктувати надійні, безпечні та ефективні системи обміну даними. Ці протоколи охоплюють увесь спектр мережевих потреб — від базових процедур встановлення з'єднання до складних механізмів шифрування, що забезпечують захист інформації. Розуміння принципів їх роботи та сфер застосування сприяє глибшому усвідомленню складності взаємодії сучасних пристроїв і мережевих систем.

2.2 Модель OSI

Модель взаємодії відкритих систем (Open Systems Interconnection, OSI), розроблена Міжнародною організацією зі стандартизації (ISO), є концептуальною основою, що стандартизує функції телекомунікаційних і обчислювальних систем шляхом поділу їх на сім абстрактних рівнів. Кожен рівень виконує визначену функцію та взаємодіє лише з безпосередньо суміжними рівнями, що забезпечує модульність і структурованість мережевої архітектури. Розуміння моделі OSI є необхідним для мережевих інженерів і проєктувальників під час діагностики несправностей та ефективного застосування мережевих протоколів.

Модель OSI складається з таких семи рівнів:

Фізичний рівень (Physical Layer). Найнижчий рівень моделі OSI, який відповідає за фізичне з'єднання між пристроями та передавання й приймання необроблених бітових потоків через фізичне середовище. Він визначає електричні, оптичні та механічні характеристики апаратних інтерфейсів мережі. Прикладами стандартів цього рівня є Ethernet, USB та RS-232.

Канальний рівень (Data Link Layer) забезпечує передавання даних між безпосередньо з'єднаними вузлами та виконує виявлення і корекцію помилок за допомогою контрольних сум і синхронізації кадрів. Поділяється на два підрівні: підрівень логічного керування каналом (LLC), який відповідає за синхронізацію кадрів, перевірку помилок і керування потоком, та підрівень керування доступом до середовища (MAC), що визначає порядок доступу до середовища передавання. На цьому рівні

функціонують протоколи Ethernet (для локальних мереж) і PPP (для з’єднань типу «точка–точка»).

Мережевий рівень (Network Layer). Основним завданням цього рівня є маршрутизація, тобто вибір фізичного або логічного шляху передавання даних від джерела до призначення через кілька мереж. Він здійснює пересилання пакетів, зокрема через проміжні маршрутизатори. Базовим протоколом цього рівня є Інтернет-протокол (IP), який забезпечує доставлення пакетів між мережами.

Транспортний рівень (Transport Layer) забезпечує наскрізний зв’язок між прикладними процесами, реалізуючи керування помилками, керування потоком та гарантуючи повноту передавання даних. Рівень може надавати надійний, орієнтований на з’єднання сервіс (TCP) або сервіс без встановлення з'єднання (UDP). Ключовими функціями є сегментація, повторне збирання даних, а також виявлення і виправлення помилок.

Сеансовий рівень (Session Layer) відповідає за встановлення, керування та завершення сеансів зв’язку між двома хостами. Він контролює діалоги між комп’ютерами, керує обміном даними та забезпечує механізми контрольних точок і відновлення у разі переривань. До цього рівня відносять такі протоколи, як RPC і SQL.

Рівень подання (Presentation Layer) часто називається синтаксичним рівнем, оскільки відповідає за перетворення даних між прикладним рівнем і мережевим форматом. Він забезпечує шифрування й дешифрування, стиснення та розпакування даних, гарантуючи коректну інтерпретацію інформації на різних системах. Протоколи шифрування, зокрема SSL, функціонують на цьому рівні.

Прикладний рівень (Application Layer). Найвищий рівень моделі OSI, який забезпечує взаємодію програмних застосунків із нижчими мережевими сервісами. Він підтримує мережевий доступ, цілісність даних і додаткові механізми безпеки. До цього рівня належать протоколи HTTP, FTP, SMTP і DNS, що безпосередньо обслуговують користувацькі застосунки та обмін даними між комп’ютерами.

Модель OSI сприяє розумінню складних процесів мережевої взаємодії шляхом поділу їх на чітко визначені функціональні рівні. Така абстракція забезпечує взаємозамінність і сумісність обладнання та програмних продуктів різних виробників, оскільки кожен компонент має відповідати стандартам лише свого рівня.

Для ілюстрації практичного застосування рівнів OSI можна розглянути приклад TCP-з’єднання, у якому задіяні кілька рівнів моделі. У цьому умовному сценарії показано процес інкапсуляції, коли дані передаються від прикладного рівня вниз до фізичного рівня, проходять мережею, а на приймальному боці піднімаються назад до прикладного рівня:

Прикладний рівень: користувач ініціює запит вебсторінки через браузер.

Рівень подання: за потреби SSL/TLS виконує шифрування HTTP-запиту.

Сеансовий рівень: встановлюється сеанс HTTP-обміну з можливістю багаторазових запитів і відповідей.

Транспортний рівень: TCP сегментує дані та додає заголовки для контролю помилок і керування потоком.

Мережевий рівень: сегменти інкапсулюються в IP-пакети з інформацією про адресацію та маршрутизацію.

Канальний рівень: пакети оформлюються у кадри з MAC-адресами для передавання мережею.

Фізичний рівень: двійкові дані передаються фізичним середовищем за допомогою електричних сигналів, кабелів або бездротових технологій.

Для поглибленого розуміння взаємодії рівнів OSI може бути використано приклад реалізації TCP Echo Server мовою Python, який демонструє, як транспортні протоколи, реалізовані засобами сокетного програмування, забезпечують надійність передавання даних і керування з'єднаннями відповідно до моделі OSI.

```
# Програма на Python, що імітує простий TCP Echo Server
import socket

def tcp_echo_server():
    host = '127.0.0.1'
    port = 65432

    # Створення TCP/IP сокета
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        # Прив'язка сокета до заданої IP-адреси та порту
        server_socket.bind((host, port))

        # Переведення сокета в режим очікування вхідних з'єднань,
        # розмір черги очікування — 5
        server_socket.listen(5)
        print("Сервер очікує з'єднання на", (host, port))

        # Очікування підключення клієнтів
        while True:
            client_socket, address = server_socket.accept()
            with client_socket:
                print("Встановлено з'єднання з", address)
                while True:
                    # Прийом даних від клієнта
                    data = client_socket.recv(1024)
                    if not data:
                        break
                    # Надсилання отриманих даних назад клієнту (echo)
                    client_socket.sendall(data)
```

```
if __name__ == '__main__':
    tcp_echo_server()
```

У наведеному вище прикладі сервер надає клієнтові кінцеву точку взаємодії на транспортному рівні, забезпечуючи інкапсуляцію повідомлень за допомогою протоколу TCP з метою надійного передавання даних. Сервер безперервно перебуває в режимі очікування вхідних з'єднань і після встановлення з'єднання повертає клієнтові всі отримані від нього дані. На кожному рівні моделі OSI інформація проходить відповідні етапи оброблення та доповнюється необхідною керувальною інформацією, що забезпечує надійність, масштабованість і коректність мережевої взаємодії.

Модель OSI відіграє ключову роль у сучасній мережевій архітектурі, оскільки використовується не лише для теоретичного осмислення принципів мережевої взаємодії, але й як еталонна основа під час діагностики та усунення мережевих несправностей. Чіткий розподіл функцій між рівнями спрощує процес локалізації помилок. Так, проблеми міжмережевої взаємодії можуть бути віднесені до мережевого рівня, тоді як збої, пов'язані з шифруванням даних, зазвичай вказують на рівень подання.

Розуміння моделі OSI не обмежується засвоєнням функціонального призначення кожного окремого рівня. Воно також передбачає усвідомлення взаємозалежної роботи всіх рівнів, спрямованої на забезпечення безперервного та ефективного передавання даних від джерела до отримувача в умовах складних мережевих інфраструктур. Незважаючи на постійний розвиток технологій і зростання вимог до комунікаційних систем, принципи моделі OSI залишаються фундаментальною основою для проєктування ефективних і надійних систем передавання інформації.

2.3 Модель TCP/IP та протоколи

Модель TCP/IP, також відома як стек інтернет-протоколів (Internet Protocol Suite), являє собою сукупність комунікаційних протоколів, які визначають правила пакування, передавання, маршрутизації та приймання даних у мережевому середовищі. Вона становить базову архітектуру Інтернету та більшості локальних мереж. На відміну від семирівневої моделі OSI, модель TCP/IP складається з чотирьох рівнів, кожен з яких забезпечує узгоджене та безперервне передавання даних між гетерогенними системами й мережами. До складу моделі входять канальний рівень, інтернет-рівень, транспортний рівень та прикладний рівень.

Канальний рівень (Link Layer), який за функціональним призначенням відповідає поєднанню фізичного та канального рівнів моделі OSI, відповідає за апаратну адресацію та фізичне передавання даних у межах однієї мережі. Він охоплює протоколи й апаратні стандарти, що забезпечують обмін даними через фізичні середовища передавання. До

основних протоколів цього рівня належать Ethernet, Wi-Fi (IEEE 802.11) та протокол «точка-точка» (PPP), які забезпечують коректне форматування пакетів для передавання певним середовищем.

Інтернет-рівень (Internet Layer) функціонує незалежно від базової мережевої інфраструктури та відповідає за маршрутизацію пакетів між різними мережами, виконуючи роль навігаційного механізму, що об'єднує множину мереж у єдину глобальну систему – Інтернет. Ключовим протоколом цього рівня є Інтернет-протокол (IP), який забезпечує адресацію та пересилання пакетів без встановлення наскрізних з'єднань. Протокол IP реалізований у двох версіях – IPv4 та IPv6.

Запровадження IPv6 дозволило подолати обмеження IPv4, зокрема дефіцит адресного простору, а також підвищити ефективність маршрутизації. До інших протоколів інтернет-рівня належать протокол керувальних повідомлень Інтернету (ICMP), що використовується для діагностики та повідомлення про помилки, а також протокол визначення адрес (ARP), який забезпечує зіставлення IP-адрес із апаратними адресами в локальних мережах.

Транспортний рівень (Transport Layer) за своїм призначенням відповідає транспортному рівню моделі OSI та забезпечує надійне доставляння повідомлень між вузлами мережі. Протоколи цього рівня використовують стандартизовані точки доступу – порти, що дає змогу розрізняти декілька прикладних процесів на одному пристрої. Основними протоколами транспортного рівня є протокол керування передаванням (TCP) та протокол дейтаграм користувача (UDP).

TCP забезпечує надійне, орієнтоване на з'єднання передавання даних із використанням механізмів тристороннього встановлення з'єднання, підтверджень отримання та повторної передачі втрачених пакетів. На відміну від нього, UDP реалізує модель обміну даними без встановлення з'єднання та застосовується в системах, де пріоритетом є швидкість передавання, наприклад у потоковому відео або мультимедійних застосуваннях.

Прикладний рівень (Application Layer) є верхнім рівнем моделі TCP/IP і поєднує функції сеансового, представницького та прикладного рівнів моделі OSI. Він забезпечує взаємодію мережевих протоколів із програмними застосунками, що беруть участь у процесі обміну даними.

До протоколів цього рівня належать HTTP для передавання вебданих, FTP для обміну файлами, SMTP для надсилання електронної пошти та DNS для перетворення доменних імен у IP-адреси. Широкий спектр прикладних протоколів забезпечує підтримку різноманітних сервісів і коректне подання інформації кінцевому користувачеві.

Протокол TCP забезпечує встановлення захищених з'єднань і надійне передавання даних між мережевими пристроями. Завдяки реалізації механізмів керування потоком, виявлення та виправлення помилок TCP гарантує цілісність даних і високу точність їх доставляння.

```
# Проста ілюстрація TCP-клієнта на Python
import socket

def tcp_client():
    server_address = ('localhost', 8080)  # Адреса та порт сервера
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Створення TCP-сокета
    client_socket.connect(server_address)  # Підключення до сервера

    try:
        message = 'Hello, TCP Server!'  # Повідомлення для відправки
        client_socket.sendall(message.encode())  # Відправка даних серверу
        response = client_socket.recv(1024)  # Отримання відповіді від сервера
        print('Отримано:', response.decode())  # Виведення отриманих даних
    finally:
        client_socket.close()  # Закриття сокета

if __name__ == "__main__":
    tcp_client()
```

Цей код на Python демонструє роботу простого TCP-клієнта, який підключається до сервера, передає та отримує дані, використовуючи властивості TCP щодо надійного обміну інформацією через встановлене з'єднання та підтвердження доставки.

Протокол датаграм користувача (UDP): UDP оптимізований для швидкого обміну даними з мінімальними накладними витратами, коли пріоритет надається швидкості, а не надійності. Він не потребує встановлення або підтримки з'єднання, що робить його придатним для застосунків у режимі реального часу, таких як онлайн-ігри або VoIP.

```
import socket

def udp_sender():
    server_address = ('localhost', 6789)

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

    try:
        message = 'Hello, UDP Server!'
        # Відправлення повідомлення на сервер
        udp_socket.sendto(message.encode(), server_address)
        print('Повідомлення надіслано на', server_address)
    finally:
```

```
        udp_socket.close()

if __name__ == "__main__":
    udp_sender()
```

Цей приклад демонструє відправку простого повідомлення на сервер за допомогою UDP, підкреслюючи швидкість і простоту безз'єднувальної комунікації.

Протокол Інтернету (IP) забезпечує навігацію даних у мережах, керуючи адресацією та маршрутизацією пакетів, що є критично важливим для комунікацій в Інтернеті. Його дві версії – IPv4 та IPv6 – відрізняються переважно довжиною адреси, причому IPv6 пропонує більший адресний простір, необхідний для сучасних мережевих вимог.

Порівняння та інтеграція з моделлю OSI: Хоча модель OSI надає комплексну теоретичну основу для мережевих комунікацій, модель TCP/IP є практичною та адаптованою до реальних інтернет-технологій. Шари моделі TCP/IP менш абстрактні, детально описують конкретні протокольні набори, що використовуються у повсякденних мережевих задачах. Розуміння взаємозв'язку між OSI та TCP/IP полегшує діагностику та аналіз мереж, надаючи різні перспективи щодо проходження даних у мережевих системах.

Модель TCP/IP істотно впливає на проєктування мережевої інфраструктури, забезпечуючи підтримку сучасних розподілених додатків, інтегруючи обчислювальні домени, розширюючи зв'язність та підтримуючи масштабовані архітектури. Завдяки чітко визначеним шарам мережеві розробники використовують TCP/IP для створення ефективних і гнучких систем, що інтегрують різноманітні протоколи та спадкові системи в єдину узгоджену структуру.

Реалізація протоколів TCP/IP. Розглянемо спрощену взаємодію клієнта та сервера, де реалізуються як TCP, так і UDP, для порівняння їхніх принципів роботи.

TCP проти UDP – реалізація сервера:

```
# Python-код, що ілюструє налаштування серверної частини TCP та UDP
import socket

def tcp_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 8090))
    server_socket.listen()
    print('TCP-сервер очікує підключень...')

    while True:
        conn, addr = server_socket.accept()
        with conn:
```

```
            print('Підключено від', addr)
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                conn.sendall(data.upper())

def udp_server():
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_socket.bind(('localhost', 8091))
    print('UDP-сервер готовий до роботи...')

    while True:
        data, addr = udp_socket.recvfrom(1024)
        print('Підключення від', addr)
        udp_socket.sendto(data.upper(), addr)

# Розкоментуйте для запуску відповідної серверної функції
# if __name__ == "__main__":
#     tcp_server()
#     udp_server()
```

TCP-сервер встановлює постійне з’єднання, забезпечуючи повну цілісність передачі даних. На противагу цьому, UDP-сервер здійснює миттєву передачу даних без встановлення з’єднання, орієнтуючись на ефективність і низьку затримку відгуку.

Відомий своєю універсальністю та надійністю набір протоколів TCP/IP залишається основою сучасних мереж, інтегруючи ключові протокольні розробки та відображаючи розвиток технологій, що гарантує швидкий і достовірний обмін даними в глобальній мережі Інтернет. Засвоєння принципів його роботи дозволяє фахівцям ефективніше проектувати та підтримувати надійні мережеві системи, адаптовані до різноманітних і динамічних умов експлуатації.

2.4 Поширені інтернет-протоколи

У середовищі цифрової комунікації поширені інтернет-протоколи забезпечують надійний обмін даними між пристроями через глобальну мережеву інфраструктуру, відому як Інтернет. Ці протоколи забезпечують встановлення з’єднань, передачу інформації та підтримку надійності й безпеки в різних мережевих середовищах. До ключових протоколів належать: Протокол передачі гіпертексту (HTTP), Протокол передачі файлів (FTP), Простий протокол пересилання пошти (SMTP) та Система доменних імен (DNS), які виконують критичні функції у вебсервісах, передачі файлів, електронній пошті та розв’язуванні доменних імен, відповідно.

