ЛЕКЦІЯ 12. МЕРЕЖНІ ЗАСОБИ ОПЕРАЦІЙНИХ СИСТЕМ

- Багаторівнева мережна архітектура і мережні протоколи
- Реалізація стека протоколів TCP/IP
- Система імен DNS
- Програмний інтерфейс сокетів Берклі
- Архітектура мережної підтримки Linux та Windows XP
- Програмний інтерфейс Windows Sockets

12.1. Загальні принципи мережної підтримки

Під *мережею* розуміють набір комп'ютерів або апаратних пристроїв *(вузлів,* nodes), пов'язані між собою каналами зв'язку, які можуть передавати інформацію один одному. Мережа має конкретну фізичну структуру (спосіб з'єднання вузлів, топологію), усі вузли підключають до мережі із використанням апаратного забезпечення, яке відповідає цій структурі. Звичайно мережа об'єднує обмежену кількість вузлів.

Під ***інтернетом*** (з малої літери) розуміють сукупність мереж, які використовують один і той самий набір *мережних протоколів* – правил, що визначають формат даних для пересилання мережею. Фізична структура окремих мереж, які входять до складу інтернету, може різнитися. Такі різнорідні мережі пов'язують одну з одною *маршрутизатори* (routers), які переадресовують пакети з однієї мережі в іншу, залежно від їхньої адреси призначення (маршрутизують їх) і при цьому перетворюють пакети між форматами відповідних мереж. Маршрутизатори підтримують міжмережну взаємодію (internetworking).

Відомий усім ***Інтернет*** (з великої літери) – це, фактично, сукупність пов'язаних між собою інтернетів, відкритих для публічного доступу, які використовують визначений набір протоколів (стек протоколів TCP/IP) і охоплюють увесь світ.

Функції забезпечення зв'язку між вузлами є досить складними. Для спрощення їхньої реалізації широко використовують багаторівневий підхід – вертикальний розподіл мережних функцій і можливостей. Він дає змогу приховувати складність реалізації функцій зв'язку: кожен рівень приховує від вищих рівнів деталі реалізації своїх функцій та функцій, реалізованих на нижчих рівнях.

Мережний сервіс – це набір операцій, які надає рівень мережної архітектури для використання її на вищих рівнях. Сервіси визначено як частину специфікації інтерфейсу рівня.

Розрізняють сервіси, *орієнтовані на з'єднання* (connection-oriented services), і *без з'єднань,* або *дейтаграмні* сервіси (connectionless services).

- Сервіси, орієнтовані на з'єднання, реалізують три фази взаємодії із верхнім рівнем: встановлення з'єднання, передавання даних і розрив з'єднання. При цьому передавання даних на верхніх рівнях здійснюють у вигляді неперервного потоку байтів.
- Дейтаграмні сервіси реалізують пересилання незалежних повідомлень, які можуть переміщатися за своїми маршрутами і приходити у пункт призначення в іншому порядку.

Зазначимо, що реалізацію сервісу на рівні ОС або у вигляді прикладної програми, що надає доступ до деякої системної функціональності через мережу, називають *мережною службою.* Визначення мережного сервісу для конкретного рівня мережної архітектури описує функціональність цього рівня, але не задає її реалізацію. Реалізацію функціональності для конкретного сервісу визначають ***мережні протоколи***.

Мережний протокол – це набір правил, що задають формат повідомлень, порядок обміну повідомленнями між сторонами та дії, необхідні під час передавання або приймання повідомлень.

Кажуть, що мережний протокол А працює поверх мережного протоколу В (А – протокол вищого рівня, а В – нижчого), коли пакети з інформацією, що відповідають протоколу А, під час передавання мережею розміщені всередині пакетів протоколу В. Процес розміщення одних пакетів усередині інших називають *інкапсуляцією пакетів* (packet encapsulation). У разі приходу пакета за призначенням відповідне програмне забезпечення по черзі «знімає конверти», переглядаючи заголовки пакетів, приймаючи рішення на основі їхнього вмісту і вилучаючи їх. Процес визначення адресата пакета за інформацією із його заголовків називають *демультиплексуванням пакетів* (packet demultiplexing). Мережний протокол надає два інтерфейси.

1. *Однорівневий, або інтерфейс протоколу* (peer-to-peer interface) призначений для організації взаємодії із реалізацією протоколу того самого рівня на віддаленому мережному вузлі. Це найважливіший інтерфейс протоколу, що реалізує безпосереднє передавання даних на віддалений вузол. Такий інтерфейс звичайно забезпечують заголовком пакета, який доповнюють реалізацією цього протоколу перед передаванням пакета мережею.
2. *Інтерфейс сервісу* (service interface) призначений для взаємодії із засобами вищого рівня; за його допомогою фактично реалізують мережний сервіс. Інтерфейс сервісу забезпечують правилами інкапсуляції пакетів вищого рівня в пакети цього протоколу.

