ДЕРЖАВНИЙ УНІВЕРСІТЕТ ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ І ЗВ'ЯЗКУ

Топалов В.В., Багачук Д. Г.

МЕРЕЖНЕ ПРОГРАМУВАННЯ

методичні вказівки до практичних занять
для здобувачів першого (бакалаврський) рівня вищої освіти
галузі знань 12 (F) «Інформаційні технології»

Одеса, 2025

УДК 004.42:004.7(07)

Укладачі:

Топалов В.В., к.т.н., доцент кафедри інформаційних та комп'ютерних систем Державного університету інтелектуальних технологій і зв'язку;

Багачук Д. Г., к.т.н., доцент кафедри інформаційних та комп'ютерних систем Державного університету інтелектуальних технологій і зв'язку

Рецензенти:

Кільдішев В.І., к.т.н., доцент кафедри Кібербезпеки та технічного захисту інформації Державного університету інтелектуальних технологій і зв'язку;

Гожий О.П., д.т.н., професор кафедри Інтелектуальних Інформаційних систем ЧНУ ім. П.Могили;

Рекомендовано до друку Навчально-методичною Радою Державного університету інтелектуальних технологій і зв'язку (протокол №4 від 30 січня 2025 р.)

Мережне програмування: методичні вказівки до практичних занять [для здобувачів першого (бакалаврський) рівня вищої освіти галузі знань 12 (F) «Інформаційні технології»]. / Уклад.: В.В. Топалов, Д.Г. Багачук. Одеса: ДУІТЗ (Електр. вид. https://metod.suitt.edu.ua), 2025. 102 с.

Методичні рекомендації орієнтовані на розв'язання практичних та лабораторних задач за дисципліною мережне програмування. Передбачено вивчення основних алгоритмів, їх реалізацій та приклади застосування, у методиці описані основні роботи за дисципліною мережне програмування, (протоколи TCP, UDP, SCTP, SMTP, IMAP, HTTP; мережевий сокет; багатопотоковість, асинхронні, синхронні, TCP, UDP, HTTP, IMAP з'єднання та інші; багатоадресна передача; архітектура клієнт-сервер та інше).

ЗМІСТ

ВСТУП

Підготовка сучасного фахівця вимагає певен володіння можливостями, надаваними комп'ютерними технологіями.

Розробка мережних додатків – це одна з важливих дисциплін інформатики, що вимагає від програміста деяких додаткових знань і вмінь. Природно, для створення додатків, що використовують можливості комп'ютерної мережі необхідно знати основи функціонування таких мереж, розуміти головні мережні протоколи, поняття маршрутизації й адресації. У переважній більшості випадків на практиці використовуються протоколи сімейства TCP/IP. Це зараз універсальний механізм передачі даних. Але самі мережні додатки можуть інтенсивно використовувати різні конкретні протоколи, що перебувають на різних рівнях моделі OSI.

Безпосередньо пов'язана з мережним обміном - це паралельне програмування. Мережні додатки звичайно дуже активно використовують багатопоточності або інші засоби забезпечення багатозадачності.

Ціль методичних рекомендацій полягає в підготовці фахівців, що володіють практичними базовими знаннями, в області розробки мережних додатків, орієнтованих на клієнт-серверну архітектуру, програмування елементів такої архітектури.

Методичні рекомендації призначені для допомоги студентам у виконанні практичних робіт з дисципліни.

У методичних рекомендаціях для вивчення представлені відомості про створення клієнт-серверних додатків, як однопоточних, так і багатопоточних. Розглядаються приклади клієнт-серверних додатків синхронних і асинхронних. Також представлені відомості про роботу із протоколами HTTP, SSL, IMAP, SMTP, FTP і т.д.

При виконанні практичних робіт рекомендується використовувати програмну мову Python.

ПРАКТИЧНА РОБОТА №1

Стек протоколів TCP/IP. Огляд протоколів TCP, UDP, SCTP. Встановлення з'єднання за протоколами TCP та SCTP

Стек протоколів TCP/IP. TCP, UDP і протоколу передачі керування потоком (SCTP).

Усі пристрої у глобальної мережі INTERNET мають IP (*Internet Protocol* — «міжмережевий протокол») адреси. IP адреси можуть бути як 32 бітні IPv4(рис.1) , так і 128 бітні IPv6 (рис. 2).

<table>
<tr><th>Біти 0-3</th><th>4-7</th><th>8-15</th><th>16-18</th><th>19-23</th><th>24-31</th></tr>
<tr><td>Версія</td><td>HLEN</td><td>Тип обслуговування</td><td colspan="3">Загальна довжина</td></tr>
<tr><td colspan="3">Ідентифікація</td><td>Прапорці</td><td colspan="2">Зміщення фрагментації</td></tr>
<tr><td colspan="2">Час життя</td><td>Протокол</td><td colspan="3">Контрольна сума заголовку</td></tr>
<tr><td colspan="6">IP-адреса відправника</td></tr>
<tr><td colspan="6">IP-адреса отримувача</td></tr>
<tr><td colspan="5">Опції</td><td>Додаток</td></tr>
<tr><td colspan="6">Дані (65535 мінус заголовок)</td></tr>
<tr><td colspan="6">...</td></tr>
</table>

Рисунок 1 – Структура IPv4

Опис полів:

– **Версія (Version)** – 4-бітове поле, що описує використовувану версію протоколу IP. Всі пристрої зобов'язані використовувати протокол IP однієї версії, пристрій що використовує іншу версію буде відкидати пакети.

– **Довжина IP-заголовку (IP header Length – HLEN)** – 4-бітове поле, що описує довжину заголовку пакету в 32-бітових блоках. Це значення – повна довжина заголовку з врахуванням двох полів змінної довжини.

– **Тип обслуговування (Type of Service – TOS)** – 8-бітове поле, що вказує на ступінь важливості інформації, що привласнена протоколом верхнього рівня.

– **Загальна довжина (Total Length)** – 16-бітове поле, що описує довжину пакету в байтах, із заголовком та даними включно. Для того щоб вирахувати довжину блока даних, потрібно від повної довжини відняти значення поля HLEN.

– **Ідентифікація (Identification)** – шістнадцятибітове поле, що зберігає ціле число, яке описує даний пакет. Це число являє собою послідовний номер.

– **Прапорці (Flags)** – 3-бітове поле, в якому два молодших біта контролюють фрагментацію пакетів. Перший біт визначає чи було пакет фрагментовано, а другий чи є цей пакет останнім фрагментом в серії фрагментів.

– **Зміщення фрагментації (Fragment Offset)** – 13-бітове поле, що допомагає зібрати разом фрагменти пакетів. Це поле дозволяє використовувати 16 бітів в сумі для прапорів фрагментації.

– **Час життя (Time-to-Live – TTL)** – 8-бітове поле – лічильник, в якому зберігаються послідовно зменшуване значення кількості пройдених вузлів (роутерів, що їх ще іноді в цьому випадку називають хопами (hops)) на шляху до місця призначення. У випадку коли лічильник пройдених хопів дорівнюватиме нулю – пакет буде відкинуто, таким чином попереджується нескінченна циклічна пересилка пакетів.

– **Протокол (Protocol)** – 8-бітове поле, що вказує на те, який протокол верхнього рівня отримає пакет, після завершення обробки IP-протоколом. Наприклад TCP або UDP.

– **Контрольна сума заголовку (Header Checksum)** – 16-бітове поле, що допомагає перевірити цілісність заголовку пакету.

– **IP-адреса відправника (Source IP address)** (адресант, сорс, відправник) – 32-бітове поле, що зберігає IP-адресу вузла-відправника.

– **IP-адреса отримувача (Destination IP adress)** (адресат, дест, отримувач) – 32-бітове поле, що зберігає адресу вузла призначення (отримувача).

– **Опції (Options)** – поле змінної довжини, що дозволяє протоколу IP реалізувати підтримку різних опцій, зокрема засобів безпеки.

– **Підкладка (Padding)** – поле, що використовується для вставки додаткових нулів, для гарантування кратності IP-заголовку 32 бітам.

– **Дані (Data)** – поле змінної довжини (64 Кбіт макс.), що зберігає інформації для верхніх рівнів.

Рисунок 2 – Структура пакета IPv6

Опис полів:

– *Version*: версія протоколу; для IPv6 це значення дорівнює 6 (значення в бітах – 0110).

– *Traffic class*: пріоритет пакета (8 бітів). Це поле містить два параметри. Старші 6 бітів використовуються DSCP для класифікації пакетів. Решта два біти використовуються ECN для контролю перевантаження.

– *Flow label*: відмітка потоку (див. відмітки потоків).

– *Payload length*: на відміну від поля *Total length* протоколу IPv4 дане поле не включає заголовок пакета (16 бітів). Максимальний розмір, що визначається розміром поля – 64 Кбайти. Для пакетів більшого розміру використовується *Jumbo payload*.

– *Next header*: вказує тип розширеного заголовка (англ. *IPv6 extension*), що розміщений одразу за основним. В останньому розширеному заголовку поле *Next header* вказує тип транспортного протоколу (TCP, UDP і т. д.)

– *Hop limit*: аналог поля *time to live* в IPv4 (8 бітів).

– *Source Address* і *Destination Address*: адреси відправника та отримувача відповідно; по 128 бітів.

Взаємодія усіх IP пристрої можливо описати за допомогою 7 рівневою моделлю взаємодії відкритих систем (Open System Interconnection, OSI).

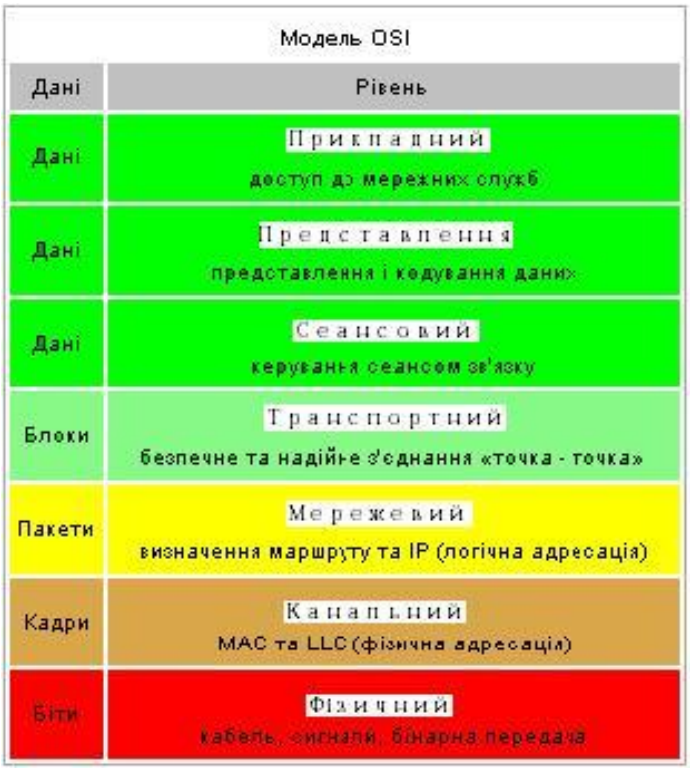

Рисунок 3 – 7-рівнева модель OSI

Транспортний протокол UDP (User Datagram Protocol - п*ротокол датаграм користувача*) є простим, ненадійним протоколом датаграм для IP (*Internet Protocol* — «міжмережевий протокол») пристроїв, тоді як транспортний TCP (*Transmission Control Protocol* — «протокол керування передаванням») є більш складним, надійним протоколом потоку байтів. SCTP (*Stream Control Transmission Protocol* — «протокол передачі з керуванням потоком») схожий на TCP як

надійний транспортний протокол, але він також забезпечує межі повідомлень, підтримку на рівні транспорту для мультидомінгу та спосіб мінімізації блокування головного рядка.

Протокол TCP інкапсулюється в протокол IP, тобто сегменти TCP вкладаються в протокол IP в якості даних.

<----------IP датаграма---------->		
	<----------TCP сегмент---------->	
Заголовок IP	Заголовок TCP	Дані TCP
20 байт	20 байт	

Рисунок 4 – Інкапсуляція TCP в протокол IP.

Розглянемо більш детально з’єднання по TCP/IP.

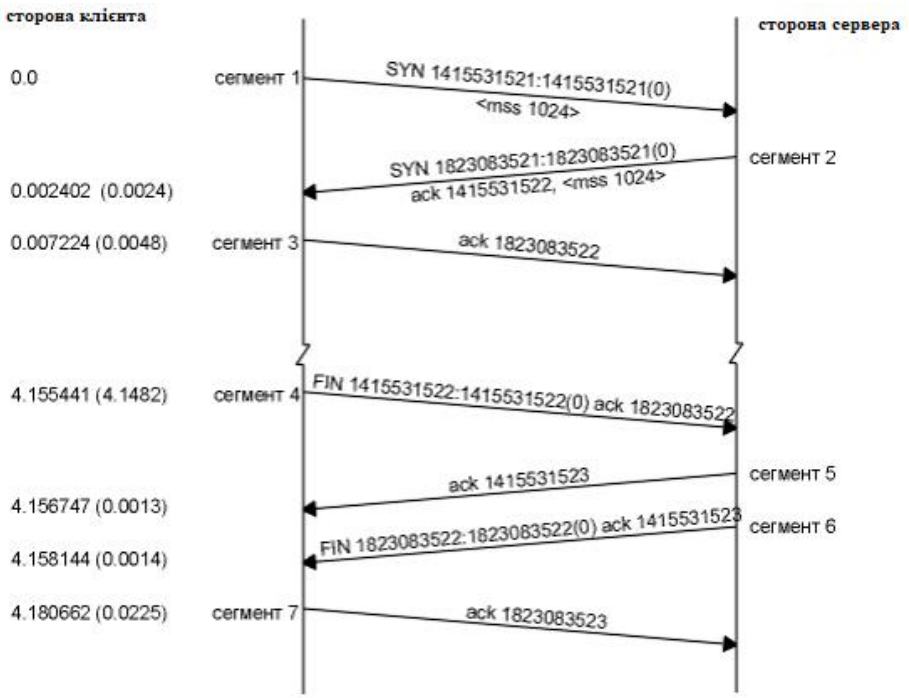

Рисунок 5 – Протокол встановлення з'єднання TCP/IP

Для встановлення з’єднання запитуюча сторона (яка, як правило, називається клієнт) відправляє SYN сегмент, вказуючи номер порту сервера, до

якого клієнт хоче приєднатися, і вихідний номер послідовності клієнта (в даному прикладі ISN, 1415531521). Це сегмент номер 1. Сервер відповідає своїм сегментом SYN, що містить вихідний номер послідовності сервера (сегмент 2). Сервер також підтверджує прихід SYN клієнта з використанням ACK (ISN клієнта плюс один). На SYN використовується один номер послідовності.

Клієнт повинен підтвердити прихід SYN від сервера з використанням ACK (ISN сервера плюс один, сегмент 3).

Цих трьох сегментів достатньо для встановлення з'єднання. Часто це називається триразовим рукостисканням (three-way handshake).

Вважається, що сторона, яка посилає перший SYN, активізує з'єднання (активне відкриття). Інша сторона, яка отримує перший SYN і відправляє наступний SYN, приймає пасивну участь у відкритті з'єднання (пасивне відкриття).

Коли кожна сторона відправила свій SYN, щоб встановити з'єднання, вона вибирає початковий номер послідовності (ISN) для цього з'єднання. ISN повинен мінятися кожен раз, тому кожне з'єднання має свій, відмінний від інших ISN. RFC 793 вказує, що ISN є 32-бітовим лічильником, який збільшується на одиницю кожні 4 мікросекунди. Завдяки номерам послідовностей, пакети, що затрималися в мережі і доставлені пізніше, не сприймаються як частина існуючого з'єднання.

Як вибирається номер послідовності? У 4.4 BSD (і в більшості Berkeley реалізацій) при ініціалізації системи початковий номер послідовності встановлюється в 1. Подібна практика засуджується вимогою до хостів Host Requirements RFC. Потім ця величина збільшується на 64000 кожні півсекунди і повертається в значення 0 через кожні 9,5 години. (Це відповідає лічильнику, який збільшується на одиницю кожні 8 мікросекунд, а не кожні 4 мікросекунди.) Крім того, кожного разу, коли встановлюється з'єднання, ця величина збільшується на 64000.

Проміжок в 4,1 секунди між сегментами 3 і 4 відповідає часу між встановленням з'єднання і введенням команди quit для telnet, щоб розірвати з'єднання.

Протокол розриву з'єднання TCP/IP.

Для того щоб встановити з'єднання, необхідно 3 сегмента, а для того щоб розірвати - 4. Це пояснюється тим, що TCP з'єднання може бути в наполовину закритому стані. Так як TCP з'єднання повнодуплексне (дані можуть пересуватися в кожному напрямку незалежно від іншого напрямку), кожен напрям має бути закритий незалежно від іншого. Правило полягає в тому, що кожна сторона повинна послати FIN, коли передача даних завершена. Коли TCP приймає FIN, він повинен повідомити додаток, що віддалена сторона розриває з'єднання і припиняє передачу даних у цьому напрямку. FIN зазвичай відправляється в результаті того, що додаток було закрито.

Отримання FIN означає тільки, що в цьому напрямку припиняється рух потоку даних. TCP, що отримав FIN, може все ще посилати дані. Незважаючи на те, що додаток все ще може посилати дані при наполовину закритому TCP з'єднанні, на практиці тільки деякі (зовсім небагато) TCP додатків використовують це. Звичайним є той сценарій, який показаний на Рис. 5.

Сегмент номер 4 на Рис. 5 приводить до закриття з'єднання і надсилається, коли Telnet клієнт припиняє роботу. Це відбувається, коли ми вводимо quit. При цьому TCP клієнт змушений послати FIN, закриваючи потік даних від клієнта до сервера.

Коли сервер отримує FIN, він відправляє назад ACK з прийнятим номером послідовності плюс один (сегмент 5). На FIN витрачається один номер послідовності, так само як на SYN. У цей момент TCP сервер також доставляє додатку ознаку кінця файлу (end-of-file) (щоб вимкнути сервер). Потім сервер закриває своє з'єднання, що змушує його послати FIN (сегмент 6), на який клієнт повинен підтвердити (ACK), збільшивши на одиницю номер прийнятої послідовності (сегмент 7).