Протокол передачі гіпертексту (HTTP) є фундаментальним протоколом у Всесвітній павутині, призначеним для передачі гіпертекстових документів між клієнтами та серверами. Протокол переважно працює поверх Протоколу управління передачею (TCP) і регламентує передачу даних через веб, інкапсулюючи запити та відповіді у зрозумілі формати для безперешкодної взаємодії користувача з веб-ресурсами. HTTP функціонує на рівні прикладного рівня моделі TCP/IP та підтримує методи ОТРИМАТИ (GET), НАДІСЛАТИ (POST), ОНОВИТИ (PUT), ВИДАЛИТИ (DELETE) для виконання різноманітних операцій над вебресурсами.

Еволюція HTTP до HTTP/2 забезпечила такі покращення, як мультиплексування, стиснення заголовків та вдосконалене керування потоком, що підвищує швидкість та ефективність роботи. HTTP/3 запроваджує нові транспортні механізми на основі швидких UDP-з'єднань в Інтернеті (QUIC).

Приклад на Python для простого HTTP-клієнта демонструє базові операції протоколу HTTP.

```
import http.client

def fetch_website(use_local=True):
    """
    Функція підключається до HTTP-сервера і отримує сторінку.
    Параметр use_local=True підключається до локального сервера,
    use_local=False підключається до www.example.com через HTTPS.
    """
    if use_local:
        host = "localhost"
        port = 8000  # переконайтеся, що локальний сервер запущений на цьому порту
        connection = http.client.HTTPConnection(host, port)
        print(f"[INFO] Підключення до локального сервера {host}:{port}")
    else:
        host = "www.example.com"
        connection = http.client.HTTPSConnection(host)
        print(f"[INFO] Підключення до віддаленого сервера {host}")

    try:
        connection.request("GET", "/")
        response = connection.getresponse()
        print("Статус відповіді:", response.status)
        print("Заголовки відповіді:", response.getheaders())
        print("Вміст сторінки:", response.read().decode())
    except Exception as e:
        print("[ERROR] Не вдалося підключитися або отримати дані:", e)
    finally:
        connection.close()
```

```
if __name__ == "__main__":
    # Підключитися до локального сервера
    fetch_website(use_local=True)

    # Підключитися до віддаленого сервера
    fetch_website(use_local=False)
```

Цей скрипт встановлює з’єднання з www.example.com, надсилає GET-запит і виводить статус, заголовки та вміст відповіді.

Протокол передачі файлів (FTP) призначений для безперебійного обміну файлами через Інтернет, забезпечуючи механізми завантаження та скачування файлів між комп’ютерами. Використовуючи окремі канали даних та керування, FTP дозволяє виконувати команди та передавати дані незалежно один від одного, часто під захищеним транспортним з’єднанням. FTP-клієнти можуть взаємодіяти із серверами, використовуючи облікові дані користувача, що забезпечує як автентифікований, так і анонімний доступ.

Активний та пасивний режими FTP

Активний режим: клієнт відкриває випадковий порт і повідомляє серверу про IP та порт, на який сервер підключається для встановлення каналу передачі даних.

Пасивний режим: сервер відкриває випадковий порт і повідомляє клієнту, дозволяючи клієнту ініціювати з’єднання для передачі даних.

Для демонстрації роботи FTP можна використати бібліотеку ftplib у Python, яка дозволяє отримувати списки файлів та завантажувати їх.

```
# Python FTP-клієнт з обробкою помилок та таймаутом
from ftplib import FTP, all_errors

def ftp_client_example():
    # Сервер для підключення: можна змінити на '127.0.0.1' для локального сервера
    ftp_server = "ftp.debian.org"
    ftp_port = 21  # стандартний FTP-порт

    try:
        # Підключення до сервера з таймаутом 10 секунд
        ftp = FTP()
        print(f"Підключення до {ftp_server}:{ftp_port}...")
        ftp.connect(host=ftp_server, port=ftp_port, timeout=10)

        # Анонімний вхід
        ftp.login()
        print("Успішно підключено і виконано логін")
```

```
        # Перехід у каталог
        ftp.cwd('/debian')
        print("Каталог змінено на /debian")

        # Отримання списку файлів (тільки перші 10 для прикладу)
        files = ftp.nlst()
        print("Файли у каталозі (перші 10):", files[:10])

        # Закриття з'єднання
        ftp.quit()
        print("З'єднання закрито")

    except all_errors as e:
        print("FTP-з'єднання не вдалося:", e)

if __name__ == "__main__":
    ftp_client_example()
```

Цей приклад демонструє підключення до FTP-сервера, отримання списку файлів у каталозі та завантаження заданого файлу.

Протокол простого пересилання пошти (SMTP) є де-факто стандартом для передачі електронної пошти через інтернет, визначаючи правила відправки, отримання та маршрутизації повідомлень. SMTP зазвичай працює поверх TCP. У той час як SMTP використовується для відправки повідомлень, протоколи, такі як Post Office Protocol (POP3) та Internet Message Access Protocol (IMAP), застосовуються для отримання пошти. SMTP використовує команди на кшталт HELO, MAIL FROM, RCPT TO, DATA та QUIT для координації обміну повідомленнями між поштовими серверами. У зв’язку з вимогами безпеки SMTP підтримує розширення, такі як STARTTLS, для шифрованої передачі пошти.

Варіант 1.

```
import smtplib

def send_mail():
    smtp_server = 'smtp.gmail.com'       # Робочий SMTP сервер
    port = 587                           # Порт TLS
    sender = 'your_email@gmail.com'      # Ваша електронна адреса
    password = 'your_app_password'       # Пароль або app password
    recipient = 'friend@example.com'     # Адреса отримувача
    message = """\
Subject: Test Email

This is a test email sent from Python."""

    try:
```

```
        # Підключення до SMTP сервера
        with smtplib.SMTP(smtp_server, port) as server:
            server.ehlo()              # Встановлення контакту з сервером
            server.starttls()          # Активуємо TLS
            server.login(sender, password)
            server.sendmail(sender, recipient, message)
            print("Email sent successfully")
    except Exception as e:
        print("Error:", e)

if __name__ == "__main__":
    send_mail()
```

Варіант 2.

```
# Python SMTP client simulation example for educational purposes

def send_mail_simulation():
    sender = 'example@example.com'       # Адреса відправника
    recipient = 'friend@example.com'     # Адреса отримувача
    subject = 'Test Email'
    body = 'This is a test email sent from Python (simulated).'

    # Формуємо текст листа
    message = f"Subject: {subject}\n\n{body}"

    # Симуляція надсилання (немає реального підключення)
    print("Підключення до SMTP-сервера... (симуляція)")
    print(f"Відправник: {sender}")
    print(f"Одержувач: {recipient}")
    print("Текст листа:")
    print(message)
    print("Лист успішно 'надіслано' (симуляція)")

if __name__ == "__main__":
    send_mail_simulation()
```

Цей скрипт встановлює з’єднання з SMTP-сервером, шифрує його за допомогою TLS і надсилає електронне повідомлення.

Система доменних імен (DNS) перетворює зрозумілі для людини доменні імена (наприклад, www.example.com) на машино-зрозумілі IP-адреси, необхідні для визначення та ідентифікації комп’ютерних сервісів і пристроїв із використанням базових мережевих протоколів. Функціонально DNS нагадує велику розподілену базу даних і здійснює розв’язання запитів через ієрархію серверів імен, включно кореневі сервери, сервери верхнього рівня домену (TLD) та авторитетні сервери імен.

Записи DNS класифікуються за типами, серед яких A, AAAA, CNAME, MX, TXT та інші, і виконують різні функції розв’язання:

A Record: відображає домен на IPv4-адресу.

AAAA Record: відображає домен на IPv6-адресу.

CNAME Record: встановлює відповідність між псевдонімом та канонічним доменним іменем.

MX Record: направляє електронну пошту на поштові сервери домену.

Бібліотека Python socket надає методи для програмного розв’язання DNS-запитів.

```
# Python DNS query example with error handling
import socket

def resolve_domain(domain):
    try:
        # Спроба отримати IP-адресу домену
        ip_address = socket.gethostbyname(domain)
        print(f"IP-адреса домену {domain} — {ip_address}")
    except socket.gaierror as e:
        # Обробка помилки DNS
        print(f"Не вдалося розв’язати домен {domain}: {e}")

if __name__ == "__main__":
    # Перевірка прикладу
    resolve_domain("www.example.com")
    resolve_domain("invalid-domain.test")  # Тест з недійсним доменом
```

Цей приклад демонструє розв’язання доменного імені в його відповідну IP-адресу за допомогою DNS-запиту.

Безпека мережевих протоколів охоплює стратегії захисту цілісності та конфіденційності даних, зокрема Transport Layer Security (TLS) та Secure Socket Layer (SSL), які додають безпечні розширення до протоколів, таких як HTTP (що призводить до HTTPS) та SMTP. Заходи безпеки, такі як шифрування, автентифікація, забезпечення цілісності та незаперечність (non-repudiation), закладають основу протоколів для протидії загрозам у процесах передавання даних.

У зв’язку з постійним зростанням Інтернету протоколи адаптуються для забезпечення підвищеної продуктивності, враховуючи експоненційне зростання кількості користувачів та обсягів обміну даними. Протоколи еволюціонують через ітеративні вдосконалення, спрямовані на зменшення затримок, стиснення даних, адаптивний контроль перевантаження та зміцнення заходів безпеки.

У сучасну цифрову еру ці протоколи виконують критично важливі функції, забезпечуючи безперервну та масштабовану комунікацію. Вони формують основу електронної комерції, онлайн-сервісів, хмарних обчислень та широкого спектра функціональностей, залежних від

Інтернету. Освоєння цих протоколів дозволяє розробникам, мережевим інженерам та IT-фахівцям оптимізувати та захищати сучасну інтернет-інфраструктуру, забезпечуючи ефективний і безпечний обмін даними.

2.5 Проблеми реалізації протоколів

Реалізація мережевих протоколів передбачає складний баланс між програмною інженерією, мережевим програмуванням та системним проєктуванням з метою забезпечення ефективної й високопродуктивної взаємодії в цифрових середовищах. Проблеми реалізації протоколів виникають унаслідок необхідності врахування різноманітних вимог щодо сумісності, масштабованості, безпеки та надійності. У міру розширення й еволюції мереж розв'язання цих проблем набуває ключового значення для розробників та інженерів, які прагнуть створювати стійкі та надійні мережеві рішення.

Забезпечення співіснування та ефективної взаємодії нових і наявних протоколів у широкому спектрі систем і пристроїв є однією з основних проблем. Різні системи можуть використовувати гетерогенне апаратне забезпечення, різні операційні середовища та програмні платформи, що зумовлює потребу у протоколах, здатних адаптуватися до таких умов без втрати сумісності. Зазначена проблема зазвичай розв'язується шляхом дотримання відкритих стандартів і специфікацій, які сприяють інтероперабельності. Водночас у реальних реалізаціях можливі відхилення в інтерпретації та виконанні протоколів, що може призводити до несумісності в мережі.

Для зменшення таких розбіжностей необхідним є ретельне тестування з використанням різноманітних тестових наборів і середовищ з метою перевірки відповідності реалізації вимогам стандартів протоколів. Важливу роль у цьому процесі відіграють симулятори та мережеві емулятори, які дозволяють моделювати різні мережеві умови та виявляти проблеми інтероперабельності.

Безпека залишається однією з найсуттєвіших проблем реалізації протоколів, що зумовлено постійною еволюцією загроз та зростанням складності кібератак, спрямованих на експлуатацію вразливостей. Реалізація захищених мережевих протоколів передбачає забезпечення конфіденційності, автентифікації, цілісності та незаперечності переданих даних.

Протоколи мають коректно впроваджувати механізми шифрування, забезпечуючи баланс між рівнем безпеки та накладними витратами на продуктивність. Наприклад, використання криптографічних процедур узгодження з'єднання, як у протоколі Transport Layer Security (TLS), призводить до додаткових затримок, проте забезпечує суттєві переваги з погляду захисту інформації. Практичним викликом є проєктування

протоколів таким чином, щоб мінімізувати вплив на продуктивність без зниження рівня безпеки.

Приклад TLS-зашифрованого з’єднання в Python із використанням модуля ssl

```
import socket
import ssl

def secure_client():
    hostname = "www.example.com"
    port = 443  # Стандартний порт HTTPS

    # Створення стандартного SSL/TLS-контексту з перевіркою сертифікатів
    context = ssl.create_default_context()

    try:
        # Встановлення TCP-з’єднання з сервером
        with socket.create_connection((hostname, port), timeout=10) as sock:
            # Обгортання сокета у захищене TLS-з’єднання
            with context.wrap_socket(sock, server_hostname=hostname) as secure_sock:
                # Формування HTTP GET-запиту
                request = (
                    "GET / HTTP/1.1\r\n"
                    "Host: www.example.com\r\n"
                    "Connection: close\r\n"
                    "\r\n"
                )
                # Надсилання запиту серверу
                secure_sock.sendall(request.encode("utf-8"))

                # Отримання відповіді від сервера
                response = secure_sock.recv(4096)
                print(response.decode("utf-8", errors="ignore"))

    except socket.gaierror:
        print("Помилка DNS: неможливо визначити IP-адресу сервера.")
    except ConnectionRefusedError:
        print("З’єднання відхилено сервером.")
    except TimeoutError:
        print("Перевищено час очікування з’єднання.")
    except Exception as e:
        print(f"Невідома помилка: {e}")

if __name__ == "__main__":
    secure_client()
```

У наведеному прикладі продемонстровано захищений обмін даними з використанням протоколу TLS, який є невід'ємною складовою забезпечення конфіденційності та цілісності HTTP-транзакцій.

Мережевий протокол має забезпечувати обробку зростаючих навантажень без зниження продуктивності, що становить одну з ключових проблем його реалізації. Масштабованість охоплює як горизонтальний аспект (обслуговування більшої кількості клієнтів), так і вертикальний аспект (підтримка складніших транзакцій). У процесі розвитку та розширення мереж протоколи мають бути спроєктовані з урахуванням зростання кількості користувачів, обсягів передаваних даних і складності обчислювальних операцій.

До основних підходів забезпечення масштабованості належать: ефективна обробка даних, використання механізмів балансування навантаження та впровадження розподілених систем. Прикладами технологій, що зменшують проблеми масштабування шляхом рівномірного розподілу навантаження між кількома серверами, є мережеві балансувальники навантаження (Network Load Balancers, NLB) та мережі доставки контенту (Content Delivery Networks, CDN).

Протилежні вимоги до мінімізації затримки та максимізації пропускної здатності створюють складну інженерну задачу. Висока пропускна здатність забезпечує ефективну передачу великих обсягів даних, тоді як низька затримка потребує мінімального часу між передаванням і прийманням інформації. Розробники протоколів мають застосовувати алгоритмічні оптимізації та відповідні мережеві механізми для досягнення збалансованого компромісу між цими параметрами.

Механізми керування перевантаженням протоколу TCP, зокрема Slow Start та Fast Retransmit, є прикладами функціональних рішень, спрямованих на підтримання високої пропускної здатності за допустимого рівня затримки. Їх коректна реалізація та всебічне тестування є необхідними для динамічної адаптації до змінних умов мережі.

Мережеві протоколи мають залишатися стійкими до втрат пакетів і помилок передавання, забезпечуючи цілісність даних і безперервність взаємодії з користувачем. Реалізація алгоритмів виявлення та виправлення помилок є фундаментальним елементом надійного проєктування протоколів. Для цього широко застосовуються такі методи, як контрольні суми, циклічні надлишкові коди (CRC) та механізми автоматичного повторного запиту (Automatic Repeat reQuest, ARQ).

Протокол UDP, попри безз'єднувальний характер, часто використовується спільно з протоколами вищих рівнів для керування неупорядкованими пакетами та спотворенням даних, що дає змогу реалізувати обробку помилок без значних накладних витрат, притаманних TCP.

```
# Програмний код Python з використанням UDP з базовим підтвердженням (ACK)
# для демонстрації обробки помилок

import socket

def udp_server_with_ack():
    host = "127.0.0.1"
    port = 65432

    # Створення UDP-сокета
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_socket.bind((host, port))

    print("UDP-сервер із механізмом підтвердження (ACK) запущено...")

    while True:
        # Прийом даних від клієнта
        data, addr = udp_socket.recvfrom(1024)
        message = data.decode("utf-8", errors="ignore")
        print(f"Отримано повідомлення від {addr}: {message}")

        # Формування та надсилання підтвердження отримання (ACK)
        ack = f"ПІДТВЕРДЖЕННЯ (ACK) для повідомлення: '{message}'"
        udp_socket.sendto(ack.encode("utf-8"), addr)

if __name__ == "__main__":
    udp_server_with_ack()
```

Наведений приклад демонструє UDP-обмін даними з використанням простого механізму підтвердження отримання, що є альтернативним підходом до зменшення втрат пакетів під час передавання інформації.