Набір протоколів різного рівня, що забезпечують реалізацію певної мережної архітектури, називають стеком протоколів (protocol stack) або набором протоколів (protocol suite).

Сукупність протоколів, які лежать в основі сучасного Інтернету, називають *набором протоколів Інтернету* (Internet Protocol Suite, IPS) або ***стеком протоколів*** *TCP/IP* за назвою двох основних протоколів. У цьому розділі наведемо основні характеристики такого набору та особливості його реалізації у сучасних ОС.

Мережна архітектура TCP/IP має чотири рівні, яки показані на рис. 12.1. Розглянемо їх знизу вгору.

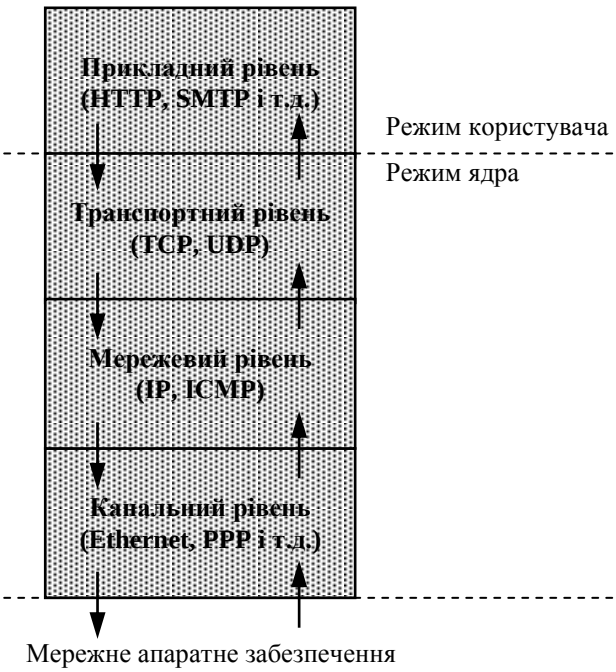

Рис. 12.1. Багаторівнева мережна архітектура TCP/IP

Канальний рівень (data link layer) відповідає за передавання кадру даних між будь-якими вузлами в мережах із типовою апаратною підтримкою (Ethernet, FDDI тощо) або

між двома сусідніми вузлами у будь-яких мережах (SLIP, PPP). При цьому забезпечуються формування пакетів, корекція апаратних помилок, спільне використання каналів. Крім того, на більш низькому рівні він забезпечує передавання бітів фізичними каналами, такими як коаксіальний кабель, кручена пара або оптоволоконний кабель (іноді для опису такої взаємодії виділяють окремий *фізичний рівень* – physical layer).

Перш ніж перейти до наступного рівня, дамо два означення. ***Хостом*** (host) є вузол мережі, де використовують стек протоколів TCP/IP. ***Мережним інтерфейсом*** (network interface) є абстракція віртуального пристрою для зв'язку із мережею, яку надає програмне забезпечення канального рівня. Хост може мати декілька мережних інтерфейсів, зазвичай вони відповідають його апаратним мережним пристроям.

На ***мережному рівні*** (network layer) відбувається передавання пакетів із використанням різних транспортних технологій. Він забезпечує доставлення даних між мережними інтерфейсами будь-яких хостів у неоднорідній мережі з довільною топологією, але при цьому не бере на себе жодних зобов'язань щодо надійності передавання даних. На цьому рівні реалізована адресація інтерфейсів і маршрутизація пакетів. Основним протоколом цього рівня у стеку TCP/IP є IP (Internet Protocol).

Транспортний рівень (transport layer) реалізує базові функції з організації зв'язку між *процесами,* що виконуються на віддалених хостах. У стеку TCP/IP на цьому рівні функціонують протоколи *TCP* (Transmission Control Protocol) і *UDP* (User Datagram Protocol). TCP забезпечує надійне передавання повідомлень між віддаленими процесами користувача за рахунок утворення віртуальних з'єднань. UDP забезпечує ненадійне передавання прикладних пакетів (подібно до IP), виконуючи винятково функції сполучної ланки між IP і процесами користувача (далі на ньому зупинятися не будемо).

Прикладний рівень (application layer) реалізує набір різноманітних мережних сервісів, наданих кінцевим користувачам і застосуванням. До цього рівня належать протоколи, реалізовані різними мережними застосуваннями (службами), наприклад, HTTP (основа організації Web), SMTP (основа організації пересилання електронної пошти).