На Рис. 6 показано типовий обмін сегментами при закритті з'єднання.

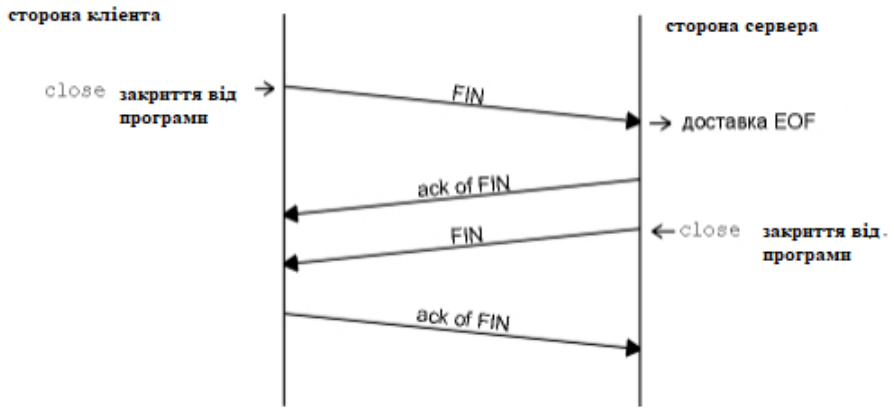

Рисунок 6 – Типовий обмін сегментами при закритті з'єднання TCP/IP

Номери послідовності на малюнку не вказано. На цьому малюнку FIN надсилаються через те, що додатки закривають свої з'єднання, тоді як ACK для цих FIN генерується автоматично програмним забезпеченням TCP.

З'єднання зазвичай встановлюється клієнтом, тобто перший SYN рухається від клієнта до сервера. Однак будь-яка сторона може активно закрити з'єднання (відправивши перший FIN). Часто, однак, саме клієнт визначає, коли з'єднання повинно бути розірване, так як процес клієнта в основному керується користувачем, який вводить щось подібне "quit", щоб закрити з'єднання. На Рис. 6 можна поміняти місцями мітки, наведені вгорі малюнка, назвавши ліву сторону сервером, а праву сторону клієнтом. Однак навіть у цьому випадку все буде працювати саме так, як показано на малюнку.

З’єднання за протоколом SCTP

Створення нового підключення в протоколах TCP і SCTP відбувається за допомогою механізму підтвердження пакетів. У протоколі TCP ця процедура отримала назву триетапне підтвердження (three-way handshake). Клієнт посилає пакет SYN (скор. Synchronize). Сервер відповідає пакетом SYN-ACK

(Synchronize-Acknowledge). Клієнт підтверджує прийом пакета SYN-ACK пакетом ACK. На цьому процедура встановлення з'єднання завершується.

Протокол TCP має потенційну вразливість, обумовлену тим, що порушник, встановивши фальшиву IP-адресу відправника, може послати серверу безліч пакетів SYN. При отриманні пакету SYN сервер виділяє частину своїх ресурсів для встановлення нового з'єднання. Обробка безлічі пакетів SYN рано чи пізно використає всі ресурси сервера і зробить неможливим обробку нових запитів. Така атака отримала назву «відмова в обслуговуванні» (Denial of Service, або скорочено DoS).

Протокол SCTP захищений від подібних атак за допомогою механізму чотирьохетапного підтвердження (four-way handshake) і введенням маркера (cookie). За протоколом SCTP клієнт починає процедуру встановлення з'єднання посилкою пакета INIT. У відповідь сервер посилає пакет INIT-ACK, який містить маркер (унікальний ключ, що ідентифікує нове з'єднання). Потім клієнт відповідає посилкою пакета COOKIE-ECHO, в якому міститься маркер, посланий сервером. Тільки після цього сервер виділяє свої ресурси новому підключенню і підтверджує це відправленням клієнту пакету COOKIE-ACK.

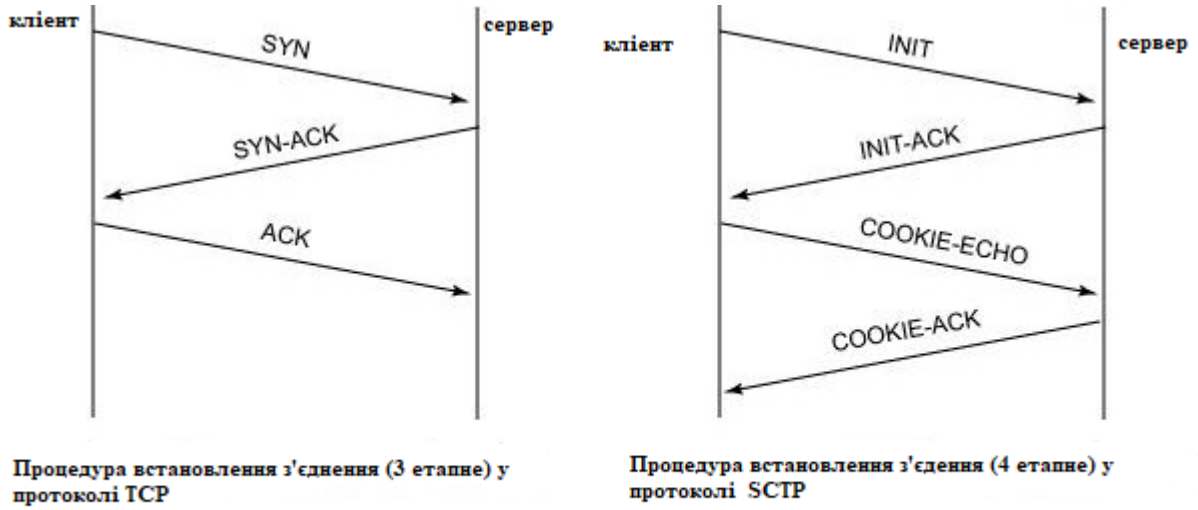

Рисунок 7 – Процедури встановлення з’єднання у TCP та SCTP

Для вирішення проблеми затримки пересилання даних при виконанні процедури чотирьохетапного підтвердження в протоколі SCTP допускається включення даних в пакети COOKIE-ECHO і COOKIE-ACK.

Поетапне завершення передачі даних

У протоколі TCP можлива ситуація, коли вузол закриває у себе сокет (виконуючи посилку пакету FIN), але продовжує приймати дані. Пакет FIN вказує кореспонденту на відсутність даних для передачі, проте до тих пір, поки кореспондент не закриє свій сокет, він може продовжувати передавати дані. Стан часткового закриття використовується додатками вкрай рідко, тому розробники протоколу SCTP вважали за потрібне замінити його послідовністю повідомлень для розриву існуючої асоціації. Коли вузол закриває свій сокет (надсилає повідомлення SHUTDOWN), обидва кореспонденти повинні припинити передачу даних, при цьому дозволяється лише обмін пакетами, що підтверджують прийом раніше відправлених даних.

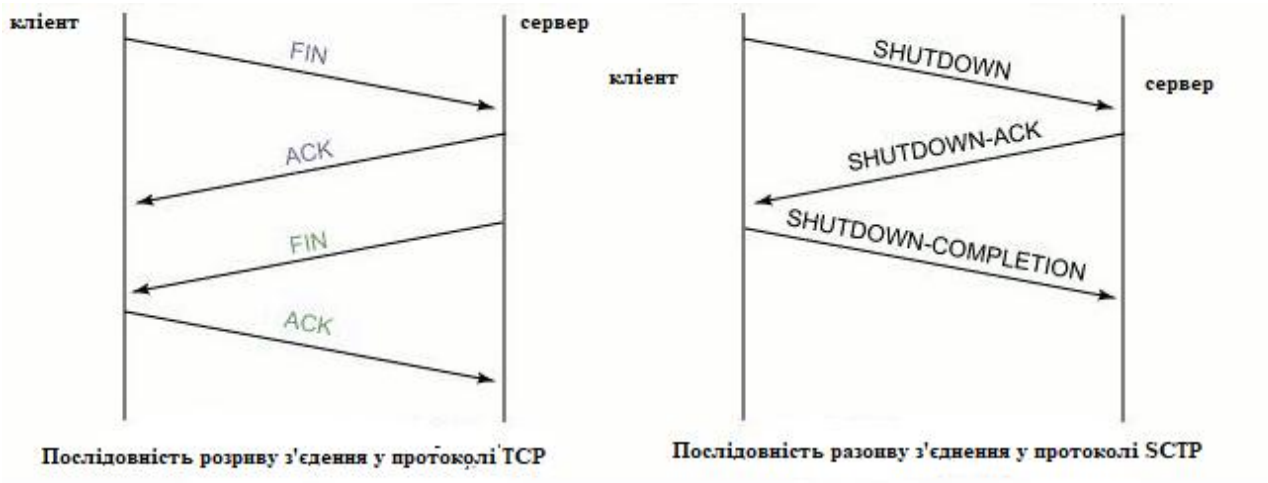

Рисунок 8 – Послідовність розриву у TCP та SCTP

Таблиця 1 – Порівняння можливостей протоколів транспортного рівня

Параметр	UDP	TCP	SCTP
Встановлення зв'язку	Ні	Так	Так
Надійна передача	Ні	Так	Так
Збереження меж повідомлення	Так	Ні	Так
Впорядкована доставка	Ні	Так	Так
Невпорядкована доставка	Так	Ні	Так
Контрольні суми даних	Так	Так	Так
Розмір контрольної суми (біт)	16	16	32
Шлях MTU	Ні	Так	Так
Управління нагромадженням	Ні	Так	Так
Багатонитевість	Ні	Ні	Так
Підтримка декількох інтерфейсів	Ні	Ні	Так
Поєднання потоків	Ні	Так	Так

Контрольні запитання

1. Що таке TCP/IP ?

2. Наведіть різницю між TCP/IP v4 та v6.

3. Поясніть призначення та застосування IP-адрес. Наведіть кілька прикладів IP-адрес.

4. Що таке UDP ?

5. Поясніть та намалюйте сеанс з’єднання TCP/IP.

6. Поясніть та намалюйте сеанс з’єднання UDP.

7. Що таке SCTP ?

8. Поясніть та намалюйте сеанс з’єднання SCTP.

9. Які відмінності UDP від TCP?

10. Які відмінності SCTP від TCP?

11. Які відмінності SCTP від UDP?

ПРАКТИЧНА РОБОТА №2

Міжпроцесна комунікація. Сокети Berkley. Сокети в Windows

Засобами міжпроцесної комунікації (IPC - Inter-Process Communication) є сигнали, канали, повідомлення, семафори, «Спільна пам'ять» і сокети. Процеси виконуються у власному адресному просторі, вони ізольовані один від одного, тому необхідні механізми для взаємодії процесів, що надаються самою операційною системою, і, як правило, розташовані в адресному просторі системи.

Комунікація між процесами необхідна для вирішення наступних завдань:

 передача даних від одного процесу до іншого.

 спільне використання загальних даних декількома процесами.

Повідомлення, семафори і «Спільна пам'ять» узагальнено називають System V IPC. Ці механізми об'єднуються в єдиний пакет, тому що їх відповідні системні виклики володіють близькими інтерфейсами, а в їх реалізації використовуються багато загальні підпрограми. Ось основні загальні властивості всіх трьох механізмів:

1. Для кожного механізму підтримується загальносистемна таблиця, елементи якої описують всіх існуючих в даний момент представників механізму.

2. Елемент таблиці містить деякий числовий ключ, який є вибраним користувачем ім'ям представника відповідного механізму. Щоб два або більше процесу могли використовувати певний механізм, вони повинні заздалегідь домовитися про іменуванні використовуваного представника цього механізму.

3. Процес, який бажає почати користуватися одним з механізмів, звертається до системи з необхідним викликом, вхідними параметрами якого є ключ об'єкту і додаткові прапори, а у відповідь параметром є числовий дескриптор, який використовується в подальших системних викликах подібно до того, як використовується дескриптор файлу при роботі з файловою системою .

4. Захист доступу до раніше створеним елементам таблиці кожного механізму ґрунтується на тих же принципах, що і захист доступу до файлів.

Сокети Берклі — прикладний програмний інтерфейс, що складається з бібліотеки для розробки програм мовою програмування С з підтримкою міжпроцесної взаємодії, що часто застосовується у комп'ютерних мережах.

Як API, сокети Берклі вперше з'явилися у операційній системі 4.2BSD Unix, що була випущена у 1983 році. Тим не менш, тільки у 1989 році Каліфорнійський університет у Берклі зміг випустити свою операційну систему і мережеві бібліотеки без ліцензійних обмежень з боку AT&T.

Сокети Берклі є де-факто стандартом абстракції для мережевих сокетів. Саме тому багато мов програмування використовують мережеві інтерфейси, подібні до API мови С.

Альтернативою сокетного API є заснований на STREAMS Інтерфейс транспортного рівня (TLI). Проте сокети Берклі значно популярніші та мають більшу кількість реалізацій.

Інтерфейс сокетів Берклі дозволяє взаємодію між хостами чи між процесами на одному комп'ютері, використовуючи концепцію Інтернет-сокетів. Дана технологія може працювати з багатьма драйверами та пристроями вводу/виводу, хоча їхня підтримка залежить від конкретної операційної системи. Реалізація інтерфейсу достатня для підтримки TCP/IP, саме тому це одна із основних технологій, на якій заснований Інтернет. Розробка технології була започаткована у Каліфорнійському університеті Берклі для застосування на ОС Unix. Всі сучасні операційні системи мають реалізацію інтерфейсу сокетів Берклі, оскільки вони є стандартним інтерфейсом для підключення до мережі Інтернет.

У модулі socket функція setlocking відповідає за блокування сокета поки обслуговується з’єднення. Без явної вказівки в модулі socket застосовується setlocking(TRUE) — блокування включено. Щоб блокування зняти потрібно явно вказати - написати в програмі setlocking(FALSE).

subprocess – модуль міжпроцесовою взаємодії.

Міжпроцесова взаємодія (англ. IPC) – це обмін даними між процесами. Як правило реалізується засобами ОС. До методів IPC належать: файли, неіменовані і

іменовані канали, черги повідомлень, сигнали, спільна пам’ять, сокети і файли, що відображаються у пам’ять. У прикладі створюється канал між стандартними потоками введення/виведення/помилок (stdin/stdout/stderr) процесів. Цей модуль створює новий процес server.py, відсилає йому дані на stdin та отримує дані з його stdout. Приклад програми наведено нижче.

```
import subprocess, pickle
data=['A','B','C'] # дані
s = pickle.dumps(data) # серіалізувати список у рядок
s=s.encode("string_escape") # перетворити в рядковий літерал Python (без "\n")
p=subprocess.Popen(["python", "server.py"],stdin = subprocess.PIPE,
stdout= subprocess.PIPE, stderr= subprocess.PIPE) # створити процес
stdout, stderr = p.communicate(input=s) # надіслати дані в stdin, отримати дані з stdout, чекати завершення процесу
s=stdout.decode("string_escape") # перетворити з рядкового літералу Python
print pickle.loads(s) # перетворити в список
```

Результат роботи програми:

['A', 'B', 'C', 'D']

Приклад, server.py – модуль сервера

Модуль отримує дані від клієнта через stdin та відсилає їх назад через stdout.

```
import pickle,sys
# ! тут заборонено використовувати print
s=sys.stdin.read().decode("string_escape") # перетворити з рядкового літералу Python
data = pickle.loads(s) # перетворити в список
data.append('D') # додати дані
s=pickle.dumps(data) # серіалізувати список у рядок
s=s.encode("string_escape") # перетворити в рядковий літерал Python (без
```

"\n")

sys.stdout.write(s) *# записати в stdout*

multiprocessing – модуль підтримки багатьох процесів

multiprocessing – це пакет, який підтримує створення процесів із використанням API, який подібний на API модуля threading. Забезпечує локальний і віддалений паралелізм, ефективно долає GIL шляхом використання підпроцесів замість потоків. У прикладі розпаралелюється задача знаходження квадратів 50 матриць 1000x1000 за допомогою класу Poll і його методу map. Зауважте, що виграш у продуктивності буде досягнуто тільки на багатопроцесорній системі. Приклад виконання програми послідовно в паралельному і звичайному режимах так:

python main.py

python main.py s

Для визначення продуктивності програми використано модуль timeit. У Windows можна також використати команду:

echo %time% & main.py & call echo %^time%

```
import numpy as np
from multiprocessing import Pool # задіяти багато процесів
#from multiprocessing.dummy import Pool # або задіяти багато потоків
import sys, timeit
def f(x): # функція, яка виконується в кожному процесі
   return x*x
if __name__ == '__main__':
   time = timeit.default_timer()
   X=[np.matrix(np.random.rand(1000,1000)) for i in range(50)] # 50 матриць 1000x1000
   if 's' in sys.argv:
      Y=map(f,X) # застосувати f для кожного у X (тільки 1 процес)
```

```
    else:
        p=Pool(4) # створити пул 4-х процесів
        Y=p.map(f, X) # задіяти 4 процеси
    print timeit.default_timer() - time
```

Результат роботи програми.

9.77497753973

15.1325679658

multiprocessing – запуск паралельних задач.

Аналог прикладу concurrent.futures на основі multiprocessing. Тут функціям ProcessPoolExecutor, submit, result, map відповідають Pool, apply_async, get, map.

```
import time
from multiprocessing import Pool
def f(x): # функція, яка буде виконуватись в окремих процесах
    time.sleep(x) # затримка (тільки для тестування паралельності)
    return x
if __name__ == '__main__':
    pool = Pool() # пул процесів
    a = pool.apply_async(f, [4]) # AsyncResult
    b = pool.apply_async(f, [2])
    while any([a,b]): # отримати результати асинхронно
        if a and a.ready(): print a.get(); a=False
        if b and b.ready(): print b.get(); b=False
    #print pool.map(f, [1,2]) # або чекати усі результати
```

Результат роботи програми.