Середовища з обмеженими ресурсами, зокрема вбудовані системи та пристрої Інтернету речей (IoT), створюють додаткові труднощі під час реалізації мережевих протоколів. Такі пристрої мають забезпечувати ефективну комунікацію за умов обмеженої оперативної пам’яті, обчислювальної потужності та енергоспоживання. У зв’язку з цим протоколи мають бути оптимізовані на мінімальні накладні витрати із збереженням базової функціональності. Саме в цьому аспекті перевагу мають легковагові протоколи, такі як MQTT та CoAP.

Проєктувальники протоколів змушені свідомо скорочувати ресурсоємні механізми, роблячи акцент на стиснених форматах повідомлень, спрощених послідовностях обміну та застосуванні полегшених методів шифрування за потреби.

Мережеві протоколи мають адаптуватися до різноманітних топологій і динамічних умов функціонування, зокрема мобільності вузлів та змін

навколишнього середовища. Мобільні ad-hoc мережі (MANET) і транспортні мережі створюють додаткові виклики, оскільки протоколи мають працювати в умовах частих змін топології та нестабільної якості радіосигналу.

Для таких сценаріїв були розроблені спеціалізовані протоколи маршрутизації, зокрема AODV (Ad hoc On-Demand Distance Vector Routing), які забезпечують самоконфігуровану, безпетельну та багатохопову маршрутизацію в ad-hoc мережах, ефективно реагуючи на динамічні зміни мережевої структури.

2.6 Контрольні питання

1. Що таке мережевий протокол та які основні функції він виконує в процесі взаємодії між мережевими пристроями?
2. У чому полягає призначення моделей OSI та TCP/IP і як вони сприяють стандартизації та сумісності мережевих протоколів?
3. За якими ознаками класифікуються мережеві протоколи та наведіть приклади протоколів комунікації, маршрутизації, управління і безпеки.
4. Порівняйте протоколи TCP та UDP за принципами роботи, надійністю передачі даних і типовими сферами застосування.
5. Яку роль відіграють протоколи безпеки (SSL/TLS) у сучасних мережах і які механізми захисту даних вони забезпечують?
6. У чому полягає призначення моделі OSI та які переваги дає поділ мережевої взаємодії на сім функціональних рівнів?
7. Які функції виконує фізичний і канальний рівні моделі OSI та які стандарти або протоколи є для них характерними?
8. Яку роль відіграють мережевий і транспортний рівні в процесі маршрутизації та надійного передавання даних між вузлами мережі?
9. Які завдання виконують сеансовий, рівень подання та прикладний рівні, і як вони забезпечують коректну взаємодію прикладних програм у мережі?
10. Як модель OSI використовується на практиці для аналізу, діагностики та усунення мережевих несправностей?
11. У чому полягає суть моделі TCP/IP та які її ключові відмінності від семирівневої моделі OSI?
12. Які функції виконує кожен із чотирьох рівнів моделі TCP/IP та які основні протоколи до них належать?
13. Яке призначення інтернет-рівня моделі TCP/IP і яку роль у ньому відіграють протоколи IP, ICMP та ARP?
14. Порівняйте протоколи TCP та UDP за принципами роботи, механізмами забезпечення надійності та типовими сферами застосування.
15. Яке значення прикладного рівня моделі TCP/IP для взаємодії мережевих протоколів із програмними застосунками та які сервіси він забезпечує?
16. Яке функціональне призначення поширених інтернет-протоколів та яку роль вони відіграють у забезпеченні взаємодії сервісів у глобальній мережі Інтернет?

17. У чому полягають принципи роботи протоколу HTTP, які основні методи він підтримує та які переваги забезпечують версії HTTP/2 і HTTP/3?

18. Які особливості архітектури протоколу FTP та в чому полягає різниця між активним і пасивним режимами передавання даних?

19. Яке призначення протоколу SMTP в системах електронної пошти та як він взаємодіє з протоколами POP3 і IMAP?

20. Як функціонує система доменних імен (DNS), які основні типи DNS-записів використовуються та яке їх значення для роботи інтернет-сервісів?

21. Які основні проблеми виникають під час реалізації мережевих протоколів і як сумісність та інтероперабельність впливають на їхню роботу?

22. Які виклики пов’язані з безпекою протоколів, і як використання TLS/SSL забезпечує конфіденційність та цілісність даних?

23. Як протоколи досягають масштабованості та високої продуктивності, і які технології застосовуються для розподілу навантаження в мережах?

24. Які методи забезпечують стійкість протоколів до втрат пакетів та помилок передачі, і як у цьому контексті застосовуються TCP та UDP з механізмом підтверджень?

25. Які особливості реалізації мережевих протоколів у ресурсозалежних середовищах (IoT, вбудовані системи) і як легковагові протоколи (MQTT, CoAP) вирішують ці проблеми?

3 ПРОГРАМУВАННЯ СОКЕТІВ У C++

У розділі подано детальне вивчення програмування сокетів мовою C++, що є фундаментальним аспектом розвитку мережевих комунікацій. Розділ розпочинається з визначення поняття сокетів та їх ключової ролі у створенні мережевих застосунків. Завдяки покроковим інструкціям здобувачі навчаються створювати та керувати сокетами, зокрема здійснювати прив'язку (binding), прослуховування (listening) та приймання з'єднань (accepting connections) для забезпечення взаємодії клієнт-сервер. Крім того, розділ охоплює основні методи надсилання та прийому даних, а також ефективні підходи до обробки помилок, що дає змогу здобувачам опанувати навички розробки надійних мережевих застосунків із використанням сокетів.

3.1 Поняття сокетів

У комп'ютерних науках сокет є фундаментальним та об'єднувальним інтерфейсом для мережевих комунікацій, який з'єднує програмні застосунки через мережу. Сокети відіграють ключову роль у забезпеченні взаємодії між різними процесами та системами, дозволяючи ефективно обмінюватися даними через стандартизовані інтерфейси. Для розуміння сокетів необхідно усвідомлювати як їх теоретичну основу, так і практичне застосування.

Поняття сокета виникло в операційних системах Berkeley Software Distribution (BSD) UNIX. Сокет являє собою кінцеву точку для надсилання та прийому даних через комп'ютерну мережу. У середовищі Unix-подібних систем сокет розглядається як дескриптор файлу, який програми використовують для встановлення каналу комунікації з іншими процесами, як локально на тій самій машині, так і на віддаленій системі.

Ця концепція поширена в більшості інших операційних систем, що робить програмування сокетів універсально застосовним навиком.

Сокети можна розглядати як рівень абстракції, який відокремлює специфіку фізичної мережі від прикладних шарів, дозволяючи розробникам зосередитися на високорівневих аспектах передачі даних без занурення в особливості самого середовища. Технічно сокет визначається IP-адресою та номером порту, утворюючи унікальний ідентифікатор для комунікацій у мережах на основі Інтернет-протоколу (IP).

Типи сокетів можна умовно класифікувати за моделлю комунікації та характером використання:

Потокові сокети (SOCK_STREAM) забезпечують двосторонні, надійні та впорядковані канали комунікації поверх протоколу TCP. Потокові сокети гарантують цілісність та послідовність даних, що робить їх придатними для застосунків, де важливі порядок доставки та надійність, таких як HTTP, FTP та SMTP. Через орієнтованість TCP на з'єднання,

потрібно встановити та підтримувати з’єднання, що передбачає додаткові витрати на керування станами та рукостисканнями (handshake).

Дейтеграмні сокети (SOCK_DGRAM) використовують протокол UDP і забезпечують передавання повідомлень без встановлення з'єднання, де кожне повідомлення є незалежним пакетом. На відміну від потокових сокетів, вони не гарантують послідовності або надійності передачі, що може бути корисним для застосунків, де пріоритетом є швидкість та ефективність, наприклад, потокові сервіси, онлайн-ігри та VoIP.

Сирі сокети (Raw Sockets) надають прямий доступ до нижчих шарів протоколів, дозволяючи маніпулювати заголовками IP-пакетів. Сирі сокети використовуються для спеціалізованих цілей, таких як тестування нових протоколів, моніторинг мереж або проведення оцінення безпеки.

Сокети з послідовними пакетами (SOCK_SEQPACKET) поєднують властивості SOCK_STREAM та SOCK_DGRAM, забезпечуючи надійну послідовність пакетів через орієнтоване на з’єднання обслуговування.

Життєвий цикл сокета передбачає виконання низки чітко визначених операцій у певній послідовності. Розуміння цього циклу є критично важливим для правильного впровадження комунікацій на основі сокетів:

Створення сокета: початковий етап передбачає створення сокета за допомогою системного виклику `socket`, який потребує визначення сімейства протоколів (зазвичай `AF_INET` для IPv4 або `AF_INET6` для IPv6), типу сокета та протоколу.

Наприклад, створення TCP-потокового сокета в C++ для IPv4 може виглядати так:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstdio>

int main() {
    // Створення TCP сокета
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Error creating socket");
        return 1;
    }
    printf("Socket successfully created with fd = %d\n", socket_fd);

    // Закриття сокета після використання
    close(socket_fd);
    return 0;
}
```

Прив’язка (Binding). Ассоціювання сокета з конкретним портом та IP-адресою на локальному комп’ютері здійснюється за допомогою виклику |bind|. Цей процес призначає сокету конкретну точку кінцевого зв’язку в межах локального мережевого інтерфейсу, що забезпечує його ідентифікацію під час надходженні даних.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstdio>
#include <cstring> // для memset

int main() {
    // Створення TCP сокета
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Error creating socket");
        return 1;
    }

    // Ініціалізація структури адреси сервера
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address)); // обнулення структури
    server_address.sin_family = AF_INET; // IPv4
    server_address.sin_port = htons(8080); // порт 8080, переведено в мережевий порядок байтів
    server_address.sin_addr.s_addr = INADDR_ANY; // приймати з будь-якого локального IP

    // Прив'язка сокета до адреси та порту
    if (bind(socket_fd, (struct sockaddr*)&server_address, sizeof(server_address)) < 0) {
        perror("Binding failed");
        close(socket_fd);
        return 1;
    }

    printf("Socket successfully bound to port 8080\n");

    // Закриття сокета після використання
    close(socket_fd);
    return 0;
}
```

Прослуховування (Listening): Насамперед актуально для серверних застосунків, сокет налаштовується на прийом вхідних з’єднань. Функція |listen| визначає максимальну кількість підключень, що можуть очікувати в черзі.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstdio>
#include <cstring>

int main() {
    // Створення TCP сокета
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Error creating socket");
        return 1;
    }

    // Ініціалізація структури адреси сервера
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(8080);
    server_address.sin_addr.s_addr = INADDR_ANY;

    // Прив'язка сокета
    if (bind(socket_fd, (struct sockaddr*)&server_address,
sizeof(server_address)) < 0) {
        perror("Binding failed");
        close(socket_fd);
        return 1;
    }

    // Встановлення сокета в режим прослуховування
    if (listen(socket_fd, 5) < 0) {
        perror("Error on listen");
        close(socket_fd);
        return 1;
    }

    printf("Server listening on port 8080...\n");

    // Закриття сокета після використання
    close(socket_fd);
```

```
    return 0;
}
```

Прийняття підключень. Особливо актуально для серверних реалізацій, для кожного підключення клієнта створюється окремий сокет внаслідок виклику системної функції |accept|. Це забезпечує виділений канал зв’язку:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstdio>
#include <cstring>

int main() {
    // Створення TCP сокета
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Error creating socket");
        return 1;
    }

    // Налаштування адреси сервера
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(8080);
    server_address.sin_addr.s_addr = INADDR_ANY;

    // Прив'язка сокета
    if (bind(socket_fd, (struct sockaddr*)&server_address,
sizeof(server_address)) < 0) {
        perror("Binding failed");
        close(socket_fd);
        return 1;
    }

    // Прослуховування
    if (listen(socket_fd, 5) < 0) {
        perror("Error on listen");
        close(socket_fd);
        return 1;
    }

    printf("Server listening on port 8080...\n");
```

```
    // Прийом з’єднання від клієнта
    struct sockaddr_in client_address;
    socklen_t client_length = sizeof(client_address);
    int client_socket = accept(socket_fd, (struct sockaddr*)&client_address,
&client_length);
    if (client_socket < 0) {
        perror("Accept failed");
        close(socket_fd);
        return 1;
    }

    printf("Client connected!\n");

    // Закриття сокетів після використання
    close(client_socket);
    close(socket_fd);

    return 0;
}
```

Передача даних. Основна функція, під час якої пакети даних надсилаються та приймаються через встановлені сокети з використанням системних викликів |send| та |recv|:

```
Варіант 1
#include <iostream>
#include <cstring>      // Для memset
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h> // Для sockaddr_in
#include <unistd.h>     // Для close()

int main() {
    // 1. Створення TCP-сокета
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Помилка створення сокета");
        return 1;
    }
    std::cout << "Сокет створено успішно.\n";

    // 2. Налаштування адреси сервера
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address)); // Очищення
структури
    server_address.sin_family = AF_INET;              // Використання IPv4
```

```
    server_address.sin_port = htons(8080);          // Порт сервера
    server_address.sin_addr.s_addr = INADDR_ANY;          // Прийом з будь-якої адреси

    // 3. Прив’язка сокета до адреси
    if (bind(socket_fd, (struct sockaddr*)&server_address, sizeof(server_address)) < 0) {
        perror("Помилка прив’язки сокета");
        close(socket_fd);
        return 1;
    }
    std::cout << "Сокет прив’язано до порту 8080.\n";

    // 4. Перехід у режим прослуховування
    if (listen(socket_fd, 5) < 0) {
        perror("Помилка встановлення прослуховування");
        close(socket_fd);
        return 1;
    }
    std::cout << "Сервер прослуховує порт 8080...\n";

    // 5. Очікування підключення клієнта
    struct sockaddr_in client_address;
    socklen_t client_length = sizeof(client_address);

    int client_socket = accept(socket_fd, (struct sockaddr*)&client_address, &client_length);
    if (client_socket < 0) {
        perror("Помилка прийому підключення");
        close(socket_fd);
        return 1;
    }
    std::cout << "Клієнт підключено.\n";

    // 6. Передача та прийом даних
    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));

    // Надсилання повідомлення клієнту
    std::string message = "Привіт від TCP-сервера!";
    int n = send(client_socket, message.c_str(), message.length(), 0);
    if (n < 0) {
        perror("Помилка надсилання даних");
    }

    // Прийом повідомлення від клієнта
    n = recv(client_socket, buffer, sizeof(buffer), 0);
```

```
    if (n < 0) {
        perror("Помилка отримання даних");
    } else {
        std::cout << "Отримано від клієнта: " << buffer << std::endl;
    }

    // 7. Закриття сокетів
    close(client_socket);
    close(socket_fd);
    std::cout << "Сокети закрито. Сервер завершив роботу.\n";

    return 0;
}
```

Варіант 2

```
#include <iostream>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // 1. Створення TCP-сокета
    int client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket < 0) {
        perror("Помилка створення сокета");
        return 1;
    }
    std::cout << "Сокет клієнта створено успішно.\n";

    // 2. Налаштування адреси сервера
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(8080); // порт сервера

    // Використовуємо localhost (127.0.0.1)
    if (inet_pton(AF_INET, "127.0.0.1", &server_address.sin_addr) <= 0) {
        perror("Невірна IP-адреса сервера");
        close(client_socket);
        return 1;
    }

    // 3. Підключення до сервера
```

```
    if (connect(client_socket, (struct sockaddr*)&server_address, sizeof(server_address)) < 0) {
        perror("Помилка підключення до сервера");
        close(client_socket);
        return 1;
    }
    std::cout << "Підключено до сервера.\n";

    // 4. Прийом повідомлення від сервера
    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));
    int n = recv(client_socket, buffer, sizeof(buffer), 0);
    if (n < 0) {
        perror("Помилка отримання даних");
    } else {
        std::cout << "Повідомлення від сервера: " << buffer << std::endl;
    }

    // 5. Надсилання повідомлення серверу
    std::string message = "Привіт, сервер!";
    n = send(client_socket, message.c_str(), message.length(), 0);
    if (n < 0) {
        perror("Помилка надсилання даних");
    }

    // 6. Закриття сокета
    close(client_socket);
    std::cout << "Клієнт завершив роботу.\n";

    return 0;
}
```

