

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

КОМП'ЮТЕРНІ МЕРЕЖІ

ЛАБОРАТОРНІ РОБОТИ З КРЕДИТНОГО МОДУЛЯ «КОМП'ЮТЕРНІ МЕРЕЖІ 2. ІНТЕРНЕТ ПРОТОКОЛИ»: виконання, оформлення та захист

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою
«Системне програмування і спеціалізовані комп'ютерні системи»
спеціальності 123 «Комп'ютерна інженерія»*

Київ
КПІ ім. Ігоря Сікорського
2022

Посібник з виконання лабораторних робіт з кредитного модуля
 «Комп'ютерні мережі 2. Інтернет протоколи» дисципліни «Комп'ютерні
 мережі» [Електронний ресурс] : навч. посіб. для студ. спеціальності 123
 «Комп'ютерна інженерія», освітньої програми «Системне програмування
 та спеціалізовані комп'ютерні системи» / КПІ ім. Ігоря Сікорського ; уклад.:
 М. М. Орлова, А. А. Крайносвіт, П. А. Сергієнко. – Електронні текстові дані (1
 файл: 2,63 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 100 с., реєстр.
 21/22-890.

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
 (протокол № 6 від 24.06.2022 р.)
 за поданням Вченої ради факультету прикладної математики
 (протокол № 9 від 30.05.2022р.)*

Електронне мережеве навчальне видання

КОМП'ЮТЕРНІ МЕРЕЖІ

«ЛАБОРАТОРНІ РОБОТИ З КРЕДИТНОГО МОДУЛЯ «КОМП'ЮТЕРНІ МЕРЕЖІ 2. ІНТЕРНЕТ ПРОТОКОЛИ»»:

виконання, оформлення та захист

Укладачі: *Орлова Марія Миколаївна, канд. техн. наук, доц.
 Крайносвіт Аркадій Артемович, асистент
 Сергієнко Павло Анатолійович, асистент*

Відповідальни
 й редактор *Тарасенко-Клятченко О.В., канд. техн. наук, доцент*

Рецензенти: *Олещенко Л.М., канд. техн. наук, доцент
 Корочкін О.В., канд. техн. наук, доцент*

Навчальний посібник розроблено для ознайомлення студентів з вимогами, правилами виконання, оформлення та оцінювання лабораторних робіт з кредитного модуля «Комп'ютерні мережі 2. Інтернет протоколи» дисципліни «Комп'ютерні мережі». Навчальне видання призначене для студентів спеціальності 123 «Комп'ютерна інженерія», освітньої програми «Системне програмування та спеціалізовані комп'ютерні системи» кафедри системного програмування і спеціалізованих комп'ютерних систем факультету прикладної математики КПІ ім. Ігоря Сікорського.

© КПІ ім. Ігоря Сікорського, 2022

ЗМІСТ

ВСТУП	4
<u>1. МЕТА ТА ЗАВДАННЯ ЛАБОРАТОРНИХ РОБІТ</u>	5
<u>2. ВИКОНАННЯ ТА ЗАХИСТ ЛАБОРАТОРНИХ РОБІТ</u>	6
<u>ЛАБОРАТОРНА РОБОТА 9. Аналіз процесів в об'єднаній комп'ютерній мережі</u>	7
<u>ЛАБОРАТОРНА РОБОТА 10. Організація доступу до комп'ютерної мережі. Технології VLAN і NAT</u>	32
<u>ЛАБОРАТОРНА РОБОТА 11. Робота з сокетамі</u>	60
<u>ЛАБОРАТОРНА РОБОТА 12. Обмін дейтаграмами</u>	84
<u>ЛАБОРАТОРНА РОБОТА 13. Мережева файлова система NFS</u>	95
<u>СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ</u>	102

ВСТУП

Даний посібник призначений для:

- визначення правил виконання та захисту лабораторних робіт з дисципліни «Комп'ютерні мережі» для здобувачів ступеня бакалавра за освітньою програмою «Системне програмування та спеціалізовані комп'ютерні системи» спеціальності 123 «Комп'ютерна інженерія» для студентів кафедри системного програмування і спеціалізованих комп'ютерних систем факультету прикладної математики КПІ імені Ігоря Сікорського;
- визначення правил і вимог до виконання лабораторних робіт;
- визначення правил і вимог до оформлення лабораторних робіт;
- правил оцінювання якості виконання лабораторних робіт та їх захисту з кредитного модуля «Комп'ютерні мережі 2. Інтернет протоколи» дисципліни «Комп'ютерні мережі».

Дисципліна «Комп'ютерні мережі» є базовою дисципліною циклу загальної підготовки бакалаврів за спеціальністю 123 «Комп'ютерна інженерія», яка ознайомлює студентів зі структурою, організацією та принципами функціонування комп'ютерних мереж (КМ). Одним з важливих видів індивідуальних завдань, що передбачені навчальною програмою дисципліни «Комп'ютерні мережі» та виконуються студентами при її вивченні, є лабораторні роботи (ЛР).

Самостійне виконання лабораторних робіт є обов'язковим при вивченні дисципліни «Комп'ютерні мережі».

Виконання лабораторних робіт сприяє розширенню та поглибленню отриманих теоретичних знань щодо організації та функціонування комп'ютерних мереж, а також обробки даних при введенні в мережу та їх передачі через комунікаційне середовище.

Посібник містить матеріали для виконання всіх лабораторних робіт кредитного модуля «Комп'ютерні мережі 2. Інтернет протоколи» дисципліни «Комп'ютерні мережі», що виконуються в другому семестрі, теоретичні відомості, які необхідні для виконання кожної лабораторної роботи, приклади та додаткові інформаційні матеріали, вимоги до виконання завдання, оформлення протоколу та захисту лабораторної роботи тощо.

1. МЕТА ТА ЗАВДАННЯ ЛАБОРАТОРНИХ РОБІТ

Метою циклу лабораторних робіт з дисципліни «Комп'ютерні мережі» є закріплення та поглиблення знань щодо принципів організації комп'ютерних мереж, принципів обробки даних, що передаються мережею, стеку протоколів мережі Інтернет, інкапсуляції тощо.

Кожна лабораторна робота призначена для вивчення конкретної теми та пов'язана з відповідними питаннями, що стосуються розгляду особливостей організації та функціонування окремих ресурсів комп'ютерної мережі та їх взаємодії. Крім того, необхідно ознайомитись та вивчити аналізатори мережевого трафіку, які надають можливість їх практичного використання в подальшій роботі.

В результаті виконання **лабораторних робіт** з кредитного модуля «Комп'ютерні мережі 2. Інтернет протоколи» студент повинен:

ЗНАТИ:

- особливості архітектури комп'ютерних мереж, організації взаємодії їх окремих модулів, функціонування окремих ресурсів КМ, процедур передачі повідомлень мережею тощо;
- загальну методику обробки та перетворення даних при їх передачі в мережу та просуванні через комунікаційне середовище;

УМІТИ:

- аналізувати вимоги до параметрів і особливостей передачі даних, аналізувати структури каналів передачі та комп'ютерної мережі в цілому;
- оцінювати особливості передачі даних з використанням різних способів та протоколів передачі, оцінювати локалізацію трафіку в IP-мережі;
- аналізувати і оцінювати результати виконаної роботи;

НАПРАЦЮВАТИ ДОСВІД:

- аналізу мережного (мережевого) трафіку за допомогою різних ресурсів (Wireshark, CISCO Packet Tracer та інших);
- перетворення DNS-імен в IP-адреси;
- роботи з сокетами різних типів;
- створення локальних мереж і їх інтеграція у зовнішні мережі;
- організації поштового зв'язку в розподіленій комп'ютерній мережі.

2. ВИКОНАННЯ ТА ЗАХИСТ ЛАБОРАТОРНИХ РОБІТ

З кредитного модуля «Комп'ютерні мережі 2. Інтернет протоколи» передбачено виконання 5 лабораторних робіт, які повинні бути виконані та захищені у визначені терміни:

- лабораторні роботи 9-11 виконуються та захищаються студентами не пізніше 6-го тижня (включно) весняного семестру (до першої атестації);
- лабораторні роботи 12-13 виконуються та захищаються студентами не пізніше 8-го тижня (включно) весняного семестру (до другої атестації);

При недотриманні даного графіку здачі нараховуються штрафні бали, а саме «- 1 бал» за кожний тиждень затримки.

Не дозволяється одночасно захищати більше двох лабораторних робіт.

Для захисту лабораторної роботи обов'язково оформляється протокол, який повинен містити наступні елементи:

- назва роботи;
- мета роботи;
- стислі теоретичні відомості за темою лабораторної роботи;
- порядок виконання роботи та отримані результати;
- висновки по роботі (обов'язково);
- в разі необхідності до протоколу додаються скріншоти, таблиці результатів тощо.

Перед захистом оформлений протокол надсилається викладачу для перевірки та попереднього аналізу виконаної роботи. В процесі захисту лабораторної роботи студент обов'язково відповідає на поставлені викладачем запитання.

Лабораторна робота може виконуватись та захищатись бригадою, яка складається з двох студентів (не більше). В цьому випадку кожен зі студентів виконує свою частину лабораторної роботи та отримує запитання при захисті роботи.

В разі поглибленого опрацювання теми лабораторної роботи, ретельної підготовки до захисту та ґрунтовних відповідей на поставлені викладачем запитання студент може отримати додаткові заохочувальні бали, але не більше «+ 2 бали» за ЛР.

Лабораторна робота 9

Аналіз процесів в об'єднаній комп'ютерній мережі при передачі поштового повідомлення з використанням симулятора мережі передачі даних Cisco Packet Tracer

Мета роботи: засвоєння принципів взаємодії мережевих пристроїв при передачі поштового повідомлення від відправника до отримувача в об'єднаній комп'ютерній мережі з використанням програми симуляції комп'ютерних мереж Cisco Packet Tracer.

План виконання лабораторної роботи

1. Побудова топології мережі, налаштування мережевих пристроїв;
2. Налаштування поштових серверів та серверів служби DNS;
3. Відправка поштового повідомлення по протоколу SMTP на сервер;
4. Отримання поштового повідомлення по протоколу POP3 від сервера;
5. Дослідження прикладних поштових протоколів в режимі симуляції;
6. Виконання індивідуального завдання.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Лабораторна робота базується на знаннях, отриманих після засвоєння наступних тем лекційного та лабораторного курсу: „Структура об'єднаної мережі”, „Адресація в IP-мережах” , „Структура стеку TCP/IP. Протоколи TCP, UDP, IP, ARP”, „Протоколи прикладного рівня DNS, SMTP та POP3.” Тому при підготовці до лабораторної роботи рекомендується повторити зазначені розділи. Стислий конспект цього теоретичного матеріалу наводиться нижче.

1.1.Протоколи SMTP і POP3

Для передачі повідомлень по TCP-з'єднанню більшість поштових агентів використовують протокол SMTP (Simple Mail Transfer Protocol).

Головною метою протоколу SMTP є надійна і ефективна доставка електронних поштових повідомлень. Для роботи протоколу потрібний надійний канал зв'язку. Середовищем для SMTP може слугувати окрема локальна мережа, система мереж або мережа Internet.

Протокол SMTP базується на наступній моделі комунікації: по запиту користувача поштова програма-відправник повідомлення встановлює двосторонній зв'язок з програмою-отримувачем (поштовим сервером).

Протокол SMTP взаємодіє з поштовим сервером використовуючи транспортний протокол TCP через порт 25.

Отримувачем може бути кінцевий або проміжний адресат. Якщо необхідно, поштовий сервер може встановити з'єднання з іншим сервером і передати повідомлення далі. Інший сервер вибирається з використанням MX-записів служби DNS.

Для того щоб отримати повідомлення із своєї поштової скриньки, поштова програма користувача з'єднується з сервером вже не по протоколу SMTP, а по спеціальному поштовому протоколу отримання повідомлень. Такий протокол дозволяє працювати з поштовою скринькою: забирати повідомлення, видаляти повідомлення, сортувати їх і виконувати інші операції. Самим популярним серед таких протоколів є протокол POP3 (Post Office Protocol v.3).

Протокол POP3 взаємодіє з поштовим сервером використовуючи транспортний протокол TCP через порт 110.

Щоб отримати доступ до POP3-сервера користувач має запустити спеціальний поштовий агент, який працює по протоколу POP3, і налаштувати його для роботи зі своїм поштовим сервером. Повідомлення доставляються клієнту по протоколу POP3, а надсилаються за допомогою SMTP. Тобто на комп'ютері користувача працюють два окремі програмні інтерфейси до поштової системи – агент доставки (POP3) і агент відправки (SMTP).

1.2.Служба DNS

Взаємодія з системою електронної пошти неможлива без системи доменних імен (DNS). В задачі служби DNS входить:

1. Перетворення символічних імен в IP-адреси;
2. Перетворення IP-адрес в символічні імена.

Додатковою функцією DNS є маршрутизація пошти.

Одиницями зберігання і передачі інформації в DNS є ресурсні записи. Для маршрутизації пошти використовується запис "MX", а коли він відсутній, то запис типу "A". Запис "A" (адресний запис) містить параметри: доменне ім'я вузла, відповідну IP-адресу.

Запис "MX" містить параметри: ім'я поштового домена, ім'я поштового сервера, пріоритет.

При отриманні поштового повідомлення МТА аналізує його службову інформацію, зокрема заголовок, визначаючи домен отримувача. Якщо

повідомлення призначене домену, який обслуговується даним МТА, виконується пошук отримувача і повідомлення поміщається в його скриньку. Якщо домен отримувача не обслуговується цим МТА, формується DNS-запит, що запитує MX-записи для даного домену. MX-запис – це особливий вид DNS-запису, який містить імена поштових серверів, які опрацьовують вхідну пошту для даного домену. MX-записів може бути кілька, тоді МТА намагається послідовно встановити з'єднання, починаючи з сервера с найвищим пріоритетом. При відсутності MX-запису використовується А-запис (запис адреси, яка порівнює доменне ім'я з IP-адресою) і виконується спроба доставити пошту на вказаний там хост. При неможливості відправити повідомлення, воно повертається відправнику (поміщається в поштову скриньку користувача) з повідомленням про помилку.

Протокол перетворення адрес DNS на транспортному рівні використовує протокол UDP (порт 53), рідше TCP (порт 53).

2. Побудова топології мережі

Для дослідження процесів, що відбуваються при взаємодії мережевих пристроїв під час передачі поштового повідомлення побудуємо тестову мережу, приклад якої зображений на рисунку 9.1.

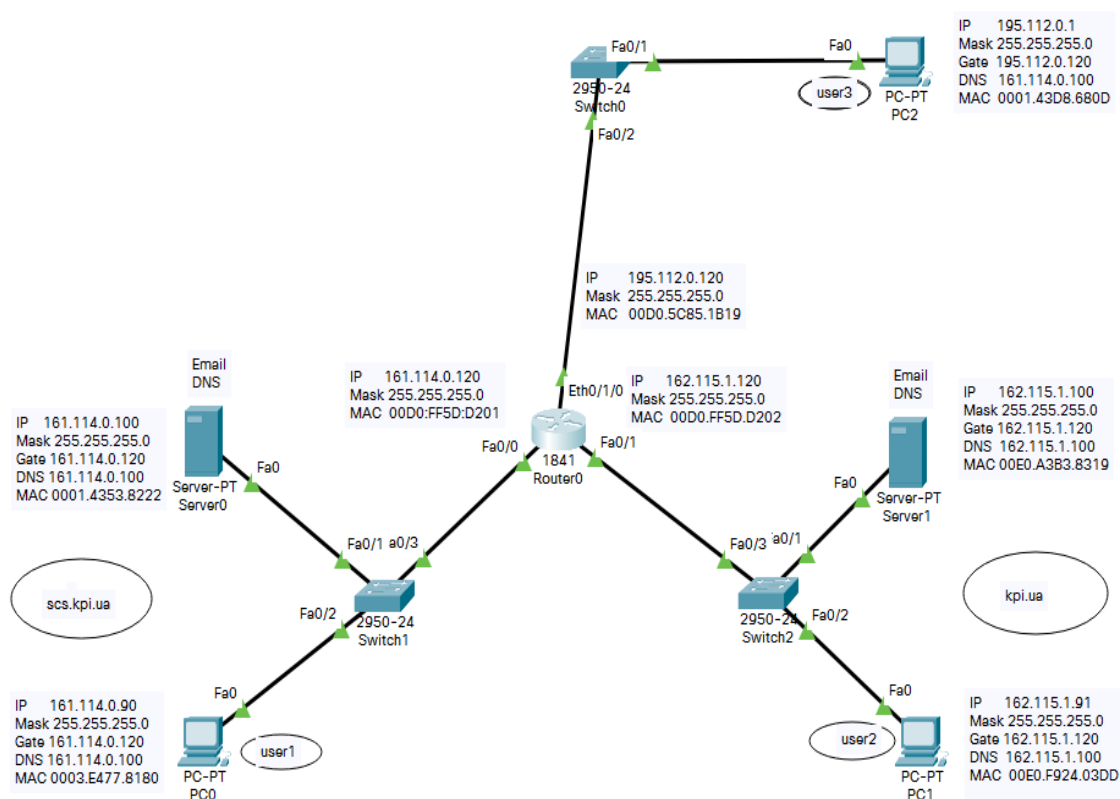


Рисунок 9.1– Топологія досліджуваної мережі

2.1. Налаштуємо мережеві пристрої згідно із заданими нижче параметрами (таблиця 9.1, таблиця 9.2):

Таблиця 9.1

Кінцеві вузли	IP-адреса	Маска мережі	IP-адреса DNS-сервера	Шлюз
PC0	161.114.0.90	255.255.255.0	161.114.0.100	161.114.0.120
PC1	162.115.1.91	255.255.255.0	162.115.1.100	162.115.1.120
PC2	195.112.0.1	255.255.255.0	161.114.0.100	195.112.0.120

Таблиця 9.2

Сервери Маршрутизатор	IP-адреса		Маска мережі	IP-адреса DNS-сервера	Шлюз
Server0	161.114.0.100		255.255.255.0	161.114.0.100	161.114.0.120
Server1	162.115.1.100		255.255.255.0	162.115.1.100	162.115.1.120
Router0	fa0/0	161.114.0.120	255.255.255.0		
	fa0/0	162.115.1.120	255.255.255.0		
	eth0/1/0	195.112.0.120	255.255.255.0		

2.2. Налаштування маршрутизатора Router0.

Маршрутизатор Cisco 1841 в стандартній комплектації обладнаний двома інтерфейсами FastEthernet0/0 і FastEthernet0/1. Для побудови мережі із заданою топологією необхідно ввести до складу маршрутизатора додатковий інтерфейсний модуль. Додавання модуля виконується наступним чином:

- 1) Один раз клікаємо на зображенні маршрутизатора;
- 2) Вибираємо вкладку „Physical“;
- 3) Вимикаємо маршрутизатор натискуючи клавішу „0“ на маршрутизаторі;
- 4) Вибираємо модуль WIC-1ENET, до складу якого входить один порт 10Mbps, і перетягуємо його на вільне гніздо маршрутизатора. Після встановлення модуля, натискуючи клавішу „1“, включаємо маршрутизатор;
- 5) Закриваємо вікно „Physical“.

Тепер переходимо до налаштування самого маршрутизатора.

Маршрутизатор в даній топології має три інтерфейси: FastEthernet0/0, FastEthernet0/1 і додатковий Ethernet 0/1/0.

- 1) Один раз клікаємо на зображенні маршрутизатора;
- 2) Вибираємо вкладку „CLI“ („Інтерфейс командного рядка“);
- 3) У вікні, що відкрилося, натискаємо клавішу „Enter“; Отримуємо запрошення:

Router>

4) Входимо в привілейований режим:

Router>enable

5) Запрошення змінює вигляд:

Router#

6) Переходимо в режим глобальної конфігурації:

Router#configure terminal

7) Переходимо в режим налаштування інтерфейсу FastEthernet0/0:

Router(config)#interface fa0/0

8) Призначаємо інтерфейсу IP-адресу і маску:

Router(config-ip)#ip address 161.114.0.120 255.255.255.0

9) Включаємо інтерфейс:

Router(config-ip)#no shutdown

10) Зберігаємо виконані налаштування:

Router(config-ip)#do write

11) Виходимо із режиму конфігурування інтерфейсу fa0/0:

Router(config-ip)#exit

12) Переходимо до налаштування інтерфейсу FastEthernet0/1:

Router(config)#interface fa0/1

13) Призначаємо інтерфейсу IP-адресу і маску:

Router(config-ip)#ip address 162.115.1.120 255.255.255.0

14) Включаємо інтерфейс:

Router(config-ip)#no shutdown

15) Зберігаємо виконані налаштування:

Router(config-ip)#do write

16) Виходимо із режиму конфігурування інтерфейсу fa0/1:

Router(config-ip)#exit

17) Переходимо до налаштування інтерфейсу Ethernet0/1/0:

Router(config)#interface ethe0/1/0

18) Призначаємо інтерфейсу IP-адресу і маску:

Router(config-ip)#ip address 195.112.0.120 255.255.255.0

19) Включаємо інтерфейс:

Router(config-ip)#no shutdown

20) Зберігаємо виконані налаштування:

Router(config-ip)#do write

21) Виходимо із режиму конфігурування інтерфейсу ethe0/1/0:

Router(config-ip)#exit

22) Виходимо із режиму глобального конфігурування:

Router(config)#exit

23) Для перевірки налаштувань портів маршрутизатора виконайте наступну команду:

Router#show ip interface brief – маємо отримати повідомлення, яке зображене на рисунку 9.2.

```
R0#sh ip interface brief
Interface                IP-Address      OK? Method Status      Protocol
FastEthernet0/0          161.114.0.120   YES NVRAM    up          up
FastEthernet0/1          162.115.1.120   YES NVRAM    up          up
Ethernet0/1/0            195.112.0.120   YES manual up          up
Vlan1                    unassigned      YES unset   administratively down down
```

Рисунок 9.2 – Перевірка налаштувань інтерфейсів маршрутизатора

24) Перевіряємо стан таблиці маршрутизації:

Router#show ip route – маємо отримати повідомлення, яке зображене на рисунку 9.3.

```
R0#sh ip route
Codes: C - connected, S - static, I - IGRP, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2, E - EGP
       i - IS-IS, L1 - IS-IS level-1, L2 - IS-IS level-2, ia - IS-IS inter area
       * - candidate default, U - per-user static route, o - ODR
       P - periodic downloaded static route

Gateway of last resort is not set

C    161.114.0.0/16 is directly connected, FastEthernet0/0
C    162.115.0.0/16 is directly connected, FastEthernet0/1
C    195.112.0.0/24 is directly connected, Ethernet0/1/0
```

Рисунок 9.3 – Перегляд стану таблиці маршрутизації

Налаштування топології досліджуваної мережі завершено. Правильність проведених налаштувань перевіряємо за допомогою утиліти *ping*.

2.3. Налаштування поштових серверів та служб DNS.

В лабораторній роботі застосовуються два сервери електронної пошти, які розташовані в різних мережах – один в мережі kpi.ua, інший в мережі scs.kpi.ua. Раніше їм були призначені IP-адреси, відповідно 162.115.1.100 і 161.114.0.100.

На кожному поштовому сервері підтримується робота SMTP- і POP3-серверів.

2.3.1. Конфігурування поштового сервера з IP-адресою 161.114.0.100:

1) Один раз клікаємо на вибраному сервері;

2) Вибираємо вкладку “*Services*”, далі *Email*;

3) Підключаємо протоколи SMTP і POP3 і вказуємо ім’я поштового домену **scs.kpi.ua**; натискаємо кнопку “*Set*”(рис.9.4);

4) Створюємо облікові записи користувачів в мережі – імена user1, user2, user3 і відповідні паролі. Натисканням кнопки “+” додаємо створені записи в систему(рис.9.4);

2.3.2. Аналогічно виконаємо конфігурування поштового сервера з IP-адресою 162.115.1.100:

1) Один раз клікаємо на вибраному сервері;

2) Вибираємо вкладку “*Services*”, далі *Email*;

3) Підключаємо протоколи SMTP і POP3 і вказуємо ім’я поштового домену **kpi.ua**; натискаємо кнопку “*Set*”(рис.9.5);

4) Створюємо облікові записи користувачів в мережі – імена user1, user2, user3 і відповідні паролі. Натисканням кнопки “+” додаємо створені записи в систему(рис.9.5).

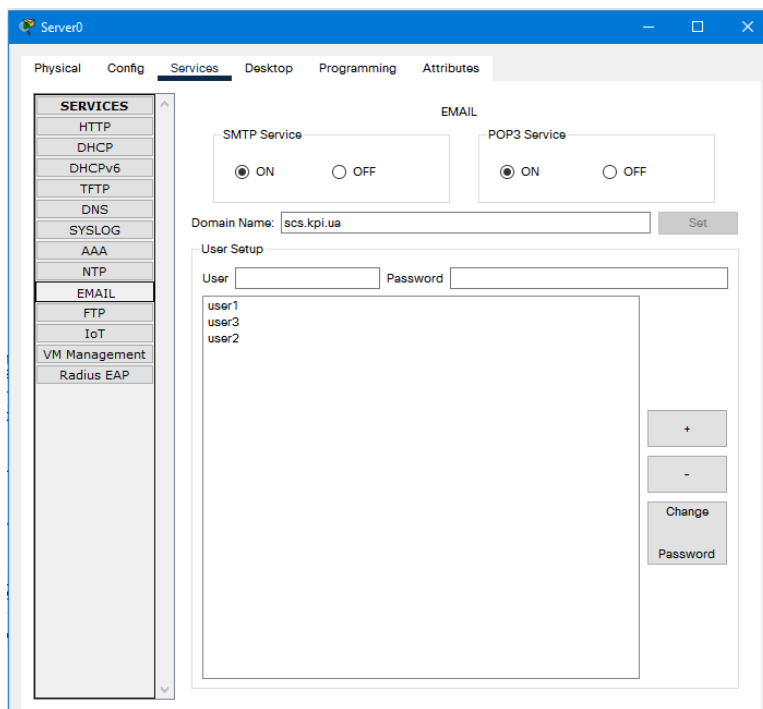


Рисунок 9.4 – Конфігурування поштового сервера Server0

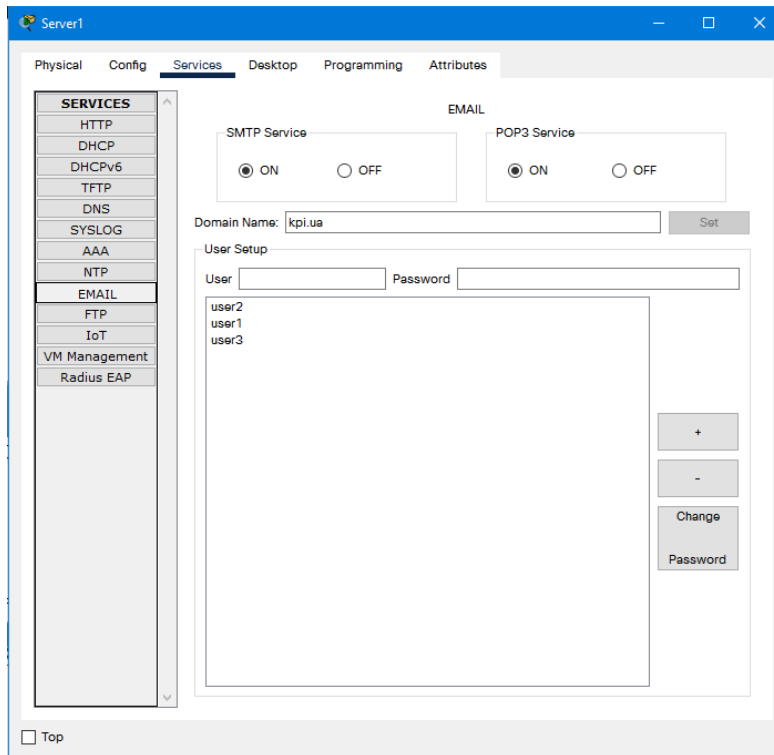


Рисунок 9.5 – Конфігурування поштового сервера Server1

2.3.3. Підключення служби DNS на поштових серверах.

- 1) Один клік по хосту 161.114.0.100;
- 2) Вибираємо вкладку “Services”, далі DNS(рис.9.6);
- 3) Почергово створюємо ресурсні записи типу **A**: доменні імена мережевих пристроїв і відповідні їм IP-адреси. Симулятор не підтримує ресурсний запис типу MX, але його можна замінити на ресурсний запис типу **A**(рис.9.6);
- 3) Почергово натискаємо кнопку “Add” і додаємо нові записи в службу DNS(на рисунку 9.6 показаний результат виконаних дій);
- 4) Активізуємо службу DNS (перемикач активності в стан “**on**”);
- 5) Аналогічні дії виконуємо на маршрутизаторі з IP-адресою 162.115.1.100. При налаштуванні враховуємо, що ім’я домену тепер **kpi.ua**(рис.9.7).

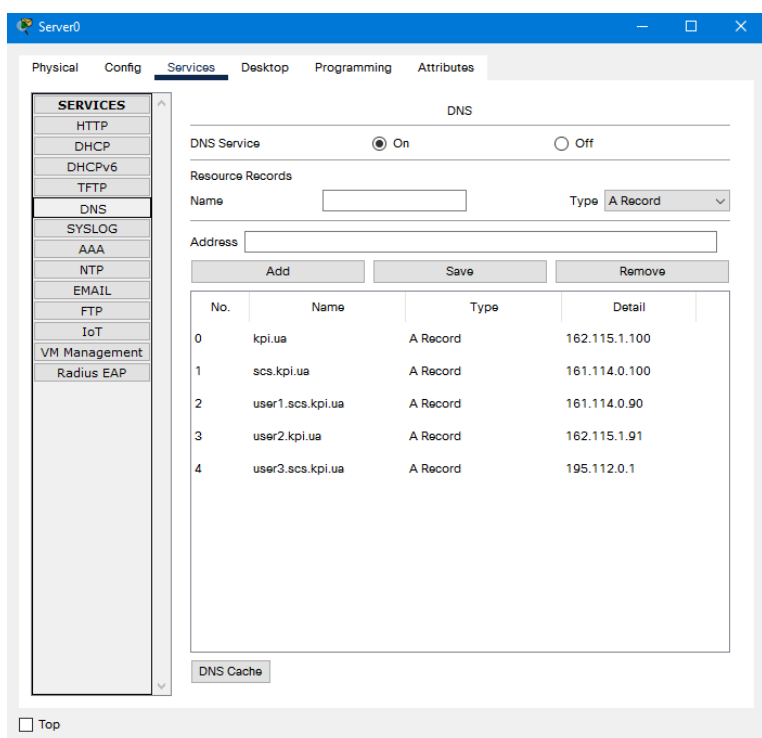


Рисунок 9.6 – Конфігурування служби DNS сервера Server0

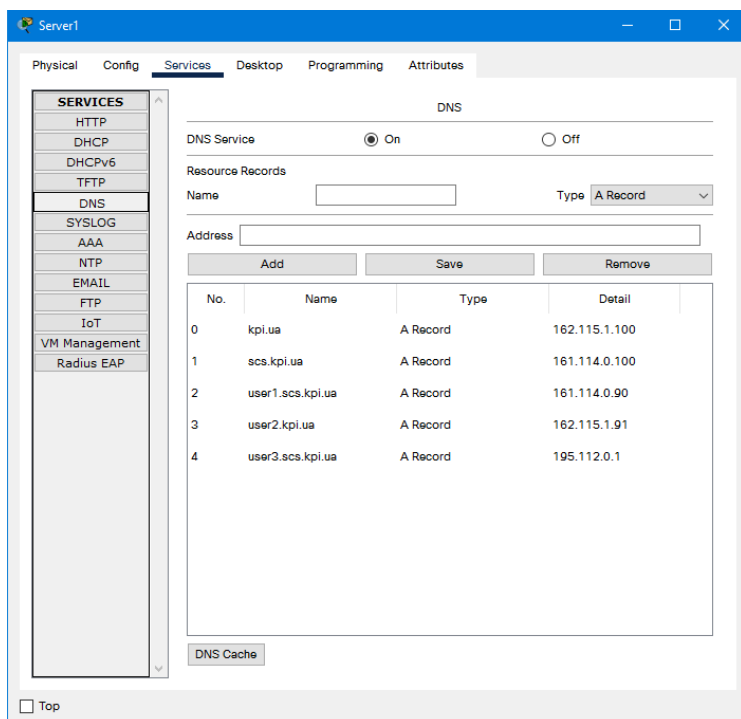


Рисунок 9.7 – Конфігурування служби DNS сервера Server1

2.4. Налаштування поштової служби на кінцевих вузлах.