2, 4

multiprocessing – міжпроцесова взаємодія

Для обміну об’єктами між процесами можна використовувати черги (Queue), канали (Pipe), спільну пам’ять (Value, Array) [5, 19]. Клас Manager створює об’єкт, що контролює серверний процес, який зберігає

об’єкти Python і дозволяє іншим процесам використовувати їх. Використання об’єктів Manager більш гнучке, ніж об’єктів спільної пам’яті, так як вони можуть підтримувати об’єкти довільних типів. Але вони більш повільні. У прикладі за допомогою Manager створюється список, у який базовий і дочірній процес паралельно додають елементи.

```
    from multiprocessing import Process, Manager
def f(L): # функція, що виконується в окремому процесі
  L.append(4)
if __name__ == '__main__':
  manager = Manager() # менеджер
  L = manager.list([1,2]) # спільний для процесів список
  p = Process(target=f, args=(L, )) # новий процес
  p.start() # стартувати процес (робота з L розпаралелюється)
  L.append(3)
  p.join() # приєднати процес
  print L
```

Результат роботи програми.

[1, 2, 3, 4]

thread – модуль створення багатьох потоків керування

Потоком виконання називають частину процесу, яка може виконуватись паралельно з іншими потоками цього процесу і використовувати спільні з ними ресурси. Синхронізація потоків і процесів – це механізм, який перешкоджає одночасному їхньому зверненню до спільно використовуваних ресурсів. Модуль thread забезпечує низькорівневі (на відміну від threading) примітиви для роботи з багатьма потоками. У прикладі створюються 4 потоки, які виконують функцію f. Звернення потоків до спільного списку A синхронізовано за допомогою простого об’єкта блокування allocate_lock. Нижче показані результати роботи програми з цим об’єктом і без нього. Зауважте, що в CPython існує глобальне блокування інтерпретатора Global Interpreter Lock (GIL), яке являє

собою механізм синхронізації потоків, що не дозволяє в один момент часу виконуватись більше ніж одному потоку. Тому застосовуйте модуль multiprocessing, якщо програмі потрібно задіяти для обчислень кілька процесорів. А багатопотоковість краще застосовувати у випадку багатьох одночасних задач введення-виведення.

```
import thread,time
def f(i): # функція виконується в окремому потоці
  mutex.acquire() # блокувати (лише один потік може виконуватись в один і той самий момент часу)
  A.append(i)
  time.sleep(1)
  A.append(i)
  mutex.release() # розблокувати
  T[i]=1 # повідомити головному потоку, що потік завершився
A=[] # глобальний список
T=[0,0,0,0] # глобальний список (якщо потік `i` завершився, то T[i]=1)
mutex = thread.allocate_lock() # створити блокуючий об'єкт
for i in range(4): # створити 4 потоки
  thread.start_new(f, (i, )) # стартувати потік 'i'
while 0 in T: # поки усі потоки не приєднаються
  pass # тут головний потік може робити щось своє
print A
```

Результат роботи програм.

[0, 0, 1, 1, 2, 2, 3, 3]

[0, 1, 2, 3, 1, 3, 2, 0]

threading – високорівневий інтерфейс потоків

Цей модуль створює високорівневі інтерфейси потоків на основі низькорівневого модуля thread [5, 19]. Потоки описуються нащадком класу threading.Thread, а їхня активність – перевизначеним методом run. У

прикладі створюються 4 потоки, які виконують код у методі run. Звернення потоків до спільного списку A синхронізовано за допомогою простого об’єкта блокування threading.Lock. Нижче показані результати роботи програми з цим об’єктом і без нього. Додатково створюється потік, який стартує через 2 секунди і додає в список A рядок ‘timer’.

```
import threading, time

class Thread(threading.Thread): # успадкований від threading.Thread
    def __init__(self, i): # конструктор
        self.i=i # ідентифікатор потоку
        threading.Thread.__init__(self) # виклик конструктора базового класу
    def run(self): # забезпечує логіку потоку
        mutex.acquire() # блокувати (лише один потік може виконуватись в один і той самий момент часу)
        #semaphore.acquire() # або так
        A.append(self.i)
        time.sleep(1)
        A.append(self.i)
        mutex.release() # розблокувати
        #semaphore.release() # або так

mutex = threading.Lock() # те саме що thread.allocate_lock()
#semaphore=threading.Semaphore(1) # або семафор (тільки 1 потік одночасно)
A=[] # глобальний список
T=[] # список потоків
for i in range(4): # створити 4 потоки
```

```
    t=Thread(i) # створити потік
    t.start() # виконати метод run в потоці
    T.append(t) # додати в список потоків
t = threading.Timer(2.0, lambda: A.append('timer')) # створити потік,
t.start() # який стартує через 2 с
T.append(t)
for t in T:
    t.join() # поки усі потоки не приєднаються
print A
```

Результат роботи програми.

[0, 0, 1, 'timer', 1, 2, 2, 3, 3]

[0, 1, 2, 3, 2, 3, 1, 0, 'timer']

Реалізувати функціонал консольного варіанту сервера чату.

Завдання

А – Блокуючі сокети

1) Виконати розробку сервера чату на базі сокету в блокуючому режимі.

2) Розробити клієнта для підключення до сервера чату (у якості альтернативи допускається використання Telnet).

3) Запустити сервер чату та виконати підключення одного клієнта. Перевірити роботу з одним клієнтом.

4) Перевірити роботу при спробі підключення декількох клієнтів, результати відобразити у протоколі (описати роботу, проблеми та надати пояснення).

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА № 3

Використання адрес і перетворення імен.

Розрізняють три типи адрес стека TCP/IP: локальні, IP-адреси та доменні імена.

Під локальною (апаратною, фізичною) адресою розуміють такий тип адреси, який використовується засобами базової технології для доставки даних в межах підмережі, що є елементом складеної інтер-мережі. В різних підмережах є допустимими різні мережеві технології, різні стеки протоколів, тому при створенні стека TCP/IP передбачалась наявність різних типів локальних адрес.

Для того, щоб в мережі Ethernet стала можливою локальна доставка фреймів, необхідна певна система адресації, тобто присвоєння імен комп'ютерам та інтерфейсум. Кожен вузол має унікальний спосіб само-ідентифікації. Ніякі дві фізичні адреси в мережі не повинні збігатись. Фізичні адреси, які в Ethernet також називають адресами керування доступом до середовища передавання (Media Access Control, MAC), записані в мережевому адаптері ПК або мережевих інтерфейсах пристроїв (маршрутизаторів, комутаторів тощо).

MAC-адреса має довжину 48 бітів і записується у вигляді дванадцяти шістнадцяткових цифр (наприклад, 00-60-2F-3A-07-BC). Перші шість цифр, що задаються IEEE, ідентифікують виробника або продавця пристрою і, містять унікальний ідентифікатор організації (Organizationally Unique Identifier – OUI). Другі шість цифр містять серійний номер інтерфейсу або інше значення, що задається конкретним виробником. MAC-адресу іноді називають прошитою (Burned-In Address – BIA), оскільки вона записана в постійній пам'яті (Read-Only Memory – ROM) інтерфейсу або пристрою. На Рис. 1 показано формат MAC-адреси.

Без MAC-адрес локальна мережа являла б собою лише групу ізольованих комп'ютерів, і доставка Ethernet-фреймів була б неможливою. Внаслідок цього на канальному рівні до даних верхніх рівнів додається заголовок, що містить MAC-

адресу пристрою та кінцевик. Заголовок та кінцевик містять службову інформацію, призначену для канального рівня пристрою, якому направляється фрейм. Дані верхніх рівнів інкапсулюються в заголовок та кінцевик канального рівня.

IP-адреси є основним типом адрес, на основі яких мережевий рівень передає пакети між мережами. Ці адреси складаються з 4 байтів і запису-ються у десятково-точковій нотації (наприклад, 195.1.7.26). Кожна з частин адреси, розділених точками, називається октетом (оскільки складається з 8 біт). IP-адреса призначається адміністратором під час конфігурування комп'ютерів та маршрутизаторів.

IP-адреси утворюють номер мережі та номер вузла. Номер мережі може бути вибраний адміністратором довільно або призначений за рекомендацією спеціального підрозділу Internet (Internet Network Information Center, InterNIC), якщо мережа повинна працювати як складова частина Internet. Зазвичай постачальники послуг Internet отримують діапазони адрес у підрозділів InterNIC, а потім розподіляють їх між своїми абонентами. Номер вузла в протоколі IP призначається незалежно від локальної адреси вузла. Маршрутизатор за визначенням входить одразу до декількох мереж. Тому кожен порт маршрутизатора має власну IP-адресу. Кінцевий вузол також може входити до декількох IP-мереж В цьому випадку комп'ютер повинен мати декілька IP-адрес, за числом мережевих зв'язків. Таким чином, IP-адреса характеризує не окремий комп'ютер або маршрутизатор, а одне мережеве з'єднання.

Символьні доменні імена. Вся мережа Internet побудована на ієрархічній системі адресації. Такий підхід дозволяє здійснювати маршрутизацію, засновану на класах адрес, а не на індивідуальних адресах. Однак використання IP-адрес не дуже зручно для користувачів. Так, різниця між адресами 194.6.197.26 та 194.6.197.62 практично непомітна, хоча обидві адреси належать до різних ресурсів мережі. Ймовірність того, що користувач може помилитьсь і ввести

неправильну IP-адресу, досить висока, оскільки числова IP-адреса ніяк не пов’язана з тематикою ресурсу.

Рисунок 1 – Формат MAC-адреси

Для прив’язування вмісту Web-сторінки та її адреси була розроблена спеціальна система доменних імен (DNS). Служба DNS призначена для трансляції IP-адрес в імена і навпаки. Домен – це група вузлів, що розташовані в одній географічній області, або ж вузлів, що використовуються зі спільною метою. Доменним іменем називають рядок символів і/або цифр, і зазвичай таке ім’я відповідає цифровій IP-адресі Web-вузла в мережі Internet. На сьогодні в мережі Internet існує більш як 200 доменів верхнього (або першого) рівня. Домени першого рівня можуть бути створені за географічною ознакою: .ua – Україна; .us – США; .de – Німеччина; .uk – Об’єднане Королівство. Крім того, існує багато загальних доменних імен: .edu – Web-сторінки, присвячені освітнім закладам; .com – комерційні Web-вузли; .gov – урядові вузли; .org – некомерційні вузли; .net – мережеві служби.

Сервер доменних імен – мережевий пристрій, який за запитом користувача перетворює доменні імена у відповідні IP-адреси і повертає результат клієнту.

Система доменних є строго ієрархічною, тому існує декілька рівнів імен і відповідно серверів DNS.

З метою отримання можливості опису мереж різного розміру та по-легшити їх класифікацію, IP-адреси було розділено на групи, які називають класами. Така схема адресації називається класовою. Кожна повна 32-бітна IP-адреса поділяється на дві частини, що описують мережу та вузол. Біт або послідовність бітів на початку кожної адреси задають її клас (Рис. 2). Є п’ять класів IP-адрес.

Рисунок 2 – Структура IP-адрес різних класів

Адреса класу A призначена для дуже великих мереж. В ній використовується тільки перший октет як ідентифікатор мережі. Три октети, що залишились, ідентифікують адресу вузлів. Перший біт в адресі класу A завжди нульовий. Враховуючи це, найменше допустиме число буде рівне 00000000 (десятковий 0), а найбільше – 01111111 (десяткове число 127). Варто відзначити, що обидва номери 0 та 127 є зарезервованими і не можуть бути використані як мережеві адреси. Будь-які адреси, що починаються з числа в діапазоні від 1 до 126 в першому октеті є адресами класу A. Мереж класу A небагато, але кількість вузлів у них може досягати 224 - 2 = = 16 777 214 вузлів (два номери ідентифікують номера мережі та широкомовну адресу).

Мережа з номером 127.0.0.0 не може бути присвоєна мережі, оскільки зарезервована для зворотного петлевого (loopback) тестування (маршрутизатори або локальні вузли можуть використовувати його для передавання пакетів самим собі).

Адреса класу B використовується для мереж середнього та великого розмірів. В IP-адресі класу B два перших октети використовується для мережевої адреси, а два других являють собою адресу вузла.

Перші два біти першого октета завжди приймають значення „1” і „0”, шість бітів, що залишились, можуть містити будь-які комбінації нулів та одиниць. Таким чином, найменше число, яке може бути використане для адрес цього класу рівне 10000000 (десяткове 128), і найбільше – 10111111 (десяткове значення рівне 191). Будь-які адреси, що містять в першому октеті числа від 128 до 191, є адресами класу B. Мережа класу B може містити максимум 216 - 2 = 65 534 вузлів.

Адреси класу C – це найчастіше використовувані адреси, призначені для використання в малих мережах. Адреса даного класу починається з двійкової комбінації 110. Отже, найменше доступне число – 11000000 (десяткове 192), а найбільше – 1101111 (десяткове значення 223). Якщо адреса в першому октеті містить числа від 192 до 223, значить він на-лежить до класу C. Максимальна кількість вузлів у мережі – 28 – 2 = 254.

Адреси класу D були створені для реалізації в IP-адресах механізму багатоадресної розсилки. Багатоадресною або груповою адресою (multicast address) називається унікальна мережева адреса, що використовується для відправлення пакетів певним групам мережевих пристроїв. Таким чином, одна мережева станція може передавати один потік даних декільком отримувачам.

Діапазон адрес класу D, які називають багатоадресними IP-адресами також певним чином обмежений. Перші чотири біти такої адреси є 1110, тому перший октет адрес цього класу може приймати значення від 11100000 до 11101111 або в десятковому записі від 224 до 239.

Адреси класу E також були описані в стандартах та виділені в окремий блок. Однак вони були зарезервовані проблемною групою проектування Internet (Internet Engineering Task Force – IETF) для власних дослідницьких потреб і не використовувались в мережі Internet. Перші чотири біти адрес класу E завжди одиничні. Значення першого октета знаходиться в діапазоні від 11110000 до 11111111 або від 240 до 255 – в десятковому вигляді.

Діапазони значень першого октета в IP-адресах для кожного з класів наведено в таблиці 1.

Таблиця 1 – Класи IP-адрес: діапазон значень першого октета

Клас	Перші біти	Мінімальний номер мережі	Максимальний номер мережі	Максимальна кількість мереж	Максимальна кіль-кількість вузлів у мережі
A	0	1.0.0.0	126.0.0.0	2^7 – 2 = 126	2^24 – 2
B	10	128.0.0.0	191.255.0.0	2^14 = 16384	2^16 – 2
C	110	192.0.0.0	223.255.255.0	2^21 = 2097152	2^8 – 2
D	1110	224.0.0.0	239.255.255.255	-	Багатоадресний
E	1111	240.0.0.0	255.255.255.255	-	Зарезервований

Застосування масок під час IP-адресації

Традиційна схема поділу IP-адреси на номер мережі та номер вузла базується на понятті класу, який визначається значеннями декількох пер-ших бітів адреси. Саме тому, що перший байт адреси 129.54.65.3 потрапляє в діапазон 128 -191, ми можемо сказати, що ця адреса відноситься до класу B, а значить, номером мережі є перші два октети, доповнені двома нульовими байтами – 129.54.0.0, а номером вузла – 0.0.65.3.

Для стандартних класів IP-адрес маски мають такі значення:

Клас A – 11111111. 00000000. 00000000. 00000000 (255.0.0.0);

Клас B – 11111111. 11111111. 00000000. 00000000 (255.255.0.0);

Клас C – 11111111.11111111.11111111.00000000 (255.255.255.0).

Доволі часто зустрічається позначення маски у вигляді числа записаного після слешу, наприклад, 129.54.65.3/16. Даний запис означає, що маска для адреси

129.54.65.3 містить 16 одиниць, тобто під номер мережі відведено 16 двійкових розрядів (2 перших байти).

В основу механізму масок покладено принцип отримання номера мережі шляхом порозрядного перемноження адреси вузла і маски. Наприклад, для IP-адреси 180.34.23.134 з маскою 255.255.0.0 маємо

10110100.00100010.00010111.10000110

11111111.11111111.00000000.00000000

00001010.00100010.00000000.00000000 = 180.34.0.0

Супроводжуючи кожну IP-адресу маскою, можна відмовитися від понять класів адрес і зробити більш гнучкою систему адресації. Напри-клад, якщо адресу 129.54.170.164 асоціювати з маскою 255.255.255.0 - номером мережі (а точніше підмережі) буде 129.54.170.0 (а не 129.54.0.0, як це визначено системою класів), якщо з маскою 255.255.248.0 - то 129.54.168.0, а якщо з маскою 255.255.255.224 - то 129.54.170.160.

Взагалі, для виділення підмережі, частина бітів, що відповідає за нумерацію вузла повинна бути визначена як мережева. Такий механізм часто називають позиченням (орендою) бітів. Процес ділення завжди починається з крайнього лівого біта вузла, положення якого залежить від класу IP-адреси.

Зазначимо також, що створення підмереж крім підвищення керованості, дозволяє мережевим адміністраторам обмежувати широкомовні розсилки та реалізувати механізм безпеки низького рівня в локальній мережі.

Безпека при використанні підмереж в ЛКМ реалізується завдяки тому, що доступ в інші підмережі організовується через маршрутизатори, які можуть бути налаштовані так, щоб дозволити або заборонити доступ до підмереж на основі різних критеріїв. Крім того, зовнішній світ “бачить” локальну мережу як єдину мережу, нічого не знаючи про її внутрішню будову. Крім підвищення рівня безпеки такий підхід також дозволяє зменшити ТМ і ефективно їх використовувати. Отримавши локальну адресу вузла 192.168.10.14, зовнішній світ

за межами ЛКМ використовує лише об’явлену основну мережеву адресу 192.168.10.0, оскільки локальна адреса 192.168.10.14 дійсна лише в її межах.

Деякі організації виявили також, що використання механізму виділення підмереж може принести додаткові прибутки за рахунок продажу чи передачі в оренду адреси, що раніше не використовувались.

Вибір необхідної кількості бітів для створення підмережі залежить від потрібної максимальної кількості її вузлів. Для визначення маски підмережі на основі кількості доступних підмереж і вузлів можна використати такі вирази: 2n - кількість використовуваних підмереж, де n – кількість позичених з вузлової частини бітів; 2m - 2 - кількість доступних вузлів, де m – кількість бітів вузлової частини, що лишились (кількість бітів вузлової частини v = n + m).

Наприклад, при запозиченні трьох бітів з вузлової частини мережі класу С для адресації вузлів будуть використовуватися 5 бітів, тому кількість вузлів у кожній підмережі рівне 25 – 2 = 30, а максимальна кількість підмереж складе 2(v-5) = 3 (тут v = 8).