Варіант 3

```
#include <iostream>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // 1. Створення TCP-сокета
    int client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket < 0) {
        perror("Помилка створення сокета");
        return 1;
```

```
    }
    std::cout << "Сокет клієнта створено успішно.\n";

    // 2. Налаштування адреси сервера
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(8080);

    // Використовуємо localhost (127.0.0.1)
    if (inet_pton(AF_INET, "127.0.0.1", &server_address.sin_addr) <= 0) {
        perror("Невірна IP-адреса сервера");
        close(client_socket);
        return 1;
    }

    // 3. Підключення до сервера
    if (connect(client_socket, (struct sockaddr*)&server_address,
sizeof(server_address)) < 0) {
        perror("Помилка підключення до сервера");
        close(client_socket);
        return 1;
    }
    std::cout << "Підключено до сервера.\n";

    // 4. Прийом повідомлення від сервера
    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));
    int n = recv(client_socket, buffer, sizeof(buffer), 0);
    if (n < 0) {
        perror("Помилка отримання даних");
    } else {
        std::cout << "Повідомлення від сервера: " << buffer << std::endl;
    }

    // 5. Надсилання повідомлення серверу
    std::string message = "Привіт, сервер!";
    n = send(client_socket, message.c_str(), message.length(), 0);
    if (n < 0) {
        perror("Помилка надсилання даних");
    }

    // 6. Закриття сокета
    close(client_socket);
    std::cout << "Клієнт завершив роботу.\n";

    return 0;
}
```

Завершення (Termination): Сокет закривається після завершення обміну даними, що звільняє зайняті ресурси системи та завершує зв’язок між клієнтом і сервером.

```
#include <iostream>
#include <cstring>      // Для memset
#include <sys/socket.h> // Для socket, bind, listen, accept
#include <netinet/in.h> // Для sockaddr_in
#include <unistd.h>     // Для close

int main() {
    // **Створення сокета**
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Помилка створення сокета");
        return 1;
    }

    // **Прив’язка сокета до IP та порта**
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address)); // Обнулення структури
    server_address.sin_family = AF_INET;              // IPv4
    server_address.sin_port = htons(8080);            // Порт 8080
    server_address.sin_addr.s_addr = INADDR_ANY;      // Прийом з будь-якого IP

    if (bind(socket_fd, (struct sockaddr*)&server_address, sizeof(server_address)) < 0) {
        perror("Помилка прив’язки сокета");
        close(socket_fd);
        return 1;
    }

    // **Прослуховування вхідних з’єднань**
    if (listen(socket_fd, 5) < 0) {
        perror("Помилка прослуховування");
        close(socket_fd);
        return 1;
    }
    std::cout << "Сервер запущено, очікування підключень на порту 8080..." << std::endl;

    while (true) {
        // **Прийом клієнтського з’єднання**
        struct sockaddr_in client_address;
```

```
        socklen_t client_length = sizeof(client_address);
        int client_socket = accept(socket_fd, (struct
sockaddr*)&client_address, &client_length);
        if (client_socket < 0) {
            perror("Помилка прийому з’єднання");
            continue; // Продовжуємо очікування інших клієнтів
        }

        std::cout << "Підключено нового клієнта." << std::endl;

        // **Обмін даними з клієнтом**
        char buffer[1024];
        memset(buffer, 0, sizeof(buffer));

        int n = recv(client_socket, buffer, sizeof(buffer), 0);
        if (n < 0) {
            perror("Помилка отримання даних");
        } else {
            std::cout << "Отримано повідомлення: " << buffer << std::endl;

            // Відправка відповіді
            n = send(client_socket, buffer, strlen(buffer), 0);
            if (n < 0) {
                perror("Помилка надсилання даних");
            }
        }

        // **Закриття клієнтського сокета**
        close(client_socket);
        std::cout << "З’єднання з клієнтом завершено." << std::endl;
    }

    // **Закриття головного сокета сервера**
    close(socket_fd);
    return 0;
}
```

Сокети також містять набір параметрів, що визначають їхню поведінку під час комунікацій, зокрема опції сокета, які можна налаштовувати за допомогою функції |setsockopt|, включно з параметрами TCP_NODELAY, SO_REUSEADDR та SO_LINGER. Ці опції змінюють способи буферизації даних, повторного використання з’єднань та обробки незавершених комунікацій.

Розуміння комунікацій через сокети передбачає врахування кількох критично важливих аспектів.

Контроль конкурентного доступу. Оскільки сокети є спільним ресурсом, а обмін даними відбувається асинхронно, одночасні

підключення можуть спричиняти складне переплетіння комунікацій, якщо їх не скеровувати належним чином. Це часто потребує використання механізмів конкурентного програмування, таких як багатопоточність або асинхронні операції введення/виведення.

Мережевий порядок байтів. Для забезпечення узгодженості подання даних у мережах використовуються функції конверсії, такі як |htonl| (host to network long) та |ntohl| (network to host long), які гарантують правильний порядок байтів на різних архітектурах. Це особливо важливо під час обміну двійковими даними між гетерогенними системами.

Блокувальне та неблокувальне введення/виведення. За замовчуванням сокети працюють у блокувальному режимі, що може призупиняти виконання процесу до завершення операцій. Неблокувальне введення/виведення дозволяє функціям повертати керування негайно, запобігаючи зависанню програми та забезпечуючи ефективне подієво-орієнтоване програмування.

Розширені операції з сокетами містять інтеграцію з технологіями Secure Sockets Layer (SSL) для шифрування комунікацій, що є критично важливим для безпечного обміну даними через публічні мережі. Бібліотеки на кшталт OpenSSL надають API, які абстрагують ці складності, дозволяючи розробникам захищати комунікації сокетів без необхідності глибокого занурення в криптографію.

Універсальність та поширеність сокетів зробили їх основою для численних мережевих застосунків, не обмежуючись лише вебсерверами, а й включно з комунікацією баз даних, мультимедійними додатками та розподіленими системами. Незалежно від того, чи йдеться про просту локальну взаємодію між процесами, чи про масштабовану багаторівневу мережу, сокети значно спрощують роботу програміста.

Розвинене розуміння сокетів суттєво допомагає у проєктуванні систем, що є ефективними, масштабованими та адаптованими до майбутніх мережевих протоколів і стандартів. Оволодіння як синтаксичною основою, так і концептуальними особливостями програмування сокетів відкриває шлях до створення складних і високочутливих мережевих застосунків.

3.2 Створення сокета

Створення сокета в C++ є початковим кроком для забезпечення мережевої комунікації між різними програмами або пристроями через мережу. Сокет слугує наріжним каменем розробки мережевих застосунків, надаючи кінцеву точку комунікації для відправки та прийому даних. У цьому підрозділі запропоновано різнобічні поради зі створення сокетів у C++ із використанням відповідних бібліотек, з акцентом на практичність та детальні пояснення основних процесів.

У C++ робота з сокетами здійснюється через системні виклики, інкапсульовані в API POSIX (Portable Operating System Interface), що містить набір стандартизованих інтерфейсів операційної системи. Це забезпечує високу портативність програм, що використовують сокети, на

різних Unix-подібних системах. Середовища Windows, хоча й подібні, потребують незначних відхилень через відмінності у системних бібліотеках та заголовочних файлах.

Реалізація сокетів у C++ зазвичай передбачає підключення певних базових бібліотек, зокрема заголовочного файлу `<sys/socket.h>` для системних викликів сокетів та `<netinet/in.h>` для роботи з інтернет-адресами. Для підтримки цих функцій можуть знадобитися додаткові заголовочні файли, такі як `<arpa/inet.h>` для роботи з IP-адресами та `<unistd.h>` для системних викликів.

Сокет у C++ ініціалізується за допомогою системного виклику `socket`, створюючи незв'язаний сокет у вказаному комунікаційному домені:

```
#include <iostream>       // Для виведення повідомлень у консоль
#include <sys/socket.h>   // Для функцій socket(), bind(), listen(), accept()
#include <netinet/in.h>   // Для структур sockaddr_in та htons()
#include <unistd.h>       // Для close()

int main() {
    // 1. Створення TCP-сокета
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        perror("Помилка створення сокета");
        return 1;
    }
    std::cout << "TCP-сокет успішно створено.\n";

    // 2. Визначення адреси сервера
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;                    // Встановлення сімейства адрес (IPv4)
    server_address.sin_port = htons(8080);                  // Встановлення порта (перетворюємо в мережевий порядок байтів)
    server_address.sin_addr.s_addr = INADDR_ANY;            // Приймаємо з'єднання на всіх інтерфейсах

    // 3. Прив'язка сокета до адреси та порта
    if (bind(socket_fd, (struct sockaddr*)&server_address, sizeof(server_address)) < 0) {
        perror("Помилка прив'язки сокета");
        close(socket_fd);
        return 1;
    }
    std::cout << "Сокет успішно прив'язано до адреси.\n";

    // 4. Встановлення сокета в режим прослуховування
    if (listen(socket_fd, 5) < 0) {
```

```
        perror("Помилка прослуховування");
        close(socket_fd);
        return 1;
    }
    std::cout << "Сервер прослуховує порт 8080...\n";

    // 5. Очікування та прийом з’єднання від клієнта
    struct sockaddr_in client_address;
    socklen_t client_length = sizeof(client_address);
    int client_socket = accept(socket_fd, (struct sockaddr*)&client_address,
&client_length);
    if (client_socket < 0) {
        perror("Помилка прийому з’єднання");
        close(socket_fd);
        return 1;
    }
    std::cout << "Підключено клієнта.\n";

    // 6. Передача та прийом даних
    char buffer[1024] = "Привіт від сервера!";
    int n = send(client_socket, buffer, sizeof(buffer), 0);
    if (n < 0) {
        perror("Помилка відправки даних");
    }

    n = recv(client_socket, buffer, sizeof(buffer), 0);
    if (n < 0) {
        perror("Помилка отримання даних");
    } else {
        std::cout << "Отримано від клієнта: " << buffer << "\n";
    }

    // 7. Закриття сокетів
    close(client_socket);
    close(socket_fd);
    std::cout << "Сокети закрито, сервер завершив роботу.\n";

    return 0;
}
```

Домен. Визначає сімейство протоколів, що використовуватиметься з сокетом. Зазвичай для мереж IPv4 застосовується AF_INET, а для IPv6 – AF_INET6. Цей параметр визначає формат адреси та сімейство протоколів для сокета. Інші домени містять AF_UNIX для локальної взаємодії через сокети.

Тип. Визначає семантику комунікації сокета. Основні значення містять:

SOCK_STREAM: для надійної, орієнтованої на з'єднання передачі даних через TCP.

SOCK_DGRAM: для безз'єднувальної комунікації через UDP.

SOCK_RAW: забезпечує роботу з «сирим» сокетом для прямого доступу до мережевого рівня.

Протокол. Зазвичай встановлюється таким, що дорівнює 0; це дозволяє операційній системі автоматично вибрати відповідний протокол залежно від зазначеного домену та типу. Цей параметр також може бути явно визначений, якщо в межах одного домену існує декілька протоколів.

Приклад створення сокета

Для створення TCP-сокета з IPv4 виклик функції socket виглядає таким чином:

```
#include <sys/socket.h>   // Для викликів socket(), bind(), listen(), accept() тощо
#include <netinet/in.h>   // Для структур sockaddr_in та AF_INET
#include <arpa/inet.h>    // Для функцій inet_addr, inet_ntoa
#include <unistd.h>       // Для close()
#include <iostream>       // Для введення/виведення через cout/cerr
#include <cstring>        // Для функцій роботи з рядками та буферами

using namespace std;

int main() {
   int sockfd;

   // Створення TCP-сокета з IPv4
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) {
      cerr << "Помилка при створенні сокета" << endl;
      return 1;
   }

   cout << "Сокет успішно створено" << endl;

   // Закриття сокета для звільнення ресурсів
   close(sockfd);

   return 0;
}
```

Код вище: містить необхідні заголовочні файли для стандартного введення/виведення та функцій, пов'язаних із сокетами. Оголошує цілу змінну `sockfd` для збереження дескриптора сокета, що повертається

функцією `socket`. Цей дескриптор аналогічний файловому дескриптору, який використовується під час роботи з файлами. Ініціалізує сокет за допомогою `AF_INET`, `SOCK_STREAM` та протоколу 0. Реалізує базову обробку помилок, перевіряючи, чи є дескриптор сокета від'ємним, що сигналізує про помилку під час створення сокета. Закриваємо сокет за допомогою системного виклику `close`. Закриття сокета є необхідним для звільнення виділених ресурсів.

Надійна обробка помилок є критичною під час роботи із сокетами. Для отримання зрозумілих для користувача повідомлень про помилки можуть використовуватися функції `perror` або `strerror(errno)`.

```
#include <iostream>      // Для введення/виведення через cout, cerr
#include <sys/socket.h>  // Для системного виклику socket, bind, listen, accept
#include <netinet/in.h>   // Для структури sockaddr_in та констант AF_INET, INADDR_ANY
#include <arpa/inet.h>   // Для функцій роботи з IP-адресами
#include <unistd.h>      // Для функції close
#include <cerrno>        // Для errno
#include <cstdio>        // Для perror
#include <cstring>       // Для memset

using namespace std;

int main() {
   int sockfd;

   // Створення TCP-сокета для IPv4
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) {
      perror("Помилка створення сокета");
      return 1;  // Завершення програми у разі помилки
   }
   cout << "Сокет успішно створено" << endl;

   // Налаштування адреси сервера
   struct sockaddr_in server_address;
   memset(&server_address, 0, sizeof(server_address));     // Очищення структури
   server_address.sin_family = AF_INET;              // Використання IPv4
   server_address.sin_port = htons(8080);            // Встановлення порту 8080
   server_address.sin_addr.s_addr = INADDR_ANY;              // Прийом підключень з будь-якої IP-адреси

   // Прив'язка сокета до зазначеної адреси
```

```
    if (bind(sockfd, (struct sockaddr*)&server_address, sizeof(server_address)) < 0) {
        perror("Помилка прив'язки сокета");
        close(sockfd);  // Закриття сокета у разі помилки
        return 1;
    }
    cout << "Сокет прив'язано до порту 8080" << endl;

    // Переведення сокета в режим прослуховування
    if (listen(sockfd, 5) < 0) {
        perror("Помилка прослуховування");
        close(sockfd);
        return 1;
    }
    cout << "Сервер готовий приймати підключення" << endl;

    // Очікування підключення клієнта
    struct sockaddr_in client_address;
    socklen_t client_len = sizeof(client_address);
    int client_socket = accept(sockfd, (struct sockaddr*)&client_address, &client_len);
    if (client_socket < 0) {
        perror("Помилка прийняття підключення");
        close(sockfd);
        return 1;
    }
    cout << "Клієнт підключений" << endl;

    // Передача та прийом даних
    char buffer[1024] = "Привіт, клієнт!";
    int n = send(client_socket, buffer, strlen(buffer), 0);
    if (n < 0) {
        perror("Помилка відправки даних");
    } else {
        cout << "Повідомлення відправлено клієнту" << endl;
    }

    n = recv(client_socket, buffer, sizeof(buffer), 0);
    if (n < 0) {
        perror("Помилка прийому даних");
    } else {
        buffer[n] = '\0';  // Завершення рядка
        cout << "Отримано від клієнта: " << buffer << endl;
    }

    // Закриття сокетів
    close(client_socket);
```

```
    close(sockfd);
    cout << "Сокети закрито, сервер завершив роботу" << endl;

    return 0;
}
```

Використання функції `perror` дозволяє отримати детальну інформацію про помилки, пов’язані з недостатністю ресурсів, проблемами дозволів або неподтримуваними конфігураціями, що допомагає відрізняти тимчасові несправності від стійких проблем.

Сучасні мережі часто потребують підтримки як IPv4, так і IPv6 схем адресації. Сокети, адаптовані для IPv6, використовують домен AF_INET6.

```
// Створення TCP-сокета для IPv6
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cerrno>
#include <cstdio>
#include <iostream>

using namespace std;

int main() {
    int sockfd;

    // Створення сокета для IPv6
    sockfd = socket(AF_INET6, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Помилка створення сокета");
        return 1;
    }

    cout << "Сокет для IPv6 створено успішно" << endl;

    // Закриття сокета після використання
    close(sockfd);

    return 0;
}
```

Сокети IPv6 зазвичай здатні обробляти IPv4-адреси, відображені в IPv6 (IPv4-mapped IPv6), що забезпечує сумісність між протоколами. Така гармонізація полегшує розробку програм, які мають працювати одночасно в обох схемах адресації без значних змін у коді.