Для роботи з SMTP- та POP-серверами на комп'ютері користувача має бути налаштований поштовий клієнт, який буде взаємодіяти з серверами.

Виконаємо налаштування на хості з IP-адресою 161.114.0.90:

- 1) Один клік на обраному хості;

2) Вибираємо вкладку “*Desktop*”, далі *Email*, *Configure Mail*. У конфігураційному вікні поштового сервісу вводимо облікові дані користувача user1(рис.9.8);

3) Натискаємо кнопку “*Save*”. Закриваємо вікно.

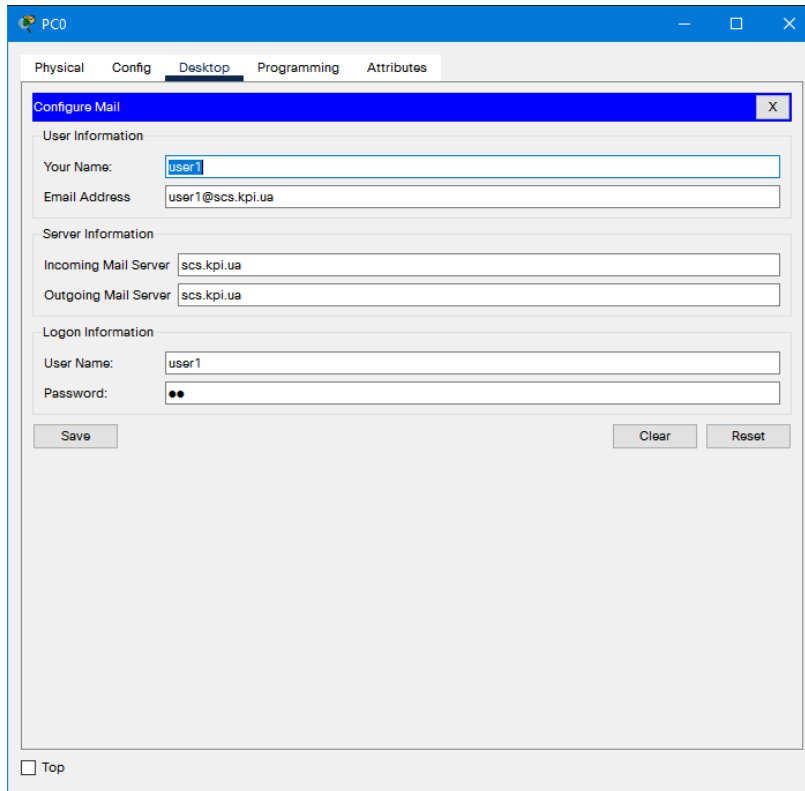


Рисунок 9.8 – Конфігурування служби Email на хості 161.114.0.90

Виконаємо налаштування на хості з IP-адресою 162.115.1.91:

1) Один клік на обраному хості;

2) Вибираємо вкладку “*Desktop*”, далі *Email*, *Configure Mail*. У конфігураційному вікні поштового сервісу вводимо облікові дані користувача user2(рис.9.9);

3) Натискаємо кнопку “*Save*”. Закриваємо вікно.

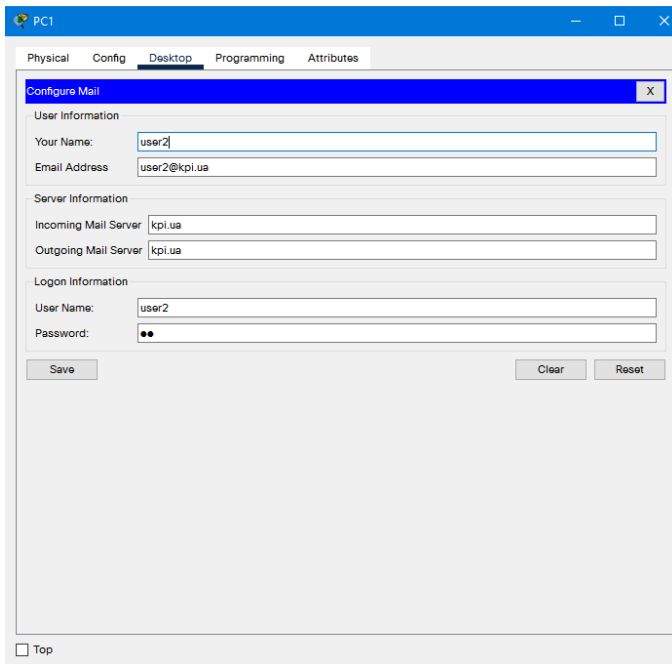


Рисунок 9.9 – Конфігурування служби Email на хості 162.115.1.91

Виконаємо налаштування на хості з IP-адресою 195.112.0.1. Він знаходиться в мережі, яка користується поштовим сервером і службою DNS із мережі **scs.kpi.ua**.

- 1) Один клік на обраному хості;
- 2) Вибираємо вкладку “*Desktop*”, далі *Email, Configure Mail*. У конфігураційному вікні поштового сервісу вводимо дані користувача user3 (рис.9.10);
- 3) Натискаємо кнопку “*Save*”. Закриваємо вікно.

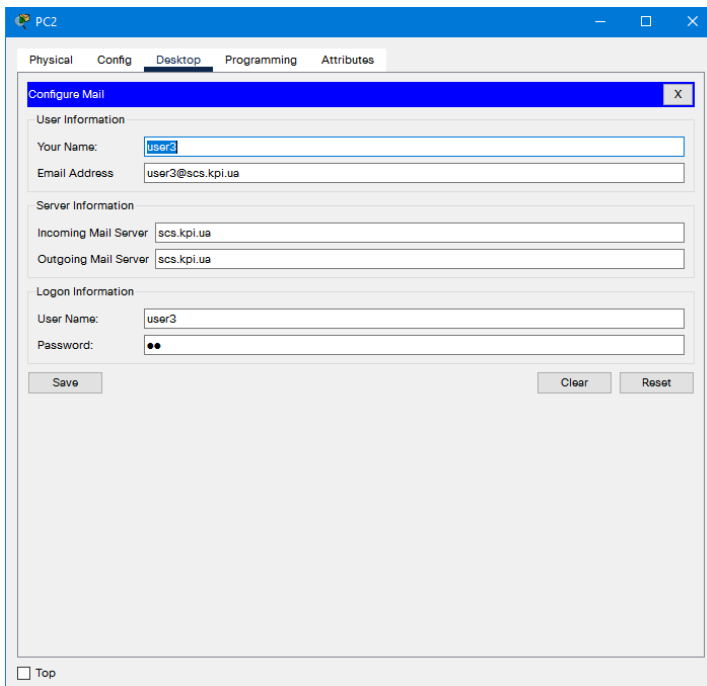


Рисунок 9.10 – Конфігурування служби Email на хості 195.112.0.1

Налаштування всіх пристроїв і необхідних служб завершено.

3. Перевірка здатності створеної моделі пересилати поштове повідомлення від користувача *user1* до користувача *user3*. Режим симуляції відключений.

3.1. Підготовка листа від *user1* (161.114.0.90) до *user3* (195.112.0.1) (рис. 9.11):

- 1) Один клік по вибраному хосту (161.114.0.90);
- 2) Вибираємо на вкладці “*Desktop*” програму “*E-mail*”;
- 3) Щоб написати листа, натискаємо на кнопку “*Compose*”. Заповнюємо форму, що з’явилася. В полі “*To*” задається адреса електронної пошти отримувача (*user3@scs.kpi.ua*). В полі “*Subject*” вказуємо тему листа. Текст листа довільний.
- 4) Натискаємо кнопку “*Send*”, щоб відправити листа.

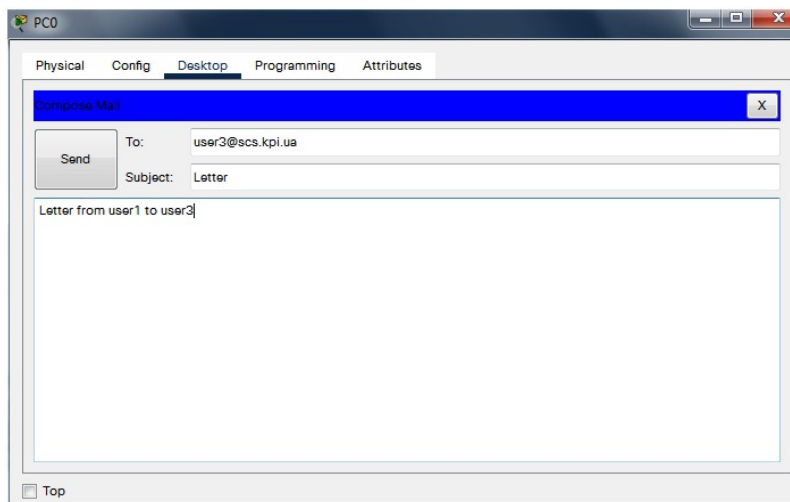


Рисунок 9.11 – Підготовка поштового повідомлення для *user3*

3.2. За допомогою протоколу SMTP ми відправили поштове повідомлення серверу *scs.kpi.ua* і тепер воно зберігається там в поштовій скриньці *user3*. Для отримання поштового повідомлення користувачу *user3* необхідно виконати такі дії:

- 1) Один клік по хосту 195.112.0.1;
- 2) Вибираємо на вкладці “*Desktop*” програму “*E-mail*”;
- 3) Натискаємо кнопку “*Receive*”, щоб отримати повідомлення(рис.9.12).

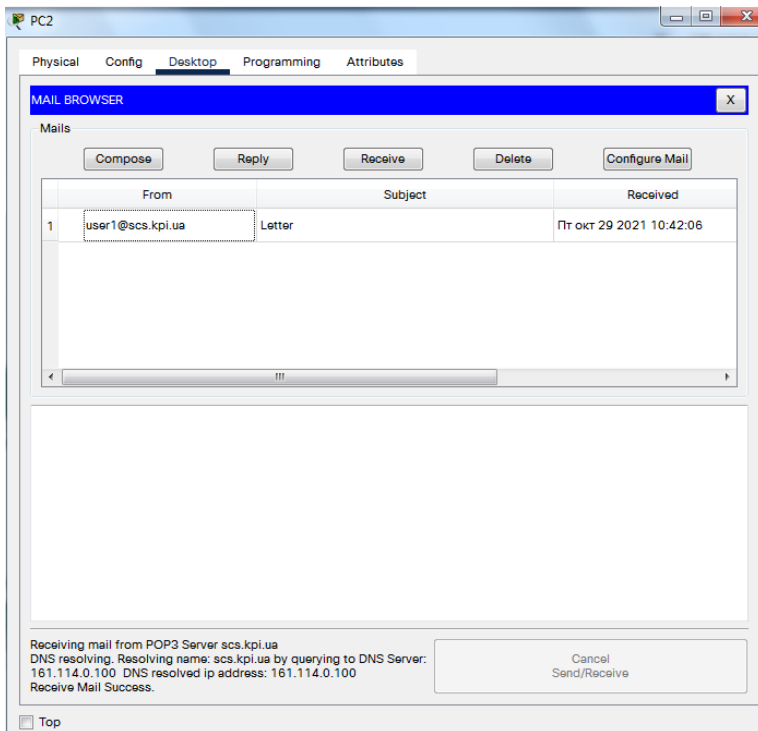


Рисунок 9.12 – Отримання поштового повідомлення user3

3.3. Переходимо до режиму симуляції Cisco Packet Tracer.

Відправимо листа з хоста 161.114.0.90 від *user1* на хост 162.115.1.91 *user2* (рис. 9.13):

- 1) Один клік по хосту 161.114.0.90;
- 2) Вибираємо на вкладці "Desktop" програму "E-mail";
- 3) Щоб написати листа, натискаємо на кнопку "Compose", і заповнюємо форму. В полі "To" задаємо адресу user2@kpi.ua. В полі "Subject" записуємо тему листа. Текст листа довільний.
- 4) Для відправки листа натискаємо кнопку "Send".

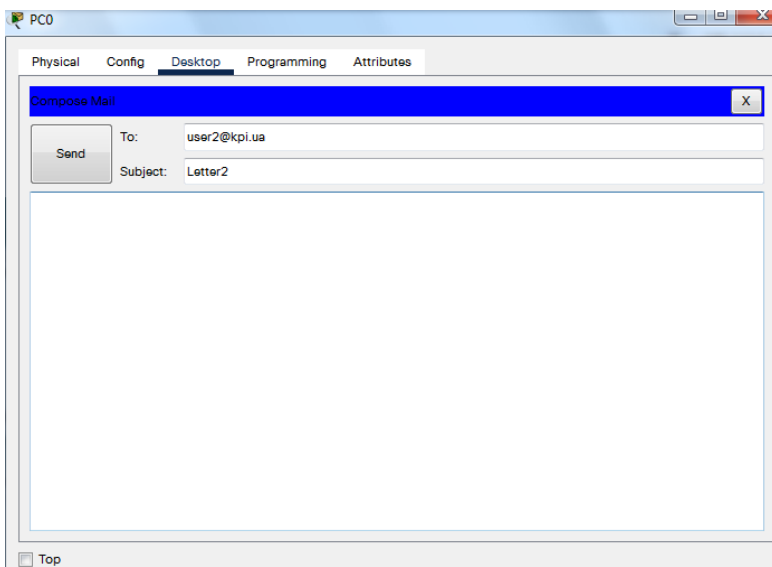


Рисунок 9.13 – Відправка поштового повідомлення user2

У вікні Event List бачимо, що на хості 161.114.0.90 сформувався пакет DNS (рис.9.14). За допомогою кнопки ”Capture/Forward“, прослідкуємо за взаємодією пакетів протоколу SMTP з пакетами протоколів TCP, DNS і ARP.

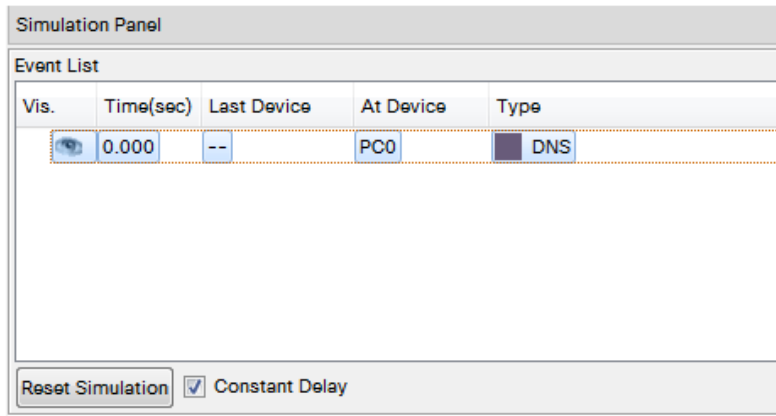


Рисунок 9.14 – Створення пакету з DNS-запитом

Розглянемо вміст цього пакету(рис. 9.15).

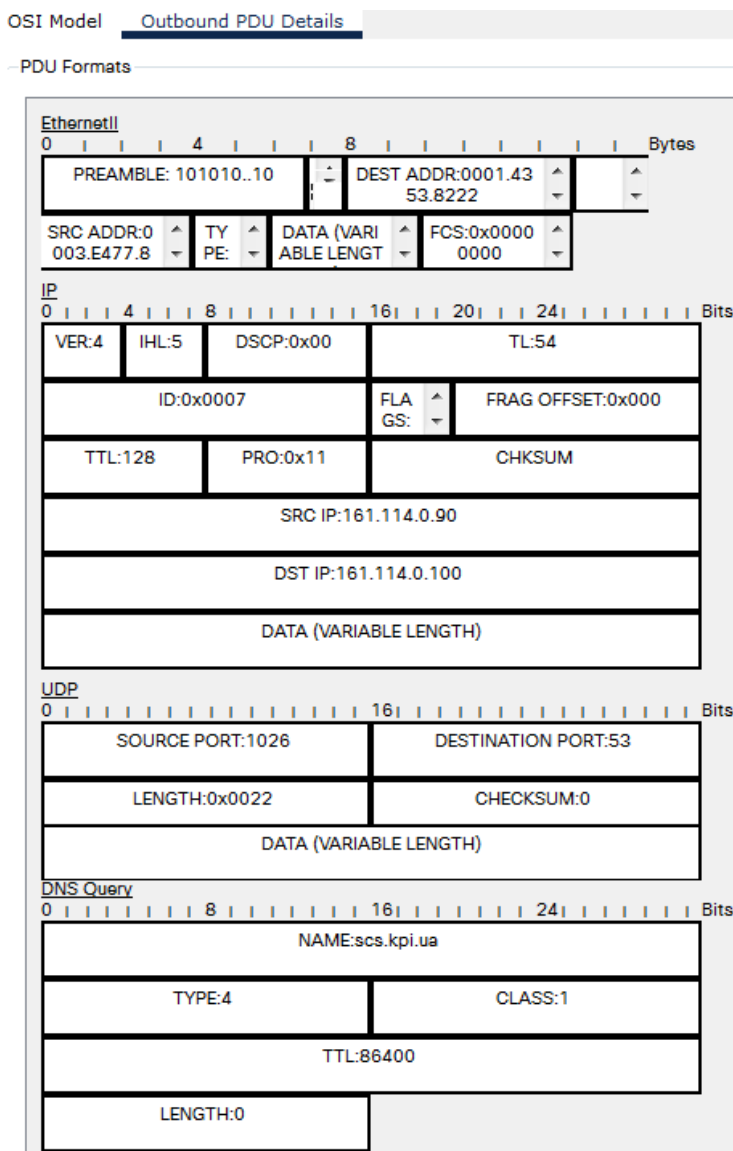


Рисунок 9.15 – Структура кадру з DNS-запитом

Як бачимо, запит DNS призначений для перетворення доменного імені SMTP-сервера *scs.kpi.ua* в IP-адресу.

Переходимо до вкладки OSI Model і бачимо, що пакет із запитом DNS знаходиться на мережевому рівні Layer 3 (рис.9.16). Для подальшого просування на канальний рівень потрібно визначити MAC-адресу призначення.

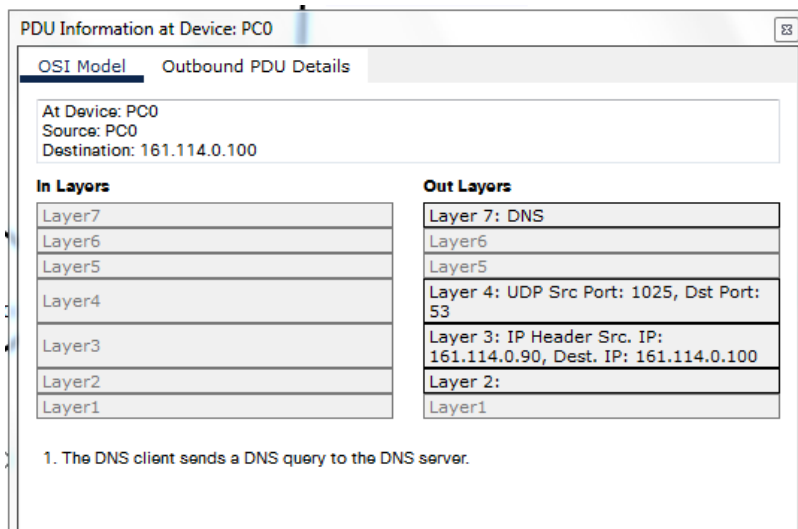


Рисунок 9.16 – Запит DNS на мережевому рівні

Тому комп'ютер PC0 створює і відправляє в мережу широкомовний запит ARP. Натискаючи на кнопку "Capture/Forward", слідкуємо за проходженням ARP-запиту. Маршрутизатор Router0 цей запит ігнорує, а сервер Server0 надсилає комп'ютеру PC0 ARP-відповідь, яка містить MAC-адресу Server0. Це дозволяє завершити формування повідомлення DNS. Пакет з DNS-запитом передається для відправки на канальний рівень Layer 2 (рис.9.17).

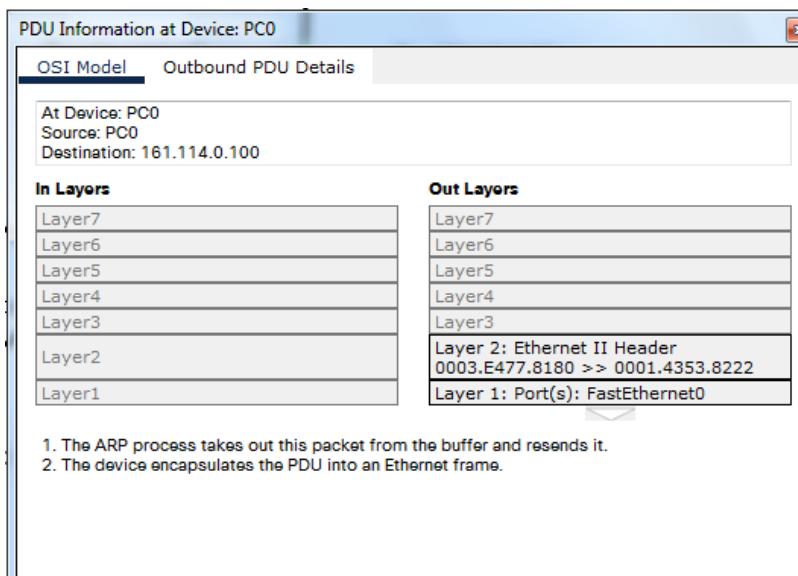


Рисунок 9.17 – Запит DNS на канальному рівні

Simulation Panel				
Event List				
Vis.	Time(sec)	Last Device	At Device	Type
	0.000	--	PC0	DNS
	0.000	--	PC0	ARP
	0.001	PC0	Switch1	ARP
	0.002	Switch1	Server0	ARP
	0.002	Switch1	Router0	ARP
	0.003	Server0	Switch1	ARP
	0.004	Switch1	PC0	ARP
	0.004	--	PC0	DNS
	0.005	PC0	Switch1	DNS
	0.006	Switch1	Server0	DNS
	0.007	Server0	Switch1	DNS
	0.008	Switch1	PC0	DNS
	0.008	--	PC0	TCP
	0.009	PC0	Switch1	TCP
	0.010	Switch1	Server0	TCP
	0.011	Server0	Switch1	TCP
	0.012	Switch1	PC0	TCP
	0.012	--	PC0	SMTP
	0.013	PC0	Switch1	TCP
	0.013	--	PC0	SMTP
	0.014	PC0	Switch1	SMTP
	0.015	Switch1	Server0	SMTP
	0.015	--	Server0	DNS
	0.015	--	Server0	DNS
	0.015	--	Server0	TCP
	0.015	--	Server0	DNS
	0.015	--	Server0	ARP
	0.016	Server0	Switch1	SMTP
	0.016	--	Server0	ARP
	0.017	Server0	Switch1	ARP
	0.017	Switch1	PC0	SMTP
	0.017	--	PC0	TCP
	0.018	Switch1	PC0	ARP
	0.018	Switch1	Router0	ARP
	0.018	PC0	Switch1	TCP
	0.019	Router0	Switch1	ARP
	0.019	Switch1	Server0	TCP
	0.020	Switch1	Server0	ARP
	0.020	Server0	Switch1	TCP
	0.020	--	Server0	TCP
	0.021	Server0	Switch1	TCP
	0.021	Switch1	PC0	TCP

Рисунок 9.18 – Передача поштового повідомлення. Фрагмент лістингу.

Натискаючи на кнопку "Capture/Forward", слідкуємо за проходженням пакету до сервера DNS. Сервер DNS виконує перетворення адрес і повертає DNS-відповідь комп'ютеру PC0(рис.9.18).

Тепер на комп'ютері PC0 почав формуватися пакет TCP для встановлення з'єднання (задіяний біт SYN в полі Flags) з сервером SMTP на хості 161.114.0.100 по порту 25(рис.9.19).

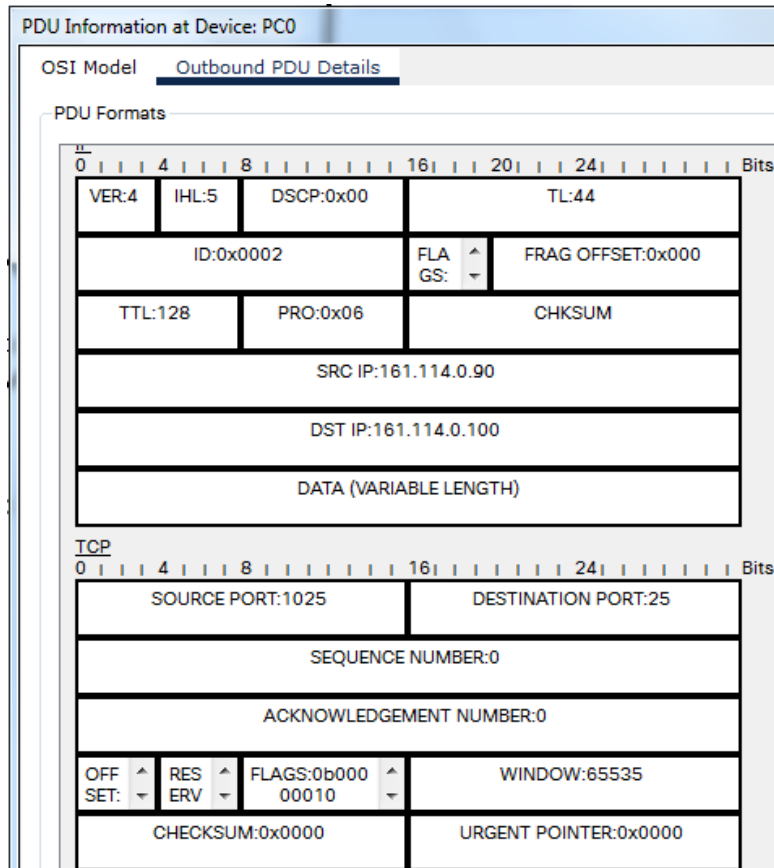


Рисунок 9.19 – Запит TCP на встановлення з'єднання

Натискаючи на кнопку "Capture/Forward", спостерігаємо, як відбувається триразове рукостискання: у відповідь на пакет з SYN сервер відправляє пакет з ACK+SYN, а потім PC0 відправляє серверу пакет з ACK.

Одночасно на PC0 починає формуватися SMTP-пакет(рис.9.20). Структура кадру з SMTP-пакетом показана на рисунку 9.21. Зараз протоколи TCP і SMTP працюють незалежно один від одного.

Після відправки пакету TCP з ACK на Switch1 відбувається передача пакету SMTP на фізичний рівень (Layer 1) для відправки в мережу(рис.9.22).

Натискаючи на кнопку "Capture/Forward" спостерігаємо за проходженням SMTP-пакету до поштового сервера Server0.

Після отримання SMTP-пакету Server0 надсилає комп'ютеру PC0 підтвердження про отримання, а PC0 відправляє серверу Server0 TCP-пакет з встановленими бітами FIN+ACK. Відбувається триразовий обмін службовими пакетами – процедура розірвання з'єднання.

Отримавши SMTP-пакет, SMTP-сервер аналізує адресу призначення в поштовому повідомленні. У нашому прикладі - це **kpi.ua**. Тому протокол SMTP викликає службу DNS для перетворення доменного імені **kpi.ua** в IP-адресу(рис. 9.23).

На Server0 створюється TCP-пакет з прапорцем SYN для встановлення з'єднання з поштовим сервером(порт 25) Server1.

Поштовий сервер Server0 отримує від служби DNS, яка працює на цьому ж комп'ютері, відповідь на DNS-запит про IP-адресу поштового сервера з доменним ім'ям **kpi.ua**

На Server0 створюється ARP-пакет, щоб визначити MAC-адресу вхідного інтерфейсу маршрутизатора Router0(рис. 9.24).