Механізм масок широко використовується в IP-маршрутизації, причому маски можуть використовуватись з різною метою. За їх допомогою адміністратор може структурувати свою мережу, не вимагаючи від постачальника послуг додаткових номерів мереж. На основі цього ж механізму постачальники послуг можуть об’єднати адресні простори декількох мереж шляхом введення так званих “префіксів” з метою зменшення об’єму таблиць маршрутизації та підвищення за рахунок цього продуктивності роботи маршрутизаторів.

Розглянемо на конкретному прикладі процес сегментації мережі із застосування масок постійної довжини (технологія FLSM - Fixed-Length Subnet Masking) для структуризації мережі.

Наприклад, адміністратору мережі треба організувати чотири мережі, а постачальником послуг виділено лише один номер мережі класу В 129.144.0.0. Дана задача може бути розв’язана за допомогою застосування масок постійної довжини. Обрана маска в даному випадку буде 255.255.192.0 (оскільки для

адресації чотирьох наших мереж треба запозичити з бітів номера вузла два старших розряди третього октету). Після накладанні такої маски на видану адресу, кількість розрядів, що відповідали номеру мережі збільшились з 16 до 18. Два додаткові біти (№ 17 і 18) у номері мережі часто інтерпретують як номери підмереж.

В результаті застосування масок схема розподілу адресного простору прийняла вигляд, як показано у Рис 3.

1 октет	2 октет	3 октет			4 октет	Опис мережі
Поле номера мережі класу B		Номер підме-режі		Поле адреса вузла		
129	44					
10000001	00101100	0	0	000000	00000000	Мережа 129.44.0.0
...	...	...	...	...	...	Маска 255.255.192.0
10000001	00101100	0	0	111111	11111111	Вузлів $2^{14} - 2$
10000001	00101100	0	1	000000	00000000	Мережа 129.44.64.0
...	...	...	...	...	...	Маска 255.255.192.0
10000001	00101100	0	1	111111	11111111	Вузлів $2^{14} - 2$
10000001	00101100	1	0	000000	00000000	Мережа 129.44.128.0
...	...	...	...	...	...	Маска 255.255.192.0
10000001	00101100	1	0	111111	11111111	Вузлів $2^{14} - 2$
10000001	00101100	1	1	000000	00000000	Мережа 129.44.192.0
10000001	00101100	1	1	000000	00000001	Маска 255.255.192.0
10000001	00101100	1	1	000000	00000010	Вузлів $2^{14} - 2$
...	...	...	...	...	...	
Невикористані адреси ($2^{14} - 4$)						
10000001	00101100	1	1	111111	11111111	

Рисунок 3 – Поділ адресного простору мережі класу B з використанням технології FLSM

Мережа, отримана в результаті такої структуризації наведена на рис. 1.3. Весь трафік із зовнішнього світу потрапляє у наші внутріш-ні мережі через маршрутизатор M1. Для структуризації інформаційних потоків у внутрішній мережі встановлено додатковий маршрутизатор M2.

В кожній з чотирьох підмереж може бути до 2^14 – 2 вузлів. Якщо такої кількості вузлів немає - адресний простір буде невикористаним.

Ззовні дана мережа виглядає як звичайна мережа класу B, а на локальному рівні – це складена мережа, що має підмережі. Це також має ще одну перевагу, оскільки дозволяє приховати від зовнішнього спостереження структуру такої мережі.

Даний підхід дозволяє у порівнянні з класовою адресацією ефектив-ніше розподіляти адресний простір. Проте, як ми побачили в дано-му випадку, залишилось незадіяними 2^14 - 4 адрес.

DNS-сервер (синонім – сервер імен) – це сервер, який містить базу даних публічних IP-адрес і пов'язаних із ними імен хостів. Як правило, DNS-сервер виконує роль перекладача, дозволяючи або переводячи імена хостів в IP-адреси.

У результаті виходить низка чисел, які стають зрозумілою людині URL-адресою. DNS-сервери використовують спеціальне програмне забезпечення і взаємодіють один з одним за окремими протоколами. У процесі обробки запитів вони призначають правильну IP-адресу URL-адресі або правильну URL-адресу IP-адресі.

DNS є невід'ємною частиною Інтернету з 1985 року завдяки впровадженню «служби каталогів», поширюваної по всьому світу.

Абревіатура DNS (*Domain Name System*) розшифровується як «система доменних імен». Вона являє собою ієрархічний децентралізований каталог іменування комп'ютерів, служб чи інших ресурсів, які під'єднані до глобальної або окремої мережі.

Відвідування будь-якого сайту або сервера можливе через введення певного IP у браузері. Як правило, користувач не знає цю конкретну IP-адресу. Він знає тільки URL, наприклад, www.ukraine.com.ua.

Якщо користувач вводить цю URL-адресу в адресний рядок свого браузера, вона відправляється на доменний сервер, який потім перенаправляє користувача на IP-адресу, прив'язану до URL-адреси. Якщо перша служба не

знаходить відповідного призначення, запит перенаправляється на наступний DNS-сервер.

Головний сервер імен, керований системою ICANN, є останнім варіантом призначення, якщо не було досягнуто збігів із попередніми серверами імен. Більшість приватних користувачів автоматично перенаправляються на DNS-сервер провайдера, коли робиться запит. Великі корпорації часто мають свої власні доменні сервери.

Ієрархічна структура системи DNS

Як вже було відмічено, існує домен кореневого рівня, який позначається крапкою. Наступний рівень ієрархії – це домени верхнього рівня. Вся структура служби DNS є ієрархічною. Існують домени першого, другого, третього, n-го рівнів.

Розглянемо доменне ім'я комп'ютера department.firma.isp.ua. Тут доменом першого рівня є ua, другого – isp, третього – firma, і четвертого – department.

Типи серверів DNS

Існує три основні типи серверів DNS, які відрізняються покладеними на них завданнями:

1) основний сервер DNS;
2) резервний (вторинний) сервер DNS;
3) кешуючий сервер DNS.

Основний сервер DNS керує зоною повноважень. Якщо потрібно додати/видалити домен або вузол або якось інакше модифікувати зону, зміни потрібно проводити на основному сервері DNS. Через певний час, який залежить від налаштувань сервера, основний сервер передасть зону резервному серверу DNS. Дане явище називається трансфером зони.

Що ж до резервних серверів, то повинен бути хоч би один резервний сервер DNS. Тому є декілька причин: якщо клієнтів багато, то наявність резервного сервера DNS дозволить знизити навантаження на основний сервер DNS і

прискорити доступ фізично віддалених від основного сервера клієнтів до бази даних доменних імен.

Контрольні запитання

1. Наведіть типи адрес стека TCP/IP з відповідними прикладами.

2. Поясніть призначення та застосування MAC-адрес. Яка структура MAC-адрес? Наведіть кілька прикладів MAC-адрес.

3. Поясніть призначення та застосування IP-адрес. Наведіть кілька прикладів IP-адрес.

4. Які адреси називають символьними? Наведіть кілька прикладів таких адрес та поясніть їх призначення.

5. Поясніть, яким чином символьні адреси перетворюються на IP-адреси. Який сенс такого перетворення?

6. Поясніть з якою метою використовується класова схема IP-адресації.

7. Перерахуйте та охарактеризуйте класи IP-адрес.

8. Поясніть, як визначити діапазон номерів мереж класів A, B та C.

9. Поясніть, як визначити максимальну кількість IP-адрес у мережах класів A, B та C.

10. Поясніть, з якою метою використовуються адреси класів D та E. Наведіть діапазони адрес цих класів.

11. Охарактеризуйте особливі IP-адреси та поясніть їх призначення. Наведіть кілька відповідних прикладів таких адрес.

12. Поясніть призначення групових IP-адрес. Чим групові адреси відрізнаються від широкомовних?

13. Поясніть з якою метою використовується широкомовне повідом-лення і обмежене широкомовне повідомлення. В чому відмінність між ними?

ПРАКТИЧНА РОБОТА №4

Загальна структура клієнт-серверного додатку, що використовує сокети в блокуючому режимі. Реалізація додатку сервера.

Сервер **сокетів Python**

Створення програми сервера завдяки сокетам python є легким прикладом застосування модуля **socket**. Щоб використовувати з'єднання сокетів python, нам потрібно імпортувати модуль сокетів import **socket** module. Потім, послідовно, нам потрібно виконати встановлення зв'язку між сервером і клієнтом. Ми можемо отримати адресу хоста за допомогою функції socket.gethostname() .

Рекомендується вказати адресу порту користувача вище значення 1024, оскільки номери портів менше 1024 зарезервовані для стандартного Інтернет-протоколу. Приклад коду сервера сокетів python представлено нижче, коментарі допоможуть вам зрозуміти код сервера з імпорт сокета.

```
import socket

def server_program():
    # get the hostname
    host = socket.gethostname()
    port = 5000  # initiate port no above 1024
    server_socket = socket.socket()  # get instance
    # look closely. The bind() function takes tuple as argument
    server_socket.bind((host, port))  # bind host address and port together
    # configure how many client the server can listen simultaneously
    server_socket.listen(2)
    conn, address = server_socket.accept()  # accept new connection
    print("Connection from: " + str(address))
    while True:
        # receive data stream. it won't accept data packet greater than 1024 bytes
```

```
            data = conn.recv(1024).decode()
            if not data:
                # if data is not received break
                break
            print("from connected user: " + str(data))
            data = input(' -> ')
            conn.send(data.encode())  # send data to the client
        conn.close()  # close the connection
    if __name__ == '__main__':
        server_program()
```

Сервер сокетів python працює на порту 5000 і буде чекати запиту клієнта. Якщо ви хочете, щоб сервер не вимикався під час закриття клієнтського з'єднання, просто видаліть умову if та оператор break. Цикл Python while використовується для невизначеного запуску серверної програми та продовження очікування запиту Клієнта.

Клієнт сокета Python

Клієнтську програму з python socket схожа на серверну програму, за винятком прив'язки (**bind**). Основна відмінність між серверною та клієнтською програмами полягає в тому, що в серверній програмі потрібно зв'язати адресу хоста та адресу порту разом. Приклад коду клієнта сокета python представлено нижче, коментар допоможе вам зрозуміти код.

```
    import socket
    def client_program():
        host = socket.gethostname()  # as both code is running on same pc
        port = 5000  # socket server port number
        client_socket = socket.socket()  # instantiate
        client_socket.connect((host, port))  # connect to the server
        message = input(" -> ")  # take input
        while message.lower().strip() != 'bye':
```

```
        client_socket.send(message.encode())  # send message
        data = client_socket.recv(1024).decode()  # receive response
        print('Received from server: ' + data)  # show in terminal
        message = input(" -> ")  # again take input
    client_socket.close()  # close the connection
if __name__ == '__main__':
    client_program()
```

Відображення роботи програм з сокет Python

Щоб побачити результат, спочатку запустіть серверну програму socket_server.py. Потім запустіть клієнтську програму socket_client.py. Після цього напишіть щось із клієнтської програми. Потім знову напишіть відповідь із серверної програми. Нарешті, напишіть **bye** з клієнтської програми, щоб завершити обидві програми. Наведене знимок екрану, яке відображує відпрацювання клієнтської та серверної частини програм з використанням socket.

```
pankaj$ python3.6 socket_server.py
Connection from: ('127.0.0.1', 57822)
from connected user: Hi
 -> Hello
from connected user: How are you?
 -> Good
from connected user: Awesome!
 -> Ok then, bye!
pankaj$
pankaj$ python3.6 socket_client.py
 -> Hi
Received from server: Hello
 -> How are you?
Received from server: Good
 -> Awesome!
```

Received from server: Ok then, bye!

-> Bye

pankaj$

Завдання:

Реалізувати функціонал консольного варіанту сервера чату.

А. Старт сервера

1) Створити сокет

2) Задати IP адресу та порт сервера чату. Порт задається вашим номером у списку групи+7000. Тобто, якщо ваш номер у групі 3 — то порт буде = 3+7000=7003.

3) Запустити сервер на прослуховування

Б. Встановлення з'єднання з клієнтом

Виконати accept()

Отримати дані про клієнта (IP & PORT) та відобразити їх в консолі сервера

Відправити запит клієнту на отримання «nickname»

Отримати «nickname» клієнта

Перевірити входження клієнта в список заблокованих юзерів (файл ban.txt):

Якщо «nickname» входить в список заблокованих клієнтів:

повідомити юзера про його блокування

розірвати з'єднанням

Якщо «nickname» = Admin:

Запитати пароль

Якщо пароль невірний:

повідомити клієнта про відмову в доступі

розірвати з'єднанням

При успішному проходженні пункту 5:

Добавити «nickname» клієнта в список ***nicknames***

Добавити клієнта в список ***clients***

Відобразити в консолі сервера інформацію про клієнта («nickname»)

Відправити клієнту повідомлення про успішне підключення до сервера чату

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА№5

Методи створення клієнтського додатку що використовує з'єднання TCP і UDP та однопотоковий сервер, який обслуговує декілька підключень.

Блокуючий і неблокуючий сокет

У клієнт-серверних додатках, коли клієнт надсилає запит серверу, сервер обробляє запит і надсилає відповідь. Для цього і клієнту, і серверу спочатку потрібно встановити з'єднання один з одним через сокети (TCP або UDP). В останніх кількох підручниках ми також бачили, як клієнт може надсилати дані у формі запиту на сервер, і сервер може працювати з ними, а потім надсилати відповідь клієнту.

Блокуючий сокет.

За замовчуванням TCP-сокети переведені в режим блокування. Це означає, що управління не повертається до вашої програми до завершення будь-якої конкретної операції.

Наприклад, якщо ви викликаєте метод connect (), з'єднання блокує вашу програму до завершення операції. У багатьох випадках ми не хочемо змушувати нашу програму чекати вічно.

В іншому прикладі, коли ми пишемо клієнт веб-браузера, який підключається до веб-сервера, ми повинні розглянути функцію зупинки, яка може скасувати активний процес підключення в середині його роботи. Цього можна досягти, перевівши сокет в неблокуючий режим.

Неблокуючий сокет

Ми можемо викликати setblocking(1) для налаштування блокування або setblocking (0) для скасування блокування. Давайте розберемося з цим на прикладі. Перш за все, давайте розглянемо блокуючий сокет:

block_client.py

```
#!usr/bin/python
import socket
sock = socket.socket()
host = socket.gethostname()
sock.connect((host, 12345))
sock.setblocking(1)
# Or simply omit this line as by default TCP sockets
# are in blocking mode
data = "Hello Python\n" *10*1024*1024    # Huge amount of data to be sent
assert sock.send(data)                   # Send data till true
```

block_server.py

```
#!usr/bin/python
#block_server.py
import socket
s = socket.socket()
host = socket.gethostname()
port = 12345
s.bind((host,port))
s.listen(5)
while True:
    conn, addr = s.accept()          # accept the connection

    data = conn.recv(1024)
    while data:                      # till data is coming
        print data
        data = conn.recv(1024)
    print "All Data Received"        # Will execute when all data is received
    conn.close()
    break
```

Тепер запустіть спочатку block_server.py, а потім block_client.py. ви помітите, що сервер продовжує друкувати Hello Python. Це триватиме до тих пір, поки не будуть відправлені всі дані. У наведеному вище коді рядок All Data Received не буде виводитися протягом тривалого часу, тому що клієнту необхідно

відправити велику кількість рядків, що зажадає часу, а до тих пір введення-виведення сокета буде заблокований.

Метод send () спробує передати всі дані на сервер, тоді як буфер запису буде заповнений. Ядро переведе процес у режим очікування, поки дані з буфера не будуть передані за призначенням і буфер знову не спорожниться. Коли буфер порожній, ядро знову запустить процес, щоб отримати наступний фрагмент даних, який потрібно передати. Коротше кажучи, ваш код заблокується і не дозволить продовжувати щось інше.

Неблокуюча socket

```
#!usr/bin/python
# non_blocking_client.py
import socket
sock = socket.socket()
host = socket.gethostname()
sock.connect((host, 12345))
sock.setblocking(0)                    # Now setting to non-blocking mode
data = "Hello Python\n" *10*1024*1024    # Huge amount of data to be sent
assert sock.send(data)                 # Send data till true
```

Тепер, якщо ми запустимо non_blocking_client.py, ви помітите, що програма буде працювати деякий час, вона надрукує останній рядок "всі дані отримані" і незабаром завершить роботу.

Клієнт відправив не всі дані. Коли ми робимо сокет неблокіруемим, викликаючи setblocking(0), він ніколи не буде чекати завершення операції. Отже, коли ми викликаємо метод send (), він поміщає якомога більше даних у буфер і повертає результат.

Завдання:

Реалізувати функціонал консольного варіанту сервера чату.

А. Старт сервера

1) Створити сокет

2) Задати IP адресу та порт сервера чату. Порт задається вашим номером у списку групи+7000. Тобто, якщо ваш номер у групі 3 — то порт буде = 3+7000=7003.

3) Запустити сервер на прослуховування

Б. Встановлення з'єднання з клієнтом

Виконати accept()

Отримати дані про клієнта (IP & PORT) та відобразити їх в консолі сервера

Відправити запит клієнту на отримання «nickname»

Отримати «nickname» клієнта

Перевірити входження клієнта в список заблокованих юзерів (файл ban.txt):

Якщо «nickname» входить в список заблокованих клієнтів:

повідомити юзера про його блокування

розірвати з'єднанням

Якщо «nickname» = Admin:

Запитати пароль

Якщо пароль невірний:

повідомити клієнта про відмову в доступі

розірвати з'єднанням

При успішному проходженні пункту 5:

Добавити «nickname» клієнта в список ***nicknames***

Добавити клієнта в список ***clients***

Відобразити в консолі сервера інформацію про клієнта («nickname»)

Відправити клієнту повідомлення про успішне підключення до сервера чату

Відправити всім клієнтам чату про приєднання нового клієнта «nickname» до чату

Завдання:

Представити код клієнта та сервера в блокуючем та неблокючем режиме з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА №6

Особливості використання сокетів в неблокуючому режимі. Багатопотокові сервери

Інтерфейс сокетів Берклі дозволяє взаємодію між хостами чи між процесами на одному комп'ютері, використовуючи концепцію Інтернет-сокетів. Дана технологія може працювати з багатьма драйверами та пристроями вводу/виводу, хоча їхня підтримка залежить від конкретної операційної системи. Реалізація інтерфейсу достатня для підтримки TCP/IP, саме тому це одна із основних технологій, на якій заснований Інтернет. Розробка технології була започаткована у Каліфорнійському університеті Берклі для застосування на ОС Unix. Всі сучасні операційні системи мають реалізацію інтерфейсу сокетів Берклі, оскільки вони є стандартним інтерфейсом для підключення до мережі Інтернет.