Продуктивність та поведінку сокета можна налаштовувати за допомогою параметрів сокета, керованих через функцію `setsockopt`. До таких параметрів належать:

- ✓ Повторне використання адреси (Address Reusability): `SO_REUSEADDR` дозволяє прив'язати сокет до адреси, яка вже використовується, уникнувши затримок під час запуску сервера.
- ✓ Параметри завершення з'єднання (Linger Options): `SO_LINGER` визначає поведінку сокета під час його закриття, зокрема, скільки часу сокет залишається активним для завершення відправки відкладених даних.

Неблокувальний режим (Non-blocking Mode): за допомогою `fcntl` сокет можна перевести у неблокувальний режим, що є корисним для асинхронних подій та моделей керування подіями. Приклад встановлення сокета в неблокувальний режим:

```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    // Створення TCP сокета
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("socket failed");
        return 1;
    }

    std::cout << "Socket created successfully, descriptor: " << sockfd << std::endl;

    // Отримання поточних прапорів сокета
    int flags = fcntl(sockfd, F_GETFL, 0);
    if (flags < 0) {
        perror("fcntl F_GETFL failed");
        close(sockfd);
        return 1;
    }

    // Встановлення неблокувального режиму
    if (fcntl(sockfd, F_SETFL, flags | O_NONBLOCK) < 0) {
        perror("fcntl F_SETFL failed");
        close(sockfd);
        return 1;
    }
```

```
    std::cout << "Socket set to non-blocking mode successfully" << std::endl;

    // Закриття сокета
    close(sockfd);
    return 0;
}
```

Блокувальна природа сокетів передбачає, що певні операції призупиняють виконання програми, доки дані не будуть повністю надіслані або отримані. Хоча такий підхід є простим у реалізації, блокувальні сокети ускладнюють створення відзивних систем у середовищах із високою конкуренцією потоків, що може призводити до неефективності, оскільки потоки очікують у стані простою.

Неблокувальні сокети вирішують цю проблему, дозволяючи продовжувати виконання операцій, навіть якщо передача даних ще не завершена.

Для ефективного керування численними запитами до сокетів застосовують методи мультиплексування введення/виведення, такі як `select`, `poll` або `epoll`, які дозволяють динамічно реагувати на змінні потреби мережі. Ці методи дозволяють одному потоку моніторити кілька сокетів одночасно, адаптуючись до вимог щодо читання або записування даних.

Хоча API POSIX переважно використовується в Unix-подібних операційних системах, у Windows для роботи з сокетами застосовується бібліотека Winsock, що призводить до невеликих відмінностей у включенні заголовочних файлів, підключенні бібліотек та у вимогах до ініціалізації. Приклад для Windows передбачає початкову ініціалізацію бібліотеки Winsock.

```
#include <iostream>

#ifdef _WIN32
    #include <winsock2.h>
    #include <ws2tcpip.h>
    #pragma comment(lib, "ws2_32.lib")
    typedef SOCKET socket_t;
#else
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <unistd.h>
    typedef int socket_t;
#endif

using namespace std;
```

```
int main() {

#ifdef _WIN32
    // Ініціалізація бібліотеки Winsock
    WSADATA wsaData;
    int result = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0) {
        cerr << "WSAStartup failed: " << result << endl;
        return 1;
    }
#endif

    // Створення TCP-сокета
    socket_t sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
#ifdef _WIN32
        WSACleanup();
#endif
        return 1;
    }

    cout << "Socket created successfully" << endl;

    // Закриття сокета
#ifdef _WIN32
    closesocket(sockfd);
    WSACleanup();
#else
    close(sockfd);
#endif

    return 0;
}
```

Тут функції WSAStartup та WSACleanup забезпечують керування життєвим циклом бібліотеки Winsock, а підключення заголовного файлу <winsock2.h> адаптує сокетно-орієнтований програмний код до середовища операційної системи Windows.

Створення сокета в мові програмування C++ є значно ширшим процесом, ніж просте відкриття каналу обміну інформацією; воно становить фундаментальний етап проєктування ефективних мережевих застосувань. Гнучкість, яку забезпечує механізм сокетів, дозволяє розробникам інкапсулювати складність базових мережевих протоколів і зосередитися на забезпеченні безпечної та надійної передачі даних. Закладення базових принципів створення сокетів робить подальші етапи – зокрема встановлення з’єднання, прив’язування до адреси та обмін даними

– більш доступними для реалізації, що, зі свого боку, спрощує процес опанування методів розроблення масштабованих і надійних мережевих застосувань мовою C++.

3.3 Прив’язування, прослуховування та приймання з’єднань

У сокетному програмуванні перехід від етапу створення сокета до безпосередньої обробки мережевих з’єднань передбачає послідовне виконання операцій прив’язування (binding), прослуховування (listening) та приймання з’єднань (accepting). Зазначені кроки готують сокет до приймання вхідних запитів, фактично створюючи основу для надійної взаємодії за моделлю клієнт–сервер. У цьому розділі детально розглядається кожен із цих процесів у контексті програмування мовою C++, з акцентом на особливості реалізації та рекомендовані практики.

Робочий процес прив’язування, прослуховування та приймання з’єднань зосереджений навколо серверного сокета, який виконує роль початкової точки контакту в архітектурі клієнт–сервер. Ефективне керування цими процесами забезпечує серверу можливість обслуговувати кілька клієнтів одночасно, організовувати ефективну обробку обміну даними та підтримувати стабільність системи за різних умов навантаження.

Прив’язування сокета полягає у встановленні відповідності між сокетом та конкретною IP-адресою і номером порту на локальній машині. Така відповідність визначає унікальну кінцеву точку, через яку можуть передаватися та прийматися дані.

Для виконання цієї операції використовується функція bind, яка приймає три основні аргументи: дескриптор сокета, вказівник на структуру адреси (наприклад, struct sockaddr_in для протоколу IPv4) та розмір структури адреси. Типова операція прив’язування передбачає обов’язкову перевірку можливих помилок, що дозволяє своєчасно виявити проблеми конфігурації або конфлікти використання адреси.

```
#include <iostream>      // стандартні засоби введення та виведення
#include <cstring>       // функції для роботи з пам’яттю (memset)
#include <sys/socket.h>  // базові функції сокетів
#include <netinet/in.h>  // структури та константи для IPv4
#include <arpa/inet.h>   // перетворення мережевих адрес
#include <unistd.h>      // системні виклики POSIX, зокрема close()

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    sockaddr_in server_address;
```

```
    std::memset(&server_address, 0, sizeof(server_address)); // ініціалізація структури адреси нулями

    server_address.sin_family = AF_INET;       // використання протоколу IPv4
    server_address.sin_port = htons(8080);    // перетворення номера порту у мережевий порядок байтів
    server_address.sin_addr.s_addr = INADDR_ANY; // прив'язка до будь-якого доступного мережевого інтерфейсу

    if (bind(sockfd,
        (struct sockaddr*)&server_address,
        sizeof(server_address)) < 0) {
      perror("Bind failed");
      close(sockfd); // звільнення ресурсів сокета у разі помилки
      return 1;
    }

    std::cout << "Socket successfully bound" << std::endl; // підтвердження успішної прив'язки сокета

    close(sockfd); // закриття сокета та звільнення системних ресурсів
    return 0;
}
```

Функція `bind` здійснює прив'язку сокета до конкретного номера порту та IP-адреси. Номер порту є критично важливим параметром, оскільки він слугує точкою доступу для вхідних повідомлень, тоді як IP-адреса визначає мережевий інтерфейс, на якому сервер очікує запити. Використання константи `INADDR_ANY` дозволяє серверу приймати з'єднання на всіх доступних мережевих інтерфейсах хоста.

Константа `AF_INET` вказує на використання протоколу IPv4. У розширених реалізаціях, орієнтованих на підтримку IPv6, застосовується структура `struct sockaddr_in6`, що відповідає адресному простору IPv6.

Застосування функції `htons` забезпечує коректне перетворення номера порту у мережевий порядок байтів, що є необхідним для міжплатформної взаємодії з огляду на відмінності в порядку байтів (endianness) на різних апаратних архітектурах.

Після успішної прив'язки сокета його необхідно перевести в режим прослуховування для підготовки до приймання вхідних з'єднань. Це здійснюється за допомогою системного виклику `listen`, який визначає роль сокета як пасивного слухача та формує чергу очікування (backlog) для збереження запитів на встановлення з'єднання, що надходять від клієнтів.

```
#include <iostream>       // std::cout, std::endl
#include <cstring>        // memset
```

```
#include <cstdio>          // perror
#include <sys/socket.h>    // socket, bind, listen
#include <netinet/in.h>    // sockaddr_in, INADDR_ANY
#include <arpa/inet.h>     // htons
#include <unistd.h>        // close

int main() {
   // Створення TCP-сокета
   int sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) {
      perror("Socket creation failed");
      return 1;
   }

   // Налаштування адреси сервера
   sockaddr_in server_address{};
   server_address.sin_family = AF_INET;
   server_address.sin_port = htons(8080);          // мережевий порядок байтів
   server_address.sin_addr.s_addr = INADDR_ANY;    // усі доступні інтерфейси

   // Прив’язка сокета
   if (bind(sockfd,
         (struct sockaddr*)&server_address,
         sizeof(server_address)) < 0) {
      perror("Bind failed");
      close(sockfd);
      return 1;
   }

   // Переведення сокета в режим прослуховування
   if (listen(sockfd, SOMAXCONN) < 0) {
      perror("Listen failed");
      close(sockfd);
      return 1;
   }

   std::cout << "Server is listening on port 8080..." << std::endl;

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

Черга очікування (Backlog Queue). Другий параметр функції `listen` визначає кількість активних, але ще не прийнятих з’єднань, які можуть перебувати в черзі в очікуванні обслуговування. Використання константи

`SOMAXCONN` дозволяє операційній системі динамічно керувати розміром цієї черги з урахуванням поточних ресурсів і навантаження.

Пасивна роль сокета (Passive Role). Режим прослуховування переводить сокет у пасивний стан, за якого він не ініціює обмін даними доти, доки не буде прийнято з'єднання. Такий підхід є критично важливим для ефективної обробки вхідних клієнт-серверних діалогів, оскільки запобігає передчасному завершенню роботи сокета або надмірному використанню системних ресурсів.

Прийняття з'єднань (Accepting Connections). Завершальним етапом підготовки сервера до обробки активних з'єднань є прийняття клієнтських запитів за допомогою функції `accept`. Ця функція створює новий, виділений сокет для кожного встановленого з'єднання з клієнтом, що забезпечує ізоляцію каналів обміну даними та підвищує надійність і масштабованість серверного застосунку.

```
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>   // функція close() для закриття сокета
#include <cerrno>

int main() {
   // 1. Створення серверного сокета
   int sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) {
      perror("Помилка створення сокета");
      return 1;
   }

   // 2. Налаштування адреси сервера
   sockaddr_in server_address{};
   server_address.sin_family = AF_INET;         // використання IPv4
   server_address.sin_port = htons(8080);              // номер порту в мережевому порядку байтів
   server_address.sin_addr.s_addr = INADDR_ANY;  // прослуховування на всіх мережевих інтерфейсах

   // 3. Прив'язка сокета до IP-адреси та порту
   if (bind(sockfd,
         (struct sockaddr*)&server_address,
         sizeof(server_address)) < 0) {
      perror("Помилка прив'язки сокета");
      close(sockfd);
      return 1;
   }
```

```
        // 4. Перехід сокета в режим прослуховування вхідних з’єднань
        if (listen(sockfd, SOMAXCONN) < 0) {
            perror("Помилка переходу в режим прослуховування");
            close(sockfd);
            return 1;
        }

        std::cout << "Сервер перебуває в режимі очікування з’єднань..." <<
std::endl;

        // 5. Прийняття клієнтського з’єднання
        sockaddr_in client_addr{};
        socklen_t client_len = sizeof(client_addr);

        int client_sockfd = accept(
            sockfd,
            (struct sockaddr*)&client_addr,
            &client_len
        );

        if (client_sockfd < 0) {
            perror("Помилка прийняття клієнтського з’єднання");
            close(sockfd);
            return 1;
        }

        std::cout << "З’єднання з клієнтом успішно встановлено" <<
std::endl;

        // 6. Закриття клієнтського та серверного сокетів і звільнення
ресурсів
        close(client_sockfd);
        close(sockfd);

        return 0;
    }
```

Очікування встановлення з’єднання (Connection Acquiescence). Функція `accept()` перебуває у блокувальному режимі до моменту надходження запиту на встановлення з’єднання, після чого створюється новий сокет, призначений для взаємодії між клієнтом і сервером. Це забезпечує формування унікального каналу зв’язку та дає змогу організувати паралельну комунікацію через кілька сокетів, створених для різних клієнтів.

Збереження адреси клієнта (Client Address Storage). Структура `client_addr` використовується для накопичення параметрів підключених клієнтів. Для кожного прийнятого з’єднання можуть бути отримані

ідентифікатори клієнта, зокрема IP-адреса та номер порту, що підвищує рівень керованості та наочності під час адміністрування серверної частини.

Керування паралельністю (Concurrency Management). Обслуговування кожного клієнтського запиту за допомогою окремого сокета сприяє ефективній роботі сервера в умовах багатоклієнтського доступу. Для цього можуть застосовуватися стратегії паралельної обробки, такі як створення дочірніх процесів (`fork`) або використання потоків виконання. Додатково застосування неблокувального введення/виведення з використанням механізмів `select()` або `poll()` підвищує швидкодію та пропускну здатність системи.

Для забезпечення одночасного обслуговування кількох клієнтських з'єднань необхідно використовувати механізми конкурентної обробки. Як приклад може бути реалізований сервер із багатопроцесною або багатопотоковою архітектурою (fork- або multithread-сервер).

Варіант 1

```
#include <iostream>
#include <cstring>
#include <pthread.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>   // close()
#include <errno.h>

using namespace std;

// Функція обробки клієнта (виконується в окремому потоці)
void* handle_client(void* arg) {
    int client_sockfd = *((int*)arg);
    char buffer[1024]{};

    // Приймання повідомлення від клієнта
    recv(client_sockfd, buffer, sizeof(buffer), 0);
    cout << "Повідомлення від клієнта: " << buffer << endl;

    // Надсилання відповіді клієнту (exo)
    send(client_sockfd, buffer, strlen(buffer), 0);

    // Закриття клієнтського сокета
    close(client_sockfd);

    return nullptr;
}
```

```
int main() {
    // Створення серверного сокета
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    // Налаштування адреси сервера
    sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // Прив’язка сокета
    if (bind(sockfd,
             (struct sockaddr*)&server_addr,
             sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(sockfd);
        return 1;
    }

    // Перехід у режим прослуховування
    if (listen(sockfd, SOMAXCONN) < 0) {
        perror("Listen failed");
        close(sockfd);
        return 1;
    }

    cout << "Server is listening..." << endl;

    // Основний цикл приймання клієнтів
    while (true) {
        sockaddr_in client_addr{};
        socklen_t client_len = sizeof(client_addr);

        int client_sockfd = accept(
            sockfd,
            (struct sockaddr*)&client_addr,
            &client_len
        );

        if (client_sockfd < 0) {
            perror("Accept failed");
            continue;
        }
```

```
        pthread_t client_thread;

        // Створення потоку для обслуговування клієнта
        if (pthread_create(
                &client_thread,
                nullptr,
                handle_client,
                (void*)&client_sockfd
            ) != 0) {

            cerr << "Не вдалося створити потік" << endl;
            close(client_sockfd);
        }

        // Відокремлення потоку
        pthread_detach(client_thread);
    }

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

Варіант 2

```
#include <iostream>
#include <cstring>

#ifdef _WIN32
    // ===== Windows (WinSock) =====
    #include <winsock2.h>
    #include <ws2tcpip.h>
    #include <windows.h>
    #pragma comment(lib, "Ws2_32.lib")
    using socket_t = SOCKET;
#else
    // ===== Linux / Unix (POSIX) =====
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
    using socket_t = int;
    #define closesocket close
#endif

int main() {
#ifdef _WIN32
    // Ініціалізація WinSock
```

```
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);
#endif

    socket_t sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(sockfd,
            (sockaddr*)&server_addr,
            sizeof(server_addr)) < 0) {
        perror("Bind failed");
        closesocket(sockfd);
        return 1;
    }

    listen(sockfd, SOMAXCONN);
    std::cout << "Server is listening..." << std::endl;

    closesocket(sockfd);

#ifdef _WIN32
    WSACleanup();
#endif

    return 0;
}
```

Створення та керування потоками: для кожного вхідного з’єднання створюється окремий потік, який відповідає за обробку взаємодії з конкретним клієнтом. Такий підхід відокремлює логіку обслуговування з’єднань від основного потоку виконання та підвищує масштабованість системи за рахунок паралельної обробки запитів.