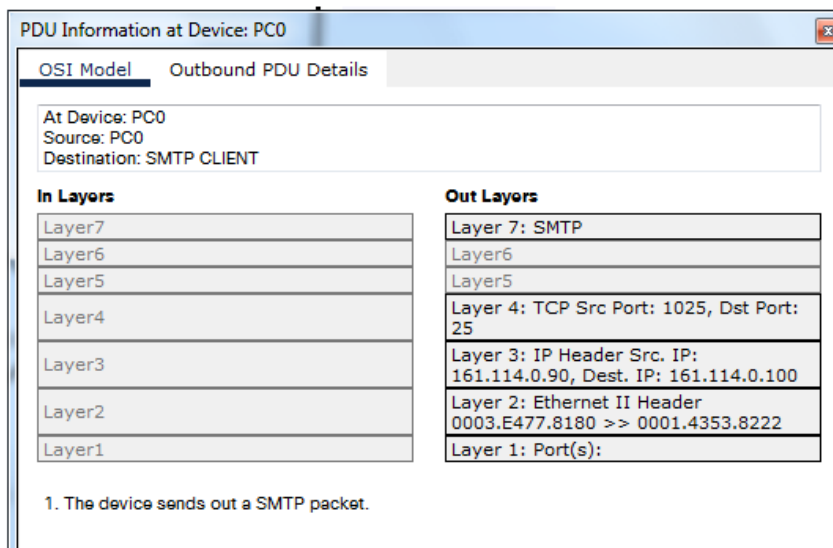


Рисунок 9.20 – Створення SMTP-повідомлення

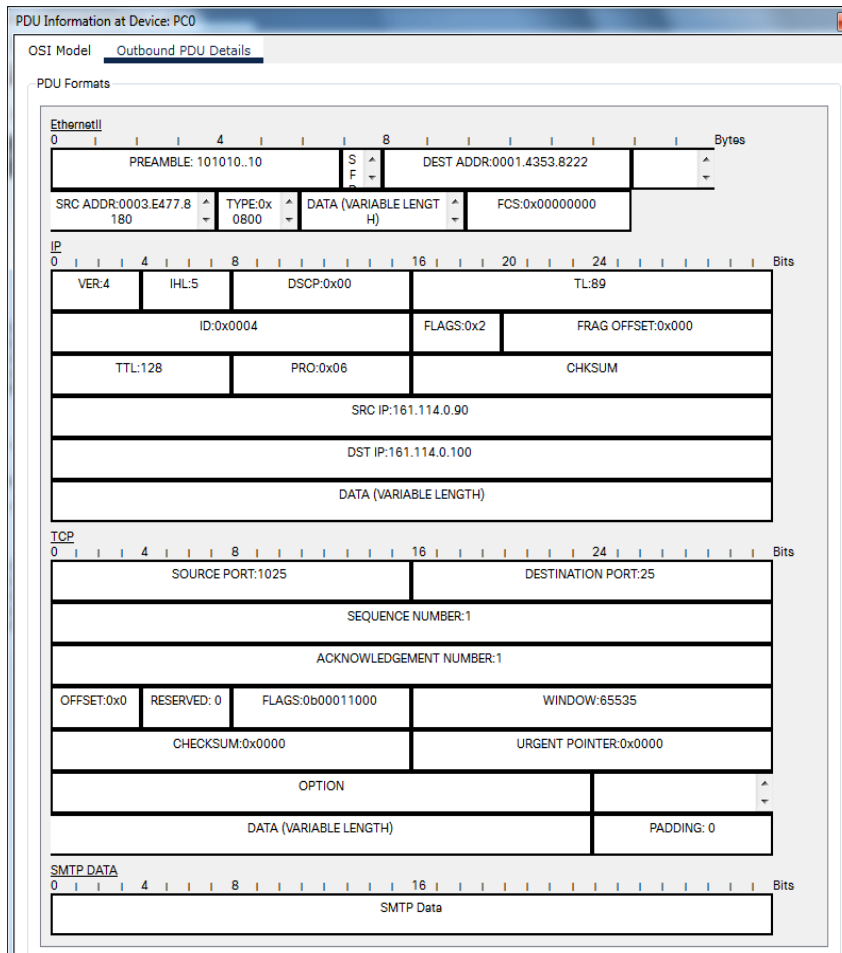


Рисунок 9.21 – Структура SMTP-повідомлення

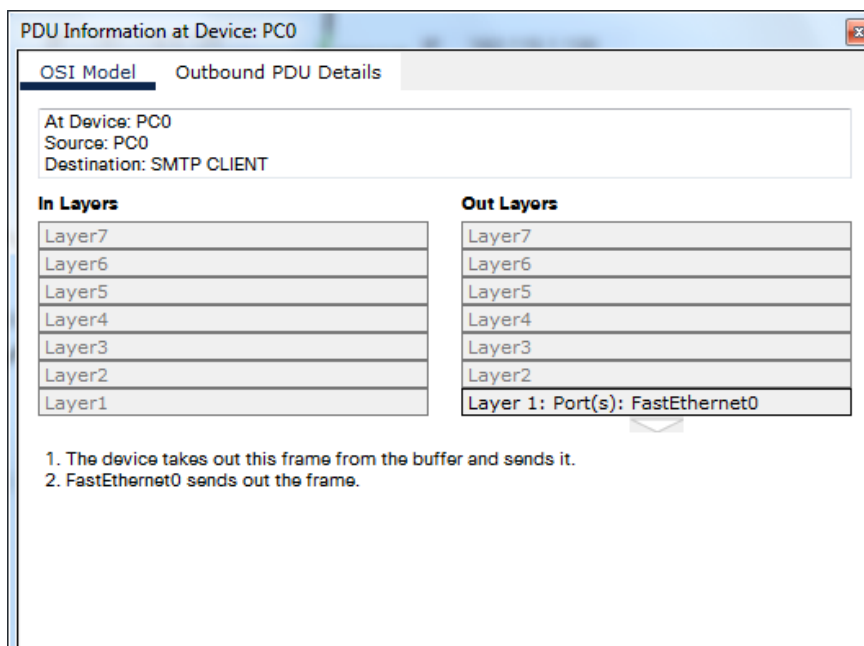


Рисунок 9.22 – Передача SMTP-повідомлення на фізичний рівень

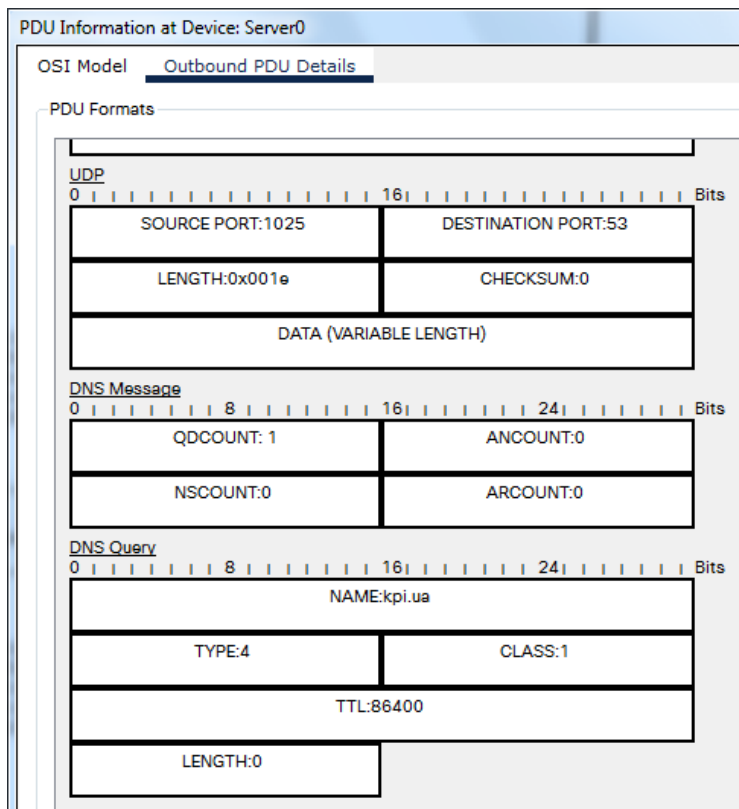


Рисунок 9.23 – Створення DNS-запиту до серверу Server1

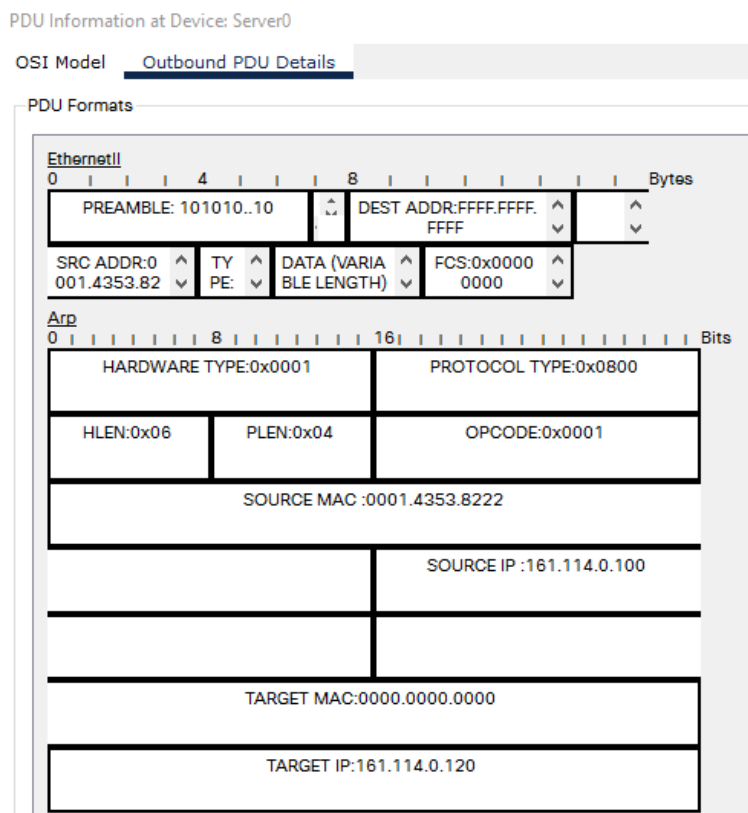


Рисунок 9.24 – Створення ARP-запиту до маршрутизатора Router0

На Server0 створюється TCP-пакет з встановленими бітами ACK+FIN до PC0.

Simulation Panel				
Event List				
Vis.	Time(sec)	Last Device	At Device	Type
	0.022	Switch1	Router0	TCP
	0.022	PC0	Switch1	TCP
	0.022	--	Router0	ARP
	0.023	Router0	Switch2	ARP
	0.023	Switch1	Server0	TCP
	0.024	Switch2	Server1	ARP
	0.024	Switch2	PC1	ARP
	0.025	Server1	Switch2	ARP
	0.026	Switch2	Router0	ARP
	0.317	--	Server0	TCP
	0.318	Server0	Switch1	TCP
	0.319	Switch1	Router0	TCP
	0.320	Router0	Switch2	TCP
	0.321	Switch2	Server1	TCP
	0.322	Server1	Switch2	TCP
	0.323	Switch2	Router0	TCP
	0.324	Router0	Switch1	TCP
	0.325	Switch1	Server0	TCP
	0.325	--	Server0	SMTP
	0.326	Server0	Switch1	TCP
	0.326	--	Server0	SMTP
	0.327	Server0	Switch1	SMTP
	0.327	Switch1	Router0	TCP
	0.328	Switch1	Router0	SMTP
	0.328	Router0	Switch2	TCP
	0.329	Router0	Switch2	SMTP
	0.329	Switch2	Server1	TCP
	0.330	Switch2	Server1	SMTP
	0.331	Server1	Switch2	SMTP
	0.332	Switch2	Router0	SMTP
	0.333	Router0	Switch1	SMTP
	0.334	Switch1	Server0	SMTP
	0.334	--	Server0	TCP
	0.335	Server0	Switch1	TCP
	0.336	Switch1	Router0	TCP
	0.337	Router0	Switch2	TCP
	0.338	Switch2	Server1	TCP
	0.339	Server1	Switch2	TCP
	0.340	Switch2	Router0	TCP
	0.341	Router0	Switch1	TCP
	0.342	Switch1	Server0	TCP
	0.343	Server0	Switch1	TCP
	0.344	Switch1	Router0	TCP
	0.345	Router0	Switch2	TCP
	0.346	Switch2	Server1	TCP
	1.830	--	Switch0	STP

Рисунок 9.25 – Передача поштового повідомлення. Фрагмент лістингу.

Комутатор Switch1 передає ARP-запит для визначення MAC-адреси вхідного інтерфейсу FastEthernet0/0 маршрутизатора Router0. Отримавши ARP-відповідь Server0 створює і надсилає в мережу TCP-пакет з бітом SYN для встановлення з'єднання з поштовим сервером (порт 25) Server1 з IP-адресою 162.115.1.100 (рис.9.25).

Отримавши цей пакет маршрутизатор за допомогою таблиці маршрутизації (рис.9.3) визначає, що мережа 162.115.0.0 підключена безпосередньо до його інтерфейсу FastEthernet0/1. Пакет пересилається на цей інтерфейс, а в мережу **kpi.ua** надсилається ARP-запит для визначення MAC-адреси поштового сервера з IP-адресою 162.115.1.100.

Після отримання ARP-відповіді, створюються всі умови для проходження TCP-пакету з SYN, що прямує від Server0 до Server1.

Натискаючи на кнопку "Capture/Forward", спостерігаємо, як відбувається триразове рукостискання: у відповідь на пакет з SYN сервер Server1 відправляє пакет з ACK+SYN, а потім Server0 відправляє серверу Server1 пакет з ACK. Процедура з'єднання поштового сервера відправника з поштовим сервером отримувача завершена.

Одночасно на поштовому сервері Server0 формується і передається в мережу SMTP-пакет.

Натискаючи на кнопку "Capture/Forward" спостерігаємо за проходженням SMTP-пакету до поштового сервера Server1. Після отримання SMTP-пакету Server1 надсилає серверу Server0 підтвердження про отримання, а Server0 відправляє серверу Server1 TCP-пакет з бітами FIN+ACK. Відбувається триразовий обмін службовими пакетами – процедура розірвання з'єднання.

3.4. Розглянемо в режимі симуляції Cisco Packet Tracer процес отримання поштового повідомлення користувачем *user2*:

- 1) Один клік по хосту 162.115.1.91;
- 2) Вибираємо на вкладці "Desktop" програму "E-mail";
- 3) Щоб отримати лист, натискаємо на кнопку "Receive".

На хості PC1 створюється DNS-запит DNS-серверу **kiev.ua** для визначення його IP-адреси(рис.9.26).

Натискаючи на кнопку "Capture/Forward" спостерігаємо за проходженням DNS-запиту до DNS-сервера Server1 та DNS-відповіді до хоста PC1.

Протокол DNS звертається до протоколу ARP для визначення MAC-адреси DNS-сервера(порт 53) з IP-адресою 162.115.1.100. Отримавши MAC-адресу, PC1 відправляє DNS-запит.

Simulation Panel				
Event List				
Vis.	Time(sec)	Last Device	At Device	Type
	0.000	--	PC1	DNS
	0.000	--	PC1	ARP
	0.001	PC1	Switch2	ARP
	0.002	Switch2	Server1	ARP
	0.002	Switch2	Router0	ARP
	0.003	Server1	Switch2	ARP
	0.004	Switch2	PC1	ARP
	0.004	--	PC1	DNS
	0.005	PC1	Switch2	DNS
	0.006	Switch2	Server1	DNS
	0.007	Server1	Switch2	DNS
	0.008	Switch2	PC1	DNS
	0.008	--	PC1	TCP
	0.009	PC1	Switch2	TCP
	0.010	Switch2	Server1	TCP
	0.011	Server1	Switch2	TCP
	0.012	Switch2	PC1	TCP
	0.012	--	PC1	POP3
	0.013	PC1	Switch2	TCP
	0.013	--	PC1	POP3
	0.014	PC1	Switch2	POP3
	0.014	Switch2	Server1	TCP
	0.015	Switch2	Server1	POP3
	0.016	Server1	Switch2	POP3
	0.017	Switch2	PC1	POP3
	0.017	--	PC1	TCP
	0.018	PC1	Switch2	TCP
	0.019	Switch2	Server1	TCP
	0.020	Server1	Switch2	TCP
	0.021	Switch2	PC1	TCP
	0.022	PC1	Switch2	TCP
	0.023	Switch2	Server1	TCP
	0.950	--	Switch0	STP

Рисунок 9.26 – Отримання поштового повідомлення. Лістинг подій.

Після отримання DNS-відповіді хост PC1 надсилає поштовому серверу Server1 TCP-пакет з бітом SYN для встановлення з'єднання.

Після завершення процедури з'єднання (триразовий обмін службовими пакетами) на хості PC1 створюється і відправляється поштовому серверу POP3-пакет, у відповідь на який поштовий сервер надсилає поштове повідомлення.

Після отримання поштового повідомлення хост PC1 ініціює процедуру розірвання з'єднання.

Приклад отриманого поштового повідомлення зображений на рисунку 9.27.

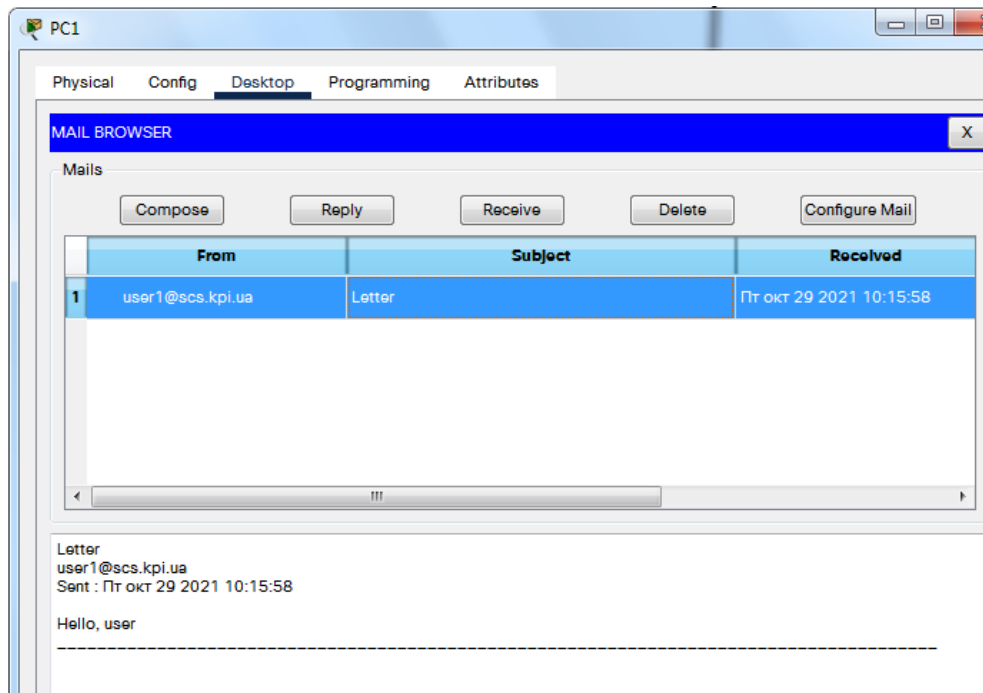


Рисунок 9.27 – Отримання поштового повідомлення користувачем *user2*

3.5. Поштові протоколи SMTP і POP3 взаємодіють за допомогою команд. Клієнту електронної пошти, щоб встановити з'єднання з сервером, відправити або отримати повідомлення, розірвати з'єднання, необхідно відправити серверу певні команди. Сервер виконує ці команди і формує відповіді. Ці відповіді містять цифровий код відповіді - успішно чи з помилкою опрацьована команда. Відповіді POP3-сервера також містять два типи повідомлень – успіх або помилка.

Завдання до лабораторної роботи

1. Побудуйте тестову мережу, приклад якої наведений на рисунку 9.1. Виконайте необхідні налаштування мережевим пристроям: комп'ютерам та маршрутизаторам.
2. Дослідіть роботу прикладних протоколів SMTP і POP3 та їхню взаємодію з мережевими протоколами TCP, UDP і ARP.
3. Самостійно дослідіть в режимі симуляції передачу поштового повідомлення від користувача *user1* до користувача *user3*. Зверніть увагу на процес пересилки поштового повідомлення у поштову скриньку користувача *user3*. Чим ця пересилка відрізняється від пересилки поштового повідомлення від користувача *user1* до користувача *user2*?
4. У звіті надайте пояснення причин утворення пакетів різних протоколів.

Лабораторна робота 10

Організація доступу до комп'ютерної мережі Технології VLAN та NAT

Мета роботи: навчити студентів створювати віртуальні комп'ютерні мережі на основі керованих комутаторів та маршрутизаторів, а також впроваджувати віртуальні мережі на базі запропонованої топології.

План виконання лабораторної роботи

1. У межах запропонованої топології (рис.10.1) створити та налаштувати три віртуальні мережі: VLAN2, VLAN3, VLAN4. Комп'ютери PC0, PC1, PC4 та PC6 помістити в VLAN2, комп'ютери PC2, PC3, PC6 та PC7 помістити в VLAN3, Web-сервер Server0 помістити в VLAN4.
2. Виконати мережеві налаштування комп'ютерів PC0 ÷ PC7 та сервера Server0. На сервері Server0 встановити службу HTTP.
3. Виконати налаштування маршрутизатора Router0 для забезпечення взаємодії комп'ютерів різних віртуальних мереж.
4. Виконати підключення створеної локальної мережі до зовнішнього сервера Server1, забезпечивши при цьому перетворення зовнішніх IP-адрес у внутрішні IP-адреси і внутрішніх IP-адрес у зовнішні IP-адреси.
5. Виконати налаштування маршрутизатора Router0 для забезпечення доступу до Web-сервера Server0 із зовнішньої мережі.

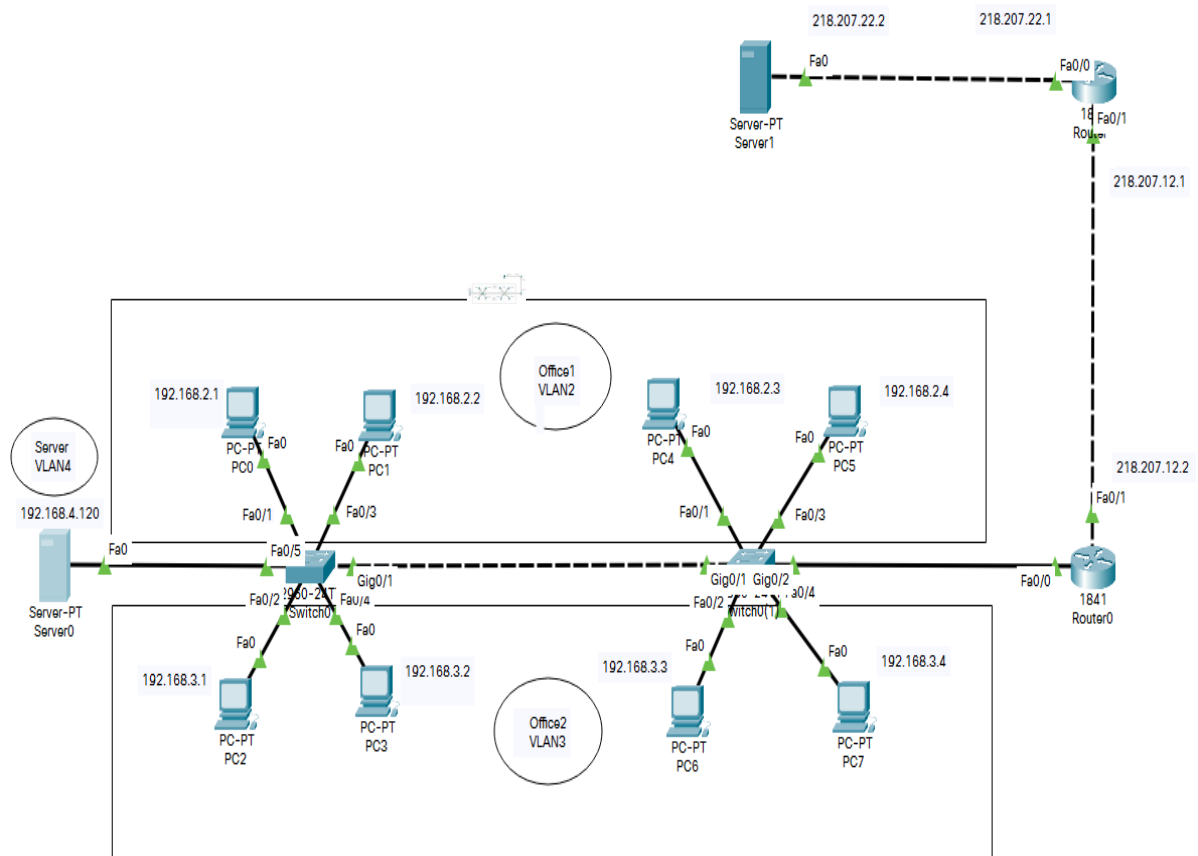


Рис.10.1 – Топологія досліджуваної мережі

1. Короткі теоретичні відомості

1.1. VLAN (Virtual Local Area Network)

VLAN (Virtual Local Area Network, віртуальна локальна мережа) — це технологія в маршрутизаторах і комутаторах, яка дозволяє на одному фізичному мережевому інтерфейсі (Ethernet, Wi-Fi-інтерфейсі) створювати кілька віртуальних локальних мереж. VLAN використовують для створення логічної топології мережі, яка не залежить від фізичної топології.

VLAN дозволяють:

- *Об'єднувати на канальному рівні в єдину мережу комп'ютери, які під'єднані до різних комутаторів.*

Допустимо, у нас є комп'ютери, які під'єднані до різних комутаторів і їх потрібно об'єднати в одну мережу(рис.10.2). Одні комп'ютери ми об'єднаємо у віртуальну локальну мережу VLAN1, а інші — в мережу VLAN2. Завдяки технології VLAN комп'ютери в кожній віртуальній мережі будуть працювати, ніби вони під'єднані до одного і того ж самого комутатора. Комп'ютери із різних віртуальних мереж VLAN1 і VLAN2 будуть недоступні один для одного.

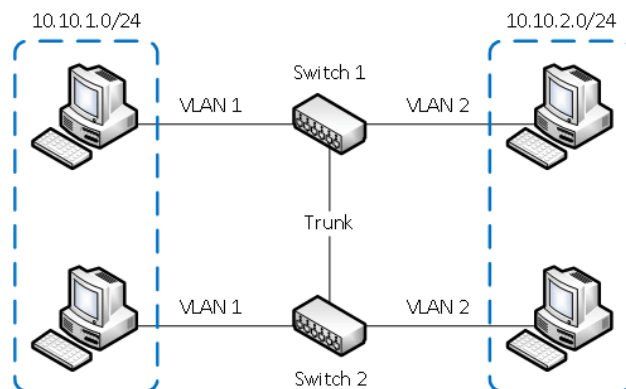


Рисунок 10.2 - Об'єднання комп'ютерів у VLAN

- *Розділяти в різні підмережі комп'ютери, які під'єднані до одного комутатора.*

На рисунку 10.3 комп'ютери фізично під'єднані до одного комутатора, але розділені між різними віртуальними мережами VLAN1 і VLAN2. Комп'ютери із різних віртуальних підмереж будуть невидимі один для одного.

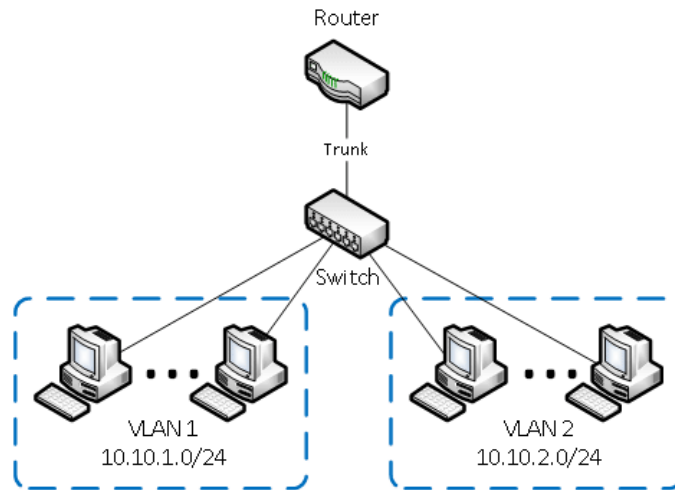


Рисунок 10.3 - Розділення комп'ютерів між різними VLAN

- *Розділяти гостьові Wi-Fi мережі і Wi-Fi мережі підприємства.*

На рисунку 10.4 до комутатора під'єднана фізично одна Wi-Fi точка доступу. На точці створені дві віртуальні Wi-Fi точки, які мають назву Gest і Office. До Gest будуть підключатися по Wi-Fi гостьові ноутбуки для доступу до інтернету, а до Office — ноутбуки підприємства. Для безпеки необхідно, щоб гостьові ноутбуки не мали доступу до мережі підприємства. Для цього комп'ютери підприємства і віртуальна Wi-Fi точка Office об'єднані у віртуальну локальну мережу VLAN1, а гостьові ноутбуки будуть знаходитися в віртуальній мережі VLAN2. Гостьові ноутбуки із мережі VLAN2 не будуть мати доступу до мережі підприємства VLAN1.

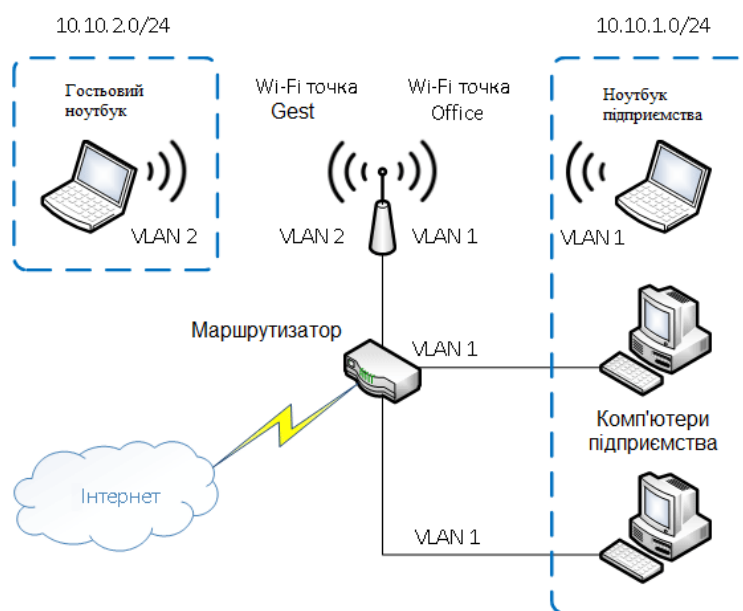


Рисунок 10.4 - Розділення гостьової Wi-Fi мережі і Wi-Fi мережі підприємства

Переваги використання VLAN

- **Гнучке розділення мережевих пристроїв на групи.**
Як правило, одному VLAN відповідає одна підмережа. Комп'ютери, які знаходяться в різних VLAN, будуть ізольовані один від одного. Також можна об'єднувати в одну віртуальну мережу комп'ютери, підключені до різних комутаторів.
- **Зменшення ширококомовного трафіку в мережі.**
Кожен VLAN являє собою окремий ширококомовний домен. Широкомовний трафік не буде транслюватися між різними VLAN. Якщо на різних комутаторах налаштувати один і той же VLAN, то порти різних комутаторів будуть створювати один ширококомовний домен.
- **Підвищення безпеки і керованості в мережі.**
У мережі, яка поділена на віртуальні підмережі, зручно застосовувати політики і правила безпеки для кожного VLAN. Політика буде застосована до всієї підмережі, а не до окремого пристрою.
- **Зменшення кількості обладнання і мережевого кабелю.**
Для створення нової віртуальної локальної мережі не потрібне придбання комутатора і прокладка мережевого кабелю. Однак є потреба в використанні більш дорогих керованих комутаторів, що підтримують VLAN.

1.2. Ідентифікація VLAN. Тегований і нетегований VLAN

Кожний VLAN-домен ідентифікується за допомогою числової мітки – *vlan id*. При використанні стандарту Ethernet II, 802.1Q вставляє тег перед полем «Тип протоколу». Згідно міжнародному стандарту 802.1Q *vlan id* може приймати значення в діапазоні від 0 до 4095. VLAN з номером 1 резервується як VLAN по замовчуванню. Трафік, який передається в цьому VLAN, не тегується (рис 10.5). Також зарезервовані такі значення *vlan id*, як 1002 і 1004 для FDDI-мереж, 1003 і 1005 для мереж Token Ring.

Типи портів на комутаторах і тегування

Порти комутатора, які підтримують VLAN, можна розділити на дві групи:

1. Нетеговані порти (або порти доступу, *access-порти* в термінології Cisco);
2. Теговані порти (або транкові порти, *trunk-порти* в термінології Cisco).

Перший тип використовується при підключенні кінцевих хостів, таких як ПК, ір-телефони, сервери і т.д. Другий тип портів призначений для такого

з'єднання між комутаторами, коли через один порт комутатора передається трафік кількох VLAN'ів.

Якщо через один транковий порт передається трафік кількох VLAN'ів, комутатору необхідно вказати, який пакет даних якому VLAN призначений на іншому кінці з'єднання (рис.10.5). Для цього використовується механізм тегування пакетів даних за допомогою VLAN-тегів. VLAN-тег вставляється в Ethernet-кадр, додаючи до нього необхідну інформацію (рис 10.6).

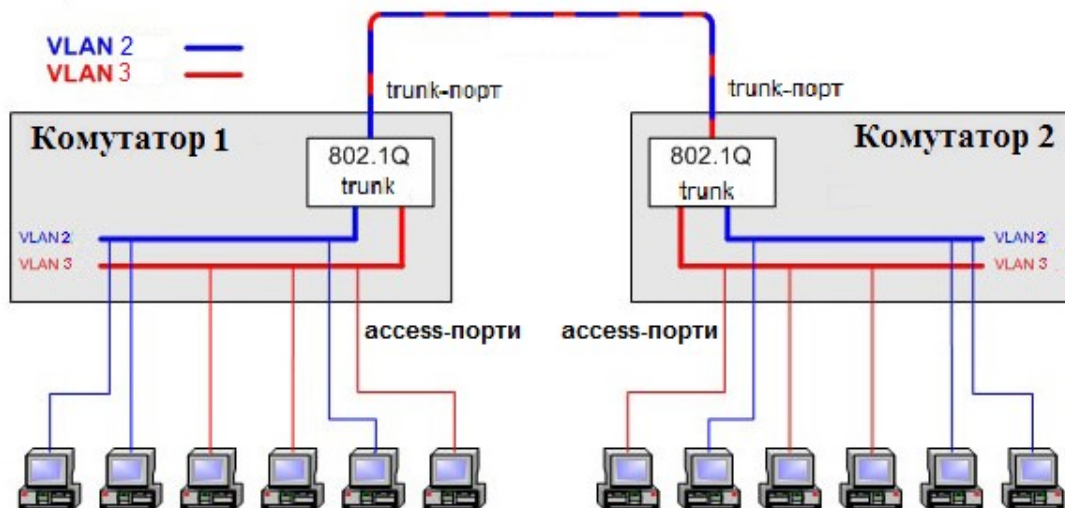


Рисунок 10.5 - Схема об'єднання двох VLAN'ів

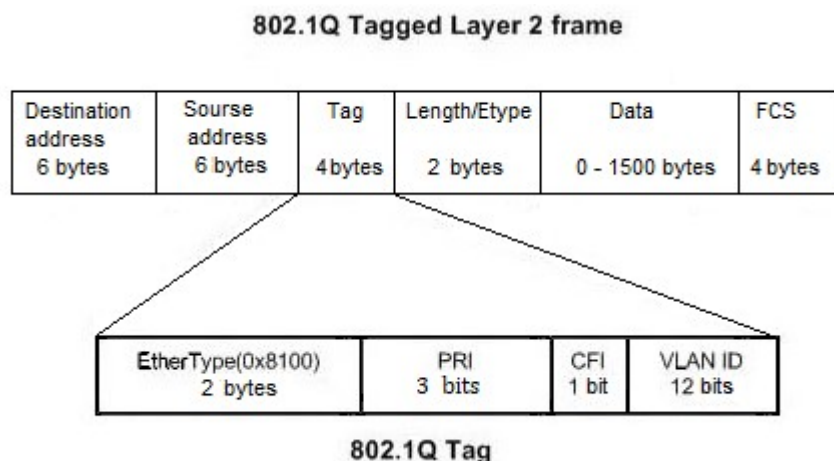


Рисунок 10.6 - Структура тегового кадру Ethernet

Стандарт 802.1Q визначає, що тег містить таку інформацію, як *vlan id* і деякі інші дані, обумовлені цим стандартом. Таким чином, теговані кадри містять інформацію про належність до VLAN, в той час як нетеговані такої інформації не мають. Типовий приклад використання тегування – це з'єднання між маршрутизатором і комутатором, до якого підключені кілька комп'ютерів із різних VLAN.