У модулі socket функція setlocking відповідає за блокування сокета поки обслуговується з'єднення. Без явної вказівки в модулі socket застосовується setlocking(TRUE) — блокування включено. Щоб блокування зняти потрібно явно вказати - написати в програмі setlocking(FALSE).

1. subprocess – модуль міжпроцесовою взаємодії.

Міжпроцесова взаємодія (англ. IPC) – це обмін даними між процесами. Як правило реалізується засобами ОС. До методів IPC належать: файли, неіменовані і іменовані канали, черги повідомлень, сигнали, спільна пам'ять, сокети і файли, що відображаються у пам'ять. У прикладі створюється канал між стандартними потоками введення/виведення/помилок (stdin/stdout/stderr) процесів. Цей модуль створює новий процес server.py, відсилає йому дані на stdin та отримує дані з його stdout. Приклад програми наведено нижче.

```
import subprocess, pickle
data=['A','B','C'] # дані
s = pickle.dumps(data) # серіалізувати список у рядок
```

```
s=s.encode("string_escape") # перетворити в рядковий літерал Python (без "\n")
p=subprocess.Popen(["python", "server.py"],stdin = subprocess.PIPE,
stdout= subprocess.PIPE, stderr= subprocess.PIPE) # створити процес
stdout, stderr = p.communicate(input=s) # надіслати дані в stdin, отримати дані з stdout, чекати завершення процесу
s=stdout.decode("string_escape") # перетворити з рядкового літералу Python
print pickle.loads(s) # перетворити в список
```

Результат роботи програми:

['A', 'B', 'C', 'D']

Приклад, server.py – модуль сервера

Модуль отримує дані від клієнта через stdin та відсилає їх назад через stdout.

```
import pickle,sys
# ! тут заборонено використовувати print
s=sys.stdin.read().decode("string_escape") # перетворити з рядкового літералу Python
data = pickle.loads(s) # перетворити в список
data.append('D') # додати дані
s=pickle.dumps(data) # серіалізувати список у рядок
s=s.encode("string_escape") # перетворити в рядковий літерал Python (без "\n")
sys.stdout.write(s) # записати в stdout
```

2. multiprocessing – модуль підтримки багатьох процесів

multiprocessing – це пакет, який підтримує створення процесів із використанням API, який подібний на API модуля threading. Забезпечує локальний і віддалений паралелізм, ефективно долає GIL шляхом використання підпроцесів замість потоків. У прикладі розпаралелюється задача знаходження квадратів 50 матриць 1000x1000 за допомогою класу Poll і його методу map. Зауважте, що виграш у продуктивності буде досягнуто тільки на

багатопроцесорній системі. Приклад виконання програми послідовно в паралельному і звичайному режимах так:

python main.py

python main.py s

Для визначення продуктивності програми використано модуль timeit. У Windows можна також використати команду:

echo %time% & main.py & call echo %^time%

```
import numpy as np
from multiprocessing import Pool # задіяти багато процесів
#from multiprocessing.dummy import Pool # або задіяти багато потоків
import sys, timeit
def f(x): # функція, яка виконується в кожному процесі
   return x*x
if __name__ == '__main__':
   time = timeit.default_timer()
   X=[np.matrix(np.random.rand(1000,1000)) for i in range(50)] # 50 матриць 1000x1000
   if 's' in sys.argv:
      Y=map(f,X) # застосувати f для кожного у X (тільки 1 процес)
   else:
      p=Pool(4) # створити пул 4-х процесів
      Y=p.map(f, X) # задіяти 4 процеси
   print timeit.default_timer() - time
```

Результат роботи програми.

9.77497753973

15.1325679658

multiprocessing – запуск паралельних задач.

Аналог прикладу concurrent.futures на основі multiprocessing. Тут функціям ProcessPoolExecutor, submit, result, map відповідають Pool, apply_async, get, map.

```
import time
from multiprocessing import Pool
def f(x): # функція, яка буде виконуватись в окремих процесах
  time.sleep(x) # затримка (тільки для тестування паралельності)
  return x
if __name__ == '__main__':
  pool = Pool() # пул процесів
  a = pool.apply_async(f, [4]) # AsyncResult
  b = pool.apply_async(f, [2])
  while any([a,b]): # отримати результати асинхронно
    if a and a.ready(): print a.get(); a=False
    if b and b.ready(): print b.get(); b=False
  #print pool.map(f, [1,2]) # або чекати усі результати
```

Результат роботи програми.

2, 4

multiprocessing – міжпроцесова взаємодія

Для обміну об’єктами між процесами можна використовувати черги (Queue), канали (Pipe), спільну пам’ять (Value, Array) [5, 19]. Клас Manager створює об’єкт, що контролює серверний процес, який зберігає об’єкти Python і дозволяє іншим процесам використовувати їх. Використання об’єктів Manager більш гнучке, ніж об’єктів спільної пам’яті, так як вони можуть підтримувати об’єкти довільних типів. Але вони більш повільні. У прикладі за допомогою Manager створюється список, у який базовий і дочірній процес паралельно додають елементи.

```
from multiprocessing import Process, Manager
```

```
def f(L): # функція, що виконується в окремому процесі
    L.append(4)
if __name__ == '__main__':
    manager = Manager() # менеджер
    L = manager.list([1,2]) # спільний для процесів список
    p = Process(target=f, args=(L, )) # новий процес
    p.start() # стартувати процес (робота з L розпаралелюється)
    L.append(3)
    p.join() # приєднати процес
    print L
```

Результат роботи програми.

[1, 2, 3, 4]

3. thread – модуль створення багатьох потоків керування

Потоком виконання називають частину процесу, яка може виконуватись паралельно з іншими потоками цього процесу і використовувати спільні з ними ресурси. Синхронізація потоків і процесів – це механізм, який перешкоджає одночасному їхньому зверненню до спільно використовуваних ресурсів. Модуль thread забезпечує низькорівневі (на відміну від threading) примітиви для роботи з багатьма потоками. У прикладі створюються 4 потоки, які виконують функцію f. Звернення потоків до спільного списку A синхронізовано за допомогою простого об’єкта блокування allocate_lock. Нижче показані результати роботи програми з цим об’єктом і без нього. Зауважте, що в CPython існує глобальне блокування інтерпретатора Global Interpreter Lock (GIL), яке являє собою механізм синхронізації потоків, що не дозволяє в один момент часу виконуватись більше ніж одному потоку. Тому застосовуйте модуль multiprocessing, якщо програмі потрібно задіяти для обчислень кілька процесорів. А багатопотоковість краще застосовувати у випадку багатьох одночасних задач введення-виведення.

```
import thread,time
def f(i): # функція виконується в окремому потоці
  mutex.acquire() # блокувати (лише один потік може виконуватись в один і той самий момент часу)
  A.append(i)
  time.sleep(1)
  A.append(i)
  mutex.release() # розблокувати
  T[i]=1 # повідомити головному потоку, що потік завершився
A=[] # глобальний список
T=[0,0,0,0] # глобальний список (якщо потік `i` завершився, то T[i]=1)
mutex = thread.allocate_lock() # створити блокуючий об'єкт
for i in range(4): # створити 4 потоки
  thread.start_new(f, (i, )) # стартувати потік 'i'
while 0 in T: # поки усі потоки не приєднаються
  pass # тут головний потік може робити щось своє
print A
```

Результат роботи програм.

[0, 0, 1, 1, 2, 2, 3, 3]

[0, 1, 2, 3, 1, 3, 2, 0]

4. threading – високорівневий інтерфейс потоків

Цей модуль створює високорівневі інтерфейси потоків на основі низькорівневого модуля thread [5, 19]. Потоки описуються нащадком класу threading.Thread, а їхня активність – перевизначеним методом run. У прикладі створюються 4 потоки, які виконують код у методі run. Звернення потоків до спільного списку A синхронізовано за допомогою простого об’єкта блокування threading.Lock. Нижче показані результати роботи програми з цим об’єктом і без нього. Додатково створюється потік, який стартує через 2 секунди і додає в список A рядок ‘timer’.

```
        import threading, time
class Thread(threading.Thread): # успадкований від threading.Thread
    def __init__(self, i): # конструктор
        self.i=i # ідентифікатор потоку
        threading.Thread.__init__(self) # виклик конструктора базового класу
    def run(self): # забезпечує логіку потоку
        mutex.acquire() # блокувати (лише один потік може виконуватись в один і той самий момент часу)
        #semaphore.acquire() # або так
        A.append(self.i)
        time.sleep(1)
        A.append(self.i)
        mutex.release() # розблокувати
        #semaphore.release() # або так

mutex = threading.Lock() # те саме що thread.allocate_lock()
#semaphore=threading.Semaphore(1) # або семафор (тільки 1 потік одночасно)
A=[] # глобальний список
T=[] # список потоків
for i in range(4): # створити 4 потоки
    t=Thread(i) # створити потік
    t.start() # виконати метод run в потоці
    T.append(t) # додати в список потоків
t = threading.Timer(2.0, lambda: A.append('timer')) # створити потік,
t.start() # який стартує через 2 с
T.append(t)
for t in T:
```

```
    t.join() # поки усі потоки не приєднаються
print A
```

Результат роботи програми.

[0, 0, 1, 'timer', 1, 2, 2, 3, 3]

[0, 1, 2, 3, 2, 3, 1, 0, 'timer']

Реалізувати функціонал консольного варіанту сервера чату.

Завдання

Б - Процеси

1) Реалізувати сервер чату з використанням процесів, для вирішення задачі обслуговування декількох клієнтів.

1) Запустити сервер та перевірити роботу з одним клієнтом.

2) Перевірити роботу з декількома клієнтами.

3) Виконати аналіз роботи сервера та відобразити у протоколі інформацію про кількість запущених процесів, тощо.

В - Потоки

4) Реалізувати сервер чату з використанням потоків, для вирішення задачі обслуговування декількох клієнтів.

5) Запустити сервер та перевірити роботу з одним клієнтом

6) Перевірити роботу з декількома клієнтами

7) Виконати аналіз роботи сервера та відобразити у протоколі інформацію про кількість запущених процесів, потоків тощо.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА №7

Протокол HTTP. Особливості використання протоколу в клієнт-серверних застосунках.

HTTP протокол (HyperText Transfer Protocol «протокол передачі гіпертексту») – це протокол прикладного рівня гіпермедійних документів, таких як HTML. Протокол був розроблений для зв'язку між веб-браузерами і веб-серверами, але також використовується і для інших цілей. HTTP протокол використовується не тільки для передачі гіпертекстових документів, але і для передачі зображень та відео або для відправки контенту на сервери. Також HTTP використовується, щоб отримати частину документа для оновлення веб-сторінки за вимогою.

Цей протокол використовує класичну модель клієнт-сервер, при якій клієнт (веб-браузер) робить запит, сервер формує відповідь і відправляє її назад клієнту у вигляді інформації, яку ми бачимо в браузері. Також необхідно знати, що клієнт і сервер не зберігають інформацію про попередні запити. Тому кожен запит містить всю необхідну інформацію.

Запити та відповіді: їх структура, формування і методи

Запити та відповіді формуються в суворій послідовності і відповідають певній структурі.

Рисунок 1 – Складові запиту HTTP

Між заголовком і тілом обов'язково повинен бути порожній рядок, який слугує роздільником.

Коли запит надсилається на сервер, сервер відправляє відповідь. Відповідь від сервера повідомляє, чи був запит успішним або ні.

Відповіді складаються з наступних компонентів:

1) код статусу (вказує був запит успішним чи ні);
2) HTTP заголовки;
3) тіло, що містить вилучений ресурс.

HTTP використовує спеціальні методи запиту для виконання різних завдань:

GET – запитує певні дані з бази даних;

HEAD – запитує певний ресурс без змісту тіла;

POST – додає вміст, повідомлення або дані в базу даних;

PUT – замінює запис, що існує, на новий;

DELETE – позбавляється від даних;

TRACE – показує будь-які внесені зміни до веб-ресурсу;

PATCH – частково змінює запис;

OPTIONS – показують, які HTTP-методи доступні для конкретної URL-адреси;

CONNECT – перетворює запитові з'єднання у прозорий TCP/IP тунель.

Метод запиту визначає дію, яку потрібно виконати на ресурсі. Він чутливий до регістру і вказується тільки великими літерами.

Приклад HTTP діалогу

Запит GET /wiki/HTTP HTTP/1.1

Host: uk.wikipedia.org

User-Agent: firefox/5.0 (Linux; Debian 5.0.8; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7

Connection: close

Відповідь HTTP/1.1 200 OK

Server: Apache
Content-Language: uk
Content-Type: text/html; charset=utf-8
Content-Length: 1234
(пустий рядок)
(далі йде текст html-сторінки)

select — модуль ефективного мультиплексування введення/виведення.

Модуль selectors забезпечує високорівневе й ефективне мультиплексування вводу/виводу, побудоване на основі примітивів модуля select. Замість цього користувачам рекомендується використовувати модуль selectors, якщо вони не бажають точного контролю над використовуваними примітивами рівня ОС.

Приклад застосування select.

```
import select
import socket
import time
PORT = 8037
TIME1970 = 2208988800L
service = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
service.bind(("", PORT))
service.listen(1)
print "listening on port", PORT
while 1:
   is_readable = [service,input]
   is_writable = []
   is_error = []
   r, w, e = select.select(is_readable, is_writable, is_error, 1.0)
   if r:
      channel, info = service.accept()
      print "connection from", info
      t = int(time.time()) + TIME1970
      t = chr(t>>24&255) + chr(t>>16&255) + chr(t>>8&255) + chr(t&255)
      channel.send(t) # send timestamp
      channel.close() # disconnect
   else:
      print "still waiting"
```

listening on port 8037 ...

Серверні обьєкти asyncio.Server – це асінхроні діспетчери контексту. При використанні async with гарантується, что коли інструкція async with завершена, то обьєкт Server закрит та не приймає нові з'єдення:

```
srv = await loop.create_server(...)
async with srv:
    # якісь код
# На цьому етапі закрити `srv` та більше на приймати підключення.
```

Метод Server.serve_forever() починає приймати підключенння, пока сопрограма не буде відмінена. Відміна завдання serve_forever приведе до закриття сервера. Цей метод може визвати, якщо сервер вже приймає підключення. На один обьект Server можно задіяти лише одне завдання Server.serve_forever().

Приклад наведен нижче.

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()
async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()
asyncio.run(main('127.0.0.1', 0))
```

Завдання:

Створити http клієнт-серверний додаток з використанням сокетов та асінхроності.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА №8

Протоколи поширених сервісів Internet. Можливості використання сервісів в клієнт-серверних застосунках

FTP (англ. *File Transfer Protocol*, укр. *протокол передавання файлів*) — стандартний мережевий протокол прикладного рівня, призначений для пересилання файлів між клієнтом та сервером в комп'ютерній мережі.

Клієнт та сервер створюють окремі канали для передачі даних та обміну командами. Можлива автентифікація клієнтів із використанням відкритого тексту, зазвичай це ім'я користувача (логін) та пароль. Також сервер може бути налаштований для роботи без автентифікації користувачів (так звані «анонімні сеанси»).

Для захисту даних (а також процесу автентифікації) використовують побудований на основі SSL/TLS варіант FTPS, або розширення протоколу SSH — *SSH File Transfer Protocol* (SFTP).

Типи даних FTP

FTP має 4 типи даних для файлів:

- ASCII — файл у форматі ASCII. Символи нового рядка конвертуються при передаванні файлів між різними системами.
- EBCDIC — аналогічно до попереднього, але в іншому кодуванні
- IMAGE — передача бінарних файлів, не змінюючи байти
- Local — для файлів в яких байти не є октетами

FTP-клієнт був інтегрований у основні веб-браузери , де перегляд файлових серверів здійснюється за допомогою префікса ftp://.

У Python завдякі модулю ftplib реалізовано взаємодія за протоколом FTP, який описан у стандарті RFC 959. Запити за FTP протоколом та HTTP протоколом також можливо надсилати за допомогою urllib.request. Дані , які будуть прийматися або відправлятися за звичай кодують у UTF-8, який описан у стандарті RFC 2640.

Модулі ftplib та urllib не присутні у WebAssembly на платформах wasm32-emscripten and wasm32-wasi.

Приклад задіювання модуля **ftplib** та підключення к FTP:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')  # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST             # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

У класі ftplib та функції FTP можливо задавати слідуючи аргументи - *host, user, passwd, acct, timeout, source_address, encoding* :

class ftplib.**FTP**(*host='', user='', passwd='', acct='', timeout=None, source_address=None, *, encoding='utf-8'*)

Нижче наведен приклад завантаження файлу на FTP сервер.