Відокремлення потоків: виклик `pthread_detach` забезпечує коректне вивільнення системних ресурсів після завершення виконання потоку, запобігаючи витокам пам’яті та надмірному накопиченню ресурсів.

Альтернативний підхід із використанням механізму `fork`: створення окремого процесу для кожного клієнтського з’єднання є доцільним у випадках, коли необхідна жорстка ізоляція протоколів або підвищені вимоги до безпеки. Водночас цей метод характеризується більшими

накладними витратами порівняно з багатопотоковою реалізацією, що потрібно враховувати під час проєктування високопродуктивних серверних застосунків.

```
#include <iostream>
#include <unistd.h>
#include <sys/socket.h>

using namespace std;

// Приклад функції обробки клієнта
void handle_client(int client_sockfd) {
    // тут логіка роботи з клієнтом
    close(client_sockfd);
}

int main() {
    int sockfd = 3;        // умовний серверний сокет
    int client_sockfd = 4; // умовний клієнтський сокет

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    }
    else if (pid == 0) {
        // Дочірній процес: обслуговування клієнта
        close(sockfd);              // серверний сокет не потрібен
        handle_client(client_sockfd); // робота з клієнтом
        _exit(0);                   // коректне завершення дочірнього процесу
    }
    else {
        // Батьківський процес
        close(client_sockfd);       // клієнтський сокет закривається
    }

    return 0;
}
```

Забезпечення своєчасного вивільнення сокетів і раціонального використання системних ресурсів дає змогу уникнути їх тривалого утримання, витоків пам’яті та деградації продуктивності сервера.

Реалізація механізмів автентифікації або застосування міжмережевих екранів (firewall) обмежує несанкціонований доступ і підвищує рівень захисту від уразливостей, що можуть бути використані в умовах відкритих мереж.

Використання балансувальників навантаження дає змогу рівномірно розподіляти вхідні запити між кількома серверами, підвищуючи швидкодію, відмовостійкість і загальну надійність системи.

Опанування процесів прив'язки, прослуховування та приймання з'єднань формує міцну основу для побудови адаптивних і надійних мережевих застосунків, здатних ефективно працювати за високих навантажень і динамічних умов. Практичне розуміння цих механізмів, підкріплене зразками коду та перевіреними практиками, сприяє підвищенню гнучкості застосунків і дозволяє розробникам створювати конкурентоспроможні паралельні мережеві рішення, адаптовані до еволюції технологій.

3.4 Встановлення клієнт-серверної взаємодії

Суть програмування з використанням сокетів полягає в успішному встановленні клієнт-серверної взаємодії – архітектури, що лежить в основі більшості мережевих застосунків. У підрозділі розглянуто етапи та ключові аспекти побудови надійної моделі клієнт-серверної взаємодії мовою C++. На основі детальних прикладів коду та аналітичних пояснень демонструється, як дві окремі сутності – клієнт і сервер – здійснюють безперешкодний обмін даними мережею.

Клієнт-серверна архітектура ґрунтується на принципі розподілу ролей, за якого сервер надає ресурси, сервіси або дані, а клієнт споживає ці сервіси шляхом надсилання запитів. Такий підхід забезпечує чітке розмежування відповідальності між компонентами системи та є ключовим не лише для вебсервісів, а й для застосунків електронної пошти, передавання файлів і доступу до баз даних.

Типовий життєвий цикл клієнт-серверної взаємодії містить такі етапи.

Ініціалізація сервера. Сервер створює сокет, прив'язує його до відповідного порту та переходить у режим прослуховування вхідних клієнтських з'єднань.

Підключення клієнта. Клієнт ініціює взаємодію, створюючи сокет і виконуючи спробу підключення до IP-адреси та порту сервера.

Обмін даними. Після встановлення з'єднання сторони здійснюють передавання даних за допомогою операцій надсилання та приймання.

Завершення з'єднання. Після завершення обміну даними клієнт і сервер коректно закривають з'єднання, вивільняючи ресурси.

Основним завданням сервера є постійна готовність до взаємодії з клієнтами. Серверна реалізація охоплює створення сокета, його прив'язку, перехід у режим прослуховування, приймання вхідних з'єднань і організацію обміну даними між клієнтом і сервером. Для забезпечення одночасної обробки кількох клієнтів комунікація зазвичай реалізується в окремих потоках, що дає змогу підтримувати паралельну взаємодію та підвищувати масштабованість системи.

```
#include <iostream>
#include <cstring>
#include <pthread.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

using namespace std;

const int PORT = 8080;

/**
 * Функція обробки клієнта (виконується в окремому потоці)
 */
void* handle_client(void* arg) {
    int client_fd = *(int*)arg;
    delete (int*)arg;   // звільнення пам’яті

    char buffer[1024]{};

    // Приймання даних від клієнта
    int bytes_received = recv(client_fd, buffer, sizeof(buffer) - 1, 0);

    if (bytes_received < 0) {
        cerr << "Помилка під час приймання даних" << endl;
    } else if (bytes_received == 0) {
        cout << "Клієнт закрив з’єднання" << endl;
    } else {
        buffer[bytes_received] = '\0';
        cout << "Повідомлення від клієнта: " << buffer << endl;

        // Надсилання відповіді
        const char* response = "Message received";
        send(client_fd, response, strlen(response), 0);
    }

    // Закриття клієнтського сокета
    close(client_fd);
    pthread_exit(nullptr);
}

int main() {
    int server_fd;
    sockaddr_in server_addr{}, client_addr{};
```

```
socklen_t client_len = sizeof(client_addr);

// 1. Створення серверного сокета
server_fd = socket(AF_INET, SOCK_STREAM, 0);
if (server_fd < 0) {
    perror("socket failed");
    return -1;
}

// 2. Налаштування адреси сервера
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(PORT);

// 3. Прив’язка сокета
if (bind(server_fd,
         (struct sockaddr*)&server_addr,
         sizeof(server_addr)) < 0) {
    perror("bind failed");
    close(server_fd);
    return -1;
}

// 4. Перехід у режим прослуховування
if (listen(server_fd, 10) < 0) {
    perror("listen failed");
    close(server_fd);
    return -1;
}

cout << "Server is listening on port " << PORT << endl;

// 5. Основний цикл приймання клієнтів
while (true) {
    int* client_fd = new int;

    *client_fd = accept(server_fd,
                        (struct sockaddr*)&client_addr,
                        &client_len);

    if (*client_fd < 0) {
        perror("accept failed");
        delete client_fd;
        continue;
    }

    cout << "Прийнято клієнтське з’єднання" << endl;
```

```
        pthread_t client_thread;
        pthread_create(&client_thread,
                       nullptr,
                       handle_client,
                       client_fd);

        pthread_detach(client_thread);
    }

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

Завдання клієнта полягає у встановленні з’єднання із сервером та ініціюванні обміну даними. Нижче наведено типовий приклад реалізації клієнтської частини на C++, яка встановлює з’єднання із сервером, надсилає повідомлення та отримує відповіді.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <iostream>
#include <cstring>

using namespace std;

const int PORT = 8080;              // Порт сервера
const char *SERVER_IP = "127.0.0.1";  // IP-адреса сервера

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[1024] = "Hello, Server!"; // Повідомлення, що надсилається серверу

    // 1. Створення сокета
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Не вдалося створити сокет");
        return -1;
    }

    // 2. Ініціалізація структури адреси сервера
    memset(&server_addr, 0, sizeof(server_addr));
```

```
    server_addr.sin_family = AF_INET;          // Використання протоколу IPv4
    server_addr.sin_port = htons(PORT);        // Порт у мережевому порядку байтів

    // 3. Конвертація IP-адреси у потрібний формат
    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        cerr << "Некоректна IP-адреса сервера" << endl;
        close(sockfd);
        return -1;
    }

    // 4. Встановлення з'єднання із сервером
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Не вдалося підключитися до сервера");
        close(sockfd);
        return -1;
    }

    cout << "Підключення до сервера встановлено" << endl;

    // 5. Надсилання повідомлення серверу
    send(sockfd, buffer, strlen(buffer), 0);

    // 6. Приймання відповіді від сервера
    int bytes_received = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
    if (bytes_received > 0) {
        buffer[bytes_received] = '\0';
        cout << "Відповідь сервера: " << buffer << endl;
    } else {
        cerr << "Не вдалося отримати відповідь від сервера" << endl;
    }

    // 7. Закриття сокета
    close(sockfd);
    return 0;
}
```

Створення сокета. Як клієнт, так і сервер створюють сокет за допомогою функції `socket`, вибираючи IPv4 (`AF_INET`) та TCP (`SOCK_STREAM`) як протокол. Обробка помилок забезпечує своєчасне виявлення та усунення проблем під час створення сокета.

Обробка IP-адреси та порту. Сервер прив'язується до порту, вказаного через `htons(PORT)`. Клієнт вказує адресу сервера за допомогою `inet_pton`, яка перетворює текстову IP-адресу у відповідний формат. Такі

перетворення критично важливі для сумісності на різних платформах і мережевих архітектурах.

Управління з'єднанням. Використання сервером функції `accept` створює виділений сокет для кожного підключеного клієнта, ізолюючи потоки комунікації. Клієнт встановлює з'єднання за допомогою функції `connect`, що ефективно ініціює канал обміну даними.

Передача та прийом даних. Функції `send` та `recv` забезпечують обмін даними між клієнтом і сервером. Правильне управління буфером, включно з встановленням нульового термінатора для рядків, гарантує цілісність даних та коректність обміну інформацією.

Паралелізм і багато поточність. Серверна багатопоточність дозволяє обслуговувати кілька клієнтських підключень одночасно, що є ключовим для адаптивних і масштабованих систем. Ця модель багатопоточності універсальна та ефективна у середовищах з високою частотою запитів за мінімального зниження продуктивності.

Обробка помилок та управління ресурсами. Комплексна перевірка помилок під час операцій із сокетами, прив'язки, прослуховування та прийняття підключень захищає від збоїв у роботі. Закриття дескрипторів сокета забезпечує звільнення ресурсів і запобігає витокам пам'яті, підтримуючи стабільність сервера та клієнта.

Покращення безпеки. Для захисту комунікацій можна інтегрувати TLS (Transport Layer Security) за допомогою бібліотек, таких як OpenSSL, що забезпечує шифрування та захист від перехоплення чи підміни даних.

Неблокувальні сокети. Використання неблокувальних I/O операцій або асинхронних патернів (наприклад, `select`, `poll`) підвищує продуктивність у сценаріях із високим навантаженням, дозволяючи виконувати інші операції під час очікування завершення I/O.

Балансування навантаження та відмово стійкість. Використання балансувальників навантаження розподіляє запити клієнтів між кількома серверами, оптимізуючи час відгуку і забезпечуючи стійкість до збоїв окремих серверів. Архітектурні підходи, такі як безстанний сервер, сприяють масштабуванню та організації відновлення після збоїв.

Проєктування протоколу. Розробка протоколу обміну даними для взаємодії клієнт-сервер потребує уважного підходу, включно з методами серіалізації (наприклад, JSON, XML) і механізмами перевірки помилок для забезпечення цілісності даних та надійності транзакцій.

Встановлення клієнт-серверного з'єднання – це не просто обмін даними; воно передбачає створення надійних, ефективних та безпечних каналів комунікації, що задовольняють потреби різних прикладних середовищ. Оволодіння цими методиками дозволяє створювати складні мережеві сервіси та системи, що забезпечують сучасні обчислювальні та інформаційні задачі.

3.5 Передача та прийом даних

Передача та прийом даних є фундаментальними процедурами в програмуванні сокетів і становлять основу мережевої комунікації. Після встановлення з'єднання між клієнтом і сервером обмін даними стає основним засобом взаємодії. У цьому підрозділі детально розглядаються техніки та нюанси передавання і прийому даних у мережах за допомогою сокетів у C++.

Розуміння передачі даних. Основою передачі даних є надійний обмін байтами, який забезпечують функції `send` та `recv` для сокетів потокового типу (TCP) або `sendto` та `recvfrom` для сокетів датаграмного типу (UDP). Ці функції формують фундамент, що дозволяє передавати інформацію між мережевими системами.

Базове використання `send` і `recv`: Функція `send` передає дані через встановлене з'єднання сокета. Для потокового сокета вона приймає такі параметри: дескриптор сокета, буфер з даними, довжину даних для відправки та прапорці, які модифікують поведінку функції.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>   // close()
#include <cstring>
#include <iostream>

using namespace std;

const int PORT = 8080;
const char* SERVER_IP = "127.0.0.1";

// Функція надійної відправки даних
int sendData(int socket_fd, const char* message) {
    int total_sent = 0;
    int bytes_left = strlen(message);
    int bytes_sent;

    while (total_sent < bytes_left) {
        bytes_sent = send(socket_fd, message + total_sent, bytes_left, 0);
        if (bytes_sent == -1) {
            perror("send failed");
            return -1;
        }
        total_sent += bytes_sent;
        bytes_left -= bytes_sent;
    }
    return total_sent;
}
```

```
int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        cerr << "Invalid server address" << endl;
        close(sockfd);
        return 1;
    }

    if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr))
< 0) {
        perror("Connection to server failed");
        close(sockfd);
        return 1;
    }

    cout << "Connected to server" << endl;

    const char* message = "Hello, Server!";
    if (sendData(sockfd, message) == -1) {
        cerr << "Failed to send data" << endl;
    } else {
        cout << "Message sent successfully" << endl;
    }

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

Надійність та повнота. Через можливі коливання мережі цикл відправки триває до тих пір, поки всі байти не будуть передані. Це забезпечує цілісність і повноту повідомлення, обробляючи часткову відправку – поширене явище у разі використання функції `send`, коли передається менше байтів, ніж запитано.

Прапорці. Наприклад, прапорець `MSG_DONTWAIT` дозволяє виконувати неблокувальні операції, що є критично важливим у високопродуктивних сценаріях, де очікування завершення введення/виведення зменшує швидкодію та чутливість системи.

На стороні прийому функція `recv` відповідає за зчитування даних із сокета з аналогічними параметрами: дескриптор сокета, буфер, довжина буфера та прапорці.

```
#include <sys/socket.h>
#include <unistd.h>
#include <cstring>
#include <iostream>
#include <cerrno>
#include <cstdio>

using namespace std;

int receiveData(int socket_fd, char* buffer, int size) {
    int bytes_received = recv(socket_fd, buffer, size - 1, 0);
    if (bytes_received < 0) {
        perror("receive failed");
        return -1;
    } else if (bytes_received == 0) {
        cout << "Connection closed by peer" << endl;
        return 0;
    }
    buffer[bytes_received] = '\0';
    return bytes_received;
}

int sendData(int socket_fd, const char* message) {
    int total_sent = 0;
    int bytes_left = strlen(message);
    int bytes_sent;

    while (total_sent < bytes_left) {
        bytes_sent = send(socket_fd, message + total_sent, bytes_left, 0);
        if (bytes_sent == -1) {
            perror("send failed");
            return -1;
        }
        total_sent += bytes_sent;
        bytes_left -= bytes_sent;
    }

    return total_sent;
}

// Тестовий main
int main() {
    cout << "Файл з функціями sendData та receiveData компілюється коректно" << endl;
    return 0;
}
```

Усвідомлення стану з'єднання. Повернення значення `0` функцією `recv` сигналізує про коректне завершення з'єднання з боку віддаленого вузла. Ця інформація є критичною для забезпечення правильних процедур завершення сеансу та ефективного керування ресурсами.

Розширені методи та аспекти. Хоча базове використання функцій `send` та `recv` забезпечує фундамент для обміну даними, для створення надійної моделі комунікації потрібно враховувати кілька додаткових аспектів.

Управління буферами та розмір буфера. Ефективне управління буферами дозволяє мінімізувати проблеми, пов'язані з пам'яттю, та оптимізувати пропускну здатність мережі. Розмір буфера потрібно підбирати відповідно до очікуваного обсягу повідомлень та наявної пропускної здатності мережі. Використання більших буферів може зменшити кількість викликів `send` та `recv`, що підвищує ефективність передачі даних.

```
#include <iostream>

// Константа розміру буфера
const int BUFFER_SIZE = 4096;

// Функція обробки даних
void processData() {
    char buffer[BUFFER_SIZE];
    std::cout << "Буфер ініціалізовано" << std::endl;
}

// Точка входу програми
int main() {
    processData();
    return 0;
}
```

Динамічне буферування. Адаптація виділення буферної пам'яті на основі аналізу розміру даних у режимі виконання забезпечує гнучкість застосунків за різних умов навантаження. Такий підхід дає змогу ефективно використовувати ресурси пам'яті та підтримувати стабільну продуктивність системи за змінних обсягів передаваних даних.