В такому випадку комутатор налаштований на використання кількох VLAN, а маршрутизатор виконує всі функції по маршрутизації між різними підмережами, які створені VLAN'ами. Схематично така взаємодія зображена на рисунку 10.7.

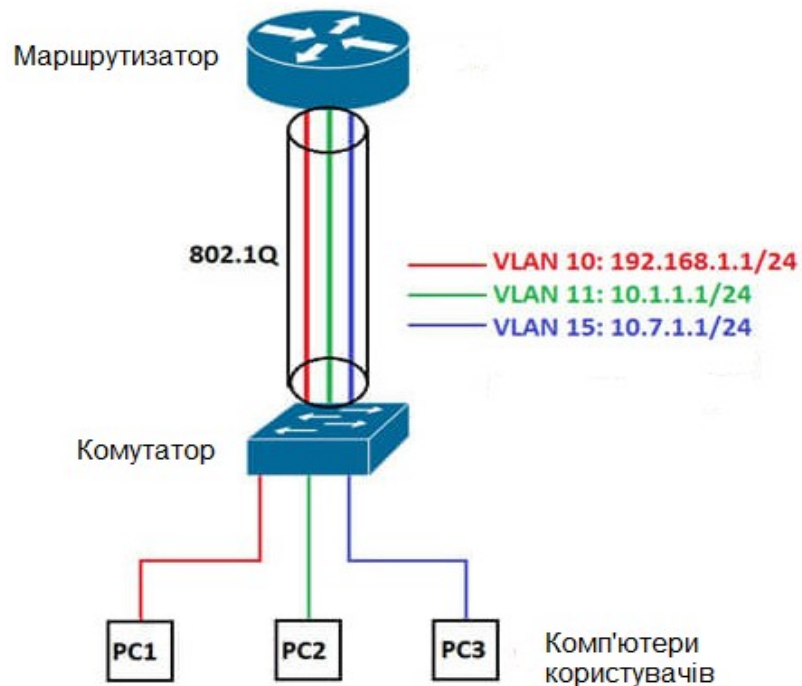


Рисунок 10.7 - Утворення теґованого трафіку між комутатором і маршрутизатором

Для того щоб маршрутизатор міг передавати трафік із одного VLAN в інший (із однієї мережі в іншу), необхідно щоб в кожній мережі у нього був інтерфейс. Для того щоб не виділяти під мережу кожного VLAN окремий фізичний інтерфейс, створюються логічні підінтерфейси на фізичному інтерфейсі маршрутизатора для кожного VLAN.

На комутаторі порт, який з'єднаний з маршрутизатором, має бути налаштований як теґований порт (в термінах Cisco - транк).

1.3.Технологія перетворення мережових адрес NAT

Однією із проблем у розвитку мереж є обмежена кількість існуючих IPv4 адрес – їх біля 4,3 мільярди. У зв'язку з поширенням інтернету і значним збільшенням кількості користувачів, стало зрозуміло, що цього недостатньо. Виникла необхідність в інструменті, який міг би вирішити цю проблему. Одним із таких інструментів стала технологія NAT (Network Address Translation).

При проектуванні мереж зазвичай застосовують приватні IP-адреси 10.0.0.0/8, 172.16.0.0/12 і 192.168.0.0/16. Їх використовують всередині локальної мережі для підтримки локальної взаємодії мережових пристроїв, а не для маршрутизації по Інтернет. Для того щоб пристрій з адресою IPv4 міг звернутися до інших пристроїв через Інтернет, його приватна адреса має бути перетворена у публічну (зовнішню). Таке перетворення - це головне, що робить NAT, спеціальний механізм перетворення приватних адрес у публічні (рис.10.8).

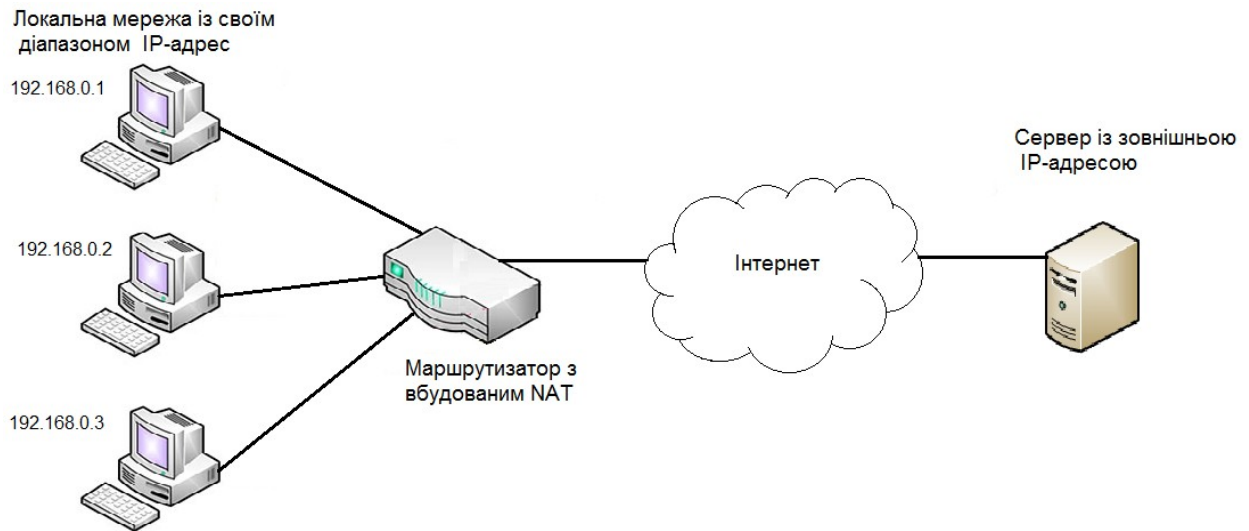


Рисунок 10.8 - Загальна схема виконання NAT

Маршрутизатори з підтримкою NAT можуть бути налаштовані з одним або з кількома зовнішніми IPv4-адресами. Ці зовнішні адреси називаються пулом NAT. Коли пристрій із внутрішньої мережі відправляє трафік у зовнішню мережу, то маршрутизатор з підтримкою NAT перетворює внутрішню IPv4-адресу пристрою на зовнішню адресу із пулу NAT. Для зовнішніх пристроїв весь трафік, що входить і виходить із мережі, має зовнішню IPv4-адресу.

Таким чином NAT дозволяє забезпечити пряму взаємодію хостів в Інтернеті з маршрутизатором, який підтримує NAT, а не з хостом в локальній мережі.

При використанні NAT, адреси IPv4 мають різні визначення в залежності від того де вони знаходяться, у внутрішній чи у зовнішній мережі, і чи є трафік вхідним чи вихідним:

- **Внутрішня адреса (Inside address)** - адреса пристрою, яка транслюється NAT;
- **Зовнішня адреса (Outside address)** - адреса пристрою призначення;
- **Локальна адреса (Local address)** – будь-яка адреса, яка відображається у внутрішній мережі;
- **Глобальна адреса (Global address)** - це будь-яка адреса, яка відображається у зовнішній мережі.

На рисунку 10.9 комп'ютер має внутрішню локальну (**Inside local**) адресу 192.168.1.5 і з його точки зору Web-сервер має зовнішню (**outside**) адресу 208.141.17.4. Коли з цього комп'ютера відправляються пакети на глобальну адресу Web-сервера, внутрішня локальна (**Inside local**) адреса комп'ютера транслюється в 208.141.16.5 (**inside global**).

На рисунку 10.9 показано, як трафік відправляється із внутрішнього комп'ютера на зовнішній Web-сервер, через маршрутизатор з підтримкою NAT, і повертається у зворотному напрямку.

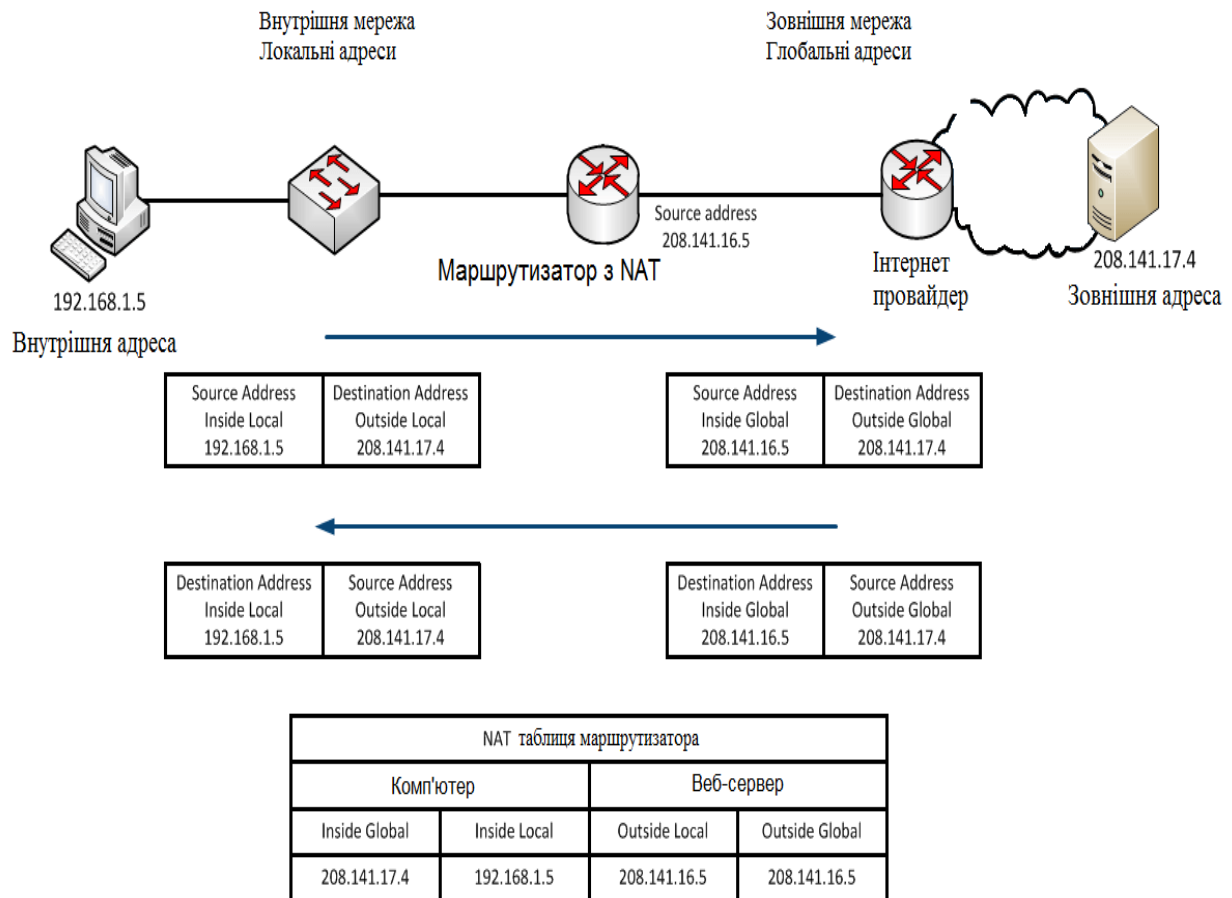


Рисунок 10.9 - Схема виконання перетворення адрес

Внутрішня локальна адреса (**Inside local address**) - адреса джерела, яку видно із внутрішньої мережі. На рисунку адресу 192.168.1.5 має комп'ютер – це і є його внутрішня локальна адреса.

Внутрішня глобальна адреса (**Inside global address**) - адреса джерела, яку видно із зовнішньої мережі. На рисунку, коли трафік з комп'ютера відправляється на Web-сервер за адресою 208.141.17.4, маршрутизатор перетворює внутрішню локальну адресу (**Inside local address**) на внутрішню глобальну адресу (**Inside global address**). У нашому випадку маршрутизатор замінює адресу джерела IPv4 з 192.168.1.5 на 208.141.16.5.

Зовнішня глобальна адреса (**Outside global address**) - адреса адресата, як її видно із зовнішньої мережі. Це глобальна IPv4-адреса, призначена хосту в Інтернеті. На схемі Web-сервер доступний за адресою 208.141.17.4. Часто зовнішні локальні і зовнішні глобальні адреси збігаються.

Зовнішня локальна адреса (**Outside local address**) - адреса отримувача, яку видно із внутрішньої мережі. У нашому прикладі комп'ютер відправляє трафік на Web-сервер за адресою 208.141.17.4.

Розглянемо весь шлях проходження пакету. Комп'ютер з адресою 192.168.1.5 намагається встановити зв'язок з Web-сервером 208.141.17.4. Коли пакет надходить на маршрутизатор з підтримкою NAT, він зчитує IPv4 адресу призначення, щоб визначити, чи відповідає пакет критеріям, вказаним для перетворення. У нашому прикладі початкова адреса відповідає критеріям і перетворюється з 192.168.1.5 (**Inside local address**) на 208.141.16.5 (**Inside global address**). Маршрутизатор додає це зіставлення локальної і глобальної адрес у таблицю NAT і відправляє пакет з перетвореною адресою джерела в пункт призначення. Web-сервер відповідає пакетом, який адресований внутрішній глобальній адресі комп'ютера (208.141.16.5). Маршрутизатор отримує пакет з адресою призначення 208.141.16.5 і перевіряє таблицю NAT, в якій знаходить запис для цього зіставлення. Він використовує цю інформацію і перетворює назад внутрішню глобальну адресу (208.141.16.5) на внутрішню локальну адресу (192.168.1.5), і пакет надсилається в напрямку комп'ютера.

Типи NAT

По способу зіставлення адрес, розрізняють такі типи трансляції NAT:

- **Статична адресна трансляція (Static NAT)** - зіставлення адрес один до одного між локальними і глобальними адресами;
- **Динамічна адресна трансляція (Dynamic NAT)** - зіставлення багатьох локальних і багатьох глобальних адрес;
- **Port Address Translation (NAT)** – багатоадресне зіставлення адрес між локальними і глобальними адресами з використанням портів. Також цей метод відомий як **NAT Overload** (або **PAT - Port Address Translation**). PAT дозволяє маршрутизатору використовувати одну зовнішню глобальну адресу для багатьох внутрішніх локальних адрес. Загальна кількість внутрішніх адрес, які можуть бути переведені на одну зовнішню адресу, теоретично може складати 65 536 на кожну IP-адресу. Проте на практиці число внутрішніх адрес, яким може бути призначена одна IP-адреса, складає біля 4000.

2. Виконання лабораторної роботи

У програмі Packet Tracer створюємо модель комп'ютерної мережі відповідно до топології, що зображена на рис. 10.1. У роботі використовуємо комутатори Cisco 2960 і маршрутизатори Cisco 1841.

Всі налаштування виконуємо користуючись інтерфейсом командного рядка (CLI).

1. Створюємо два VLAN з іменами Office1 та Office2 з одним комутатором і налагоджуємо на інтерфейсах порти *access*, до яких потім будуть підключені комп'ютери (рис 10.10).

```

Switch>enable
Switch#conf t
Switch(config)#vlan 2
Switch(config-vlan)#name Office1
Switch(config-vlan)#exit
Switch(config)#int fa0/1
Switch(config-if)#switchport mode access
Switch(config-if)#switchport access vlan 2
Switch(config-if)#exit
Switch(config)#int fa0/3

```

переходимо в привілейований режим
 переходимо в режим глобальної конфігурації
 створюємо VLAN 2
 надаємо ім'я створеному VLAN
 переходимо до конфігурування інтерфейсу
 задаємо режим роботи *access* інтерфейсу
 поміщаємо інтерфейс в VLAN 2

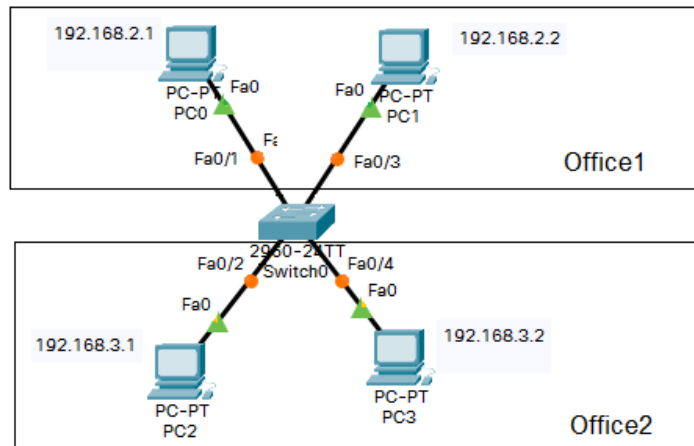


Рисунок 10.10 - Схема VLAN з одним комутатором

```

Switch(config-if)#switchport mode access
Switch(config-if)#switchport access vlan 2
Switch(config-if)#exit
Switch(config)#vlan 3
Switch(config-vlan)#name Office2
Switch(config-vlan)#exit
Switch(config)#int fa0/2
Switch(config-if)#switchport mode access
Switch(config-if)#switchport access vlan 3
Switch(config-if)#exit
Switch(config)#int fa0/4
Switch(config-if)#switchport mode access
Switch(config-if)#switchport access vlan 3
Switch(config-if)#end

```

Перевіримо виконані налаштування:

```
Switch#show vlan brief
```

показати створені VLAN (скорочено)


```
Switch#sh vlan brief
```

VLAN Name	Status	Ports
1 default	active	Fa0/5, Fa0/6, Fa0/7, Fa0/8 Fa0/9, Fa0/10, Fa0/11, Fa0/12 Fa0/13, Fa0/14, Fa0/15, Fa0/16 Fa0/17, Fa0/18, Fa0/19, Fa0/20 Fa0/21, Fa0/22, Fa0/23, Fa0/24 Gig0/1, Gig0/2
2 Office1	active	Fa0/1, Fa0/3
3 Office2	active	Fa0/2, Fa0/4
1002 fddi-default	active	
1003 token-ring-default	active	
1004 fddinet-default	active	
1005 trnet-default	active	

Рисунок 10.11- Створені VLAN

Отже, VLAN1 та VLAN2 створені.

- Виконаємо мережеві налаштування комп'ютерів PC0÷PC3 (таблиця 10.1).

Таблиця 10.1. Схема адресації мережі

Комп'ютер	IP-адреса	Маска
PC0	192.168.2.1	255.255.255.0
PC1	192.168.2.2	255.255.255.0
PC2	192.168.3.1	255.255.255.0
PC3	192.168.3.2	255.255.255.0

- Перевіримо доступність комп'ютера, який розташований в тому ж VLAN, що і комп'ютер PC0. Із командного рядка PC0 виконаємо команду **ping**:

C:\>ping 192.168.2.2

```
Packet Tracer PC Command Line 1.0
C:\>ping 192.168.2.2

Pinging 192.168.2.2 with 32 bytes of data:

Reply from 192.168.2.2: bytes=32 time<1ms TTL=128
Reply from 192.168.2.2: bytes=32 time<1ms TTL=128
Reply from 192.168.2.2: bytes=32 time=1ms TTL=128
Reply from 192.168.2.2: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.2.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

- Перевіримо доступність комп'ютера, який розташований в іншому VLAN. Із командного рядка PC0 виконаємо команду **ping**:

C:\>ping 192.168.3.2

Спостерігаємо, що комп'ютер з іншого VLAN недоступний.

```

C:\>ping 192.168.3.2

Pinging 192.168.3.2 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.3.2:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

```

5. Перевіримо тепер таблицю комутації:
Switch#show mac address-table

```

Switch#sh mac address-table
      Mac Address Table
-----
Vlan    Mac Address      Type    Ports
----    -
Switch#
Switch#sh mac address-table
      Mac Address Table
-----
Vlan    Mac Address      Type    Ports
----    -
      2    0050.0f14.6585    DYNAMIC Fa0/3
      2    0060.5c03.e2e7    DYNAMIC Fa0/1

Switch#

```

6. Додамо до досліджуваної схеми ще кілька сегментів: комутатор Switch0(1) з комп'ютерами PC4÷ PC7 та сервер Server0. Сервер Server0 виділяємо в окрему підмережу VLAN4 (рис.10.12).

Додавання сегменту мережі з комутатором Switch0(1) та комп'ютерами PC4÷ PC7 виконуємо методом копіювання і вставки раніше вже налаштованого сегменту мережі з комутатором Switch0 та комп'ютерами PC0 ÷ PC3.

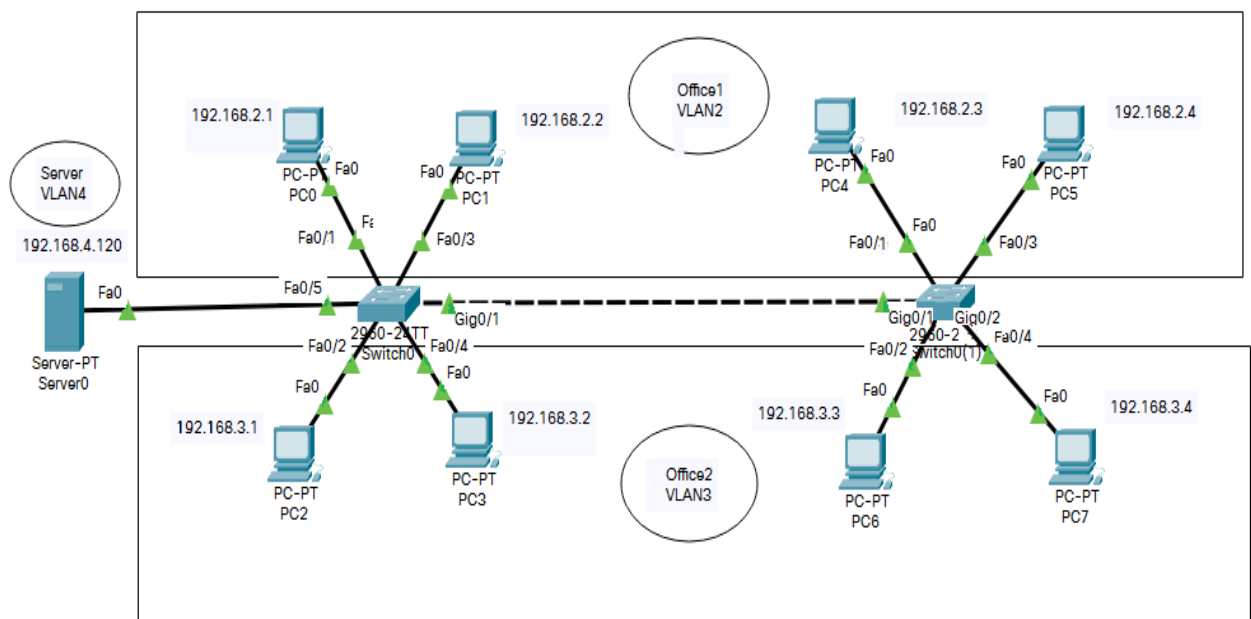


Рисунок 10.12 - Схема VLAN з двома комутаторами

Комутатори Switch0 та Switch0(1) з'єднаємо через швидкісні порти Gigabit, а сервер Server0 із VLAN4 підключимо до порту fa0/5 комутатора Switch0. Виконаємо налаштування на комутаторі Switch0. Спочатку до інтерфейсу fa0/5 підключимо Server0:

```
Switch#conf t
Switch(config)#vlan 4           створюємо VLAN 4
Switch(config-vlan)#name Server надаємо ім'я створеному VLAN
Switch(config-vlan)#exit
Switch(config)#int fa0/5
Switch(config-if)#switchport mode access
Switch(config-if)#switchport access vlan 4
Switch(config-if)#end
```

Збережемо конфігурацію:

```
Switch#wr mem
```

Тепер налаштуємо гігабітний порт gig0/1. Через цей порт буде проходити трафік усіх підмереж: VLAN2, VLAN3 та VLAN4. В термінології Cisco такі порти називаються магістральними або транковими (trunk).

```
Switch#conf t
Switch(config)#int gig0/1
Switch(config-if)#switchport mode trunk
Switch(config-if)#switchport trunk allowed vlan 2,3,4
Switch(config-if)#end
Switch#write mem
```

Такі ж налаштування виконаємо на порту gig0/1 комутатора Switch0(1):

```
Switch#conf t
Switch(config)#vlan 4
Switch(config)#exit
Switch(config)#int gig0/1
Switch(config-if)#switchport mode trunk
Switch(config-if)#switchport trunk allowed vlan 2,3,4
Switch(config-if)#end
Switch#wr mem
```

7. Виконаємо мережеві налаштування комп'ютерів PC4÷PC7 та сервера Server0 (таблиця 10.2).

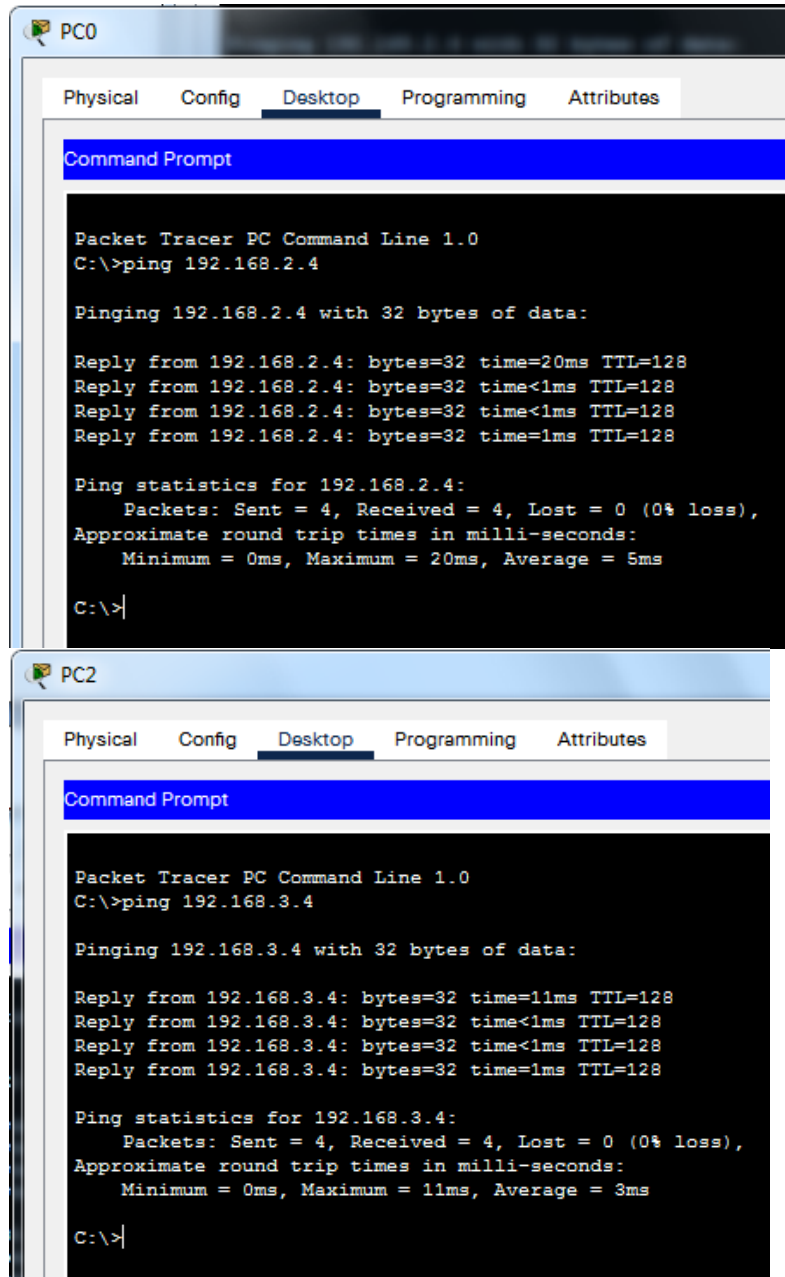
Таблиця 10.2. Схема адресації мережі

Комп'ютер	IP-адреса	Маска
PC4	192.168.2.3	255.255.255.0
PC5	192.168.2.4	255.255.255.0
PC6	192.168.3.3	255.255.255.0
PC7	192.168.3.4	255.255.255.0
Server0	192.168.4.120	255.255.255.0

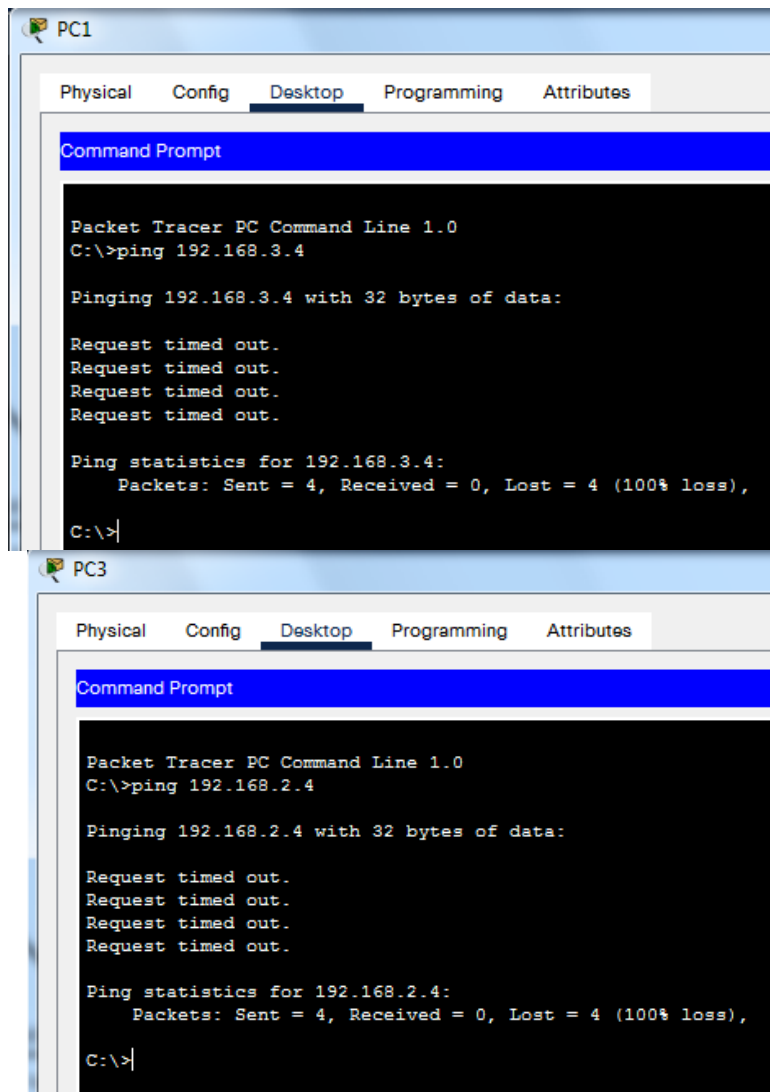
8. Налаштування портів fa0/1, fa0/2, fa0/3 і fa0/4 комутатора Switch0(1)

виконується аналогічно налаштуванням відповідних інтерфейсів комутатора Switch0.

9. Отже, створена схема складається із трьох VLAN'ів. Доступність комп'ютерів, які входять до складу одного VLAN перевіряємо за допомогою утиліти **ping**.



10. За допомогою утиліти **ping** можна впевнитися, що комп'ютери, які знаходяться у різних підмережах (VLAN) не можуть обмінюватися пакетами. Для їхньої взаємодії потрібний пристрій, що працює на мережевому рівні (маршрутизатор).



11. Щоб забезпечити доступ комп'ютерів різних VLAN один до одного, додамо до нашої схеми маршрутизатор Cisco 1841 (рис.10.13).

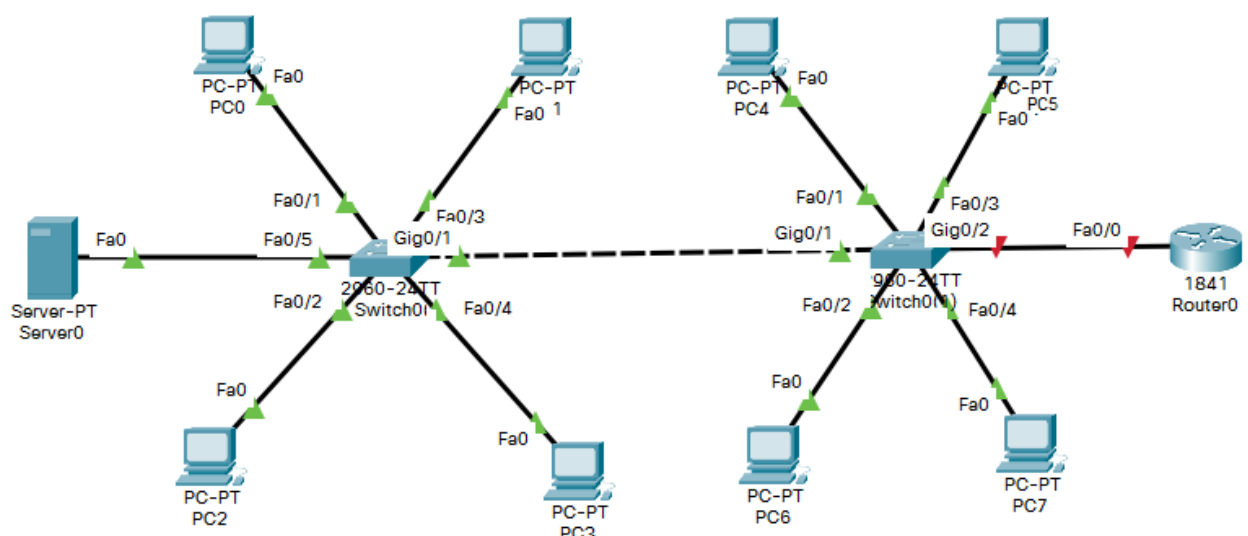


Рисунок 10.13 - Схема VLAN з двома комутаторами і маршрутизатором

12. Виконуємо налаштування інтерфейсу gig0/2 на комутаторі Switch0(1).