```
# Import Module
import ftplib
```

```
        # Fill Required Information
        HOSTNAME = "ftp.dlptest.com"
        USERNAME = "dlpuser@dlptest.com"
        PASSWORD = "eUj8GeW55SvYaswqUyDSm5v6N"
        # Connect FTP Server
        ftp_server = ftplib.FTP(HOSTNAME, USERNAME, PASSWORD)
        # force UTF-8 encoding
        ftp_server.encoding = "utf-8"
        # Enter File Name with Extension
        filename = "File Name"
        # Read file in binary mode
        with open(filename, "rb") as file:
          # Command for Uploading the file "STOR filename"
          ftp_server.storbinary(f"STOR {filename}", file)
        # Get list of files
        ftp_server.dir()
        # Close the Connection
        ftp_server.quit()
```

Завдання:

А - Скачування файлів з FTP серверу

1) Підключитися до FTP серверу: **ftp.ubuntu.com**
2) Перейти до каталогу *ubuntu/dists/*
3) Отримати зміст каталогу *ubuntu/dists/* та зберегти його в файл на локальний диск: . *./downloads/ubuntudists.txt*

4) Завантажити на локальний диск файл маніфесту (*MANIFEST)* для всіх доступних оновлень *(bionic-updates, trusty-updates, xenial-updates, ...)*

Наприклад, для **trusty-updates** повний шлях до файлу MANIFEST: ubuntu/dists/**trusty-updates**/main/mstaller-i386/20101020ubuntu318.46/images/ **MANIFEST**

Зміст trusty-updates MANIFEST:

■ MANIFEST - Блокнот

Файл Правка Формат Вид Справка

|cdrom-xen/cdroiii/xen/initrd,gz

cdrom-xen/cdrom/xen/vmlinuz

cdrom-xen/cdrom/xen/xm-debian.i

cdrom/debian-cd_info.tar,gz

cdrom/initrd.gz

cdrom/vmlinuz

hd-media/boot.img.gz

hd-media/initrd.gz

hd-media/wmlinuz

netboot/boot.img.g2

initrd for installing under xen

kernel image for installing under Xen

example Xen configuration isolinux config files for CD

 initrd for use with isolinux to build a CD kernel for use with isolinux to build a CD

800 mb image (compressed) for USB memory stick for use on USB memory sticks

for use on USB memory sticks

compressed network install image for USB memory stick

Б - Завантаження файлів на FTP сервер Бекап файлів звіту

На локальному диску розміщено каталог *ZVIT* з набором звітних файлі у форматі txt, docx, pdf. Розробити програмний додаток для автоматизації створення бекапу звітів за поточну добу. Бекап зберігається на FTP сервері в каталозі /data/arc/

1) Виконати аналіз каталогу *ZVIT* на наявність звітних матеріалів за поточну добу

2) Створити в каталозі *FTP ARC* файл, що містить назви звітних файлів за поточну добу та їх хеш код, формат назви файлу *yyyy_mm_dd_hh_mm_ss*

3) Підключитися до FTP сервера

4) Перейти на FTP сервері до каталогу */data/arc/*

5) Створити в /data/arc/ каталог для бекапу звітних файлів за поточну добу, назва каталогу поточна дата у форматі *yyyy_mm_dd*

6) Завантажити на FTP сервер всі файли з каталогу *ZVIT* у яких поточна дата створення/оновлення

7) Розірвати з’єднання

Примітка. При неможливості використання безкоштовного FTP сервера використати умовні дані у якості параметрів FTP, та пояснити роботу програмного додатку без демонстрації підключення до FTP сервера.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА №9

Протоколи почтових сервісів SMTP, POP3 та IMAP

Електронною поштою можна надсилати не тільки письмові повідомлення, але й документи, графіку, аудіо-, відеофайли, програми, тощо. Електронна пошта дуже корисна, якщо немає повноцінного доступу до Інтернету. Через електронну пошту можна отримати послуги інших сервісів мережі. Це найуніверсальніша

послуга Інтернет ще й досі є найпопулярнішою у всіх сферах діяльності користувачів.

В Інтернеті для роботи з електронною поштою використовуються протоколи SMTP, POP та IMAP.

Протокол SMTP (Simple Mail Transfer Protocol -простий протокол передачі пошти) підтримує передачу повідомлень між різними вузловими комп`ютерами Інтернету згідно зі стандартами RFC 821, RFC 822, RFC 2821, RFC 5321. Маючи механізми проміжного зберігання пошти, протокол SMTP допускає використання різних транспортних служб та поштових серверів. Він може працювати навіть в мережах які не підтримують протокол TCP/IP. Протокол SMTP дозволяє групувати повідомлення, які приходять на адресу одного користувача, а також розсилати копії e-mail повідомлення по різних адресах.

Протокол POP (PostOffice Protocol - протокол поштового офісу) дає кінцевому користувачу доступ до надійшовших на його адресу електронних повідомлень. POP-клієнти при спробі одержання пошти вимагають пароль, що підвищує конфіденційність переписки. На сьогодні актуальна версія протоколу POP 3.

Протокол IMAP (Internet Message Access Protocol - протокол доступу до поштових повідомлень через Інтернет). Програма IMAP-клієнт дає доступ до поштових каталогів на IMAP сервері з будь-якої платформи комп`ютера користувача розташованого будь-де в Інтернеті. Це перспективний новий протокол, основною перевагою протоколу IMAP перед POP - протоколом є можливість використання більше ніж одного комп`ютера для роботи з поштою. Ще одною перевагою є здатність IMAPудо селективного доступу до різних частин листів (наприклад, з поштового сервера на Ваш комп`ютер не буде стягуватися доданий до листа аудіофайл розміром 2 Мб, без Вашої на те згоди). Це дуже важливо для для повільних з`єднань (напр. dialup). Сьогодні використовується версія IMAP4.

Будова адреси електронної пошти в Інтернеті

В загальному випадку адреса електронної пошти складається з трьох частин і виглядає приблизно так: user@computer.site.ua

Де:@ - спеціальний символ притаманний адресам електронної пошти, відділяє ім`я комп`ютера від імені користувача. user - ім`я користувача. Властиво, назва поштової скриньки, яка зареєстрована на поштовому сервері на певну особу чи організацію. computer.site.ua - доменне ім`я комп`ютера (ім`я під яким комп`ютер зареєстрований в Інтернеті) на якому запущено поштовий сервер.

Будова листа електронної пошти

Кожен лист електронної пошти складається з двох частин - заголовків та тіла листа, подібно як звичайний лист складається з конверта та властиво листа.

Заголовки листа, складаються багатьох полів, більшість з яких містять службову інформацію, нецікаву для пересічного користувача. Звичайно поштові програми-клієнти за замовчуванням показують лише декілька основних полів: To:, From:, Date:, Subject:, Cc:, Bcc:, і Attachments:. Поля To:, Subject:, Cc:, і Bcc: можуть бути безпосередньо відредаговані. Для того щоб відправити лист обов`язково заповнити поле To: і бажано поле Subject:. Всі решта поля можна залишити порожніми. Для переміщення курсору між полями використовуйте клавішу Tab або клацніть мишкою на потрібному полі. При введенні інформації в поле використовуйте стандартні засоби редагування тексту, доступні в меню Edit (редагувати). Нижче описані основні поля заголовку листа:

To: (кому). Електронна адреса одержувача або визначений Вами псевдонім (nickname). Можна відправити один лист зразу на декілька адрес, в цьому випадку різні адреси повинні розділятися комами. Якщо лист прийшов Вам то в полі To: повинна бути Ваша адреса. From: (від кого). Електронна адреса відправника, деколи ним може бути не людина, а програма-робот. Date:(дата). Точний час та дата відправлення листа. Subject: (тема). Невеликий текст, що пояснює зміст Вашого листа. Це поле можна залишити порожнім, що, однак, не рекомендується з міркувань етикету. Cc: (копії). E-mail адреса або псевдонім людини, якій буде

відправлена копія Вашого листа. Можна писати декілька адрес через кому. Bcc: (приховані копії). Адресати, перераховані в полях Cc: та Bcc: отримають копію листа. Проте, на відміну від адресатів, перерахованих в Cc:, адресати, які були в полі Bcc: не будуть видимі в заголовкахотриманого листа. Це корисно, тоді коли Ви хочете послати комусь копію листа так, щоб інші одержувачів не знали про це. Можна писати декілька адрес через кому. Attachments: (приєднання). Список файлів, що будуть послані разом з листом.

У python за використання smtp протоколу відповідає модуль smtplib. Нижче наведен приклад SMTP клієнта. Зверніть увагу, що заголовки повинні бути включені до введеного повідомлення згідно стандарту RFC 822. Зокрема, адреси "To» і "From" повинні бути включені в заголовки повідомлень явним чином:

```
import smtplib
def prompt(prompt):
    return input(prompt).strip()
fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")
# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line
```

```
print("Message length is", len(msg))
server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Також у роботі потрібно реалізовати SMTP сервер прийому повідомлень. Нижче наведен приклад простої реалізації SMTP сервера.

```
from datetime import datetime
import asyncore
from smtpd import SMTPServer
class EmlServer(SMTPServer):
    no = 0
    def process_message(self, peer, mailfrom, rcpttos, data):
        filename = '%s-%d.eml' % (datetime.now().strftime('%Y%m%d%H%M%S'),
                self.no)
        f = open(filename, 'w')
        f.write(data)
        f.close
        print '%s saved.' % filename
        self.no += 1
def run():
    foo = EmlServer(('localhost', 25), None)
    try:
        asyncore.loop()
    except KeyboardInterrupt:
        pass
if __name__ == '__main__':
        run()
```

Практичне завдання *SMTP*

А - Система розсилки почтових повідомлень (SMTP+SOCKET) Реалізувати:

 сервер відправки електронного листа (SMTP);

 сервер отримання замовлень (файлів) з інформацією про email розсилку (socket);

 клієнт відправки замовлення (файлу) (socket).

Клієнт відправляє на сервер файл, що містить список адресатів та текст електронного листа:

[Recipients:]

user1@pth.ua, user2@pth.ua, user3@pth.ua [Subject:]

Session is soon!

[Body:]

Hello [login]!

We are happy to inform you about the upcoming exam. Complete all assignments and submit all reports.

Regards, your nerves:)

1) Сервер отримання замовлень приймає файл та зберігає на диску в директорію **incoming**

2) Сервер розсилки перевіряє наявність файлів в директорії **incoming,** опрацьовує їх та переміщає до директорії **ARC,** додаючи до імені файлу дату та час розсилки.

3) За результатам розсилки відправити лист на службу підтримки, що містить:

 ім’я вхідного файлу;

 загальну кількість адресатів;

 списком адресатів, яким не вдалося відправити електронний лист.

Б - Система розсилки почтових повідомлень (SMTP+FTP) Реалізувати:

 сервер відправки електронного листа (SMTP);

 FTP сервер (для тестування допускається використання локального сервера, наприклад, OpenServer)

 клієнт відправки замовлення (файлу) на FTP сервер.

Клієнт завантажує на FTP сервер в директорію **incoming** файл, що містить список адресатів та текст електронного листа:

[Recipients:]

user1@pth.ua, user2@pth.ua, user3@pth.ua [Subject:]

Session is soon!

[Body:]

Hello [login]!

We are happy to inform you about the upcoming exam. Complete all assignments and submit all reports.

Regards, your nerves:)

Сервер розсилки перевіряє наявність файлів в директорії **incoming,** опрацьовує їх та переміщає до директорії **ARC,** додаючи до імені файлу дату та час розсилки.

За результатам розсилки відправити лист на службу підтримки, що містить:

 загальну кількість адресатів;

 списком адресатів, яким не вдалося відправити електронний лист;

 вхідний файл (прикріплення файлу до поштового листа).

Примітка. У якості відправника та отримувача допустимо використання одної електронної адреси.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ПРАКТИЧНА РОБОТА №10

Багатоадресна розсилка. Особливості використання в клієнт-серверних застосунках

Багатоадресна розсилка

Багатоадресна передача за своєю суттю не є механізмом, орієнтованим на з'єднання, тому такі протоколи, як TCP, який дозволяє повторно передавати пропущені пакети, не підходять. Для таких програм, як потокове передавання аудіо та відео, випадкові відкинуті пакети не є проблемою. Але для поширення важливих даних потрібен механізм запиту повторної передачі.

Однією з таких схем, запропонованою Cisco, є PGM (спочатку досить хороша багатоадресна передача, але змінена з міркувань торгової марки на Pragmatic General Multicast), [потрібне цитування] задокументовано в RFC 3208. У цій схемі багатоадресні пакети мають порядкові номери, і коли пакет пропущений, одержувач може вимагати повторної багатоадресної передачі пакета з іншими членами багатоадресної групи, ігноруючи дані заміни, якщо вони не потрібні. Розширена версія, PGM-CC, намагалася зробити багатоадресну IP-адресу більш "дружньою до TCP", зменшивши пропускну здатність всієї групи до пропускної здатності, доступної гіршому приймачу.

Дві інші схеми, задокументовані робочою групою з інженерного забезпечення Інтернету (IETF): протокол, заснований на стандартах, Nack-орієнтована надійна багатоадресна передача (NORM), задокументований у RFC 5740 та RFC 5401, та протокол доставки файлів односпрямованим транспортом (FLUTE), задокументований у RFC 6726. Для них існують реалізації з відкритим кодом, на додаток до власних. Існують інші подібні протоколи, такі як масштабована надійна багатоадресна передача, і вони визначаються багатьма джерелами. Такі протоколи відрізняються за допомогою засобів виявлення помилок, механізмів, що використовуються для відновлення помилок, масштабованості такого відновлення та основоположних ідей щодо того, що

означає бути надійним. Список надійних протоколів багатоадресної передачі, представлений на семінарі ACM SIGCOMM Multicast Workshop, 27 серпня 1996 р., документує ряд підходів до проблеми.

Незалежні групи, такі як ініціатива стандартизації багатоадресної передачі Інтернет-протоколу (IPMSI), заявили, що відсутність дійсно масштабованого безпечного протоколу багатоадресної передачі IP, подібного до запропонованого Secure Multicast for Advanced Repeating of Television (SMART), перешкоджала впровадженню багатоадресної передачі IP у міждоменній маршрутизації. Відсутність широко розповсюдженої системи безпеки рівня AES та масштабованої надійності заважали ЗМІ транслювати спортивні події (наприклад, Суперкубок) та / або останні новини в загальнодоступному Інтернеті.[потрібне цитування]

Надійні протоколи IP Multicasting, такі як PGM і SMART, є експериментальними; єдиним протоколом, орієнтованим на стандарти, є NORM (редакція RFC 3941, орієнтована на стандарти, вказана в RFC 5401, редакція RFC 3940, орієнтована на стандарти, вказана в RFC 5740).

Протоколи на основі багатоадресної розсилки

Оскільки багатоадресна розсилка відрізняється від одноадресної, з багатоадресною розсилкою можна розумно використовувати лише протоколи, розроблені для багатоадресної розсилки. Більшість існуючих прикладних протоколів, що використовують багатоадресну передачу, працюють поверх протоколу дейтаграм користувача (UDP).

У багатьох додатках транспортний протокол реального часу (RTP) використовується для передачі мультимедійного вмісту через багатоадресну передачу; Протокол резервування ресурсів (RSVP) може використовуватися для резервування пропускної здатності в мережі, що підтримує багатоадресну передачу. Багатоадресна DNS (mDNS) може використовуватися для вирішення імен доменів або хостів без виділеного DNS-сервера за допомогою багатоадресної передачі.

Розгортання

Багатоадресна IP-розсилка широко використовується на підприємствах, комерційних фондових біржах і в мережах доставки мультимедійного контенту. На підприємствах IP multicast зазвичай використовується для IPTV додатків, таких як пряма трансляція телепрограм і корпоративних зборів по телебаченню.

Багатоадресна розсилка

Для приєднання до групи багатоадресної розсилки Python використовує власний інтерфейс сокета операційної системи. Завдяки портативності та стабільності середовища Python багато параметрів сокета безпосередньо перенаправляються до власного виклику socket setsockopt. Режим багатоадресної передачі, такий як приєднання до групи та видалення членства в ній, можна виконати лише за допомогою setsockopt.

Базова програма для прийому багатоадресного IP-пакету може виглядати наступним чином:

```
from socket import *
multicast_port  = 55555
multicast_group = "224.1.1.1"
interface_ip    = "10.11.1.43"
s = socket(AF_INET, SOCK_DGRAM )
s.bind(("", multicast_port ))
mreq = inet_aton(multicast_group) + inet_aton(interface_ip)
s.setsockopt(IPPROTO_IP, IP_ADD_MEMBERSHIP, str(mreq))
while 1:
   print s.recv(1500)
```

Спочатку створює сокет, прив'язує його і запускає приєднання до групи багатоадресної розсилки шляхом видачisetsockopt. В самому кінці він отримує пакети назавжди.

Надсилання багатоадресних IP-кадрів просте. Якщо у вашій системі є один мережевий адаптер, надсилання таких пакетів не відрізняється від звичайного надсилання кадрів UDP. Все, про що вам потрібно подбати, це просто встановити правильну IP-адресу призначення в методі sendto().

Python перенаправляє виклик методу setsockopt на власний інтерфейс сокета C. документація сокета Linux (див man 7 ip) представляє дві форми ip_mreqn структури для параметра IP_ADD_MEMBERSHIP. Найкоротша форма має довжину 8 байт, а довша - 12 байт. Наведений вище приклад генерує 8-байтовий setsockopt дзвінок, де перші чотири байти визначають multicast_group, а другі чотири байти визначають interface_ip.

Широкомовна передача

Широкомовна адреса-умовний (не присвоєний ніякому пристрою в мережі) адреса, який використовується для передачі широкомовних пакетів в комп'ютерних мережах.

Це єдина адреса, яка може використовуватися для зв'язку з усіма хостами мережі:

>>> net.network_address

IPv4Address('192.4.2.0')

Найчастіше ви будете стикатися з довжиною префікса кратної 8.

Prefix Length	Number of Addresses	Netmask
8	16,777,216	255.0.0.0
16	65,536	255.255.0.0
24	256	255.255.255.0
32	1	255.255.255.255

Рисунок 1 – Довжина профіксу

Будь-яке ціле число від 0 до 32 є допустимим, але такий варіант зустрічається рідше:

>>> net = IPv4Network("100.64.0.0/10")

>>> net.num_addresses

4194304

>>> net.netmask

IPv4Address('255.192.0.0')

Поширений спосіб розбиття на підмережі-це збільшення довжини префікса на 1

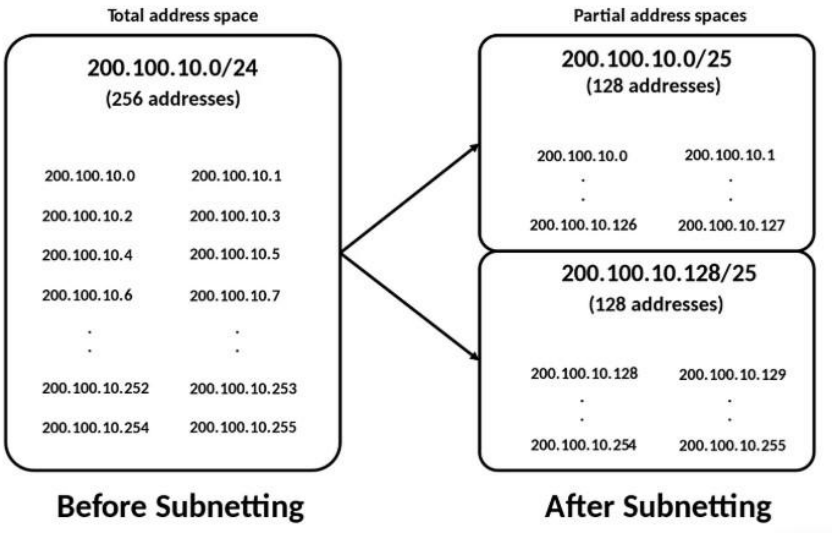

Рисунок 2 – Розбиття на подмережі

На щастя, IPv4 Network розрахунки підмереж підтримуються вбудованим методом subnets():

>>> for sn in net.subnets():

... print(sn)

200.100.10.0/25

200.100.10.128/25

У переданому subnets() аргументі можна задати, яким повинен бути новий префікс:

>>> for sn in net.subnets(new_prefix=28):

... print(sn)

200.100.10.0/28

200.100.10.16/28

200.100.10.32/28

...