Оброблення передавання великих обсягів даних. Передавання великих наборів даних потребує їх сегментації на керовані фрагменти, що забезпечує коректну та надійну передачу й приймання кожного з них. Поділ даних на блоки спрощує контроль цілісності передавання, зменшує ризик втрат і сприяє підвищенню ефективності мережевої взаємодії.

```
#include <algorithm>
#include <cstddef>
#include <cstdio>
#include <cstring>
```

```
#include <cerrno>
#include <sys/types.h>
#include <sys/socket.h>
#include <iostream>

const std::size_t BUFFER_SIZE = 4096;

/*
 * Функція сегментованого передавання великих обсягів даних
 * через сокет із використанням буферизації
 */
void sendLargeData(int socket_fd, const char* large_data, std::size_t size)
{
   std::size_t bytes_sent = 0;

   while (bytes_sent < size)
   {
      std::size_t segment_size =
         std::min(size - bytes_sent, BUFFER_SIZE);

      int sent = send(socket_fd,
                  large_data + bytes_sent,
                  segment_size,
                  0);

      if (sent < 0)
      {
         perror("Error in sending large data");
         break;
      }

      bytes_sent += static_cast<std::size_t>(sent);
   }
}

/*
 * Функція приймання та обробки даних
 */
void processData()
{
   char buffer[BUFFER_SIZE];
   // Приймання та обробка даних (навчальний приклад)
}

/*
 * Головна функція програми
 */
```

```
int main()
{
   std::cout << "Program started successfully." << std::endl;

   processData();

   // sendLargeData() зазвичай викликається після встановлення
   // сокетного з’єднання (тут наведено лише структуру)

   return 0;
}
```

Реалізація протоколу фреймінгу є критично важливою під час проєктування мережевих застосунків, оскільки вона забезпечує чітке відмежування окремих фрагментів даних. Це, зі свою боку, сприяє коректній інтерпретації та відновленню повідомлень на стороні приймача, незалежно від особливостей передавання даних у мережі.

Кожен фрейм має наперед визначений розмір, що спрощує обробку даних і зменшує складність реалізації протоколу. Водночас такий підхід може призводити до додаткових накладних витрат у разі передавання повідомлень змінної довжини.

Для позначення меж фреймів використовуються спеціальні символьні послідовності (наприклад, символи нового рядка). Цей метод є простим у реалізації, однак потребує обережного вибору розділювачів, щоб уникнути їхнього неоднозначного трактування в переданих даних.

Фреймінг із полем довжини передбачає використання заголовка, що містить інформацію про розмір подальшого корисного навантаження. Такий підхід забезпечує гнучкість, ефективність і надійність передавання, особливо в системах зі змінною довжиною повідомлень і складною структурою даних.

```
#include <cstdint>
#include <cstring>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <iostream>

const std::size_t BUFFER_SIZE = 4096;

// Функція обробки даних
void processData() {
   char buffer[BUFFER_SIZE];
   // Приймання та обробка даних (приклад)
}

// Функція для відправки великих даних
void sendLargeData(int socket_fd, const char* large_data, size_t size) {
```

```
    size_t bytes_sent = 0;
    while (bytes_sent < size) {
        size_t segment_size = std::min(size - bytes_sent, BUFFER_SIZE);
        int sent = send(socket_fd, large_data + bytes_sent, segment_size, 0);
        if (sent < 0) {
            perror("Error in sending large data");
            break;
        }
        bytes_sent += sent;
    }
}

// Приклад фреймінгу з полем довжини
void sendFramedMessage(int socket_fd, const char* message) {
    uint32_t length = htonl(strlen(message));
    send(socket_fd, &length, sizeof(length), 0);
    send(socket_fd, message, strlen(message), 0);
}

// Головна функція програми
int main() {
    std::cout << "Програма запущена успішно." << std::endl;

    processData();

    // Тут можна додати приклад виклику sendLargeData або sendFramedMessage
    // int sockfd = ... ; const char* msg = "Test"; sendFramedMessage(sockfd, msg);

    return 0;
}
```

Тут заголовок кодує довжину повідомлення, що забезпечує коректне вилучення приймачем саме того обсягу байтів, який передбачався.

Використання подієво-орієнтованого або асинхронного введення/виведення підвищує відзивність програми, дозволяючи продовжувати обробку даних під час очікування завершення операцій введення/виведення.

За допомогою механізмів select, poll або epoll розробники можуть відстежувати кілька файлових дескрипторів одночасно та визначати, коли можливе введення/виведення без блокування виконання програми.

```
#include <sys/select.h>
#include <unistd.h>
#include <iostream>
```

```
/*
 * Приклад обробки введення-виведення за допомогою select
 * server_fd — файловий дескриптор серверного сокета
 */
void handleIOSelect(int server_fd) {
    fd_set read_fds;
    struct timeval timeout;

    FD_ZERO(&read_fds);
    FD_SET(server_fd, &read_fds);

    timeout.tv_sec = 5;
    timeout.tv_usec = 0;

    int activity = select(server_fd + 1, &read_fds, NULL, NULL, &timeout);
    if (activity < 0) {
        perror("Помилка select");
    } else if (FD_ISSET(server_fd, &read_fds)) {
        std::cout << "Server socket ready for I/O" << std::endl;
    }
}

// Головна функція для запуску програми
int main() {
    int fake_server_fd = 0; // Просто приклад, для демонстрації
    handleIOSelect(fake_server_fd);

    return 0;
}
```

Функція select дозволяє перевіряти готовність декількох сокетів одночасно, що забезпечує ефективне управління кількома клієнтами та підвищує продуктивність за рахунок зменшення затримок під час обробки запитів.

Для застосунків, що залежать від специфічних протоколів понад базовий TCP/IP, необхідно враховувати додаткові аспекти:

Обробка двійкових даних: потрібно враховувати різну послідовність байтів (endianness) на різних системах, наприклад, шляхом використання функцій `htonl` або `ntohl` для конвертації багатобайтових цілих чисел.

Шифрування даних: інтеграція бібліотек шифрування, таких як OpenSSL, забезпечує безпечну передачу даних, захищаючи їх від несанкціонованого доступу.

Стиснення даних: застосування алгоритмів стиснення (наприклад, Zlib) зменшує вплив на пропускну здатність мережі за рахунок зменшення розміру повідомлень.

```
/*
   Лістинг 1. Повний TLS-сервер на Windows
   Використовує WinSock + OpenSSL
   Мета: Демонстрація захищеного клієнт-серверного зв'язку за протоколом TLS
*/

#include <winsock2.h>
#include <ws2tcpip.h>
#include <windows.h>
#include <iostream>
#include <string>

// OpenSSL
#include <openssl/ssl.h>
#include <openssl/err.h>

#pragma comment(lib, "Ws2_32.lib")
#pragma comment(lib, "libssl.lib")
#pragma comment(lib, "libcrypto.lib")

const int PORT = 4433;        // Порт для TLS
const char* CERT_FILE = "server.crt";  // Сертифікат сервера
const char* KEY_FILE  = "server.key";  // Приватний ключ

int main() {
   // 1. Ініціалізація WinSock
   WSADATA wsaData;
   if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
      std::cerr << "WinSock initialization failed." << std::endl;
      return 1;
   }

   // 2. Ініціалізація OpenSSL
   SSL_library_init();
   SSL_load_error_strings();
   OpenSSL_add_all_algorithms();

   const SSL_METHOD* method = TLS_server_method();
   SSL_CTX* ctx = SSL_CTX_new(method);
   if (!ctx) {
      std::cerr << "Unable to create SSL context." << std::endl;
      ERR_print_errors_fp(stderr);
      WSACleanup();
      return 1;
   }
```

```
    // 3. Завантаження сертифікату та ключа
    if (SSL_CTX_use_certificate_file(ctx, CERT_FILE,
SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        SSL_CTX_free(ctx);
        WSACleanup();
        return 1;
    }

    if (SSL_CTX_use_PrivateKey_file(ctx, KEY_FILE,
SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        SSL_CTX_free(ctx);
        WSACleanup();
        return 1;
    }

    // 4. Створення сокета
    SOCKET server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == INVALID_SOCKET) {
        std::cerr << "Socket creation failed." << std::endl;
        SSL_CTX_free(ctx);
        WSACleanup();
        return 1;
    }

    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    // 5. Прив'язка
    if (bind(server_fd, (sockaddr*)&addr, sizeof(addr)) ==
SOCKET_ERROR) {
        std::cerr << "Bind failed." << std::endl;
        closesocket(server_fd);
        SSL_CTX_free(ctx);
        WSACleanup();
        return 1;
    }

    // 6. Прослуховування
    if (listen(server_fd, SOMAXCONN) == SOCKET_ERROR) {
        std::cerr << "Listen failed." << std::endl;
        closesocket(server_fd);
        SSL_CTX_free(ctx);
        WSACleanup();
```

```
        return 1;
    }

    std::cout << "TLS Server listening on port " << PORT << "..." << std::endl;

    // 7. Основний цикл приймання клієнтів
    while (true) {
        sockaddr_in client_addr;
        int client_len = sizeof(client_addr);
        SOCKET client_fd = accept(server_fd, (sockaddr*)&client_addr, &client_len);
        if (client_fd == INVALID_SOCKET) {
            std::cerr << "Accept failed." << std::endl;
            continue;
        }

        std::cout << "Client connected." << std::endl;

        // 8. Створення SSL об'єкта
        SSL* ssl = SSL_new(ctx);
        SSL_set_fd(ssl, (int)client_fd);

        // 9. TLS handshake
        if (SSL_accept(ssl) <= 0) {
            ERR_print_errors_fp(stderr);
            SSL_free(ssl);
            closesocket(client_fd);
            continue;
        }

        // 10. Приймання і відправка даних
        const char* reply = "Hello from TLS Server!";
        char buffer[1024] = {0};
        int bytes = SSL_read(ssl, buffer, sizeof(buffer)-1);
        if (bytes > 0) {
            buffer[bytes] = '\0';
            std::cout << "Received from client: " << buffer << std::endl;

            SSL_write(ssl, reply, strlen(reply));
        }

        // 11. Закриття клієнтського SSL і сокета
        SSL_shutdown(ssl);
        SSL_free(ssl);
        closesocket(client_fd);
    }
```

```
    // 12. Очистка ресурсів
    closesocket(server_fd);
    SSL_CTX_free(ctx);
    WSACleanup();

    return 0;
}
```

Складність процесу надсилання та отримання даних через сокети потребує глибокого розуміння динаміки мереж та парадигм комунікації. Завдяки ретельному опрацюванню питань буферизації, кадрування, одночасного обслуговування та вимог протоколів, розробники можуть створювати ефективні та надійні системи, які задовольняють різноманітні потреби у зв'язку. Опанування цих технік дозволяє будувати масштабовані та стійкі мережеві додатки, здатні забезпечувати динамічну взаємодію у різноманітних обчислювальних середовищах. Таким чином, глибоке розуміння цих елементів є необхідним для фахівців, які прагнуть ефективно використовувати можливості мережевих комунікацій у сучасних програмних екосистемах.

3.6 Обробка помилок у програмуванні сокетів

Обробка помилок у програмуванні сокетів є критично важливою для забезпечення надійності та стабільності мережевих додатків. Мережеве середовище може бути непередбачуваним, і на роботоздатність впливають численні фактори, що потенційно призводять до збоїв: обмеження ресурсів, перебої в мережі або помилки програмного забезпечення. Ефективне управління помилками передбачає їх виявлення, відповідну реакцію та реалізацію стратегій, що запобігають повторним проблемам.

Обробка помилок створює «захисну сітку», дозволяючи програмам відпрацьовувати некритичні збої без втрати даних або переходу в нефункціональний стан. Вона полегшує процес відлагодження, підвищує якість користувацького досвіду та забезпечує стабільність системи. Антиципація можливих відмов і впровадження надійних механізмів обробки дозволяє створювати додатки, стійкі до непередбачуваних умов мережевого середовища.

Помилки у програмуванні сокетів можна умовно поділити на три основні категорії:

1. *Обмеження системних ресурсів*. Недостатня пам'ять, обмежена кількість дескрипторів файлів або інші ресурси можуть заважати роботі сокетів.

2. *Мережеві проблеми*. Втрата пакетів, затримки або розриви з'єднання можуть порушувати канали комунікації.

3. *Помилки коду та логіки*. Програмні баги або некоректна логіка можуть спричиняти непередбачувану поведінку або аварійне завершення.

Обробка помилок у сокет-програмуванні зазвичай містить:

- перевірку значень, що повертаються функціями,
- аналіз глобальної змінної `errno`,
- впровадження систем журналювання або повідомлень, специфічних для додатка.

Більшість функцій сокетів у разі виникнення помилки повертають від'ємне значення або `-1` і одночасно встановлюють змінну `errno` у код конкретної помилки. Наприклад:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h> // для AF_INET
#include <arpa/inet.h>
#include <unistd.h>
#include <cstdio>

int main() {
    // Створення TCP-сокета
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    // Перевірка на помилку
    if (sockfd == -1) {
        perror("Помилка під час відкриття сокета");
        return -1;
    }

    // Закриття сокета перед завершенням програми
    close(sockfd);
    return 0;
}
```

У цьому прикладі функція `perror` зіставляє значення `errno` з описовим текстом, що дозволяє отримати миттєвий зворотний зв'язок щодо природи помилки.

Поширені значення `errno`:

EACCES: доступ заборонено (Permission denied).

EADDRINUSE: адреса вже використовується (Address already in use).

EAFNOSUPPORT: сімейство адрес не підтримується (Address family not supported).

ECONNRESET: з'єднання скинуто віддаленим вузлом (Connection reset by peer).

ENOMEM: недостатньо пам'яті (Insufficient memory available).

ETIMEDOUT: перевищено час очікування з'єднання (Connection timed out).

На кожному етапі взаємодії через сокети можуть виникати помилки, що потребують відповідних стратегій обробки. Створення сокета може завершитися невдачею через нестачу системних ресурсів, що потребує

реалізації механізмів резервного відновлення або повідомлень користувачу для коректного реагування на помилку.

```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <cerrno>
#include <cstring>

using namespace std;

int main() {
   // Створення сокета TCP/IPv4
   int sockfd = socket(AF_INET, SOCK_STREAM, 0);

   // Перевірка на помилку створення сокета
   if (sockfd == -1) {
      switch(errno) {
         case EACCES:
            cerr << "Доступ заборонено (Permission denied)" << endl;
            break;
         case EMFILE:
            cerr << "Занадто багато відкритих дескрипторів файлів (Too many file descriptors in use)" << endl;
            break;
         default:
            cerr << "Помилка створення сокета: " << strerror(errno) << endl;
      }
      return -1; // Завершення програми у разі помилки
   }

   cout << "Сокет успішно створено!" << endl;

   return 0;
}
```

Встановлення з’єднання, особливо у клієнтських застосунках, схильне до тайм-аутів або недоступних мережевих маршрутів.