```
Switch>enable
Switch#conf t
Switch(config)#int gig0/2
Switch(config-if)#switchport mode trunk
Switch(config-if)#switchport trunk allowed vlan 2,3,4
Switch(config-if)#end
Switch#wr mem
```

13. Виконуємо налаштування інтерфейсу fa0/0 маршрутизатора Router0. Переходимо на маршрутизатор. По замовчуванню всі інтерфейси маршрутизатора відключені, тому спочатку потрібно включити інтерфейс fa0/0 командою **no shutdown** (скорочено **no shut**).

Через інтерфейс fa0/0 маршрутизатора проходить трафік VLAN2, VLAN3 і VLAN4. Тому на інтерфейсі fa0/0 потрібно створити три підінтерфейси (subinterfaces).

```
Router>enable
Router#conf t
Router#(config)#int fa0/0
Router#(config-if)#no shut
Router# (config-if)#exit
Router# (config)#int fa0/0.2
Router# (config-subif)#encapsulation dot1Q 2
Router# (config-subif)#ip address 192.168.2.10 255.255.255.0
Router# (config-subif)#no shut
Router# (config-subif)#exit
Router(config)#int fa0/0.3
Router(config-subif)#encapsulation dot1Q 3
Router(config-subif)#ip address 192.168.3.10 255.255.255.0
Router(config-subif)#no shut
Router(config-subif)#exit
Router(config)#int fa0/0.4
Router(config-subif)#encapsulation dot1Q 4
Router(config-subif)#ip address 192.168.4.10 255.255.255.0
Router(config-subif)#no shut
Router(config-subif)#end
Router#wr mem
```

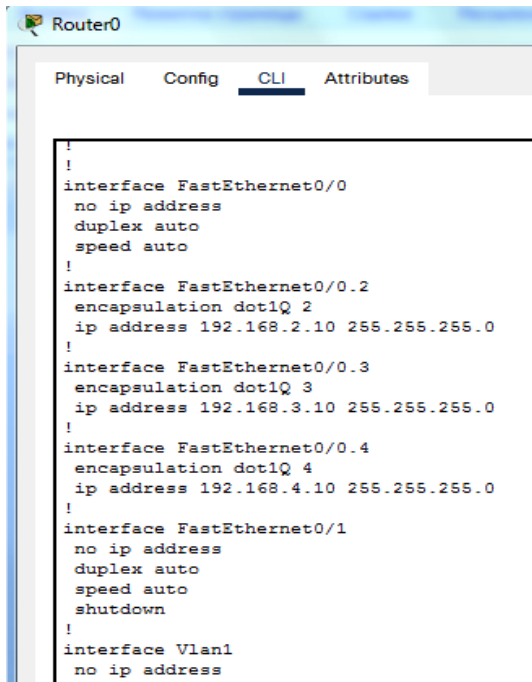
створення підінтерфейсу
підінтерфейс для VLAN 2
адреса VLAN 2 і маска

створення підінтерфейсу
підінтерфейс для VLAN 3
адреса VLAN 3 і маска

створення підінтерфейсу
підінтерфейс для VLAN 4
адреса VLAN 4 і маска

14. Перевіряємо виконані налаштування.

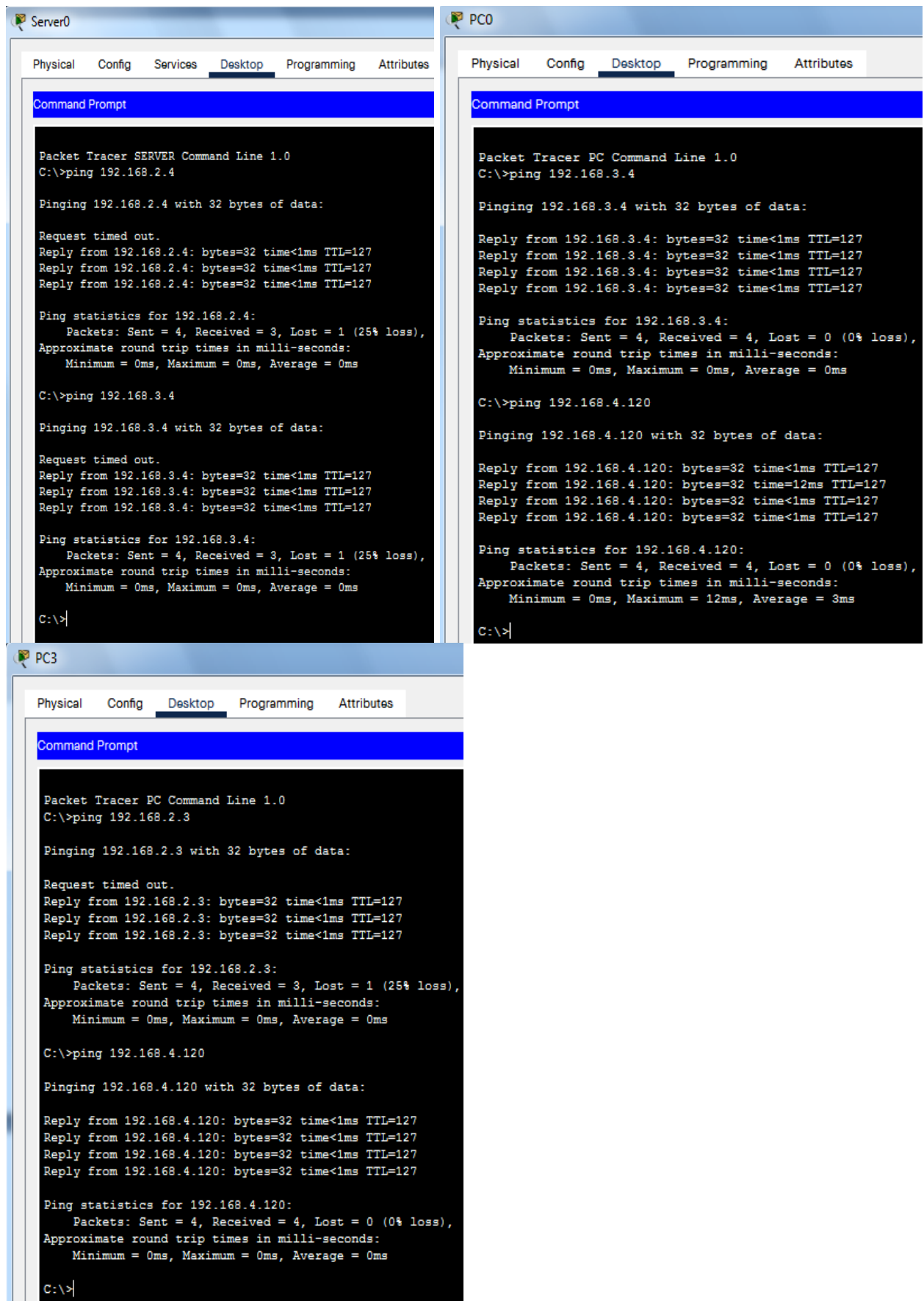
Router# show running-config (або **sh run**) вивести поточну конфігурацію



15. Тепер потрібно повернутися до мережевих налаштувань сервера Server0 і комп'ютерів PC0÷PC7 і вказати в налаштуваннях IP-адресу шлюзу по замовчуванню – це IP-адреси відповідних підінтерфейсів маршрутизатора Router0:

для хостів у VLAN2 – 192.168.2.10
 для хостів у VLAN3 – 192.168.3.10
 для хостів у VLAN4 – 192.168.4.10

16. За допомогою утиліти **ping** знову перевіряємо доступність комп'ютерів, які входять до складу різних VLAN і впевнюємось, що комп'ютери всіх VLAN доступні один для одного.



17. Тепер змодельюємо ситуацію підключення створеної локальної мережі до зовнішнього сервера Server1 в Інтернеті за допомогою маршрутизатора Router1, який має публічну IP-адресу 218.207.22.2.

Для такого підключення нам виділена публічна IP-адреса 218.207.12.2 (рис 10.14).

18. Виконаємо налаштування маршрутизатора провайдера Router1, який має два інтерфейси і відповідно дві IP-адреси: зовнішню fa0/0 218.207.22.1, яка звернена в напрямку зовнішнього сервера провайдера Server1, і внутрішню fa0/1 218.207.12.1, яка звернена до маршрутизатора Router0.

```
Router>enable
Router#conf t
Router(config)#int fa0/0
Router(config-if)#ip address 218.207.22.1 255.255.255.252
Router(config-if)#no shut
Router(config-if)#exit
Router(config)#int fa0/1
Router(config-if)# ip address 218.207.12.1 255.255.255.252
Router(config-if)#no shut
Router(config-if)#end
Router#wr mem
```

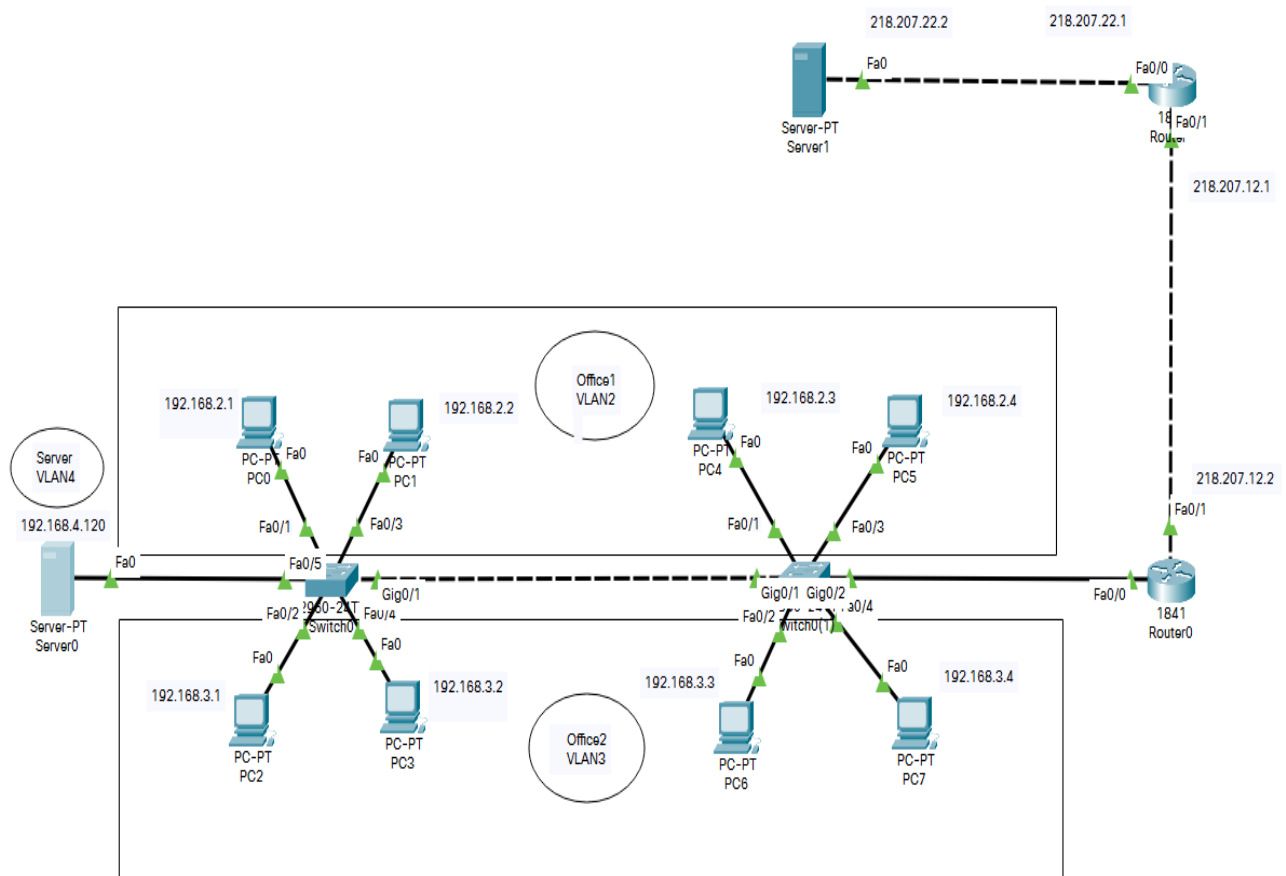


Рисунок 10.14 - Підключення локальної мережі до Інтернет

19. Виконаємо налаштування інтерфейсу fa0/1 маршрутизатора Router0:

```
Router>enable
```

```

Router# conf t
Router(config)#int fa0/1
Router(config-if)#ip address 218.207.12.2 255.255.255.252
Router(config-if)#no shut
Router(config-if)#exit
Router(config)#ip route 0.0.0.0 0.0.0.0 218.207.12.1      шлюз по замовчуванню
Router(config)#end
Router#wr mem

```

20. Виконаємо мережеві налаштування зовнішнього сервера Server1:

IP-адреса	218.207.22.2
Маска	255.255.255.252
IP-адреса шлюзу	218.207.22.1

21. Перевіряємо зв'язок Router0 з провайдером (рис.10.15):

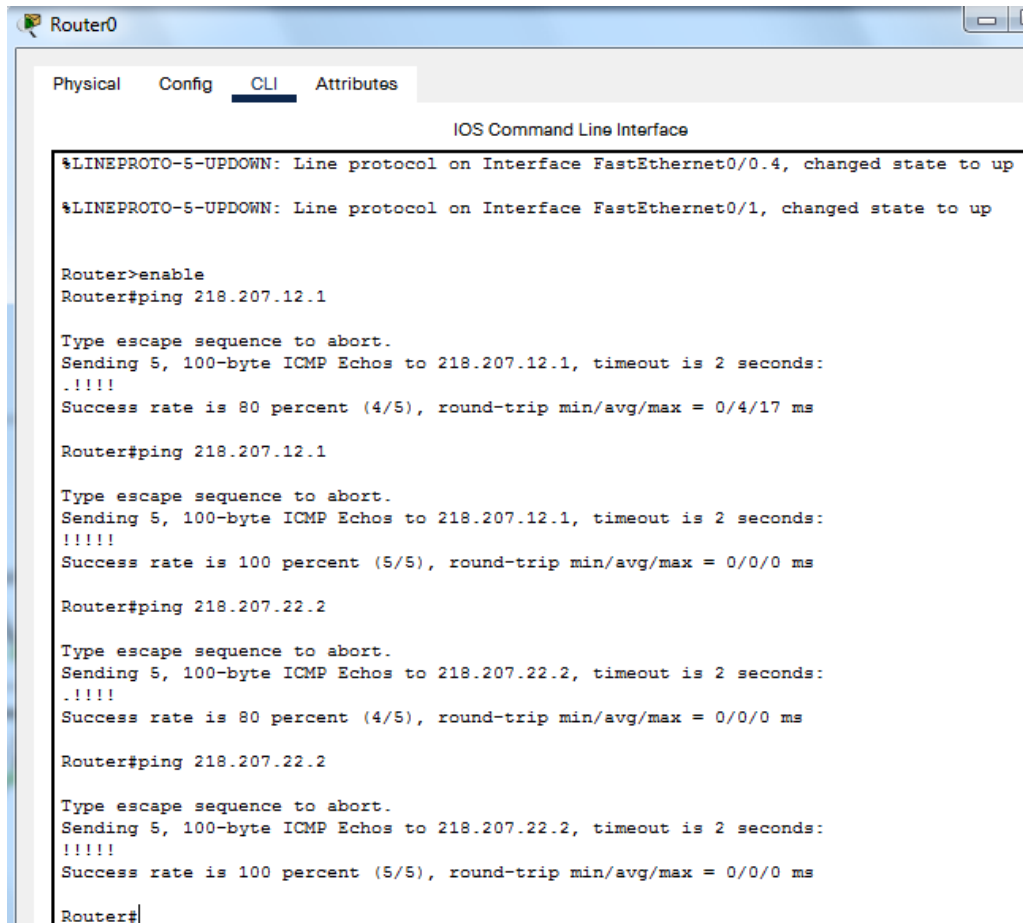
```
Router#ping 218.207.12.1
```

22. Перевіряємо доступність сервера Server1 (рис.10.15):

```
Router#ping 218.207.22.2
```

Якщо інтерфейси не запрацювали, то на маршрутизаторі Router1 виконуємо команду **no shutdown** на інтерфейсі fa0/0:

```
Router(config-if)#no shut
```



```

Router0
Physical  Config  CLI  Attributes
IOS Command Line Interface

%LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/0.4, changed state to up
%LINEPROTO-5-UPDOWN: Line protocol on Interface FastEthernet0/1, changed state to up

Router>enable
Router#ping 218.207.12.1

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 218.207.12.1, timeout is 2 seconds:
!!!!
Success rate is 80 percent (4/5), round-trip min/avg/max = 0/4/17 ms

Router#ping 218.207.12.1

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 218.207.12.1, timeout is 2 seconds:
!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 0/0/0 ms

Router#ping 218.207.22.2

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 218.207.22.2, timeout is 2 seconds:
!!!!
Success rate is 80 percent (4/5), round-trip min/avg/max = 0/0/0 ms

Router#ping 218.207.22.2

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 218.207.22.2, timeout is 2 seconds:
!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 0/0/0 ms

Router#

```

Рисунок 10.15 – Перевірка доступу від маршрутизатора до провайдера і публічного сервера

Налаштування NAT (Network Address Translation)

23. Тепер спробуємо зв'язатися з публічним сервером Server1 з локального комп'ютера (наприклад, з Server0), пінг не проходить, тому що ми використовуємо внутрішні IP-адреси, а маршрутизатор не має ніякої інформації про локальну мережу (рис.10.16).

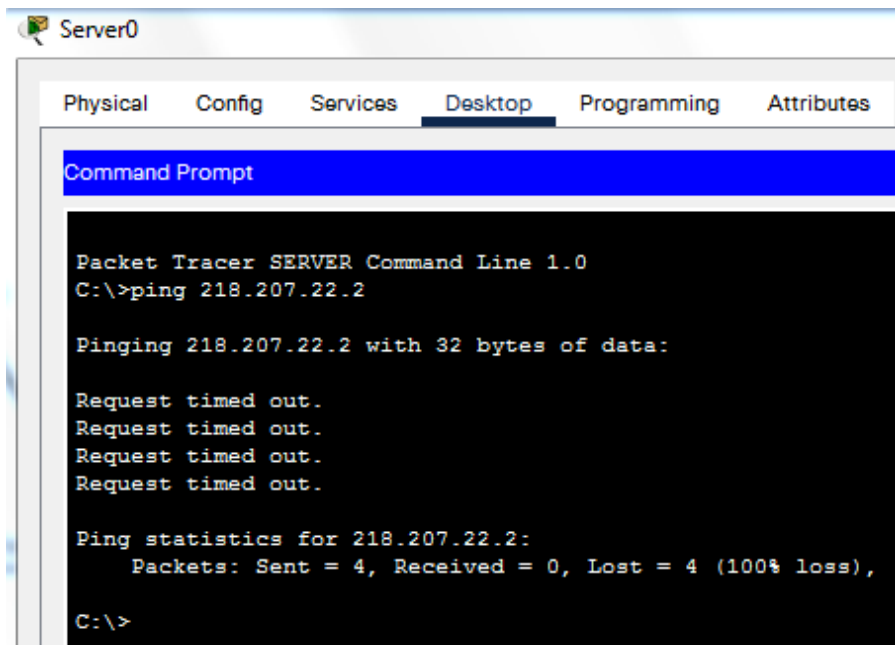


Рисунок 10.16 - Локальна мережа не має доступу до Інтернету

24. За допомогою технології NAT налаштуємо доступ локальних комп'ютерів і WEB-сервера до мережі Інтернет.

24.1. Спочатку вказуємо маршрутизатору Router0 який інтерфейс буде для NAT зовнішнім, а який внутрішнім.

```

Router#conf t
Router(config)#int fa0/1
Router(config-if)#ip nat outside
Router(config-if)#exit
Router(config)#int fa0/0.2
Router(config-subif)#ip nat inside
Router(config-subif)#exit
Router(config)#int fa0/0.3
Router(config-subif)#ip nat inside
Router(config-subif)#exit
Router(config)#int fa0/0.4
Router(config-subif)#ip nat inside
Router(config-subif)#end
Router# wr mem

```

24.2. Тепер створюємо access-list. Який містить перелік IP-адрес усіх підмереж внутрішньої мережі для яких необхідно буде виконувати трансляцію адрес.

```

Router#conf t
Router(config)#ip access-list standard FOR-NAT    присвоєння імені списку доступу
Router(config-std-nacl)#permit 192.168.2.0 0.0.0.255  IP-адреса VLAN2 і обернена маска
Router(config-std-nacl)#permit 192.168.3.0 0.0.0.255  IP-адреса VLAN3 і обернена маска

```

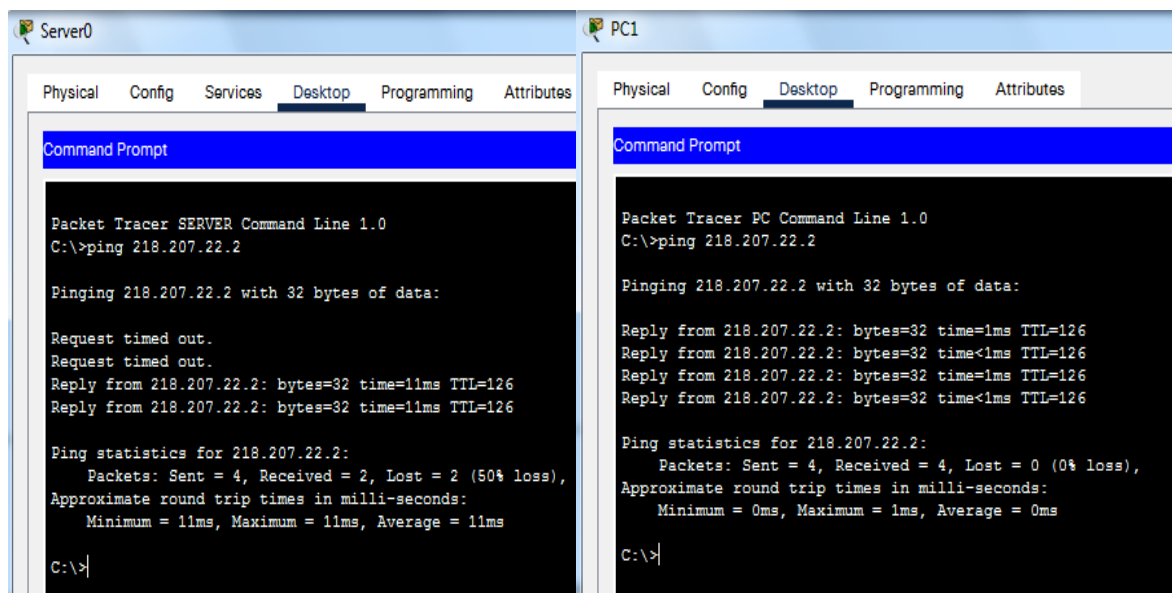
```
Router(config-std-nacl)#permit 192.168.4.0 0.0.0.255  IP-адреса VLAN4 і обернена маска
Router(config-std-nacl)#sh run                      перевірка налаштувань
Router(config-std-nacl)#end
```

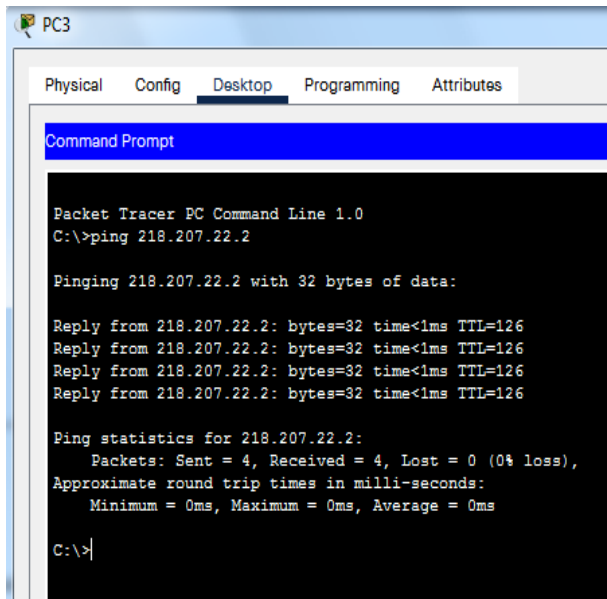
24.3. Останньою командою включаємо режим роботи NAT, який дозволяє відображати кілька внутрішніх IP-адрес в одну зовнішню, використовуючи різні порти (**NAT overload**).