200.100.10.208/28

200.100.10.224/28

200.100.10.240/28

Приклад коду широкомовної передачі серва та клієнта вказан в лабораторній роботі №3.

Завдання:

Створити клієнт-серверний додаток багатоадресної розсилки.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ЛАБОРАТОРНА РОБОТА №1

Створення програми з реалізацією протоколу HTTP з використанням у клієнт-серверних застосунках

У Python є багато HTTP-клієнтів (бібліотек); найбільш широко використовуваний і простий у робота з requests. Це стандарт де-фактора в наші дні.

Постійні з'єднання

Перший спосіб, який необхідно взяти до уваги, — це постійне підключення до веб-сервера. Постійні з'єднання є стандартом, починаючи з HTTP 1.1, хоча багато програм не використовують їх. Відсутність оптимізації в ньому легко пояснити, якщо ви знаєте, що при використанні запитів у простому режимі (наприклад, з функцією get) з'єднання закривається при отримання відповіді від сервера. Щоб уникнути цього, програмі потрібно використовувати Об'єкт Session, який дозволяє повторно використовувати вже відкрите з'єднання.

Використання сеансу (Session) із запитами

import requests

```
session = requests.Session()
session.get("http://example.com")
# Connection is re-used
session.get("http://example.com")
```

Кожне з'єднання зберігається в Пулі з'єднань (за замовчуванням поміщає 10 з'єднань), розмір пулу також налаштовується:

Зміна розміру пулу

```
 import requests
session = requests.Session()
adapter = requests.adapters.HTTPAdapter(
pool_connections=100,
pool_maxsize=100)
session.mount('http://', adapter)
response = session.get("http://example.org")
```

Повторне використання TCP-з'єднання для надсилання декількох запитів HTTP має ряд переваг у продуктивності:

– Зниження використання процесора і пам'яті (менша кількість одночасно відкритих з'єднань).

– Зменшена затримка при наступних запитах (без TCP-handshaking).

Винятки можуть бути підняті без штрафу закриття TCP-з'єднання. Протокол HTTP також забезпечує конвеєризацію (pipelining), яка дозволяє надсилати кілька запитів по одному з'єднанню, не чекаючи отримання відповідей (думаю, пакет). На жаль, це не підтримується бібліотекою requests. Однак конвеєризація запитів може бути не такою швидкою, як їх паралельне надсилання. Так як, протокол HTTP 1.1 змушує відправляти відповіді в тому ж порядку, в якому були відправлені запити — першим прийшов — першим вийшов.

Паралелізм

Requests також мають один істотний недолік: ця бібліотека синхронна. Виклик request.get («http://example.org") блокує програму, поки сервер HTTP не відповість повністю. Недоліком може бути те, що програма під час запиту очікує відповіді і нічого не робить. Цілком можливо, що програма могла б робити щось інше, а не сидіти.

Розумний додаток може пом'якшити цю проблему, використовуючи пул потоків, подібних до тих, що надаються concurrent.futures. Це дозволяє дуже швидко розпаралелювати HTTP-запити.

Використання futures з requests

```
from concurrent import futures
import requests
with futures.ThreadPoolExecutor(max_workers=4) as executor:
futures = [
executor.submit(
lambda: requests.get("http://example.org"))
for _ in range(8)
]
results = [
f.result().status_code
for f in futures
]
print("Results: %s" % results)
```

Цей шаблон досить корисний, він був упакований у бібліотеку requests-futures. За допомогою його можна легко використовувати об'єкт Session:

```
from requests_futures import sessions
session = sessions.FuturesSession()
futures = [
session.get("http://example.org")
```

```
for _ in range(8)
]
results = [
f.result().status_code
for f in futures
]
print("Results: %s" % results)
```

За замовчуванням створюється worker з двома потоками, але програма може легко налаштувати це значення, передавши аргумент max_workers або навіть власного виконавця об'єкту FuturSession — наприклад, так: FuturesSession (executor = ThreadPoolExecutor (max_workers = 10)).

Асинхронність

Як пояснювалося раніше, requests повністю Синхронний. Він блокує додаток в очікуванні відповіді сервера, сповільнюючи роботу програми. Створення HTTP-запитів у потоках є одним із рішень, але потоки мають власні накладні витрати, і це передбачає паралельність, яку не завжди всі раді бачити в програмі.

Починаючи з версії 3.5, Python пропонує асинхронність всередині свого ядра за допомогою aiohttp. Бібліотека aiohttp надає асинхронний HTTP-клієнт, побудований поверх asyncio. Ця бібліотека дозволяє надсилати запити послідовно, але не чекаючи першої відповіді, перш ніж надсилати нову. На відміну від конвеєра HTTP, aiohttp надсилає запити через кілька з'єднань паралельно, уникаючи проблеми, описаної раніше.

Використання aiohttp

```
import aiohttp
import asyncio
async def get(url):
async with aiohttp.ClientSession() as session:
async with session.get(url) as response:
```

```
return response
loop = asyncio.get_event_loop()
coroutines = [get("http://example.com") for _ in range(8)]
results = loop.run_until_complete(asyncio.gather(*coroutines))
print("Results: %s" % results)
```

Усі ці рішення (за допомогою Session, thread, futures або async io) пропонують різні підходи до прискорення роботи HTTP-клієнтів.

Продуктивність

Нижче наведено фрагмент клієнта HTTP, який надсилає запити на httpbin.org, HTTP-API, який забезпечує (серед іншого) кінцеву точку, що імітує довгий запит. Цей приклад реалізує всі методи, перераховані вище.

Програма для порівняння продуктивності використання різних запитів

```
import contextlib
import time
import aiohttp
import asyncio
import requests
from requests_futures import sessions
URL = "http://httpbin.org/delay/1"
TRIES = 10
@contextlib.contextmanager
def report_time(test):
t0 = time.time()
yield
print("Time needed for `%s' called: %.2fs"
% (test, time.time() - t0))
with report_time("serialized"):
for i in range(TRIES):
requests.get(URL)
```

```
session = requests.Session()
with report_time("Session"):
for i in range(TRIES):
session.get(URL)
session = sessions.FuturesSession(max_workers=2)
with report_time("FuturesSession w/ 2 workers"):
futures = [session.get(URL)
for i in range(TRIES)]
for f in futures:
f.result()
session = sessions.FuturesSession(max_workers=TRIES)
with report_time("FuturesSession w/ max workers"):
futures = [session.get(URL)
for i in range(TRIES)]
for f in futures:
f.result()
async def get(url):
async with aiohttp.ClientSession() as session:
async with session.get(url) as response:
await response.read()
loop = asyncio.get_event_loop()
with report_time("aiohttp"):
loop.run_until_complete(
asyncio.gather(*[get(URL)
for i in range(TRIES)]))
```

Запуск цієї програми дає наступний висновок:

```
Time needed for `serialized' called: 12.12s
Time needed for `Session' called: 11.22s
Time needed for `FuturesSession w/ 2 workers' called: 5.65s
```

Time needed for `FuturesSession w/ max workers' called: 1.25s

Time needed for `aiohttp' called: 1.19s

Не дивно, що повільніший результат приходить із серіалізованою версією, оскільки всі запити виконуються один за одним без повторного використання з'єднання — 12 секунд на 10 запитів.

Використання Об'єкта Session і, отже, повторне використання з'єднання означає економію 8% часу, що вже є великим і легким виграшем. Як мінімум, ви завжди повинні використовувати Session.

Якщо ваша система та програма дозволяють використовувати потоки, рекомендується використовувати їх для розпаралелювання запитів. Однак потоки мають деякі накладні витрати, і вони не менш вагові. Вони повинні бути створені, запущені і потім приєднані.

Якщо ви не використовуєте старіші версії Python, то, без сумніву, використання aiohttp має бути вашим вибором, якщо ви хочете написати швидкий та асинхронний клієнт HTTP. Це найшвидше та наймасштабніше рішення, оскільки воно може обробляти сотні паралельних запитів.

Потік

Ще одна ефективна оптимізація швидкості-це потокове передавання запитів. При відправці запиту за замовчуванням все тіло відповіді завантажується негайно. Кращий спосіб не завантажувати весь контент в пам'ять відразу при запиті. Для цього є параметра stream, в бібліотеці requests або атрибут content в aiohttp.

Потокове передавання з requests

```
import requests
# Use `with` to make sure the response stream is closed and the connection
can
# be returned back to the pool.
with requests.get('http://example.org', stream=True) as r:
    print(list(r.iter_content()))
```

Потокове передавання з aiohttp

```
import aiohttp
import asyncio
async def get(url):
async with aiohttp.ClientSession() as session:
async with session.get(url) as response:
return await response.content.read()
loop = asyncio.get_event_loop()
tasks = [asyncio.ensure_future(get("http://example.com"))]
loop.run_until_complete(asyncio.wait(tasks))
print("Results: %s" % [task.result() for task in tasks])
```

Не завантажувати повний контент вкрай важливо, щоб уникнути непотрібного виділення сотень мегабайт пам'яті. Якщо вашій програмі не потрібен доступ до всього вмісту в цілому, але вона може працювати з частинами. Наприклад, якщо ви збираєтеся зберегти та записати вміст у файл, читання лише шматка та одночасне записування буде набагато ефективнішим, ніж читання всього тіла HTTP , виділяючи величезну купу пам'яті, і лише після цього записати його на диск.

Завдання:

Створити http клієнт-серверний додаток з використанням асінхроної обробки та потоків.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ЛАБОРАТОРНА РОБОТА №2

Створення програми з реалізацією протоколів почтових сервісів SMTP, POP3 та IMAP.

TCP – "гарантований" протокол з попереднім встановленням з'єднання ("рукостисканням").

Протокол:

– дає впевненість в безпомилковості одержуваних даних і дотриманні послідовності відправки;

– дозволяє регулювати навантаження на мережу і зменшувати час очікування даних при передачі на великі відстані.

UDP (англ. User Datagram Protocol-протокол користувацьких дейтаграм) - » негарантований«протокол передачі без встановлення з'єднання (»рукостискання").

Протокол:

– на відміну від TCP не дозволяє упевнитися в доставці повідомлення адресату, а також порядку отримання пакетів;

– зазвичай швидше, ніж TCP і корисний для серверів, що відповідають на невеликі запити від величезного числа клієнтів (онлайн-ігри, потокові Мультимедіа Додатки та ін.).

Протокол TCP / IP має важливу перевагу - апаратну незалежність (не прив'язаний до особливостей мережевого апаратного забезпечення). Використовуючи даний протокол, можна організувати мережу між будь-якими пристроями (зазвичай званих хостами або вузлами).

Електронна пошта (англ. electronic mail, email, e-mail) - технологія і надані нею послуги з пересилання та отримання електронних повідомлень («листів» або «електронних листів») по комп'ютерній мережі.

За складом елементів і принципом роботи Електронна пошта практично повторює систему звичайної (паперової) Пошти, запозичуючи як терміни (пошта, лист, конверт, вкладення, ящик, доставка та ін.), так і характерні особливості —

простоту використання, затримки передачі повідомлень, достатню надійність і в той же час відсутність гарантії доставки.

Перевагами електронної пошти є:

– легко сприймаються і запам'ятовуються людиною адреси виду імя_пользователя@імя_домена (наприклад ,somebody@example.com);

– можливість передачі як простого тексту, так і форматованого, а також довільних файлів;

– незалежність серверів (в загальному випадку вони звертаються один до одного безпосередньо);

– досить висока надійність доставки повідомлення;

– простота використання людиною і програмами.

Недоліками електронної пошти є:

– спам (масові рекламні та вірусні розсилки);

– можливі затримки доставки повідомлення (до декількох діб);

– обмеження на розмір одного повідомлення і на загальний розмір повідомлень в поштовій скриньці (персональні для користувачів).

В системі електронної пошти виділяється ряд компонентів

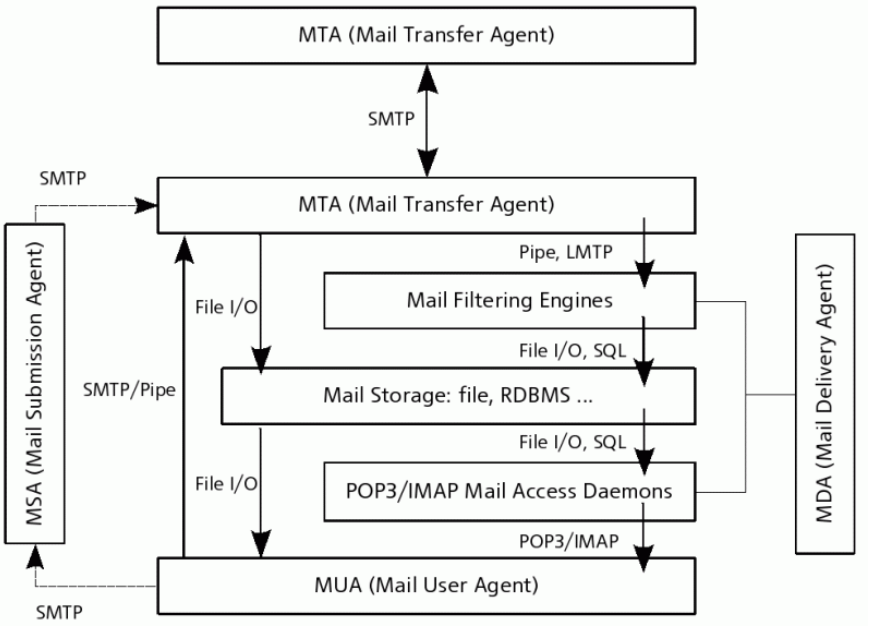

Рисунок 1 – Копоненти електронної пошти

MTA (англ. Mail Transfer Agent-агент пересилання пошти) - відповідає за пересилання пошти між поштовими серверами; як правило, перший MTA в ланцюжку отримує повідомлення від MUA, останній передає повідомлення до MDA;

MDA (англ. Mail Delivery Agent-агент доставки пошти — відповідає за доставку пошти кінцевому користувачеві;

MUA (англ. Mail User Agent-Поштовий агент користувача; в російській нотації закріпився термін поштовий клієнт — програма, що забезпечує користувальницький інтерфейс, що відображає отримані листи і надає можливість відповідати, створювати, перенаправляти листи;

MRA (англ. Mail Retrieve Agent) - поштовий сервер, що забирає пошту з іншого сервера за протоколами, призначеними для MDA.

При передачі по протоколу SMTP електронний лист складається з наступних частин:

Дані SMTP-конверта, отримані сервером (Службова Інформація для сервера).

Повідомлення (DATA): складається з двох частин, розділених порожнім рядком:

– заголовки листа (англ. Headers): Службова Інформація та позначки поштових серверів, через які пройшов лист, позначки про пріоритет, вказівка на адресу та ім'я відправника і одержувача листа, тема листа та інша інформація;

– тіла (англ. Body) листи: безпосередньо повідомлення.

SMTP

class*smtplib.SMTP(*host=''*, *port=0*, *local_hostname=None*,** [timeout,***]***source_address=None*)**¶

Створює SMTP-об'єкт, який інкапсулює роботу з SMTP-протоколом. Деякі параметри:

– host (str) – сервер, на якому працює SMTP-сервер (IP-адреса або доменне ім'я);

– port (int) – порт SMTP-сервера (зазвичай 25).

sendmail(*from_addr, to_addrs, msg, mail_options=[], rcpt_options=[]*)

Виконує відправку пошти.

Деякі параметри:

– from_addr (str) - адреса відправника;

– to_addrs (list – - адреси одержувачів;

– msg-текст повідомлення в спеціальному форматі.

send_message(*msg, from_addr=None, to_addrs=None, mail_options=[], rcpt_options=[]*)

Виконує відправку пошти.

Параметр: msg (email.message.Message) - повідомлення.

Інші параметри аналогічні функції smtplib.SMTP.sendmail().

Приклад роботи з протоколом SMTP представлено у додатку 1.

POP3

***class*poplib.POP3(*host, port=POP3_PORT*[, *timeout*])**

Створює POP3-об'єкт, при створенні ініціалізується з'єднання.

Параметр:

– host (str) – сервер, на якому працює POP3-сервер (IP-адреса або доменне ім'я);

– port (int) - порт сервера POP3 (за замовчуванням 110); timeout-час очікування з'єднання.

***class*poplib.POP3_SSL(*host, port=POP3_PORT*[, *timeout*])**

Створює POP3-об'єкт, використовує шифрування SSL.

Параметр:

– host (str) – сервер, на якому працює POP3-сервер (IP-адреса або доменне ім'я);

– port (int) - порт сервера POP3 (за замовчуванням 995); timeout-час очікування з'єднання.

***exception*poplib.error_proto**

Клас-виняток для будь-яких помилок модуля.

***class*poplib.POP3**

getwelcome()

Повертає привітання сервера.

capa()

Повертає словник, що містить можливості сервера (див.RFC 2449).

user(username)

Надсилає ім'я користувача на сервер. Відповідь повинна містити запит пароля.

pass_(password)

Надсилає пароль на сервер. Відповідь повинна містити кількість повідомлень і розмір поштової скриньки.

Після входу, ящик залишається заблокованим до виклику poplib.POP3.quit().

stat()

Повертає статус поштової скриньки у форматі:

(кортеж: (кількість повідомлень, розмір поштової скриньки)).

list([which])

Запитує та повертає список повідомлень у форматі:

(response, ['mesg_num octets', ...], octets).