```
#include <sys/socket.h>     // Основні функції для роботи з сокетами (socket, connect тощо)
#include <netinet/in.h>   // Структури для IPv4 адрес (sockaddr_in)
#include <arpa/inet.h>    // Функції для конвертації IP-адрес (inet_pton)
#include <unistd.h>       // Функція close() для закриття сокета
#include <cerrno>         // Глобальна змінна errno для обробки помилок
```

```
#include <cstring>          // Функція strerror() для отримання текстового опису помилки
#include <iostream>        // Для cout та cerr

using namespace std;

int main() {
    // 1. Створення сокета
    // AF_INET - використання IPv4
    // SOCK_STREAM - TCP-з’єднання
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        cerr << "Помилка створення сокета: " << strerror(errno) << endl;
        return -1; // Завершення програми у разі помилки
    }

    // 2. Налаштування адреси сервера
    struct sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;          // Вказуємо тип адреси IPv4
    server_addr.sin_port = htons(8080);              // Встановлення порту сервера у мережевому порядку байтів

    // Конвертація IP-адреси з текстового формату у бінарний
    if (inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr) <= 0) {
        cerr << "Неправильна IP-адреса сервера" << endl;
        close(sockfd); // Закриття сокета перед завершенням
        return -1;
    }

    // 3. Підключення до сервера
    int status = connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr));
    if (status == -1) {
        // Обробка різних типів помилок підключення
        if (errno == ECONNREFUSED) {
            cerr << "З’єднання відхилене сервером" << endl;
        } else if (errno == ETIMEDOUT) {
            cerr << "Час очікування з’єднання перевищено" << endl;
        } else {
            cerr << "Невідома помилка з’єднання: " << strerror(errno) << endl;
        }
        close(sockfd); // Закриття сокета у разі помилки
        return -1;
    }

    // Виведення повідомлення про успішне підключення
```

```
    cout << "Підключення до сервера успішне!" << endl;

    // 4. Закриття сокета після завершення роботи
    close(sockfd);

    return 0;
}
```

Навіть під час встановленого з’єднання передача та приймання даних можуть стикатися з проблемами через мережеві аномалії, що потребує повторних спроб або застосування альтернативних дій.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>     // close()
#include <cerrno>
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    // 1. Створення сокета
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        cerr << "Помилка створення сокета: " << strerror(errno) << endl;
        return -1;
    }

    // 2. Налаштування адреси сервера
    struct sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080); // порт сервера
    if (inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr) <= 0) {
        cerr << "Неправильна IP-адреса сервера" << endl;
        close(sockfd);
        return -1;
    }

    // 3. Підключення до сервера
    if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr))
== -1) {
        cerr << "Помилка підключення: " << strerror(errno) << endl;
        close(sockfd);
        return -1;
    }
```

```
    cout << "Підключення до сервера успішне!" << endl;

    // 4. Підготовка повідомлення для відправки
    const char* message = "Привіт, сервер!";
    size_t message_length = strlen(message);

    // 5. Відправка даних із обробкою помилок
    int bytes_sent = send(sockfd, message, message_length, 0);
    if (bytes_sent == -1) {
        if (errno == EINTR) {
            cerr << "Операцію перервано, повторна спроба" << endl;
            // Тут може бути логіка повторної відправки
        } else {
            cerr << "Помилка під час відправлення даних: " <<
strerror(errno) << endl;
        }
        close(sockfd);
        return -1;
    }

    cout << "Відправлено байт: " << bytes_sent << endl;

    // 6. Закриття сокета після завершення роботи
    close(sockfd);
    return 0;
}
```

Удосконалення механізмів обробки помилок. Використання винятків. Хоча у програмуванні сокетів винятки C++ застосовуються рідше через історично C-орієнтований характер відповідних API, їх використання може суттєво підвищити структурованість і зрозумілість коду. Винятки дають змогу інкапсулювати помилки, відокремити логіку обробки помилкових ситуацій від основного алгоритму та спростити передавання інформації про збої між рівнями програми. Це особливо корисно в складних програмних системах, де необхідно централізовано обробляти помилки та забезпечувати керований і передбачуваний перебіг виконання.

```
// Клас винятку для помилок сокетних операцій
class SocketException : public std::runtime_error {
public:
    // Конструктор винятку з передаванням тексту повідомлення про
помилку
    explicit SocketException(const std::string &message)
        : std::runtime_error(message) {}
};
```

```
// Функція надсилання даних через сокет із використанням винятків
void sendData(int socket_fd, const char* message) {
    // Спроба передати повідомлення через сокет
    if (send(socket_fd, message, strlen(message), 0) == -1) {
        // У разі помилки генерується виняток із описом, отриманим з errno
        throw SocketException(strerror(errno));
    }
}
```

У цьому прикладі замість традиційної перевірки кодів повернення використовується механізм винятків C++. Це дозволяє:

- інкапсулювати інформацію про помилку в окремому класі;
- відокремити логіку обробки помилок від основного алгоритму передавання даних;
- централізовано перехоплювати та обробляти помилки на вищих рівнях програми.

Такий підхід особливо доцільний у складних клієнт-серверних застосунках, де важливо зберігати читабельність коду та забезпечувати надійне реагування на збої мережевої взаємодії.

```
#include <sys/socket.h>   // send()
#include <unistd.h>      // close()
#include <cerrno>        // errno
#include <cstring>       // strlen(), strerror()
#include <string>        // std::string
#include <stdexcept>     // std::runtime_error
#include <iostream>

// Клас винятку для помилок сокетних операцій
class SocketException : public std::runtime_error {
public:
    explicit SocketException(const std::string& message)
        : std::runtime_error(message) {}
};

// Функція надсилання даних через сокет
void sendData(int socket_fd, const char* message) {
    if (send(socket_fd, message, strlen(message), 0) == -1) {
        throw SocketException(strerror(errno));
    }
}

// Точка входу програми
int main() {
    int dummy_socket = -1; // приклад (у реальній програмі тут буде socket())
```

```
    try {
        sendData(dummy_socket, "Test message");
    } catch (const SocketException& ex) {
        std::cerr << "Помилка сокетної операції: " << ex.what() << std::endl;
    }

    return 0;
}
```

Для підвищення надійності програмних систем доцільно впроваджувати розвинені механізми журналювання, здатні фіксувати детальну інформацію про помилки та аномальні стани. Накопичені журнальні дані забезпечують цінну аналітичну інформацію, яка є критично важливою як на етапі розроблення й тестування, так і під час експлуатації системи, сприяючи оперативному виявленню, діагностиці та усуненню збоїв.

```
#include <fstream>   // Для роботи з файловими потоками
#include <string>    // Для використання std::string
#include <iostream> // (необов'язково) для можливого виведення повідомлень

// Функція журналювання помилок сокетних операцій
// Записує повідомлення про помилку у файл журналу в режимі додавання
void logError(const std::string &error_message) {
    // Відкриття файлу журналу помилок у режимі дописування
    std::ofstream log_file("socket_errors.log", std::ios_base::app);

    // Перевірка успішності відкриття файлу
    if (!log_file.is_open()) {
        std::cerr << "Не вдалося відкрити файл журналу помилок" << std::endl;
        return;
    }

    // Запис повідомлення про помилку з переходом на новий рядок
    log_file << error_message << std::endl;

    // Закриття файлу (відбудеться автоматично у деструкторі)
}
```

`#include <string>` є обов'язковим для використання типу `std::string`. Клас `std::ofstream` використовується для запису даних у файл.

Прапорець `std::ios_base::app` забезпечує додавання нових записів без перезапису файлу.

Перевірка `is_open()` дозволяє коректно обробити ситуацію, коли файл не вдалося створити або відкрити (наприклад, через відсутність прав доступу).

Журналювання помилок є важливим елементом моніторингу мережевих застосунків і значно полегшує аналіз збоїв під час експлуатації.

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

// Функція журналювання помилок
void logError(const std::string &error_message) {
    std::ofstream log_file("socket_errors.log", std::ios_base::app);
    if (log_file.is_open()) {
        log_file << error_message << std::endl;
    }
}

// Точка входу програми
int main() {
    std::cout << "Програма запущена успішно" << std::endl;

    // Приклад використання журналювання
    logError("Тестовий запис журналу");

    return 0;
}
```

Особливо в мережевих операціях доцільним є застосування експоненційних стратегій зворотної затримки під час повторних спроб виконання операцій. Такий підхід дає змогу ефективно керувати тимчасовими мережевими збоями, поступово збільшуючи інтервали між повторними запитами, що запобігає перевантаженню системи та мережевої інфраструктури. Застосування back-off стратегій підвищує стабільність і надійність мережевих застосунків, особливо в умовах нестабільних або змінних характеристик мережевого середовища.

```
#include <iostream>
#include <cstring>
#include <cerrno>
#include <cmath>
#include <thread>
#include <chrono>

#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

using namespace std;

constexpr int MAX_RETRIES = 5;

int main() {
    // Створення сокета
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        cerr << "Помилка створення сокета: " << strerror(errno) << endl;
        return -1;
    }

    // Налаштування адреси сервера
    sockaddr_in server_addr{};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr);

    int retry_count = 0;

    // Спроби підключення з експоненційною затримкою
    while (retry_count < MAX_RETRIES) {
        int result = connect(sockfd,
                             reinterpret_cast<sockaddr*>(&server_addr),
                             sizeof(server_addr));

        if (result == 0) {
            cout << "З’єднання встановлено успішно" << endl;
            break;
        }

        int backoff_time = static_cast<int>(pow(2, retry_count));
        cerr << "Спроба " << retry_count + 1
             << " не вдалася. Повтор через "
             << backoff_time << " с." << endl;

        std::this_thread::sleep_for(std::chrono::seconds(backoff_time));
        retry_count++;
    }

    if (retry_count == MAX_RETRIES) {
        cerr << "Досягнуто максимальну кількість спроб. З’єднання не
встановлено."
```

```
            << endl;
        close(sockfd);
        return -1;
    }

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

Експоненційна стратегія зворотної затримки передбачає подвоєння часу очікування між повторними спробами з'єднання (1 с, 2 с, 4 с, 8 с тощо). Це дозволяє уникнути перевантаження мережі та сервера у випадку тимчасових збоїв і є стандартною практикою в сучасних мережевих протоколах і розподілених системах.

Плавна деградація функціональності передбачає проєктування інтерфейсів і сервісів таким чином, щоб у разі виникнення помилок або відмов система продовжувала роботу в обмеженому режимі, інформуючи користувача про проблему без аварійного завершення програми чи зупинення сервісу. Такий підхід підвищує надійність програмного забезпечення та мінімізує негативний вплив збоїв на користувацький досвід.

Для застосунків, що функціонують у розподілених системах або використовують сучасні технології, такі як хмарні сервіси та Інтернет речей (IoT), обробка помилок потребує гнучкості та узгодженої взаємодії між численними компонентами системи.

Реалізація резервування компонентів системи забезпечує безперервність роботи навіть у разі відмови окремих вузлів. Такий підхід є критично важливим для хмарних застосунків, де доступність сервісів має першочергове значення.

Системи мають проєктуватися з використанням протоколів, стійких до збоїв, які здатні динамічно перенаправляти обмін даними або коректно обробляти часткові відмови без повного припинення функціонування системи.

Інтеграція механізмів сповіщення дозволяє оперативно інформувати адміністраторів про критичні помилки в режимі реального часу, що забезпечує швидке реагування та мінімізацію негативних наслідків.

3.7 Контрольні питання

1. Що таке сокет у комп'ютерних мережах і яку роль він відіграє у клієнт-серверній взаємодії?
2. Які основні типи сокетів існують у C++ (SOCK_STREAM, SOCK_DGRAM, Raw Sockets, SOCK_SEQPACKET) і в яких сценаріях кожен із них застосовується?
3. Опишіть життєвий цикл сокета в C++, включно з етапами створення, прив'язки (bind), прослуховування (listen), прийому з'єднання (accept), передачі даних (send/recv) і завершення (close).

4. Які механізми та функції забезпечують узгодженість порядку байтів, конкурентний доступ та блокувальне/неблокувальне введення/виведення під час роботи з сокетами?

5. Які опції сокета (setsockopt) та розширені технології (SSL/TLS) використовуються для покращення продуктивності, повторного використання з'єднань і безпечного обміну даними?

6. Яке призначення функції socket() у C++ і які параметри вона приймає для створення TCP або UDP сокета?

7. Які заголовочні файли та бібліотеки необхідні для роботи із сокетами у C++ на Unix-подібних системах та як це відрізняється від Windows (Winsock)?

8. Поясніть різницю між доменом (AF_INET/AF_INET6), типом сокета (SOCK_STREAM, SOCK_DGRAM) та протоколом у разі створення сокета.

9. Які методи обробки помилок застосовуються під час створення сокета та прив'язки до адреси (наприклад, perror, strerror(errno)), і чому це критично для надійності програми?

10. Які способи налаштування сокета існують після його створення (неблокувальний режим, SO_REUSEADDR, SO_LINGER) і яку роль вони відіграють у продуктивності та керуванні мережевими з'єднаннями?

11. Яка роль функції bind() у сокетному програмуванні та які параметри необхідні для прив'язки сокета до IP-адреси та порту?

12. Поясніть призначення функції listen(); що таке черга очікування (backlog) і як використання константи SOMAXCONN впливає на роботу сервера.

13. Як працює функція accept() і чому для кожного клієнтського з'єднання створюється новий виділений сокет?

14. Які методи керування одночасними підключеннями клієнтів існують у C++ (потоки, fork, select/poll) і в чому полягають їхні переваги та недоліки?

15. Чому важливо коректно закривати сокети та звільняти ресурси після завершення роботи з'єднань, і які проблеми можуть виникнути у разі недотримання цього правила?

16. Які основні етапи життєвого циклу клієнт-серверної взаємодії і яка роль кожного з них?

17. Як клієнт ініціює з'єднання із сервером у C++ і які функції для цього застосовуються?

18. Поясніть, чому сервер створює окремий сокет для кожного клієнтського підключення і яку роль відіграє багатопоточність у цьому процесі.

19. Які методи забезпечують коректний обмін даними між клієнтом і сервером та як обробляються помилки за використання send() і recv()?

20. Які практики потрібно застосовувати для управління ресурсами та безпекою під час встановлення клієнт-серверного з'єднання (закриття сокетів, TLS, балансування навантаження)?

21. Які основні функції використовуються для передачі та прийому даних у сокетах TCP та UDP у C++ і які їхні ключові параметри?

22. Чому необхідно враховувати можливість часткової передачі даних у разі використання функції send, і як це реалізується на практиці?

23. Поясніть принцип фреймінгу повідомлень і чим відрізняються методи фреймінгу за допомогою символів-розділювачів та поля довжини.

24. Як розмір буфера та динамічне виділення пам’яті впливають на ефективність передачі даних і обробку великих повідомлень?

25. Які підходи підвищення безпеки та продуктивності передачі даних можна реалізувати, включно з TLS, стисненням і подіє-орієнтованим введенням/виведення (select, poll, epoll)?

26. Які основні категорії помилок можуть виникати під час роботи з сокетами і як вони впливають на стабільність мережевих застосунків?

27. Як у C++ перевіряється наявність помилки під час викликів функцій сокетів і яку роль відіграють errno та функції perror і strerror?

28. Поясніть принцип використання винятків (exceptions) для обробки помилок у сокет-програмуванні та їхні переваги порівняно з традиційною перевіркою кодів повернення.

29. Які стратегії підвищення надійності роботи мережевих застосунків передбачають повторні спроби операцій і експоненційну затримку (back-off)?

30. Яким чином журналювання помилок та механізми сповіщення сприяють ефективному моніторингу та налагодженню мережевих програм?

ВИСНОВКИ

Реалізація мережевих протоколів перебуває на перетині інноваційності та складності в галузі мережевої інженерії. Вирішення питань сумісності, безпеки, масштабованості, продуктивності, стійкості до помилок, обмеженості ресурсів і динаміки мереж є визначальним для створення надійних та ефективних систем зв'язку. Застосовуючи обґрунтовані підходи до проєктування та використовуючи прогресивні технології, сучасні інженери здатні розробляти протоколи, які не лише задовольняють поточні потреби, але й еволюціонують з урахуванням майбутніх вимог у все більш взаємопов'язаному цифровому середовищі.

Ефективна обробка помилок у сокетному програмуванні є не допоміжним елементом, а фундаментальною складовою проєктування та розроблення програмного забезпечення. Усвідомлення можливих джерел помилок і впровадження надійних стратегій їх обробки дає змогу створювати стійкі застосунки, здатні коректно функціонувати навіть за несприятливих умов. Опанування методів управління помилками сприяє збільшенню життєвого циклу програмних продуктів, підвищує довіру користувачів і рівень їхньої задоволеності. Завдяки ретельному журналюванню, адаптивним механізмам реагування та раціональному використанню ресурсів сокетні застосунки можуть підтримувати високу доступність і продуктивність навіть у складних та непередбачуваних мережевих середовищах.

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 [Електронний ресурс]. 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. В., Коломієць А. А., Грабчак Н. В., Грицаюк Д. Ю., Костьянін В. Ю. 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 [Текст] / 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 [Текст] / 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 [Текст] / М. В. Васильківський, А. В. Прикмета, А. Олійник, Д. В. Нікітович // *Вісник Хмельницького національного університету. Серія «Технічні науки».* 2023. № 1. (317). С. 33–41.

31. Дослідження архітектури штучного інтелекту для інфокомунікаційних мереж 6G [Текст] / 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

Мережеве програмування: проєктування ефективних систем зв'язку

Навчальний посібник

Рукопис оформив *М. Васильківський*

Редактор *Т. Старічек*

Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 14.07.2026 р.
Гарнітура Times New Roman.
Зам. № Р2026-091.
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.