```
Router#conf t
Router(config)#ip nat inside source list FOR-NAT interface fa0/1 overload
Router(config)#end
Router#wr mem
```

Ця команда говорить маршрутизатору, що у всіх пакетів, отриманих на внутрішній інтерфейс і дозволених списком доступу FOR-NAT, адреса відправника буде транслюватися в адресу інтерфейсу fa0/1. Ключ *overload* вказує, що трансляції будуть перезавантажені, що дозволить кільком внутрішнім вузлам транслюватися на одну IP-адресу. Тепер NAT налаштований.

25. Перевіримо можливість виходу в Інтернет із будь-якого комп'ютера, тобто перевіримо доступ до сервера 218.207.22.2.





26. На маршрутизаторі Router0 виконаємо команду

Router#show ip nat translations

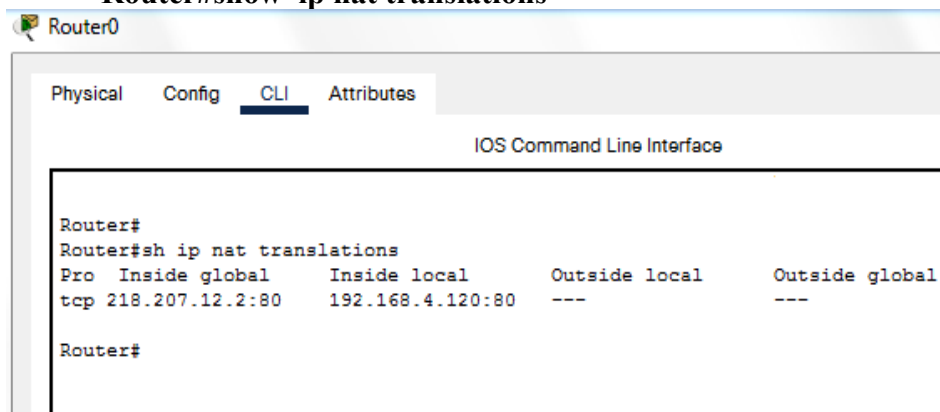


Рисунок 10.17- Таблиця трансляції NAT

27. Розглянемо, як працює NAT у нашому прикладі.

Комп'ютер із внутрішньої мережі (наприклад, Server0) відправляє дейтаграму в напрямку маршрутизатора Router0. Маршрутизатор отримує дейтаграму, замінює в ній IP-адресу відправника 192.168.4.120 на IP-адресу 218.207.12.2, розташовану в глобальній мережі, також відбувається заміна старого номеру порту призначення на новий (рис.10.18).

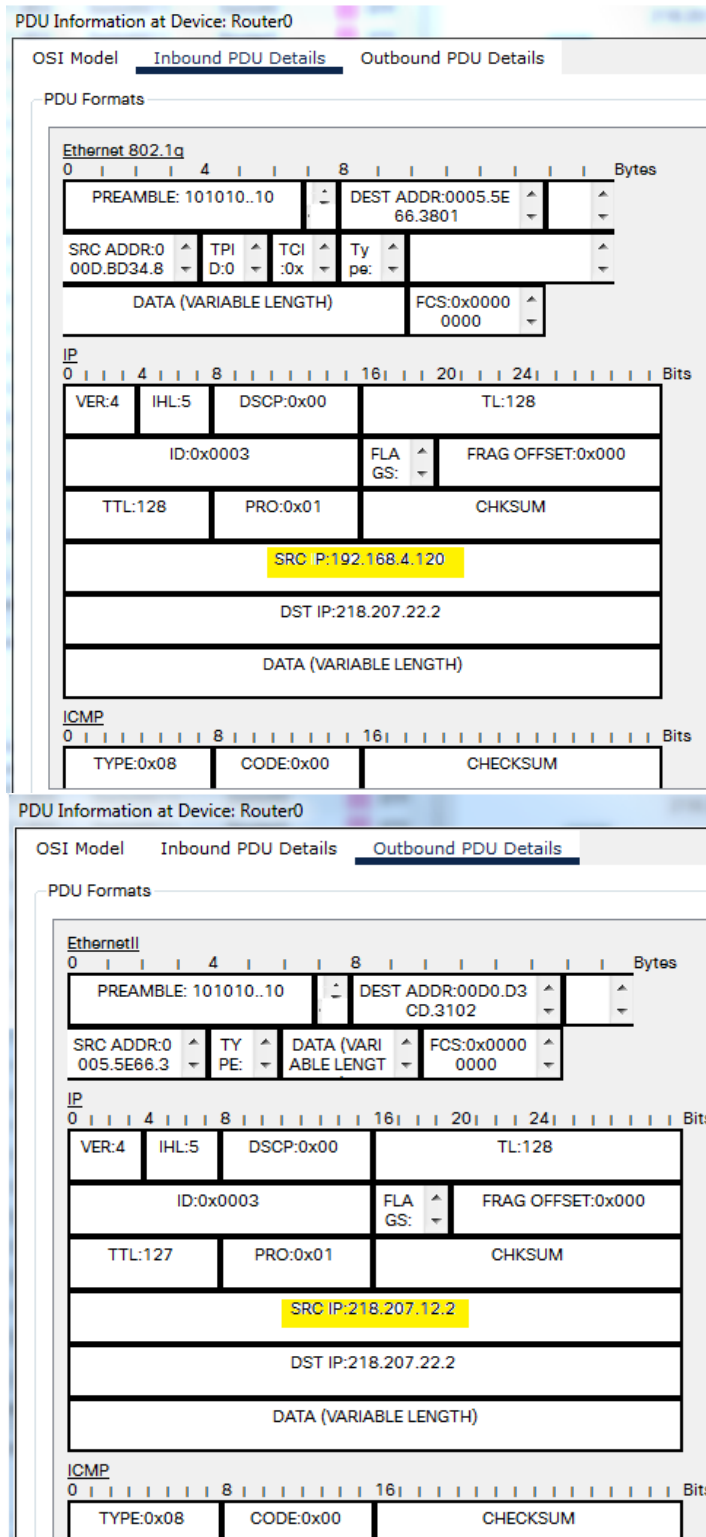


Рисунок 10.18- Заміна внутрішньої адреси на зовнішню

28. На сервері Server0 запусимо сервіс HTTP. Сервіс HTTP дозволяє будувати нескладні веб-сторінки і перевіряти проходження пакетів на порт 80 сервера.

Виконаємо http-запит із зовнішнього сервера Server1 до внутрішнього Web-сервера Server0.

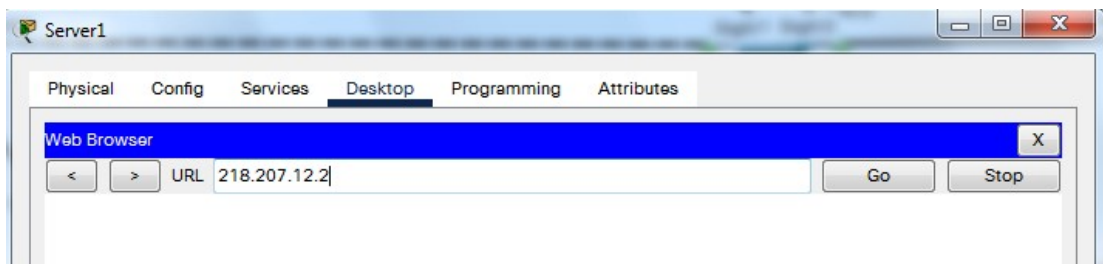
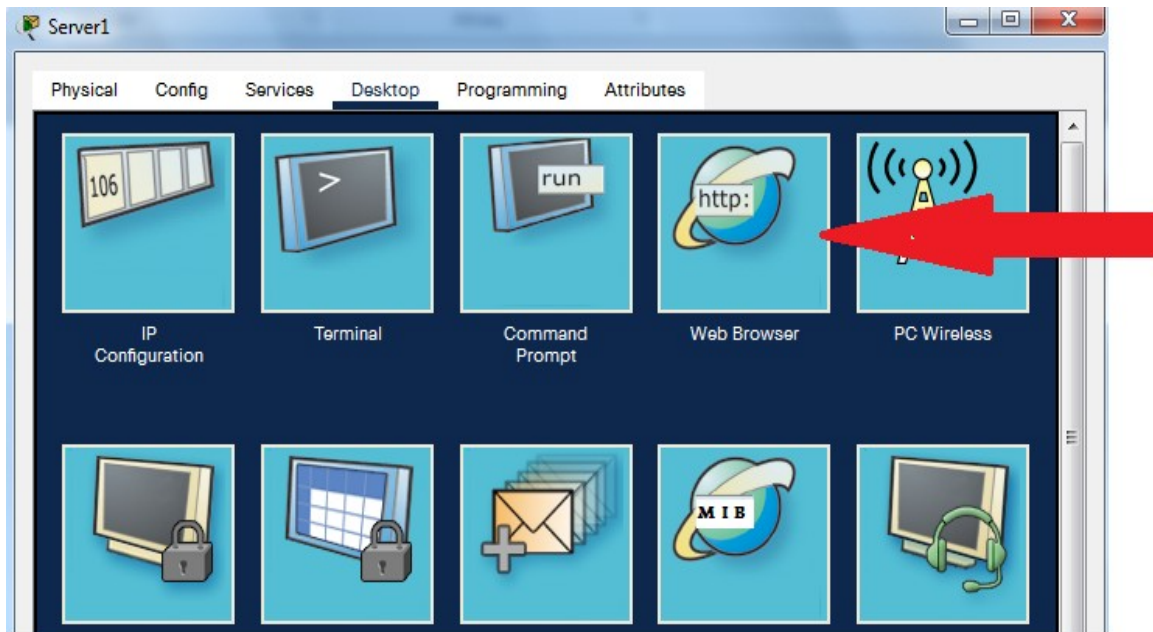
Для цього на Router0 необхідно налаштувати правило для статичного перетворення IP-адрес (Static NAT):

Router>enable

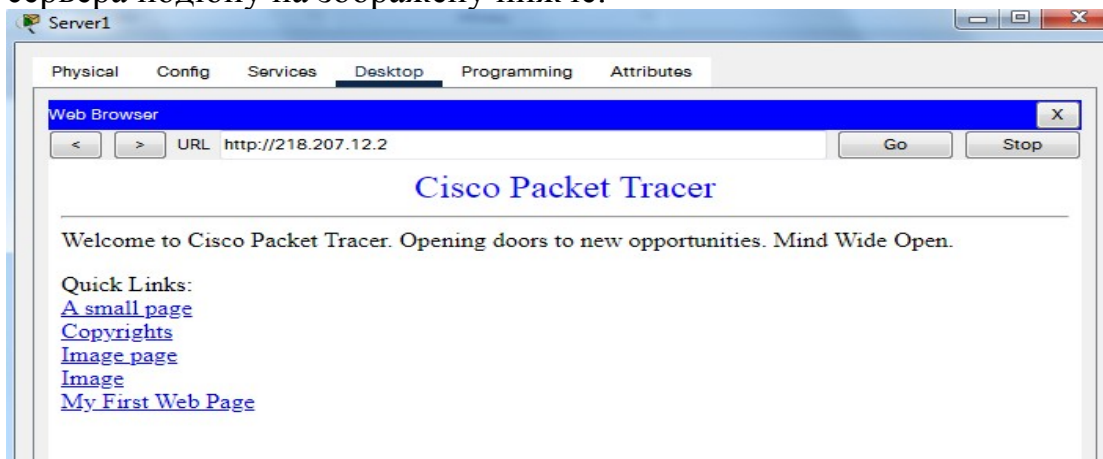
Router#conf t

Router(config)#ip nat inside source static tcp 192.168.4.120 80 218.207.12.2 80

29. Заходимо на Server1, відкриваємо вкладку Desktop. Запускаємо на сервері Web-браузер:



В полі **URL** вказуємо глобальну IP-адресу NAT-маршрутизатора 218.207.12.2 і натискаємо клавішу **GO**. Маємо отримати відповідь Web-сервера подібну на зображену нижче:



Завдання до лабораторної роботи

1. Побудувати модель комп'ютерної мережі, яка зображена на рисунку 10.1.
2. Покроково виконати необхідні мережеві налаштування мережевих пристроїв. Рекомендується після кожного кроку перевіряти виконані налаштування
3. У режимі симуляції за допомогою утиліти **ping** дослідити рух службових пакетів по створеній мережі:
 - від хоста у VLAN2 до хоста у VLAN3;
 - від хоста у VLAN2 до хоста у VLAN4;
 - від хоста у VLAN3 до хоста у VLAN2;
 - від хоста у VLAN3 до хоста у VLAN4;
 - від хоста у VLAN4 до хоста у VLAN2;
 - від хоста у VLAN2 до хоста у VLAN3.

Результати спостережень занесіть у звіт.

4. У режимі симуляції за допомогою утиліти **ping** дослідіть рух службових пакетів із внутрішньої мережі до сервера Server1 і в зворотному напрямку. Звернути увагу на заміну внутрішньої IP-адреси на зовнішню і зовнішньої на внутрішню при проходженні пакету через маршрутизатор Router0.

Результати спостережень занесіть у звіт.

5. Із сервера Server1 виконайте http-запит до Web-сервера Server0.

Отриманий результат занесіть у звіт.

Контрольні запитання

1. З якою метою створюються віртуальні мережі?
2. Які команди використовуються для призначення VLAN на інтерфейси?
3. Як забезпечити зв'язок між вузлами різних віртуальних мереж?
4. Як побудувати VLAN на кількох комутаторах?
5. Що таке ідентифікатор кадру (tag)? Де він розташований?
6. Що таке транк? Які переваги надає використання транкових з'єднань?
7. Які команди використовуються для створення транкових з'єднань?
8. Назвіть усі можливі схеми роботи служби NAT.
9. Перерахуйте етапи налаштування служби NAT.
10. Як перевірити роботу служби NAT на маршрутизаторі?.

Лабораторна робота 11

Робота з сокетами

Мета роботи: вивчити роботу з потоковими сокетами. Створити сервер, який буде відповідати на запити клієнтів.

План виконання лабораторної роботи

1. Ознайомитися та засвоїти теоретичні відомості, викладені в методичці до лабораторної роботи.
2. Виконати завдання до лабораторної роботи. Скласти звіт.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Поняття сокету

Сокет (socket) - це кінцевий пункт мережевих комунікацій. Він є своєрідним "порталом", через який можна відправляти байти в зовнішній світ. Додаток просто пише дані в сокет; їхня подальша буферизація, відправка і транспортування виконується стеком протоколів і мережевою апаратурою. Зчитування даних з сокету відбувається аналогічним чином.

Інтерфейс сокетів (API – Application Programming Interface) – це набір системних викликів і/або бібліотечних функцій, розділених на чотири групи:

1. локального керування;
2. встановлення з'єднання;
3. обміну даними (вводу/виводу);
4. закриття з'єднання.

API-інтерфейси сокетів логічно розташовані між прикладним і транспортним рівнями. API сокетів не є окремим рівнем в моделі зв'язку. API сокетів дозволяють додаткам взаємодіяти з транспортним або мережевим рівнями. Стрілки на рисунку 11.1 показують взаємодію API сокету з комунікаційними рівнями.

Socket API вперше був реалізований в операційній системі Berkley UNIX. Зараз цей **програмний інтерфейс** доступний практично в будь-якій модифікації Unix, в тому числі в Linux. Хоча всі реалізації чимось відрізняються одна від другої, основний набір функцій в них збігається. Спочатку сокети використовувались в програмах на C/C++, але зараз засоби для роботи з ними надають багато інших мов (Perl, PHP, Python, [Tcl](#), [Javascript](#), Ruby і ін.).

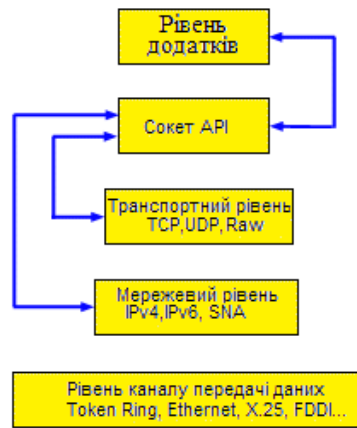


Рисунок 11.1 – Місце API сокетів в моделі зв'язку

В програмі сокет ідентифікується **дескриптором**, який є змінною типу **int**. Програма отримує дескриптор від операційної системи при створенні сокету, а потім передає його сервісам socket API для позначення сокету, з яким необхідно виконати ті або інші дії..

Сокети надають досить потужний і гнучкий механізм міжпроцесної взаємодії (IPC). Вони можуть використовуватися для організації взаємодії програм на одному комп'ютері, в локальній мережі або через Internet, що дозволяє створювати розподілені застосування різної складності. Крім того, за їх допомогою можна організовувати взаємодію з програмами, що працюють під керуванням інших операційних систем. Наприклад, під Windows існує інтерфейс Window Sockets, побудований на основі socket API.

Сокети підтримують багато стандартних мережевих протоколів (конкретний їх перелік залежить від реалізації) і надають уніфікований інтерфейс для роботи з ними. Частіше за все сокети використовуються для роботи в IP-мережах. У цьому випадку їх можна використовувати для взаємодії додатків не тільки по спеціально розробленим, але і по стандартним протоколам – HTTP, FTP, Telnet і т. д. Наприклад, можна написати власний Web-броузер або Web-сервер, здатний обслуговувати одночасно багатьох клієнтів.

Створюючи програму, необхідно мати можливість користуватися як орієнтованими, так і не орієнтованими на з'єднання протоколами. Інтерфейс сокетів дозволяє програмам використання обох цих типів протоколів.

В орієнтованих на з'єднання протоколах дані передаються як єдиний потік байтів без будь-якого розділення на блоки. В неорієнтованих на з'єднання протоколах дані передаються у вигляді окремих блоків, які називаються дейтаграмами.

За допомогою інтерфейсу сокетів можна створювати як програми-сервери, так і програми-клієнти, однак кожний тип додатків обслуговується різними функціями. Набір цих функцій залежить від того, серверу чи клієнту

призначений сокет, а також від того, чи буде сокет орієнтований на з'єднання, чи ні.

Усі мережеві додатки побудовані на технології клієнт-сервер; це значить, що в мережі існує хоча б один додаток, який є сервером, типова задача якого – це очікування запиту на підключення від додатків-клієнтів, яких може бути теоретично скільки завгодно, і виконання різних процедур у відповідь на запити клієнтів. Для клієнт-серверної технології абсолютно не важливо, де розташовані клієнт і сервер - на одній машині або на різних машинах. Звичайно, для успішного з'єднання клієнта з сервером клієнту необхідно мати мінімальний набір даних про розташування сервера – для мереж TCP/IP це IP-адреса комп'ютера, де розташований сервер, і адреса порту, на якому сервер чекає запити від клієнтів.

Сокети, незалежно від виду, поділяються на три типи: потокові, сирі і дейтаграмні. Потокові сокети працюють з встановленням з'єднання, забезпечуючи надійну ідентифікацію обох сторін і гарантують цілісність і успішність доставки даних, спираючись на протокол TCP. Дейтаграмні сокети працюють без встановлення з'єднання і не забезпечують ні ідентифікації відправника, ні контролю успішності доставки даних, зате вони швидше поточкових, спираючись на протокол UDP. Сирі сокети надають можливість ручного формування заголовків транспортного та мережевого рівнів.

Також існує 2 види сокетів:

- синхронні – затримують управління на час виконання операції;
- асинхронні – повертають управління, але продовжують виконувати роботу в фоні та після закінчення повідомляють про це.

У випадку з синхронними (блокуючими) сокетами сервер прийнявши нового клієнта працює з ним (обмінюється інформацією), але інші клієнти чекають в черзі, поки сервер не завершить роботу з цим. Асинхронні (не блокуючі) сокети працюють паралельно - витягує клієнта з черги, породжує потік/процес, передає йому дескриптор клієнта (який повернула функція асерт), цей потік/процес починає працювати у фоновому режимі в свою чергу сервер знову витягує нового клієнта із черги і так далі.

Асинхронні сокети слід використовувати там, де є велике навантаження при передачі даних.

1.2. Робота сокетів

Для прикладу розглянемо механізм роботи асинхронного сокету. Нехай на серверній стороні запускається серверний сокет, який після запуску відразу переходить в режим прослуховування (тобто очікування запиту на з'єднання від клієнтів). На стороні клієнта створюється сокет, для якого вказується IP-адреса і порт сервера і виконується команда на з'єднання.

Коли сервер отримує запит на з'єднання, ОС створює новий екземпляр сокету, за допомогою якого сервер може обмінюватися даними з клієнтом. При цьому сокет, який створений для прослуховування, продовжує знаходитися в режимі прийому з'єднань, таким чином, програміст може створити сервер, що працює з кількома з'єднаннями із клієнтами.

Розглянемо випадок, в якому основним сокетом, що працює на сервері, створюється дочірній процес для опрацювання нового з'єднання.

Нехай серверний сокет запускається на деякому вузлі з IP-адресою 206.62.226.35 і виконує пасивне відкриття з'єднання, використовуючи свій наперед визначений номер порту (наприклад 21). Тепер він очікує надходження запиту від клієнта (рис.11.2).

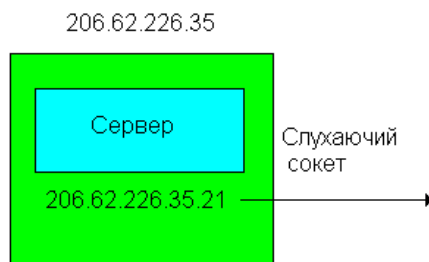


Рисунок 11.2 - Запуск сокету на сервері

Пізніше клієнтський сокет запускається на вузлі з IP-адресою 198.69.10.2 і виконує активне відкриття з'єднання з сервером з IP-адресою 206.62.226.35. В цьому прикладі ми вважаємо, що динамічно призначений порт, вибраний клієнтом TCP, - це порт 1500 (рис.11.3).

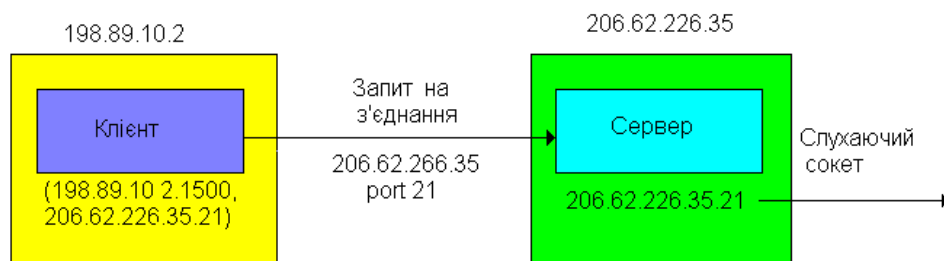


Рисунок 11.3 - Запит на з'єднання від клієнта

Клієнт характеризується своєю парою сокетів 198.69.10.2.1500, 206.62.226.35.21.

Коли серверний сокет отримує і виконує з'єднання з клієнтом, він створює за допомогою функції **fork(0)** свою копію, даючи змогу дочірньому процесу опрацювати запит клієнта, як показано на рис.11.4.

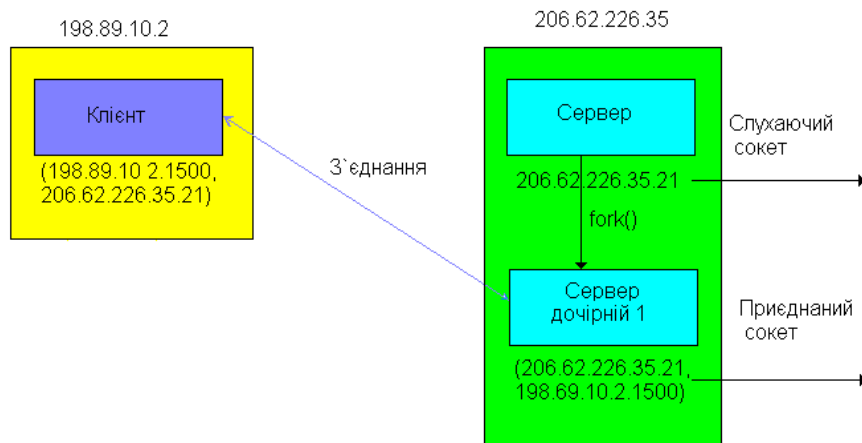


Рисунок 11.4 - Створення першого приєднаного сокету

Зверніть увагу на те, що приєднаний сокет використовує той же локальний порт 21, що і слухаючий сокет.

При виклику `fork()` створюється новий процес (дочірній процес), який майже ідентичний батьківському процесу. дочірній процес успадковує такі ознаки батьківського процесу:

- сегмент коду, даних і стеку програми;
- таблицю файлів, в якій знаходяться стани прапорців дескрипторів файлів, що вказують на те, чи читається файл, чи пишеться. Крім того, в таблиці файлів зберігається поточна позиція вказівника запису-зчитування;
- робочий та кореневий каталоги;
- реальний і ефективний номери користувача та номер групи;
- пріоритети процесу;
- контрольний термінал;
- маску сигналів;
- обмеження по ресурсам;
- відомості про середовище виконання;
- сегменти пам'яті, що використовуються.

Дочірній процес не успадковує такі ознаки:

- ідентифікатор процесу (PID, PPID);
- використаний час ЦП (він обнуляється);
- сигнали батьківського процесу, що вимагають відповіді;
- блоковані файли (record locking).

При виконанні наступного кроку припускається, що інший клієнтський процес на клієнтському вузлі вимагає з'єднання з тим же сервером. Код TCP-клієнта присвоює новому сокету клієнта невикористаний номер порту, що призначається динамічно, наприклад, 1501.

На стороні сервера розрізняють два з'єднання: пара сокетів для першого з'єднання відрізняється від пари сокетів для другого з'єднання, оскільки TCP-клієнт вибирає невикористаний порт 1501 для другого з'єднання.

З цього прикладу видно, що TCP не може демультимплексувати вхідні сегменти, проглядаючи тільки номери портів отримувача. TCP повинен розглядати всі чотири елементи в парі сокетів, щоб визначити адресу отримувача сегменту. На рис.11.5 показані три сокети з одним і тим же локальним портом 21.

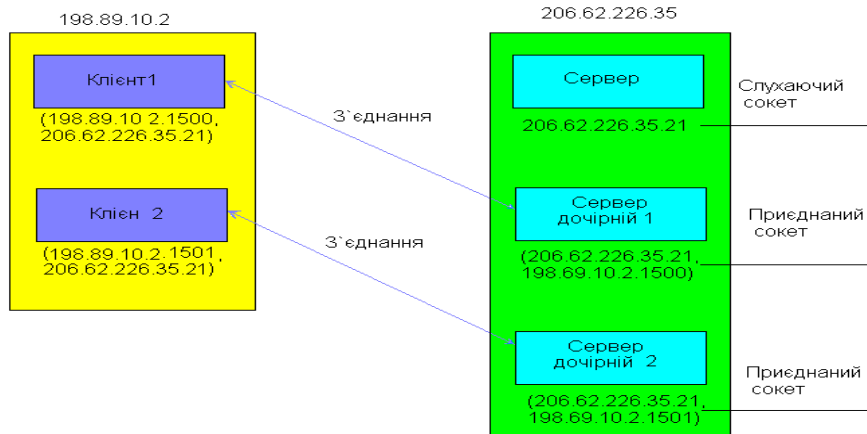


Рисунок 11.5 - Створення другого приєднаного сокету

Якщо сегмент надходить з IP-адреси 198.69.10.2, порт 1500 і адресований на IP-адресу 206.62.226.35, порт 21, він доставляється першому дочірньому процесу. Якщо ж сегмент надходить з IP-адреси 198.69.10.2, порт 1501, і адресований на IP-адресу 206.62.226.35, порт 21, він доставляється другому дочірньому процесу. Всі інші сегменти TCP, що надходять на порт 21, доставляються сокету, що слухає.

1.3. Створення сокету

На рис. 11.6 зображені системні виклики інтерфейсу сокетів, які, як правило, використовуються програмами, орієнтованими на з'єднання. Ліва частина діаграми показує виклики на стороні сервера, а права — на стороні клієнта.

Програма-сервер створює сокет, викликаючи функцію **socket**. Інтерфейс сокетів створює структуру даних сокету і повертає дескриптор сокету, який буде використаний для наступних викликів мережевих функцій.

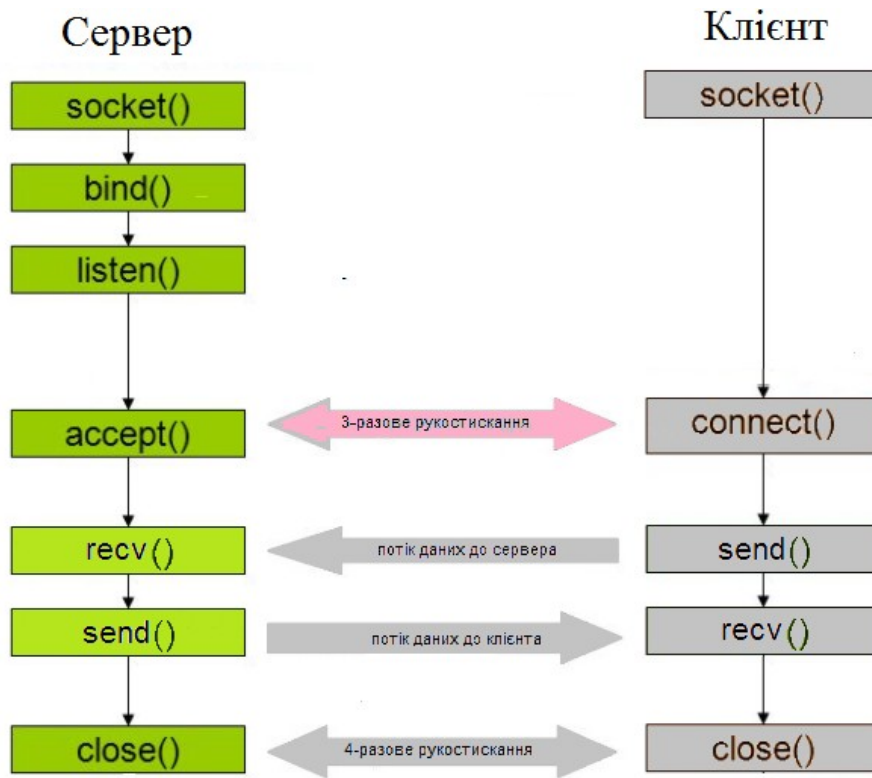


Рисунок 11.6 – Схема роботи простого сокета TCP

При створенні сокету необхідно вказати три атрибути:

- *домен (групу, до якої належить протокол);*
- *тип сокету;*
- *протокол.*

Ці атрибути задаються при створенні сокету і залишаються незмінними на протязі всього часу його існування. Для створення сокету використовується функція **socket**, яка має такий прототип.

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

```
int socket(int domain, int type, int protocol);
```

Домен визначає простір адрес, в якому розташований сокет, і множину протоколів, які використовуються для передачі даних. Частіше за інших використовуються домени Unix і Internet, які задаються константами **AF_UNIX** і **AF_INET** відповідно (префікс **AF** означає "address family" - "сімейство адрес"). При значенні **AF_UNIX** для передачі даних використовується файлова система вводу/виводу Unix. У цьому випадку сокети використовуються для міжпроцесної взаємодії на одному комп'ютері і не можуть використовуватися для роботи по мережі. Константа **AF_INET** відповідає Internet-домєну. Сокети, які розміщені у цьому домені, можуть використовуватися для роботи в будь-якій IP-мережі. Існують і інші домени (**AF_IPX** для протоколів Novell, **AF_INET6** для нової модифікації протоколу IP – IPv6 і т. д.).

Тип сокету визначає спосіб передачі даних по мережі.. Частіше за інші застосовуються:

- **SOCK_STREAM.** Передача потоку даних з попереднім встановленням з'єднання. Забезпечується надійний канал передачі даних, при якому фрагменти відправленого блоку не втрачаються, не переупорядковуються і не дублюються.
- **SOCK_DGRAM.** Передача даних у вигляді окремих повідомлень (дейтаграм). Попереднє встановлення з'єднання не потрібне. Обмін даними відбувається швидше, але є ненадійним: повідомлення можуть втрачатися, дублюватися і переупорядковуватися. Допускається передача повідомлення кільком отримувачам (multicasting) і ширококомвна передача (broadcasting).
- **SOCK_RAW.** Цей тип присвоюється низькорівневим (так званим "простим") сокетам. Їх відмінність від звичайних сокетів полягає у тому, що за їхньою допомогою програма може взяти на себе формування деяких заголовків, що додаються до повідомлення.

Головна мета використання простих сокетів полягає в обході механізму, за допомогою якого комп'ютер опрацьовує TCP/IP. Це досягається особливою реалізацією стеку TCP/IP, яка замінює механізм, що надається стеком TCP/IP в ядрі — пакет безпосередньо передається застосуванню і, як наслідок, опрацьовується ефективніше, ніж при проході через головний стек протоколів клієнта.

Отже, *простий сокет* — це сокет, який приймає пакети, обминає рівні TCP і UDP в стеку TCP/IP і направляє їх безпосередньо додатку.

При використанні простих сокетів обов'язок правильно опрацьовувати всі дані і виконувати такі дії, як видалення заголовків і розбір полів, лягає на застосування.

Робота з простими сокетами вимагає глибокого розуміння базових протоколів TCP/UDP/IP.

Низькорівневий сокет дозволяє програмі використовувати напряму ті низькорівневі протоколи, які зазвичай використовуються мережевими протоколами більш високого рівня. Наприклад, програма **ping** створює простий сокет, щоб напряму використати протокол керування повідомленнями Інтернет (ICMP). Як правило, програми-застосування не використовують ICMP. Вони надають можливість самій мережі вирішувати проблеми, пов'язані з помилками. Транспортні протоколи Інтернет самостійно надсилають повідомлення про помилку, яке адресоване програмі-застосуванню. Однак програма-застосування може напряму звернутися до рівня IP або ICMP. Для цього їй доведеться створити простий сокет.

Необхідно звернути увагу, що не всі домени допускають використання довільного типу сокету. Наприклад, разом з доменом Unix використовується

тільки тип **SOCK_STREAM**. З іншого боку, для Internet-домену можна задавати будь-який із перерахованих типів. У цьому випадку для реалізації **SOCK_STREAM** використовується протокол TCP, для реалізації **SOCK_DGRAM** - протокол UDP, а тип **SOCK_RAW** використовується для низькорівневої роботи з протоколами IP, ICMP і т. д.

Нарешті, останній атрибут визначає **протокол**, який використовується для передачі даних. Слід відзначити, що часто протокол однозначно визначається по домену і типу сокету. У цьому випадку третьому параметру функції **socket** можна присвоювати значення 0, що відповідає протоколу по замовчуванню. Однак, іноді (наприклад, при роботі з низькорівневими сокетами) потрібно задавати протокол явно.

Отже, щоб створити сокет, програма викликає функцію **socket**. Вона, в свою чергу, повертає дескриптор сокету, подібний до дескриптора файлу. Іншими словами, дескриптор сокету вказує на таблицю дескрипторів, яка містить опис властивостей і структуру сокету, тобто дескриптор вказує на член системної таблиці дескрипторів, який відповідає даному сокету.

Створення сокету — це процес виділення системної пам'яті для розміщення в ній структури даних, що описують даний сокет.

Структура даних сокету в спрощеному вигляді показана на рисунку 11.7.

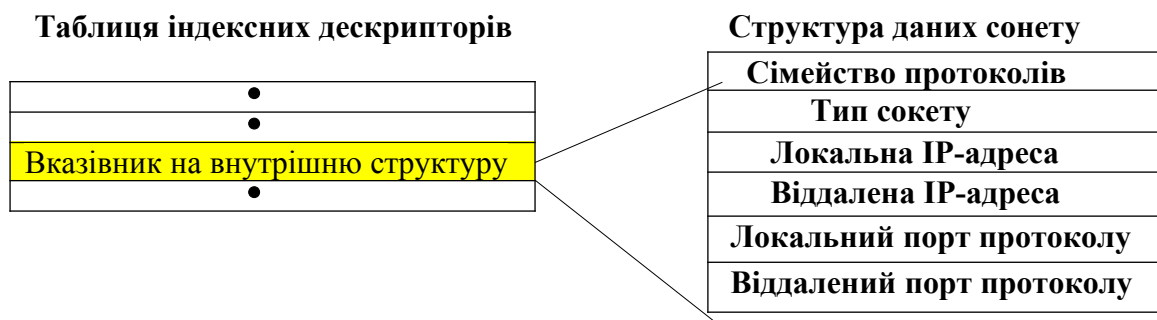


Рисунок 11.7 – Спрощена структура даних сокету.

Як видно із рисунку, структура даних сокету містить елементи для зберігання аргументів, з якими була викликана функція **socket**. Крім того, в структурі розташовані чотири адреси: локальна IP-адреса, віддалена IP-адреса, адреси локального і віддаленого портів. Кожного разу, коли програма викликає функцію **socket**, реалізація сокетів відводить машинну пам'ять для нової структури даних, а потім поміщає в неї сімейство адрес, тип сокету і протоколу. В таблиці дескрипторів розташований вказівник на цю структуру. Отриманий програмою дескриптор — це індекс (порядковий номер) в таблиці дескрипторів.

Функція **socket** створює структуру даних сокету, не заповнюючи при цьому поля адрес. Щоб зв'язати сокет з певною мережевою адресою, необхідно викликати інші функції, що входять до складу API, так, як це буде показано в наступних розділах.

Адреси

Сокет створюється без імені. Віддалений процес не може посилатися на сокет, поки сокету не буде надана адреса. Перш ніж передавати дані через сокет, його необхідно зв'язати з адресою в локальному домені (цю процедуру називають іменуванням сокету). Вид адреси залежить від вибраного типу домена. В Unix-домени це текстовий рядок – ім'я файлу, через який відбувається обмін даними. В Internet-домени – адреса задається комбінацією IP-адреси і 16-бітного номеру порту. Іноді зв'язування виконується неявно (всередині функцій **connect** і **accept**), але виконувати його необхідно у всіх випадках.

Функція **bind** дозволяє явно вказати сокету локальну адресу: IP-адресу і порт. IP-адреса визначає хост в мережі, а порт - конкретний сокет на цьому хості. Протоколи TCP і UDP використовують різні простори портів. Її прототип має такий вигляд:

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

int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

Перший параметр, що передається, являє собою дескриптор сокету, який треба прив'язати до заданої адреси. Другий параметр, **addr**, – це вказівник на структуру з адресою, а третій – це довжина цієї структури. Таким чином, прототип структури з адресою має такий вигляд:

```
struct sockaddr {
    unsigned short sa_family;    // Сімейство адрес, AF_ххх
    char          sa_data[14];  // 14 байтів для зберігання
    адреси
};
```

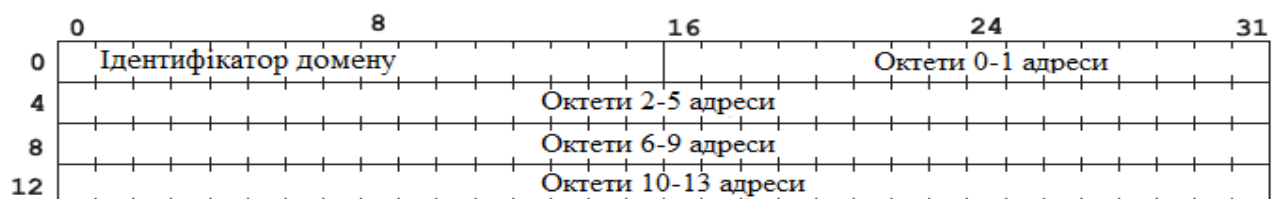


Рисунок 11.8 - Структура *sockaddr*, яка використовується при передачі протокольної адреси інтерфейсу сокетів

Поле **sa_family** містить ідентифікатор домену, той же, що і перший параметр функції **socket**. В залежності від значення цього поля по-різному інтерпретується вміст масиву **sa_data**. Зрозуміло, працювати з цим масивом напряду не дуже зручно, тому можна використати замість **sockaddr** одну із альтернативних структур, що має вигляд **sockaddr_XX** (XX - суфікс, що позначає домен: "un" - Unix, "in" - Internet и т. д.) – рис.11.9.

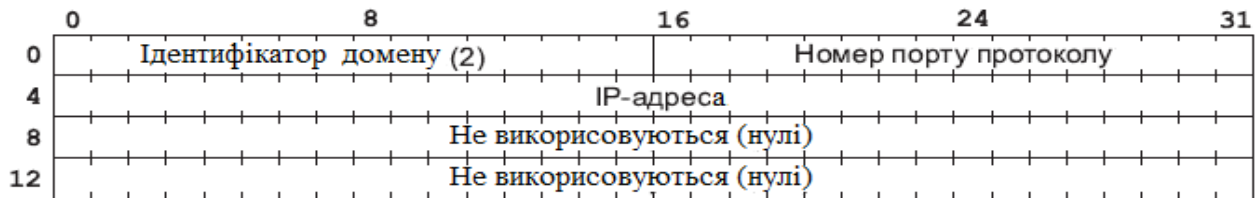


Рисунок 11.9 - Структура *sockaddr_in* при використанні протоколів TCP/IP

При передачі в функцію **bind** вказівник на цю структуру приводиться до вказівника на **sockaddr**. Розглянемо, як приклад, структуру **sockaddr_in**.

```
struct sockaddr_in {
    short int          sin_family; // Сімейство адрес
    unsigned short int sin_port;   // Номер порту
    struct in_addr     sin_addr;   // IP-адреса
    unsigned char      sin_zero[8]; // "Доповнення" до розміру
    структури sockaddr
};
```

Тут поле **sin_family** відповідає полю **sa_family** в **sockaddr**, в **sin_port** записується номер порту, а в **sin_addr** - IP-адреса хосту. Поле **sin_addr** саме є структурою, яка має такий вигляд:

```
struct in_addr {
    unsigned long s_addr;
};
```

Для чого потрібно залучати всього одне поле в структуру? Справа в тому, що раніше **in_addr** являла собою об'єднання (union), яке вміщало набагато більшу кількість полів. Зараз, коли в ньому залишилось всього одне поле, вона продовжує використовуватись для забезпечення зворотної сумісності.

І ще одне важливе застереження. Існує два порядки зберігання байтів: в слові і подвійному слові. Один із них називається *порядком хосту* (host byte order), другий - *мережевим порядком* (network byte order) зберігання байтів. Якщо вказується IP-адреса і номер порту, то необхідно перетворювати число із порядку хоста в мережевий. Для цього використовуються функції **htons** (Host TO Network Short) і **htonl** (Host TO Network Long). Зворотне перетворення виконують функції **ntohs** і **ntohl**.

ПРИМІТКА

На деяких машинах (PC це не стосується) порядок хосту і мережевий порядок зберігання байтів збігаються. Незважаючи на це, функції перетворення краще застосовувати і тут, оскільки це покращить мобільність програми. Це ніяк не вплине на продуктивність, оскільки передпроцесор сам вилучить всі "зайві" виклики цих функцій, залишивши їх тільки там, де перетворення дійсно необхідне.

Встановлення з'єднання (сервер)

Встановлення з'єднання на стороні сервера виконується за чотири кроки, ні один із яких не може бути пропущений. Спочатку сокет створюється і зв'язується з локальною адресою. Якщо комп'ютер має кілька мережевих інтерфейсів з різними IP-адресами, приймати з'єднання можна тільки з одного із них, передавши його адресу функції **bind**. При готовності з'єднання з клієнтами через будь-який інтерфейс, необхідно задати адресу за допомогою константи **INADDR_ANY**. Що стосується номера порту, то можна задати конкретний номер або 0 (в цьому випадку система сама вибере довільний, що не використовується в даний момент, номер порту).

Наступним кроком створюється черга запитів на з'єднання. При цьому сокет переводиться в режим очікування запитів з боку клієнтів. Все це виконує функція **listen**.

```
int listen(int sockfd, int backlog);
```

Перший параметр - дескриптор сокета, а другий задає розмір черги запитів. Кожного разу, коли черговий клієнт намагається з'єднатися з сервером, його запит ставиться в чергу, оскільки сервер може бути зайнятий опрацюванням інших запитів. Якщо черга заповнена, всі наступні запити ігноруються. Коли сервер готовий опрацювати черговий запит, він використовує функцію **accept**.

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

```
int accept(int sockfd, void *addr, int *addrlen);
```

Функція **accept** вибирає із черги з'єднань, що очікують обробки, перший запит і створює для спілкування з клієнтом *новий* сокет з майже такими ж властивостями, як і у сокета **sockfd** (див. матеріал на стор.3).

Якщо в момент виклику **accept()** в черзі не було запитів на з'єднання, то поведінка функції залежить від того, в якому режимі знаходиться сокет, блокуючому чи неблокуючому. В першому випадку програма блокується до приходу запиту на з'єднання, в другому - функція **accept()** повертається з помилкою (errno буде **EWOULDBLOCK** або **EAGAIN**). Функція повертає дескриптор створеного сокета, який потрібно використовувати тільки для обміну даними, і який не може бути використаний для прийому з'єднань. Первинний сокет **sockfd** залишається відкритим і служить для прийому

наступних з'єднань на цьому порті. Другий параметр заповнюється самою функцією і після завершення виклику буде містити інформацію про адресу того сокету, який приєднався (ім'я віддаленого сокету). Третій параметр одночасно є як вхідним, так і вихідним. При виконанні виклику він має містити розмір об'єкту, на який показує вказівник **addr**, а після завершення – він буде містити фактичну довжину адреси.

Необхідно звернути увагу, що отриманий від **accept** новий сокет зв'язаний з тією ж самою адресою, що і слухаючий сокет. Спочатку це виглядає дивно. Але справа в тому, що адреса TCP-сокету не обов'язково має бути унікальною в Internet-домени. Унікальними мають бути тільки *з'єднання*, для ідентифікації яких використовуються *дві* адреси сокетів, між якими відбувається обмін даними.

Встановлення з'єднання (клієнт)

На стороні клієнта для встановлення з'єднання використовується функція **connect**, яка має такий прототип.

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

```
int connect(int sockfd, struct sockaddr *serv_addr, int
addrlen);
```

Тут **sockfd** – дескриптор сокету, який буде використовуватися для обміну даними з сервером, **serv_addr** містить вказівник на структуру з адресою сервера, а **addrlen** - довжину цієї структури. Зазвичай сокет не вимагає попереднього прив'язування до локальної адреси, тому що функція **connect** виконує це сама, підібравши підходящий вільний порт. Можна примусово призначити клієнтському сокету деякий номер порту, використовуючи **bind** перед викликом **connect**. Робити це слід у випадку, коли сервер з'єднується тільки з клієнтами, які використовують певний порт (прикладом таких серверів є rlogind и rshd). В решті випадків простіше і надійніше надати самій системі можливість вибирати порт.

Обмін даними

Після того як з'єднання встановлене, можна починати обмін даними. Для цього використовуються функції **send** і **recv**. В Unix для роботи з сокетами можна використовувати також файлові функції **read** і **write**, але вони мають менші можливості, і, крім того, не будуть працювати на інших платформах (наприклад, під Windows), тому використовувати їх не рекомендується.

Функція **send** використовується для відправки даних і має таку структуру:

```
int send(int sockfd, const void *msg, int len, int flags);
```

Тут `sockfd` - це, як завжди, дескриптор сокету, через який відправляються дані, **`msg`** - вказівник на буфер з даними, **`len`** - довжина буферу в байтах, а **`flags`** - набір бітових прапорців, які керують роботою функції (якщо прапорці не використовуються, функції необхідно передати 0). Ось деякі із них (повний перелік можна знайти в документації):

- **MSG_OOB**. Примушує відправити дані як *термінові* (out of band data, OOB). Концепція термінових даних дозволяє мати два паралельних канали даних в одному з'єднанні. Іноді це буває зручно. Наприклад, Telnet використовує термінові дані для передачі команд типу Ctrl+C. Зараз використовувати їх не рекомендується із-за проблем з сумісністю (існує два різних стандарти їх використання, які описані в RFC793 і RFC1122). Безпечніше просто створювати для термінових даних окреме з'єднання.
- **MSG_DONTROUTE**. Забороняє маршрутизацію пакетів. Нижче розташовані транспортні рівні можуть проігнорувати цей прапорець.

Функція **`send`** повертає число байтів, які фактично були відправлені (або -1 якщо була помилка). Це число може бути менше вказаного розміру буфера. Якщо необхідно відправити увесь буфер цілком, то треба написати свою функцію і викликати в ній **`send`**, поки всі дані не будуть відправлені. Це може мати такий вигляд:

```
int sendall(int s, char *buf, int len, int flags)
{
    int total = 0;
    int n;

    while(total < len)
    {
        n = send(s, buf+total, len-total, flags);
        if(n == -1) { break; }
        total += n;
    }
    return (n==-1 ? -1 : total);
}
```

Використання **`sendall`** нічим не відрізняється від використання **`send`**, але вона відправляє цілком увесь буфер з даними.

Для зчитування даних із сокету використовується функція **`recv`**.

```
int recv(int sockfd, void *buf, int len, int flags);
```

Її використання аналогічне **`send`**. Вона теж приймає дескриптор, вказівник на буфер і набір прапорців. Прапорець **MSG_OOB** використовується для прийому термінових даних, а **MSG_PEEK** дозволяє "підглянути" дані, отримані від віддаленого хосту, не видаляючи їх із системного буферу (це означає, що при наступному зверненні до **`recv`** будуть

отримані ті ж самі дані). Повний перелік прапорців можна знайти в документації. По аналогії з **send** функція **recv** повертає кількість прочитаних байтів, яка може бути меншою ніж розмір буфера. Існує ще один особливий випадок, коли **recv** повертає 0. Це означає, що з'єднання було разірване.

Закриття сокету

Закінчивши обмін даними, сокет закривається за допомогою функції **close**. Відбувається розрив з'єднання:

```
#include <unistd.h>
```

```
int close(int fd);
```

Також можна заборонити передачу даних в якомусь одному напрямку, використовуючи **shutdown**:

```
int shutdown(int sockfd, int how);
```

Параметр **how** може приймать одне із таких значень:

- 0 – заборонити зчитування із сокету
- 1 – заборонити запис в сокет
- 2 – заборонити і зчитування і запис

Хоча після виклику **shutdown** з параметром **how**, рівному 2, втрачається можливість використовувати сокет для обміну даними, тим не менше, потрібно викликати **close**, щоб вивільнити пов'язані з ним системні ресурси.

Обробка помилок

Всі розглянуті нами функції повертають -1, записуючи в глобальну змінну **errno** код помилки. Відповідно, можна проаналізувати значення цієї змінної і вжити заходи по відновленню нормальної роботи програми, не перериваючи її виконання. А можна просто видати діагностичне повідомлення (для цього зручно використовувати функцію **perror**), а потім завершити програму за допомогою **exit**.

Налаштування програм

Мережеву програму можна налаштовувати без мережі. Для цього достатньо запустити програми клієнта і сервера на одній машині, а потім використати для з'єднання адресу *інтерфейсу внутрішньої петлі* (loopback interface). В програмі їй відповідає константа **INADDR_LOOPBACK** (до неї потрібно застосувати функцію **htonl!**). Пакети, які направляються за цією

адресою, в мережу не попадають. Замість цього вони передаються стеку протоколів TCP/IP, як тільки що прийняті. Таким чином, моделюється наявність віртуальної мережі, в якій можна проводити налаштування мережових застосувань.

Приклади створення ехо-клієнта і ехо-сервера

В прикладі (лістинг 1) ехо-клієнт надсилає серверу повідомлення "Hello there!" і виводить на екран відповідь сервера. Ехо-сервер читає все, що передає йому клієнт, а потім просто відправляє отримані дані назад клієнту. Код програми ехо-сервер наведений в лістингу 2.

Лістинг 1. Ехо-клієнт.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char message[] = "Hello there!\n";
char buf[sizeof(message)];

int main()
{
    int sock;
    struct sockaddr_in addr;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425); // або інший порт
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    if(connect(sock, (struct sockaddr *)&addr, sizeof(addr)) <
0)
    {
        perror("connect");
        exit(2);
    }

    send(sock, message, sizeof(message), 0);
    recv(sock, buf, sizeof(message), 0);

    printf(buf);
    close(sock);

    return 0;
}
```

Лістинг 2. Ехо-сервер.

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
    int sock, listener;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) <
0)
    {
        perror("bind");
        exit(2);
    }

    listen(listener, 1);

    while(1)
    {
        sock = accept(listener, NULL, NULL);
        if(sock < 0)
        {
            perror("accept");
            exit(3);
        }

        while(1)
        {
            bytes_read = recv(sock, buf, 1024, 0);
            if(bytes_read <= 0) break;
            send(sock, buf, bytes_read, 0);
        }

        close(sock);
    }

    return 0;
}

```

Паралельне обслуговування клієнтів

Наступне важливе питання, яке ми будемо обговорювати, – це паралельне обслуговування клієнтів. Ця проблема стає актуальною, коли серверу доводиться обслуговувати значну кількість запитів. Зрозуміло, на машині з одним процесором справжньої паралельності досягнути не вдасться. Але навіть на одній машині можна досягти суттєвого виграшу в продуктивності. Припустимо, сервер відправив якісь дані клієнту і чекає підтвердження. Поки воно мандрує по мережі, сервер міг би працювати з іншими клієнтами. Для реалізації такого алгоритму обслуговування існує багато способів, але частіше застосовуються два з них.

Спосіб 1

Цей спосіб полягає в створенні дочірнього процесу для обслуговування кожного нового клієнта. При цьому батьківський процес виконує тільки прослуховування порту і прийом з'єднань. Щоб забезпечити таку поведінку, відразу після **accept** сервер викликає функцію **fork** для створення дочірнього процесу. Далі аналізується значення, яке повернула ця функція. В батьківському процесі воно містить ідентифікатор дочірнього, а в дочірньому процесі дорівнює нулю. Використовуючи це значення, ми переходимо до чергового виклику **accept** у батьківському процесі, а дочірній процес обслуговує клієнта і завершується (**exit**).

Цей спосіб створення паралельного сервера застосований в лістингу 4.

Лістинг 4. Ехо-сервер (версія **fork**)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
    int sock, listener;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = INADDR_ANY;
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) <
```

```

0)
{
    perror("bind");
    exit(2);
}

listen(listener, 1);

while(1)
{
    sock = accept(listener, NULL, NULL);
    if(sock < 0)
    {
        perror("accept");
        exit(3);
    }

    switch(fork())
    {
    case -1:
        perror("fork");
        break;

    case 0:
        close(listener);
        while(1)
        {
            bytes_read = recv(sock, buf, 1024, 0);
            if(bytes_read <= 0) break;
            send(sock, buf, bytes_read, 0);
        }

        close(sock);
        _exit(0);

    default:
        close(sock);
    }
}

close(listener);

return 0;
}

```

Очевидна перевага такого підходу полягає в тому, що він дозволяє писати доволі компактні, зрозумілі програми, в яких код встановлення з'єднання відділений від коду обслуговування клієнта. На жаль, у нього є і недоліки. По-перше, якщо клієнтів дуже багато, створення нового процесу для обслуговування кожного із них може виявитися доволі затратною операцією. По-друге, такий спосіб неявно має на увазі, що всі клієнти обслуговуються незалежно один від другого. Однак це може бути і не так. Якщо, наприклад, ви пишете чат-сервер, то ваша основна задача –

підтримувати взаємодію всіх клієнтів, які приєдналися до нього. В цих умовах межі між процесами стануть для вас серйозною перешкодою. В подібному випадку вам слід серйозно розглянути інший спосіб обслуговування клієнтів.

Спосіб 2

Другий спосіб базується на використанні *неблокуючих сокетів* (nonblocking sockets) і функції **select**. Спочатку давайте подивимось, що таке неблокуючі сокети. Сокети, які ми до цього часу використовували, були *блокуючими* (blocking). Ця назва означає, що на час виконання операції з таким сокетом ваша програма блокується. Наприклад, якщо ви викликали **recv**, а даних на вашому кінці з'єднання немає, то в очікуванні їх надходження ваша програма "засинає". Аналогічна ситуація спостерігається, коли ви викликаєте **accept**, а черга запитів на з'єднання порожня. Цю поведінку можна змінити, використовуючи функцію **fcntl**.

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
```

Ця нескладна операція перетворює сокет в неблокуючий. Виклик будь-якої функції з таким сокетом буде повертати керування негайно. Причому, якщо затребувана операція не була виконана до кінця, функція поверне -1 і запише в **errno** значення **EWOULDBLOCK**. Щоб дочекатися завершення операції, ми можемо опитувати всі наші сокети в циклі, поки якась функція не поверне значення, відмінне від **EWOULDBLOCK**. Як тільки це відбудеться, ми можемо запустити на виконання наступну операцію з цим сокетом і повернутися до нашого опитуючого циклу. Така тактика (називається polling) дієздатна, але дуже неефективна, оскільки процесорний час витрачається даремно на багаторазові (і безрезультатні) опитування.

Щоб виправити ситуацію, використовують функцію **select**. Ця функція дозволяє відслідковувати стан кількох файлових дескрипторів (а в Unix до них належать і сокети) одночасно.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
```

```
FD_SET(int fd, fd_set *set);
FD_ZERO(int fd);
```

Функція **select** працює з трьома множинами дескрипторів, кожна з яких має тип **fd_set**. В множину **readfds** записуються дескриптори сокетів, із яких нам потрібно зчитувати дані (сокети, які слухають, додаються у цю ж множину). Множина **writfds** має містити дескриптори сокетів, в які ми збираємося записувати, а **exceptfds** - дескриптори сокетів, які потрібно контролювати на виникнення помилки. Якщо якась множина вас не цікавить, ви можете передати замість вказівника на неї **NULL**. Що стосується інших параметрів, в **n** потрібно записати максимальне значення дескриптора по всім множинам плюс одиниця, а в **timeout** - величину таймауту. Структура **timeval** має такий формат:

```
struct timeval {
    int tv_sec;        // секунди
    int tv_usec;       // мікросекунди
};
```

Поле "мікросекунд" виглядає вражаюче. Але на практиці вам не досягнути такої точності вимірювання часу при використанні **select**. Реальна точність виявиться близько 100 мілісекунд.

Тепер розглянемо множини дескрипторів. Для роботи з ними передбачені функції **FD_XXX**, згадані вище; їх використання повністю приховує від нас деталі внутрішнього облаштування **fd_set**. Розглянемо їх призначення.

- **FD_ZERO(fd_set *set)** – очищує множину **set**
- **FD_SET(int fd, fd_set *set)** – додає дескриптор **fd** у множину **set**
- **FD_CLR(int fd, fd_set *set)** – видаляє дескриптор **fd** із множини **set**
- **FD_ISSET(int fd, fd_set *set)** - перевіряє, чи є дескриптор **fd** в множині **set**

Якщо хоча б один сокет готовий до виконання заданої операції, **select** повертає ненульове значення, а всі дескриптори, які призвели до "спрацювання" функції, записуються у відповідні множини. Це дозволяє нам проаналізувати розташовані в цих множинах дескриптори і виконати з ними необхідні дії. Якщо спрацював таймаут, **select** повертає нуль, а у випадку помилки -1. Розширений код записується в **errno**.

Програми, які використовують неблокуючі сокети разом з **select**, стають доволі заплутаними. Якщо при використанні **fork** ми будуємо логіку програми, начебто всього для одного клієнта, то тут програма вимушена відслідковувати дескриптори *всіх* клієнтів і працювати з ними паралельно. Щоб проілюструвати цю методику, в черговий раз перепишемо ехо-сервер з використанням **select**. Нова версія приведена в лістингу 5. Зверніть увагу, що ця програма, на відміну від всіх інших, написана на C++ (а не на C). Скористуємось класом **set** із бібліотеки STL мови C++, щоб полегшити роботу з набором дескрипторів і зробити її більш зрозумілою.

Лістинг 5. Ехо-сервер (неблокуючі сокети і select).

```

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
    int listener;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }

    fcntl(listener, F_SETFL, O_NONBLOCK);

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = INADDR_ANY;
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) <
0)
    {
        perror("bind");
        exit(2);
    }

    listen(listener, 2);

    set<int> clients;
    clients.clear();

    while(1)
    {
        // Заповнюємо множину сокетів
        fd_set readset;
        FD_ZERO(&readset);
        FD_SET(listener, &readset);

        for(set<int>::iterator it = clients.begin(); it !=
clients.end(); it++)
            FD_SET(*it, &readset);
    }
}

```

```

// Задаємо таймаут
timeval timeout;
timeout.tv_sec = 15;
timeout.tv_usec = 0;

// Чекаємо події в одному із сокетів
int mx = max(listener, *max_element(clients.begin(),
clients.end()));
if(select(mx+1, &readset, NULL, NULL, &timeout) <= 0)
{
    perror("select");
    exit(3);
}

// Визначаємо тип події і виконуємо відповідні дії
if(FD_ISSET(listener, &readset))
{
    // Надійшов новий запит на з'єднання, використовуємо
accept
    int sock = accept(listener, NULL, NULL);
    if(sock < 0)
    {
        perror("accept");
        exit(3);
    }

    fcntl(sock, F_SETFL, O_NONBLOCK);

    clients.insert(sock);
}

for(set<int>::iterator it = clients.begin(); it !=
clients.end(); it++)
{
    if(FD_ISSET(*it, &readset))
    {
        // Надійшли дані від клієнта, читаємо їх
        bytes_read = recv(*it, buf, 1024, 0);

        if(bytes_read <= 0)
        {
            // З'єднання розірвано, видаляємо сокет із
МНОЖИНИ
            close(*it);
            clients.erase(*it);
            continue;
        }

        // Відправляємо отримані дані клієнту
        send(*it, buf, bytes_read, 0);
    }
}
}

```



```
    return 0;  
}
```

Контрольні запитання:

1. Що таке сокет? Які існують типи сокетів?
2. Яка функція використовується при створенні сокету?
3. Яка функція використовується для надання сокету локальної адреси?
4. Якими функціями визначаються адреси віддалених клієнтів?
3. Опишіть принцип роботи потокових сокетів.
4. В чому полягає різниця між дейтаграмними і потоковими сокетами?
5. Назвіть функції для отримання і відправлення повідомлень?
6. Які поля містить структура *sockaddr*?
7. Коли бажано використовувати асинхронні сокети?

Завдання:

1. Набрати коди прикладів сокетів. Запустити і показати викладачу.
2. Відповісти на запитання викладача по матеріалу, який викладений в описі даної лабораторної роботи.

Обмін дейтаграмами. Низькорівневі сокети.

Мета роботи: вивчити роботу з дейтаграмними сокетами. Створити програму-sender і програму-receiver. Продемонструвати їхню взаємодію.

План виконання лабораторної роботи

1. Ознайомитися та засвоїти теоретичні відомості викладені в методичці до лабораторної роботи.
2. Виконати завдання до лабораторної роботи. Скласти звіт.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Поняття дейтаграмного сокету

Дейтаграми використовуються в програмах досить давно. В більшості випадків надійність передачі критична для прикладних додатків, і замість того, щоб винаходити власний надійний протокол поверх UDP, програмісти надають перевагу використанню TCP. Тим не менше, іноді дейтаграми виявляються корисними. Наприклад, їх зручно використовувати при трансляції звуку або відео по мережі в реальному часі, особливо при ширококомовному трансляванні, коли важливість швидкості передачі перевищує вимоги до надійності доставки пакетів.

Оскільки для обміну дейтаграмами не потрібно встановлювати з'єднання, використовувати їх набагато простіше (рис 12.1). Створивши сокет за допомогою **socket** і **bind**, ми можемо відразу використовувати його для відправки або отримання даних. Для цього ми будемо використовувати функції **sendto** і **recvfrom**.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

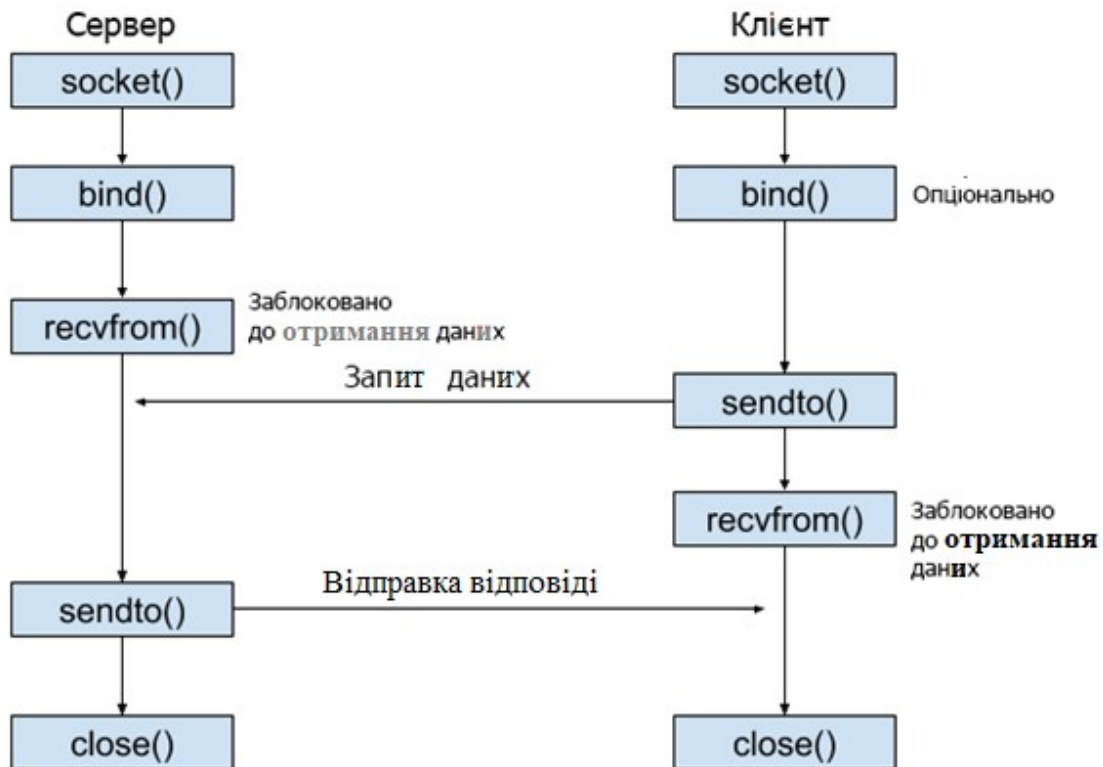


Рисунок 12.1 – Схема роботи дейтаграмного сокета UDP

На рисунку 12.1:

socket() — насамперед сокет визначається як для сервера, так і для клієнта. Це не обов'язково має відбуватися одночасно.

bind() — сокету надається адреса(IP-адреса і порт) на локальному комп'ютері. Для клієнтського сокету це необов'язково, тому що навіть якщо клієнтський сокет не прив'язаний до адреси, прив'язування виконується автоматично кожного разу, коли клієнт ініціює підключення до сервера.

recvfrom() — після прив'язування до порту комп'ютера серверний сокет очікує підключення від клієнтського сокету. Виконання поточного сокету призупиняється (блокується) до того часу, поки серверний сокет не отримає з'єднання. Те ж саме відбувається і з клієнтським сокетом при очікуванні відповіді сервера.

sendto() — після з'єднання з клієнтом серверний сокет відправляє дані клієнту. Цей же метод використовується клієнтським сокетом для виконання запиту на підключення до сервера.

close() — після успішного обміну даними обидва сокети закриваються, тобто звільнюються ресурси системи, які були виділені для сокетів.

Функція **sendto** дуже схожа на **send**. Два додаткові параметри **to** і **toalen** використовуються для запису адреси отримувача. Для запису адреси використовується структура **sockaddr**, як і у випадку з функцією **connect**. Функція **recvfrom** працює аналогічно **recv**. Отримавши чергове повідомлення, вона записує його адресу в структуру, на яку посилається **from**, а записану кількість байтів — в змінну, що адресується вказівником **fromlen**. Як ми знаємо, аналогічно працює функція **accept**.