Якщо параметр which встановлено, повертається конкретне повідомлення.

retr(which)

Завантажує повідомлення під номером which у форматі:

(response, ['line', ...], octets) і відзначає його прочитаним.

dele(which)¶

Ставить повідомлення під номером which в чергу на видалення. На більшості серверів видалення відбувається під час виходу (poplib.POP3.quit()).

noop()

Не робить нічого. Зазвичай періодично викликається для підтримки з'єднання з сервером для уникнення обриву з'єднання.

quit()

Вихід: застосування змін, розблокування поштової скриньки і закриття з'єднання.

Приклад коду з протоколом POP3 у додатку 1.

IMAP

***class* imaplib.IMAP4(*host=''*, *port=IMAP4_PORT*)¶**

Створює IMAP-об'єкт, при створенні ініціалізується з'єднання. Параметр:

host (str) – сервер, на якому працює IMAP-сервер (IP-адреса або доменне ім'я); port (int) – порт IMAP-сервера (за замовчуванням 143).

***class*imaplib.IMAP4_SSL(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*)¶**

Створює IMAP-об'єкт, використовує шифрування SSL.

Приклад коду до IMAP у додатку 1.

Завдання:

Створити e-mail (виористання протоколів на ваш розсуд та обгрунтування) клієнт-серверний додаток з використанням асінхроної обробки та потоків.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

ЛАБОРАТОРНА РОБОТА №3

Створення програми багатоадресною розсилки з використанням у клієнт-серверних застосунках

Багатоадресна розсилка

Багатоадресна передача за своєю суттю не є механізмом, орієнтованим на з'єднання, тому такі протоколи, як TCP, який дозволяє повторно передавати пропущені пакети, не підходять. Для таких програм, як потокове передавання аудіо та відео, випадкові відкинуті пакети не є проблемою. Але для поширення важливих даних потрібен механізм запиту повторної передачі.

Однією з таких схем, запропонованою Cisco, є PGM (спочатку досить хороша багатоадресна передача, але змінена з міркувань торгової марки на Pragmatic General Multicast), [потрібне цитування] задокументовано в RFC 3208. У цій схемі багатоадресні пакети мають порядкові номери, і коли пакет пропущений, одержувач може вимагати повторної багатоадресної передачі пакета з іншими членами багатоадресної групи, ігноруючи дані заміни, якщо вони не потрібні. Розширена версія, PGM-CC, намагалася зробити багатоадресну IP-адресу більш "дружньою до TCP", зменшивши пропускну здатність всієї групи до пропускної здатності, доступної гіршому приймачу.

Дві інші схеми, задокументовані робочою групою з інженерного забезпечення Інтернету (IETF): протокол, заснований на стандартах, Nack-орієнтована надійна багатоадресна передача (NORM), задокументований у RFC 5740 та RFC 5401, та протокол доставки файлів односпрямованим транспортом (FLUTE), задокументований у RFC 6726. Для них існують реалізації з відкритим кодом, на додаток до власних. Існують інші подібні протоколи, такі як масштабована надійна багатоадресна передача, і вони визначаються багатьма джерелами. Такі протоколи відрізняються за допомогою засобів виявлення помилок, механізмів, що використовуються для відновлення помилок, масштабованості такого відновлення та основоположних ідей щодо того, що

означає бути надійним. Список надійних протоколів багатоадресної передачі, представлений на семінарі ACM SIGCOMM Multicast Workshop, 27 серпня 1996 р., документує ряд підходів до проблеми.

Протоколи на основі багатоадресної розсилки

Оскільки багатоадресна розсилка відрізняється від одноадресної, з багатоадресною розсилкою можна розумно використовувати лише протоколи, розроблені для багатоадресної розсилки. Більшість існуючих прикладних протоколів, що використовують багатоадресну передачу, працюють поверх протоколу дейтаграм користувача (UDP).

Розгортання

Багатоадресна IP-розсилка широко використовується на підприємствах, комерційних фондових біржах і в мережах доставки мультимедійного контенту. На підприємствах IP multicast зазвичай використовується для IPTV додатків, таких як пряма трансляція телепрограм і корпоративних зборів по телебаченню.

Багатоадресна розсилка

Для приєднання до групи багатоадресної розсилки Python використовує власний інтерфейс сокета операційної системи. Завдяки портативності та стабільності середовища Python багато параметрів сокета безпосередньо перенаправляються до власного виклику socket setsockopt. Режим багатоадресної передачі, такий як приєднання до групи та видалення членства в ній, можна виконати лише за допомогою setsockopt.

Базова програма для прийому багатоадресного IP-пакету може виглядати наступним чином:

```
from socket import *
multicast_port  = 55555
multicast_group = "224.1.1.1"
interface_ip    = "10.11.1.43"
s = socket(AF_INET, SOCK_DGRAM )
s.bind(("", multicast_port ))
mreq = inet_aton(multicast_group) + inet_aton(interface_ip)
s.setsockopt(IPPROTO_IP, IP_ADD_MEMBERSHIP, str(mreq))
while 1:
```

print s.recv(1500)

Спочатку він створює сокет, прив'язує його і запускає приєднання до групи багатоадресної розсилки шляхом видачisetsockopt. В самому кінці він отримує пакети назавжди.

Надсилання багатоадресних IP-кадрів просте. Якщо у вашій системі є один мережевий адаптер, надсилання таких пакетів не відрізняється від звичайного надсилання кадрів UDP. Все, про що вам потрібно подбати, це просто встановити правильну IP-адресу призначення в методі sendto().

Python перенаправляє виклик методу setsockopt на власний інтерфейс сокета C. документація сокета Linux (див man 7 ip) представляє дві форми ip_mreqn структури для параметра IP_ADD_MEMBERSHIP. Найкоротша форма має довжину 8 байт, а довша - 12 байт. Наведений вище приклад генерує 8-байтовий setsockopt дзвінок, де перші чотири байти визначають multicast_group, а другі чотири байти визначають interface_ip.

Широкомовна передача

Широкомовна адреса-умовний (не присвоєний ніякому пристрою в мережі) адреса, який використовується для передачі широкомовних пакетів в комп'ютерних мережах.

Це єдина адреса, яка може використовуватися для зв'язку з усіма хостами мережі:

>>> net.network_address

IPv4Address('192.4.2.0')

Найчастіше ви будете стикатися з довжиною префікса кратної 8.

Prefix Length	Number of Addresses	Netmask
8	16,777,216	255.0.0.0
16	65,536	255.255.0.0
24	256	255.255.255.0
32	1	255.255.255.255

Приклад коду широкомовної передачі серва та клієнта:

client.py

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP) # UDP

# Enable port reusage so we will be able to run multiple clients and servers on single (host, port).
# Do not use socket.SO_REUSEADDR except you using linux(kernel<3.9): goto https://stackoverflow.com/questions/14388706/how-do-so-reuseaddr-and-so-reuseport-differ for more information.
# For linux hosts all sockets that want to share the same address and port combination must belong to processes that share the same effective user ID!
# So, on linux(kernel>=3.9) you have to run multiple servers and clients under one user to share the same (host, port).

client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

# Enable broadcasting mode
client.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

client.bind(("", 37020))
while True:
    # Thanks @seym45 for a fix
    data, addr = client.recvfrom(1024)
    print("received message: %s"%data)
```

server.py

```
import socket
import time

server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

# Enable port reusage so we will be able to run multiple clients and servers on single (host, port).
# Do not use socket.SO_REUSEADDR except you using linux(kernel<3.9): goto https://stackoverflow.com/questions/14388706/how-do-so-reuseaddr-and-so-reuseport-differ for more information.
# For linux hosts all sockets that want to share the same address and port combination must belong to processes that share the same effective user ID!
# So, on linux(kernel>=3.9) you have to run multiple servers and clients under one user to share the same (host, port).

server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

# Enable broadcasting mode
server.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

# Set a timeout so the socket does not block
# indefinitely when trying to receive data.
server.settimeout(0.2)
message = b"your very important message"
while True:
    server.sendto(message, ('<broadcast>', 37020))
    print("message sent!")
    time.sleep(1)
```

Завдання:

Створити клієнт-серверний додаток багатоадресною розсилки з використанням асинхронної обробки та потоків.

Представити код клієнта та сервера з поясненнями, відображення праці додатків в усіх консолях, де працював код та зробити висновки.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. W. R. Stevens, A. R. Stephen Advanced Programming in the UNIX : Addison Wesley Professional, 2013. 1034 p.

2. Pradeeban Kathiravelu, Dr. M. O. Faruque Sarker Python Network Programming Cookbook, second edition : Packt Publishing, 2017. 442 p.

3. Мельник В.М., Ройко О.Ю. Мережне програмування в середовищах операційних систем UNIX та Linux: частина перша. Навчальний посібник : «Вежа друк», 2017. 192 с.

ДОДАТОК 1

Приклад роботи з протоколом SMTP, використовуючи модуль smtplib і пакет email наведено нижче

```
import os.path
# не показує вод паролю
import getpass
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
from email.mime.application import MIMEApplication
if __name__ == "__main__":
    # 1. Данные сервера и письма
    # 'email_from'        : адрес відправника
    # 'email_to'          : адрес отримувача
    # 'email_from_password': пароль email_from
    email_from = input("Адрес відправника : ")
    email_to = input("Адрес  отримувача: ")
    # вод пароля використовує функцфя getpass.getpass,
    # не відображає вод
    email_from_password = getpass.getpass("Пароль для {}: ".format(email_from))
    # Текст повідомлення  у HTML-формате.
    # Можливо використовувати і звичайний текст.
    html = """\
    <html>
      <body>
       <h1>Прівет!</h1>
```

```
    <p>?</p>
    <p>
      Силка на сайт Python:
      <a href="http://www.python.org">http://www.python.org</a>.
    </p>
    <p>
      К повідомленню додан:.
      <ul>
        <li>рисунок;</li>
        <li>та  код.</li>
      </ul>
    </p>
  </body>
</html>
"""
# 2. створювання контейнера
msg = MIMEMultipart()
msg["Subject"] = "Силка на сайт Python"
msg["From"] = email_from
msg["To"] = email_to
# 2.1. Текст
# додавання вложення за допомогою attach у контейнер.
# у MIMEText якщо на  HTML а текст, тоді
# другим параметром буде "plain".
msg.attach(MIMEText(html, "html"))
# 2.2. Файл - малюнок
img_filename = input("Им’я файла з малюнком: ")
with open(img_filename, "rb") as image:
    attachment = MIMEImage(image.read())
```

```
    # позначаємо вложення та формуємо ім'я
    attachment.add_header("Content-Disposition", "attachment",
                          filename=os.path.basename(img_filename))
    msg.attach(attachment)
    # 2.3. Файл - Код
    with open(__file__, "rb") as f:
        attachment = MIMEApplication(f.read())
    # позначаємо вложення та формуємо ім'я
    attachment.add_header("Content-Disposition", "attachment",
                          filename=os.path.basename(__file__))
    msg.attach(attachment)
    # 3. Підключення к серверу та відправка повідомлення
    server = smtplib.SMTP("smtp.gmail.com", 587)
    try:
        # Встановлення захищеного з'єдненання
        server.starttls()
        server.login(email_from, email_from_password)
        # Відправка повідомлення
        server.send_message(msg)
        print("Повідомлення відправлено!")
    except smtplib.SMTPException as err:
       print("Помилка у відправкі:", err)
    finally:
        server.quit()
```

Приклад роботи з протоколом POP3

```
import sys
 import getpass
 import poplib
```

```
 import email
from email.header import decode_header
def do_decode_header(header):
 """Декодувати заголовок за ключом 'header'.
  Параметри:
   header (str): заголовок
                "=?UTF-8?B?UmU6INCT0LvQtdCxINCf0L7Rh9GC0LA=?=".
  Базові стандарти RFC 822 и RFC 2822 кажуть , що
  заголовок є набор ASCII-символів
  Python позволяет декодировати заголовки за домогою
   decode_header модуля email.header.
  """
  header_parts = decode_header(header)
  # 'decode_header' повертаємо список  вида
  # [(b'\xd0\x9f\xd0\xb0\xd0\xb2\xd0\xb5\xd0\xbb
   #     \xd0\x9f\xd0\xb0\xd0\xbd\xd1\x84\xd0\xb8\xd0\xbb
   #     \xd0\xbe\xd0\xb2', 'utf-8'),
  #  (b' <***@gmail.com>', None)]
    res = []
  for decoded_string, encoding in header_parts:
     if encoding:
        decoded_string = decoded_string.decode(encoding)
     elif isinstance(decoded_string, bytes):
        decoded_string = decoded_string.decode("ascii")
      res.append(decoded_string)
   # На виході 'res' є заголовок у декодованому виді
  return "".join(res)
def get_part_info(part):
 "    Параметри:
```

```
        - part: частина повідомлення email.Message.
      Результат:
        - message (str): повідомлення;
        - encoding (str): кодировка повідомлення ;
        - mime (str): MIME-тип.
      """
       encoding = part.get_content_charset()
       #
       if not encoding:
          encoding = sys.stdout.encoding
       mime = part.get_content_type()
       message = part.get_payload(decode=True).decode(encoding,
errors="ignore").strip()
       return message, encoding, mime
```

Послідуюча частина схоже з попередньої від SMTP

Приклад роботи з протоколом IMAP

```
import sys
import getpass
import re
import imaplib
import email
from email.header import decode_header
def do_decode_header(header):
   """Декодувати заголовок за ключом 'header'.
   Параметри:
     header (str): заголовок вида
                "=?UTF-8?B?UmU6INCT0LvQtdCxINCf0L7Rh9GC0LA=?=".
```

стандарти RFC 822 и RFC 2822 кажуть , що заголовок є набор ASCII-

```
символів
        """
        header_parts = decode_header(header)
        # 'decode_header' возвращает список кортежей вида
        # [(b'\xd0\x9f\xd0\xb0\xd0\xb2\xd0\xb5\xd0\xbb
        #     \xd0\x9f\xd0\xb0\xd0\xbd\xd1\x84\xd0\xb8\xd0\xbb
        #     \xd0\xbe\xd0\xb2', 'utf-8'),
        #  (b' <***@gmail.com>', None)]
        res = []
        for decoded_string, encoding in header_parts:
            if encoding:
                decoded_string = decoded_string.decode(encoding)
            elif isinstance(decoded_string, bytes):
                decoded_string = decoded_string.decode("ascii")
            res.append(decoded_string)
        return "".join(res)
    def get_part_info(part):
        """Получити текст повідомлення у правильномк коддуванні.
        Параметри:
          - part: часттна повідомлення email.Message.
        Результат:
          - message (str): повідомлення;
          - encoding (str): кодировка повідомлення;
          - mime (str): MIME-тип.
        """
        encoding = part.get_content_charset()
          if not encoding:
            encoding = sys.stdout.encoding
        mime = part.get_content_type()
```

```
            message = part.get_payload(decode=True).decode(encoding,
errors="ignore").strip()
            return message, encoding, mime
        def get_message_info(message):
            """Отримати текст повідомлення у правильній кодировці.
            Параметры:
              - message: повідомлення email.Message.
            Результат:
              - message (str): повідомлення или строка ;
              - encoding (str): кодировка повідомлення ;
              - mime (str): MIME-тип .
            """
            # Алгоритм отримання повідомлення :
            # - якщо повідомлення  включає деклька частин
            #   (message.is_multipart()) - необхідно пройтись по составним
            #   частинам повідомлення : "text/plain" фбо "text/html"
            # - якщо ні - текст можливо отримати напряму
            message_text, encoding, mime = "Не має тіла повідомлення", "-", "-"
            if message.is_multipart():
                for part in message.walk():
                    if part.get_content_type() in ("text/html", "text/plain"):
                        message_text, encoding, mime = get_part_info(part)
                        break  # Тільки перше входження
            else:
                message_text, encoding, mime = get_part_info(message)
            return message_text, encoding, mime
        if __name__ == "__main__":
            # 1. Підключення до серверу
            server = 'imap.gmail.com'
```

```
port = '993'
login = input('Адрес пошти: ')
password = getpass.getpass("Пароль для {}: ".format(login))
# Уся нформація зберігається у файлі
with open(__file__ + "_data.txt", "w", encoding="utf-8") as fh:
    try:
        # 2. Cnворення захищеного з’єдення
        mailserver = imaplib.IMAP4_SSL(server, port)
        mailserver.login(login, password)

        mailserver.select()
                response, messages_nums = mailserver.search(None, "ALL")
        if response != "OK":
            raise imaplib.IMAP4.error("Не вдалось получити список повідомлень.")
        # Кожне повідомлення має
        # номер від 1 до len(messages)
        # 'message_nums' - список вида
        # [b'1 2 3 4 5 6 7 8 9 10 11 12']
        messages_nums = messages_nums[0].split()
        print("\nВсего писем: {}".format(len(messages_nums)), file=fh)
                for message_num in reversed(messages_nums[-3:]):
            # message_parts = "(RFC822)" - отримання повідомлення разом
            response, data = mailserver.fetch(message_num,
                                    message_parts="(RFC822)")
            message_num = int(message_num.decode())
            if response != "OK":
                print("Не вдалось получити повідомлення №", message_num, file=fh)
```

```
                continue
            message_size = int(re.findall(r"{(\d+)}", data[0][0].decode())[0])
            raw_message = data[0][1]
            message = email.message_from_bytes(raw_message)
            # отримання текст повідомлення
            text, encoding, mime = get_message_info(message)
            # Вивод інформації о  повідомленні
            print("\n№{} ---"
                "\n  - від: '{}'"
                "\n  - тема: '{}'"
                "\n  - дата: '{}'"
                "\n  - size: {:.2f} Кб."
                "\n  -  (тип): MIME: {}, Кодировка: {}, multipart: {}"
                "\n  -  (перші 100 симв.):\n\"{}\"".
                format(message_num,
                    do_decode_header(message["From"]),
                    do_decode_header(message["Subject"]),
                    message["Date"],
                    message_size / 2**10,
                    mime, encoding, message.is_multipart(),
                    text[:100]),
                file=fh)
        print("Інформація о поштовой скринькі та повідомлення отримані та збережені.")
    except imaplib.IMAP4.error as err:
        print("Виникла помилка:", err)
    finally:
        ma
ilserver.logout()
```