Деяку плутанину вносять *приєднані дейтаграмні сокети* (connected datagram sockets). Справа в тому, що для сокету з типом **SOCK_DGRAM** також можна викликати функцію **connect**, а потім використати **send** і **recv** для обміну даними. Потрібно розуміти, що ніякого з'єднання при цьому не встановлюється. Операційна система просто запам'ятовує адресу, яку ми передали функції **connect**, а потім використовує його при відправці даних. Зверніть увагу, що приєднаний сокет може отримувати дані *тільки* від сокету, з яким він з'єднаний.

Для ілюстрації процесу обміну дейтаграмами напишіть дві невеличкі програми - **sender** (лістинг 1) і **receiver** (лістинг 2). Перша відправляє повідомлення "Hello there!" і "Bye bye!", а друга отримує їх і виводить на екран. Програма **sender** демонструє застосування як звичайного, так і приєднаного сокету, а **receiver** використовує звичайний.

Лістинг 1. Програма **sender**.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char msg1[] = "Hello there!\n";
char msg2[] = "Bye bye!\n";

int main()
{
    int sock;
    struct sockaddr_in addr;

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    sendto(sock, msg1, sizeof(msg1), 0,
```

```

        (struct sockaddr *)&addr, sizeof(addr));

connect(sock, (struct sockaddr *)&addr, sizeof(addr));
send(sock, msg2, sizeof(msg2), 0);

close(sock);

return 0;
}

```

Лістинг 2. Програма receiver.

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

int main()
{
    int sock;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("bind");
        exit(2);
    }

    while(1)
    {
        bytes_read = recvfrom(sock, buf, 1024, 0, NULL, NULL);
        buf[bytes_read] = '\0';
        printf(buf);
    }

    return 0;
}

```

1.2. Використання низькорівневих сокетів

Низькорівневі сокети надають нові можливості. Вони надають програмісту повний контроль над вмістом пакетів, які відправляються в мережу. З іншого боку, вони більш складні в використанні. Тому використовувати їх можна тільки у випадку необхідності. Наприклад, без них не обійтись при створенні системних утиліт типу **ping** і **traceroute**.

Спочатку з'ясуємо, чим низькорівневі сокети відрізняються від звичайних. Працюючи із звичайними сокетами, ми передаємо системі "чисті" дані, а вона

сама турбується про додавання до них необхідних заголовків (а іноді ще і кінцевників). Наприклад, коли ми надсилаємо повідомлення через UDP-сокети, до нього додається спочатку UDP-заголовок, потім IP-заголовок, а в самому кінці – заголовок апаратного протоколу, який використовується в нашій локальній мережі (наприклад, Ethernet). В результаті створюється кадр, показаний на рисунку 12.2.

Заголовок Ethernet	Заголовок IP	Заголовок UDP	Дані
--------------------	--------------	---------------	------

Рисунок 12.2 – Приклад структури кадру.

Низькорівневі сокети дозволяють включати в буфер з даними заголовки деяких протоколів. Наприклад, ви можете включити в ваше повідомлення TCP- або UDP-заголовок, надавши системі можливість сформувати для вас IP-заголовок, а можете взагалі сформувати усі заголовки самостійно. Звісно, при цьому вам доведеться вивчити роботу відповідних протоколів і суворо дотримуватись форматів їхніх заголовків, інакше програма працювати не буде.

При роботі з низькорівневими сокетами вам доведеться вказувати в третьому параметрі функції **socket** той протокол, до заголовку якого ви хочете отримати доступ. Константи для основних протоколів Internet оголошені в файлі **netinet/in.h**. Вони мають вигляд

IPPROTO_XXX, де XXX–назва протоколу: **IPPROTO_TCP**, **IPPROTO_UDP**, **IPPROTO_RAW** (в останньому випадку ви отримаєте можливість працювати із "сирим" IP і формувати IP-заголовки вручну).

Застереження

Всі числові дані в заголовках мають бути записані в мережевому форматі. Тому не забувайте застосовувати функції **htons** і **htonl**.

Щоб проілюструвати все це прикладом, переписіть програму sender із попереднього розділу з використанням низькорівневих UDP-сокетів. При цьому вручну формується UDP-заголовок повідомлення, що відправляється. Виберемо для прикладу UDP, тому що у цього протоколу заголовок має доволі простий вигляд (рисунок 12.3).

Порт відправника (16 біт)	Порт отримувача (16 біт)
Довжина (заголовок+дані) (16 біт)	Контрольна сума (16 біт)

Рисунок 12.3 – Структура UDP-заголовку.

Код прикладу приведений в лістингу 3. Звернімо увагу на кілька моментів. По-перше, не задається номер порту в структурі **sockaddr_in**. Оскільки цей

номер присутній в UDP-заголовку, від поля **sin_port** вже нічого не залежить. По-друге, замість контрольної суми записаний нуль, щоб не турбувати себе її розрахунком. Протокол UDP є ненадійним, тому він допускає таку свободу. Але інші протоколи (наприклад, IP) можуть і не допускати. Нарешті, зверніть увагу, що всі дані UDP-заголовку форматуються з використанням **htons**.

Лістинг 3. Програма sender з використанням низькорівневих сокетів.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct UdpHeader
{
    u_short src_port;
    u_short targ_port;
    u_short length;
    u_short checksum;
};

char message[] = "Hello there!\n";
char msgbuf[1024];

int main()
{
    int sock;
    struct sockaddr_in addr;
    struct UdpHeader header;

    sock = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
    if(sock < 0)
    {
        perror("socket");
        exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

    header.targ_port = htons(3425);
    header.length = htons(sizeof(header)+sizeof(message));
    header.checksum = 0;

    memcpy((void *)msgbuf, (void *)&header, sizeof(header));
    memcpy((void *) (msgbuf+sizeof(header)), (void *)message,
sizeof(message));

    sendto(sock, msgbuf, sizeof(header)+sizeof(message), 0,
        (struct sockaddr *)&addr, sizeof(addr));

    close(sock);

    return 0;
}
```

1.3. Функції для роботи з адресами і DNS

В цьому розділі ми обговоримо кілька функцій, без яких написати тестовий приклад можна, але без яких навряд чи обійдеться реальна програма. Оскільки для ідентифікації хостів в Internet широко використовуються доменні імена, ми маємо вивчити механізм перетворення їх в IP-адреси. Крім того, ми вивчимо кілька зручних допоміжних функцій.

IP-адреси, зазвичай, записують у вигляді чотирьох чисел, розділених крапками. Для перетворення адреси, записаної в такому форматі, в число і навпаки використовується сімейство функцій **inet_addr**, **inet_aton** і **inet_ntoa**.

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

int inet_aton(const char *cp, struct in_addr *in_p);
unsigned long int inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

Функція **inet_addr** часто використовується в програмах. Вона приймає рядок і повертає адресу (вже з мережевим порядком слідування байтів). Проблема з цією функцією полягає у тому, що значення -1, яке вона повертає у випадку помилки, в той же час є коректною адресою 255.255.255.255 (широкомовна адреса). Ось чому зараз рекомендується використовувати більш нову функцію **inet_aton** (Ascii TO Network). Для зворотнього перетворення використовується функція **inet_ntoa** (Network TO Ascii). Обидві ці функції працюють з адресами в мережевому форматі. Зверніть увагу, що в разі помилки вони повертають 0, а не -1.

Для перетворення доменного імені в IP-адресу використовується функція **gethostbyname**.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Ця функція отримує ім'я хосту і повертає вказівник на структуру із його описом. Розглянемо цю структуру більш детально.

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

- **h_name.** Ім'я хосту.
- **h_aliases.** Масив рядків, які містять псевдоніми хосту. Масив закінчується значенням NULL.
-

- **h_addrtype**. Тип адреси. Для Internet-домену - **AF_INET**.
- **h_length**. Довжина адреси в байтах.
- **h_addr_list**. Масив, що містить адреси всіх мережевих інтерфейсів хосту. Завершується нулем. Зверніть увагу, що байти кожної адреси зберігаються у мережевому порядку, тому **htonl** викликати не потрібно.

Як бачимо, **gethostbyname** повертає достатньо повну інформацію. Якщо нас цікавить адреса хосту, ми можемо вибрати її із масиву **h_addr_list**. Часто беруть саму першу адресу (як ми бачили вище, для посилання на неї визначений спеціальний макрос **h_addr**). Для визначення імені хосту по адресі використовується функція **gethostbyaddr**. Замість рядка вона отримує адресу (в вигляді **sockaddr**) і повертає вказівник на ту ж саму структуру **hostent**. Використовуючи ці дві функції, потрібно пам'ятати, що вони повідомляють про помилку не так, як інші: замість вказівника повертається **NULL**, а розширений код помилки записується в глобальну змінну **h_errno** (а не **errno**). Відповідно, для виводу діагностичного повідомлення потрібно використовувати **herror** замість **perror**.

ЗАСТЕРЕДЖЕННЯ

Треба мати на увазі, що функції **gethostbyname** і **gethostbyaddr** повертають вказівник на статичну область пам'яті. Це означає, що кожне нове звернення до однієї з цих функцій викличе перезаписування даних, отриманих у попередньому зверненні.

Підсумовуючи, розглянемо ще одне сімейство корисних функцій - **gethostname**, **getsockname** і **getpeername**.

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

Функція **gethostname** використовується для отримання імені локального хосту. Далі його можна перетворити в адресу за допомогою **gethostbyname**. Це дає нам спосіб у будь-який момент програмно отримати адресу машини, на якій виконується наша програма, що може бути корисним у багатьох випадках.

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

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

Функція **getpeername** дозволяє в будь-який момент дізнатися про адресу сокету на протилежному кінці з'єднання. Вона отримує дескриптор сокету, з'єданого з віддаленим хостом, і записує адресу цього хосту в структуру, на яку вказує **addr**. Фактична кількість записаних байтів записується за адресою **addrlen** (не забудьте записати туди розмір структури **addr** до виклику **getpeername**). Отриману адресу, при необхідності, можна перетворити в рядок, використовуючи **inet_ntoa** або **gethostbyaddr**. Функція **getsockname** по

призначенню зворотня `getpeername` і дозволяє визначити адресу сокету на "нашому кінці" з'єднання.

1.4. Робота по стандартним протоколам

Нагадаємо, що сокети можуть бути використані при написанні додатків, що працюють по протоколам прикладного рівня Internet (HTTP, FTP, SMTP і т. д.). При цьому взаємодія клієнта і сервера відбувається по тій же самій схемі, що і взаємодія ехо-клієнта і ехо-сервера в нашому прикладі. Різниця у тому, що дані, якими обмінюються клієнт і сервер, інтерпретуються у відповідності з приписами відповідного протоколу.

Наприклад, веб-сервер може працювати по такому алгоритму:

1. Створюємо слухаючий сокет і прив'язуємо його до 80-го порту (стандартний порт для HTTP-сервера).
2. Приймаємо черговий запит на з'єднання.
3. Читаємо HTTP-запит від клієнта (він має стандартний формат і описаний в RFC2616).
4. Обробляємо запит і відправляємо клієнту відповідь, яка також має стандартний формат.
5. Розриваємо з'єднання.

Веб-браузер, який є клієнтом для веб-серверу, може використовувати схожий алгоритм.

1. З'єднуємося з сервером за заданою адресою.
2. Відправляємо йому HTTP-запит.
3. Отримуємо і обробляємо відповідь сервера (наприклад, форматуємо і виводимо на екран отриману HTML-сторінку).
4. Розриваємо з'єднання.

Як бачимо, в роботі по стандартним протоколам немає нічого складного або принципово нового.

2. Сокети на платформі Windows

В світі Internet взаємодія програм, що працюють на різних платформах, зустрічається дуже часто. Так, практично щосекунди черговий Internet Explorer під'єднується до веб-серверу Apache, а черговий Netscape Navigator абсолютно спокійно під'єднується до IIS. Ось чому доволі корисно писати програми так, щоб їх можна було легко переносити на інші платформи. В цьому розділі ми розглянемо, як переносити Linux-програми, які використовують сокети, на платформу Windows.

Перелік основних відмінностей socket API від Winsock API приблизно має такий вигляд:

- В Windows набір файлів заголовків суттєво зменшений. Власне кажучи, вам потрібно включити всього один файл `winsock.h` (або `winsock2.h`, якщо ви хочете використовувати розширені можливості Winsock 2).
- В Windows бібліотеку Winsock необхідно явно проініціювати до звернення до будь-яких інших функцій із неї. Це виконується за допомогою функції **WSAStartup**. Крім того, існує функція **WSACleanup**, яку слід викликати після завершення роботи з сокетом.
- Ми вже знаємо, що в Linux дескриптори сокетів мають тип **int**. У Windows сокети не є файловими дескрипторами, тому для них введений свій тип **SOCKET**. Хоча цей тип і оголошений як **u_int**, сподіватися на це в програмі не варто.
- В Windows для роботи з сокетом не використовуються функції файлового вводу/виводу (**read** і **write**). Замість **close** використовується **closesocket**.
- В Windows глобальна змінна **errno** не використовується. Замість цього код останньої помилки зберігається системою для кожного потоку окремо. Щоб його отримати, використовується функція **WSAGetLastError**.
- В Windows введені додаткові константи, які слід застосовувати замість конкретних чисел. Так, значення, які повертаються функціями Winsock, слід порівнювати з константами **INVALID_SOCKET** або **SOCKET_ERROR**, а не з -1.

Якщо переписати наш ехо-клієнт з врахуванням приведених особливостей Winsock API, а потім зкомпілювати його під Windows (наприклад, за допомогою Visual C++), він повністю буде в змозі взаємодіяти з ехо-сервером, що працює під Linux. Таким чином, сокети дозволяють вирішити проблему кросплатформної взаємодії двох додатків.

На жаль, відмінності socket API і Winsock не обмежуються приведеним вище переліком. При присвоюванні портів складніших програм починають виникати більш принципові проблеми. Наприклад, під Windows існують обмеження щодо підтримки низькорівневих сокетів (вони вперше з'явилися в специфікації Winsock 2, а можливість напряму маніпулювати IP-заголовками доступна тільки під Windows 2000). Крім того, проблеми можуть виникнути з функціями, які не мають прямого стосунку до socket API. Так, в Windows немає прямого аналогу функції **fork**, і для організації паралельного обслуговування клієнтів доведеться вдатися до інших засобів.

Завдання:

- 1) Вивчити викладений матеріал.
- 2) Набрати приклади, продемонструвати роботу сокетів викладачу.
- 3) Бути готовим відповісти на запитання по викладеному матеріалу.

Лабораторна робота 13

Мережева файлова система *NFS*

Мета роботи: ознайомитися з принципами організації мережевої файлової системи та отримати практичні навички в налаштуванні сервера мережевої файлової системи.

План виконання лабораторної роботи

1. Ознайомитися та засвоїти теоретичні відомості про мережеву файлову систему NFS.
2. Встановити, налаштувати та продемонструвати на своєму комп'ютері взаємодію клієнта з сервером мережевої файлової системи.

1. Короткі теоретичні відомості

Мережева файлова система (далі - NFS) є одним із видів файлових систем, відомих як розподілені файлові системи. Вона дозволяє користувачам звертатися до файлів, які зберігаються на віддалених системах, навіть не підозрюючи, що вони працюють через мережу. Це дозволяє файловим системам бути розподіленими серед багатьох комп'ютерів. Ці віддалені системи можуть знаходитися в тій же кімнаті або на значній відстані.

Щоб отримати доступ до таких файлів, необхідно виконати дві дії. Перше: дистанційна система повинна зробити файли доступними для інших систем у мережі. Друге: ці файли повинні бути змонтовані на локальній системі, щоб до них можна було звертатися. Процес монтування файлів призводить до того, що вони виглядають так, ніби знаходяться в локальній системі. Система, яка робить свої файли доступними із мережі, називається сервером, а система, яка використовує віддалений файл, називається клієнтом.

NFS надає клієнтам прозорий доступ до файлів і файлової системи сервера. На відміну від FTP, який забезпечує повну передачу файлів, NFS здійснює доступ тільки до тих частин файлу, до яких звернувся процес, і основна перевага NFS полягає в тому, що він робить цей доступ прозорим. Це означає, що будь-який додаток клієнта, який може працювати з локальним файлом, з таким же успіхом може працювати і з NFS-файлом, без додаткових модифікацій самої програми.

На рисунку 13.1 показана типова схема взаємодії NFS-клієнта і NFS-сервера.

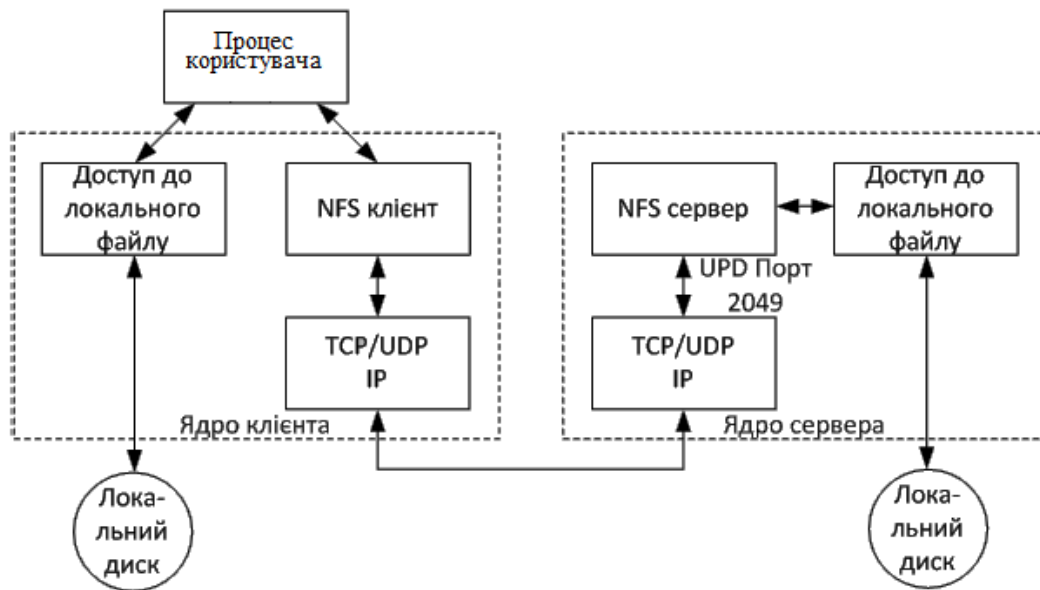


Рисунок 13.1 – Типова схема взаємодії NFS-клієнта і NFS- сервера

Розглядаючи рисунок 13.1 слід зазначити:

1. Мережева файлова система дозволяє клієнту отримати прозорий доступ до файлів на інших комп'ютерах, тобто з боку користувача це виглядає так, ніби ці файли розташовані на локальному комп'ютері. Після того як файл відкритий, ядро передає всі звернення до локальних файлів через модуль, позначений як "доступ до локальних файлів", а всі посилання на NFS-файли передаються через модуль "NFS-клієнт".

2. NFS-клієнт відправляє RPC-запити (Remote Procedure Calls - віддалений виклик процедур) NFS-серверу через модуль TCP/IP. NFS зазвичай використовує протокол UDP, однак більш нові реалізації можуть використовувати протокол TCP.

3. NFS-сервер отримує запити від клієнта у вигляді UDP-дейтатаграм на порт 2049. Незважаючи на те, що NFS може працювати з перетворювачем портів, що дозволяє серверу використовувати динамічно призначені порти, UDP-порт 2049 жорстко закріплений за NFS у більшості реалізацій.

4. Коли NFS-сервер отримує запит від клієнта, він передає його локальній підпрограмі доступу до файлу, яка забезпечує доступ до локального диску на сервері.

Більшість Unix-хостів може функціонувати як NFS-клієнт і як NFS-сервер, або як і те й інше одночасно. Більшість реалізацій Windows мають тільки реалізації NFS-клієнта. Більшість IBM мейнфреймів надає тільки функції NFS-сервера.

Крім підтримки багатьох інших типів файлових систем, у *FreeBSD* вбудована підтримка мережевої файлової системи (*Network File System*). *NFS* дозволяє системі використовувати каталоги і файли разом з іншими

машинами, за допомогою мережі. Використовуючи *NFS* користувачі і програми можуть отримувати доступ до файлів на віддалених системах так само, як би це були файли на локальних дисках.

Ось деякі найбільш помітні переваги, які надає використання *NFS*:

- Окремо взяті робочі станції менше використовують власний дисковий простір, оскільки загальні дані можуть зберігатися на одній окремій машині і бути доступними для інших машин в мережі.
- Пристрої зберігання інформації, такі, як приводи *CD-ROM* і пристрої *Zip®*, можуть використовуватися іншими машинами в мережі.
- У великих мережах може виявитись більш зручним налаштувати центральний сервер *NFS*, на якому розташовуються усі домашні каталоги користувачів. Ці домашні каталоги можуть потім експлуатуватися в мережі так, що користувачі завжди будуть мати один і той же домашній каталог незалежно від того, на якій робочій станції вони працюють.
- Кілька машин можуть мати загальний каталог */usr/ports/distfiles*. Таким чином, коли вам потрібно буде встановити порт (тут порт – це програма) на кілька машин, ви зможете швидко отримати доступ до текстів без їх завантаження на кожну машину.

При неправильному налаштуванні *NFS* також створює потенційну небезпеку несанкціонованого доступу до диску через мережу, і, як наслідок, розголошення і псування інформації. Тому при використанні мережових файлових систем потрібно приділяти багато уваги безпеці та контролю доступу.

NFS складається із двох основних частин: сервера і одного або кількох клієнтів. Клієнт звертається до даних, які знаходяться на сервері, в режимі віддаленого доступу. Для того, щоб це нормально функціонувало, потрібно налаштувати і запустити кілька процесів.

На сервері працюють такі демони:

Демон	Опис
<i>nfsd</i>	Демон <i>NFS</i> , який обслуговує запити від клієнтів <i>NFS</i> .
<i>mountd</i>	Демон монтування <i>NFS</i> , який виконує запити, що передаються йому від <i>nfsd</i> .
<i>portmap</i>	Демон відображення портів дозволяє клієнтам <i>NFS</i> визначити порт, який використовується сервером <i>NFS</i> .

Застереження: В *FreeBSD 5.X*, утиліта *portmap* була замінена на утиліту *rpcbind*.

Клієнт може запустити також демон, який називається *nfsiod* і обслуговує запити, що надходять від сервера *NFS*. Він не обов'язковий, збільшує продуктивність, проте для нормальної і правильної роботи не потрібен.

2. Налаштування *NFS*

Налаштування *NFS* полягає у забезпеченні запуску всіх необхідних процесів під час завантаження. Для цього на *NFS*-сервері у файлі */etc/rc.conf* необхідно виконати такі налаштування:

```
portmap_enable="YES"
```

```
nfs_server_enable="YES"
```

```
nfs_server_flags="-u -t -n 4"
```

```
mountd_flags="-r" (mountd запускається автоматично, якщо задіяна функція сервера NFS).
```

На клієнті треба впевнитися, що у файлі */etc/rc.conf* присутній такий параметр:

```
nfs_client_enable="YES"
```

Файл */etc/exports* визначає, які файлові системи на нашому сервері *NFS* будуть експортуватися (іноді їх називають такими, що спільно використовуються). Кожний рядок в */etc/exports* задає файлову систему, яка буде експортуватися і які машини будуть мати до неї доступ. Крім машин, що мають доступ, можуть бути вказані і інші параметри, які впливають на характеристики доступу. Існує повний набір параметрів, які можна використовувати, але ми розглянемо лише деякі з них. Опис решти параметрів можна знайти на сторінках довідкової системи по *exports* (<http://www.freebsd.org/cgi/man.cgi?query=exports&sektion=5>).

Ось кілька із можливих рядків у файлі */etc/exports*:

```
/cdrom -ro host1 host2 host3
```

цей запис дозволяє експортувати каталог */cdrom* для трьох машин, які знаходяться в тому ж самому домені, що і сервер (тому відсутнє доменне ім'я для кожної машини) або для яких є записи у файлі */etc/hosts*. Прапорець **-ro** вказує на використання файлової системи, що екпортується, в режимі "тільки читати". З цим прапорцем віддалена система не зможе ніяким чином змінити файлову систему, що екпортується;

```
/home -alldirs 192.168.0.2 192.168.0.3 192.168.0.4
```

цим рядком екпортується файлова система */home*, яка стає доступною трьом хостам, вказаних їхніми *IP*-адресами. Це корисно, якщо у вас є власна мережа без налаштованого сервера *DNS*. Як варіант, файл */etc/hosts* може містити внутрішні імена хостів. Прапорець **-alldirs** дозволяє розглядати підкаталоги як точки монтування. Іншими словами, це не монтування підкаталогів, а дозвіл клієнтам монтувати тільки каталоги, які їм потрібні;

```
/a -maproot=root host.example.com box.example.org
```

цим рядком файлова система */a* екпортується таким чином, що вона доступна двом клієнтам із інших доменів. Параметр **-maproot=root** дозволяє користувачу *root* віддаленої системи здійснювати запис у файлову систему, що екпортується, як користувачу *root*. Якщо параметр **-maproot=root** не заданий, то навіть якщо користувач має права доступу *root* на віддаленій

системі, він не може модифікувати файли у файловій системі, що експортується.

Для того, щоб клієнт міг звернутися до файлової системи, він повинен мати право це зробити. Перевірте, що клієнт вказаний у вашому файлі */etc/exports*.

У файлі */etc/exports* кожен рядок містить інформацію про експортування для окремої файлової системи для окремо взятого хоста. Віддалений хост може бути заданий тільки один раз для кожної файлової системи, і може мати тільки один запис, який використовується по замовчуванню, для кожної локальної файлової системи. Наприклад, допустимо, що */usr* є окремою файловою системою. Тоді приведені нижче заповнення файлу */etc/exports* будуть некоректним:

```
# Invalid when /usr is one file system
/usr/src client
/usr/ports client
```

Помилка полягає в тому, що одна файлова система, */usr*, має два рядки, що задають експортування для одного і того ж хосту, *client*. Правильний формат в цьому випадку такий:

```
/usr/src /usr/ports client
```

Властивості окремої файлової системи, що експортується деякому хосту, мають задаватися в одному рядку. Рядки без зазначення клієнта сприймаються як окремих хост. Це обмежує можливості експортування файлової системи, але для більшості випадків це не проблема.

Нижче приведений приклад правильного списку експортування, де */usr* і */exports* є локальними файловими системами:

```
# Експортуємо src і ports для client01 і client02, але
# тільки client01 має права користувача root на них
/usr/src /usr/ports -maproot=root client01
/usr/src /usr/ports client02
# Клієнтські машини мають користувача root і можуть монтувати все в
# каталозі /exports. Хто завгодно може монтувати /exports/obj в режимі
читання
/exports -alldirs -maproot=root client01 client02
/exports/obj -ro
```

Ви маєте перезапустити **mountd** після того, як змінили */etc/exports*, щоб зміни вступили в силу. Це може бути досягнуто посилкою сигналу *HUP* процесу *mountd*:

```
# kill -HUP `cat /var/run/mountd.pid`
```

Як варіант, при перезавантаженні *FreeBSD* все налаштується правильно. Хоча виконувати перезавантаження зовсім не обов'язково. Виконання наступних команд користувачем *root* запустить все, що потрібно.

На сервері *NFS*:

```
# portmap
```

```
# nfsd -u -t -n 4
```

```
# mountd -r
```

На клієнті *NFS*:

```
# nfsiod -n 4
```

Тепер все має бути готовим до реального монтування віддаленої файлової системи. Якщо ви тільки хочете тимчасово змонтувати віддалену файлову систему, або всього лише протестувати ваші налаштування, то просто запустіть команди, подібні тим, що приведені тут, від імені користувача *root* на клієнтській машині:

```
# mount server:/home /mnt
```

По цій команді файлова система */home* на сервері буде змонтована в каталог */mnt* на клієнті. Якщо все налаштовано правильно, ви зможете увійти в каталог */mnt* на клієнті і побачити файли, які знаходяться на сервері.

Якщо ви хочете автоматично монтувати віддалену файлову систему при кожному завантаженні комп'ютера, додайте файлову систему в */etc/fstab*. Наприклад:

```
server:/home /mnt nfs rw 0 0
```

На сторінках довідкової системи по *fstab* перераховані всі доступні параметри (<http://www.freebsd.org/cgi/man.cgi?query=fstab&sektion=5>).

Автоматичне монтування з використанням *amd*

Демон автоматичного монтування *amd* автоматично монтує віддалену файлову систему, як тільки відбувається звернення до файлу або каталогу в цій файловій системі. Крім того, файлові системи, які були неактивні деякий час, будуть автоматично розмонтовані демоном *amd*. Використання *amd* є простою альтернативою статичному монтуванню, оскільки в останньому випадку зазвичай все має бути описане у файлі */etc/fstab*.

Коли відбувається звернення до файлу в одному із каталогів: *host* або */net*, *amd* шукає відповідний віддалений ресурс для монтування і автоматично його монтує. Каталог */net* використовується для монтування експортованої файлової системи за адресою *IP*, а каталог */host* використовується для монтування ресурсу по віддаленому імені хоста.

Звернення до файлу в каталозі */host/foobar/usr* покаже *amd*, що була спроба монтування ресурсу */usr*, який знаходиться на хості *foobar*.

Приклад монтування ресурсу за допомогою *amd*

Ви можете переглянути доступні для монтування ресурси віддаленого хоста командою *showmount*. Наприклад, щоб переглянути ресурси хоста з іменем *foobar*, ви можете використати:

```
% showmount -e foobar
Exports list on foobar:
/usr      10.10.10.0
/a        10.10.10.0
% cd /host/foobar/usr
```

Як видно із прикладу, *showmount* показує */usr* як експортований ресурс. При переході в каталог */host/foobar/usr* демон *amd* намагається перетворити ім'я хоста *foobar* і автоматично монтувати потрібний ресурс.

Демон ***amd*** може бути запущений із скриптів початкового завантаження, якщо помістити такий рядок у файл */etc/rc.conf*:

```
amd_enable="YES"
```

Крім того, демону ***amd*** можуть бути передані налагоджувальні прапорці через параметр *amd_flags*. По замовчуванню *amd_flags* налаштований наступним чином:

```
amd_flags="-a /.amd_mnt -l syslog /host /etc/amd.map /net /etc/amd.map"
```

Файл */etc/amd.map* задає опції, які використовуються по замовчуванню при монтуванні експортованих ресурсів. В файлі */etc/amd.conf* задані налаштування деяких більш складних можливостей ***amd***.

Завдання до лабораторної роботи.

1. Налаштуйте *NFS*-сервер.
2. Забезпечте автоматичне монтування віддаленої файлової системи.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Комп'ютерні мережі: підручник / Азаров О.Д., Захарченко С.М., Кадук О.В., Орлова М.М., Тарасенко В.П. – Вінниця: ВНТУ. – 2020. – 378 с.
2. Комп'ютерні мережі: навчальний посібник / Азаров О.Д., Захарченко С.М., Кадук О.В., Орлова М.М., Тарасенко В.П.. – Вінниця: ВНТУ. – 2013. – 371 с.
3. Таненбаум Э., Уэзерпол Д. Компьютерные сети. – СПб.: Питер. – 2012. – 960 с.
4. В.Г.Олифер, Н.А. Олифер. Компьютерные сети. Принципы, технологии, протоколы. Учебник. ПИТЕР. – 2020. – 467 с.
5. В. А. Ганжа, В. В. Шиманский. Компьютерные сети. Минск БГУИР, 2015.
6. К. Дуглас. Тср/ір крупным планом. MirKnig, 2009.
7. Крис Сандерс. Анализ пакетов: практическое руководство по использованию Wireshark и tcpdump для решения реальных проблем в локальных сетях. 3-е издание. Издательский дом Вильямс, 2018.