Основна відмінність прикладного рівня полягає в тому, що у більшості випадків його підтримка реалізована в режимі користувача (звичайно за це відповідають різні прикладні програми-сервери), а підтримка інших рівнів – у ядрі ОС. Завдання мережної служби прикладного рівня – реалізувати сервіс для кінцевого користувача (пересилання електронної пошти, передавання файлів тощо), який не має інформації про особливості переміщення даних мережею. Інші рівні, навпаки, не мають інформації про особливості застосувань, які обмінюватимуться даними за їхньою допомогою.

Реалізація канального рівня звичайно включає драйвер мережного пристрою ОС і апаратний мережний пристрій та приховує від програмного забезпечення верхнього рівня та прикладних програм деталі взаємодії з фізичними каналами, надаючи їм абстракцію мережного інтерфейсу. Передавання даних мережею у програмному забезпеченні верхнього рівня відбувається між мережними інтерфейсами.

Крім того, виділяють спеціальний *інтерфейс зворотного зв'язку* (loopback interface); усі дані, передані цьому інтерфейсу, надходять на вхід реалізації стека протоколів того самого хоста.

Протокол IP надає засоби доставлення дейтаграм неоднорідною мережею без встановлення з'єднання. Він реалізує доставлення за заданою адресою, але при цьому надійність, порядок доставлення і відсутність дублікатів не гарантовані. Усі засоби щодо забезпечення цих характеристик реалізуються у протоколах вищого рівня (наприклад, TCP).

Кожний мережний інтерфейс в IP-мережі має унікальну адресу. Такі адреси називають *IP-адресами.* Стандартною версією IP, якою користуються від початку 80-х років XX століття, є IP версії 4 (IPv4), де використовують адреси завдовжки 4 байти, їх зазвичай записують у крапково-десятковому поданні (чотири десяткові числа, розділені крапками, кожне з яких відображає один байт адреси). Прикладом може бути 194.41.233.1. Спеціальну адресу зворотного зв'язку 127.0.0.1 (loopback address) присвоюють інтерфейсу зворотного зв'язку і використовують для зв'язку із застосуваннями, запущеними на локальному хості.

Як зазначалося, IP доставляє дейтаграми мережному інтерфейсу. Пошук процесу на відповідному хості забезпечують протоколи транспортного рівня (наприклад, TCP). Пакети цих протоколів інкапсулюють в IP-дейтаграми.

Суттєвим недоліком протоколу IPv4 є незначна довжина IP-адреси. Кількість адрес, які можна відобразити за допомогою 32 біт, є недостатньою з огляду на сучасні темпи росту Інтернету. Сьогодні нові IP-адреси виділяють обмежено.

Для вирішення цієї проблеми було запропоновано нову реалізацію IP-протоколу – *IP версії 6* (IPv6), основною відмінністю якої є довжина адреси – 128 біт (16 байт).

Крім IP, на мережному рівні реалізовано й інші протоколи. Для забезпечення мережної діагностики застосовують протокол ICMP (Internet Control Message Protocol), який використовують для передавання повідомлень про помилки під час пересилання IP-дейтаграм, а також для реалізації найпростішого *луна-протоколу,* що реалізує обмін запитом до хосту і відповіддю на цей запит. ICMP-повідомлення інкапсулюють в IP-дейтаграми.

Більшість сучасних ОС мають утиліту ping, яку використовують для перевірки досяжності віддаленого хоста. Ця утиліта використовує луна-протокол у рамках ICMP.

Засоби підтримки мережного рівня, як зазначалося, є частиною реалізації стека протоколів у ядрі ОС. Головними їхніми завданнями є інкапсуляція повідомлень транспортного рівня (наприклад, TCP) у дейтаграми мережного рівня (наприклад, IP) і передавання підготовлених дейтаграм драйверу мережного пристрою, отримання дейтаграм від драйвера мережного пристрою і демультиплексування повідомлень транспортного рівня, маршрутизація дейтаграм.

Пакет з TCP-заголовком називають *TCP-сегментом.* Основні характеристики протоколу TCP такі.

- *Підтримка комунікаційних каналів* між клієнтом і сервером, які називають *з'єднаннями* (connections). TCP-клієнт встановлює з'єднання з конкретним сервером, обмінюється даними з сервером через це з'єднання, після чого розриває його.
- *Забезпечення надійності передавання даних.* Коли дані передають за допомогою TCP, потрібне підтвердження їхнього отримання. Якщо воно не отримане впродовж певного часу, пересилання даних автоматично повторюють, після чого протокол знову очікує підтвердження. Час очікування зростає зі збільшенням кількості спроб. Після певної кількості безуспішних спроб з'єднання розривають. Неповного передавання даних через з'єднання бути не може: або воно надійно пересилає дані, або його розривають.
- *Встановлення послідовності даних* (data sequencing). Для цього кожний сегмент, переданий за цим протоколом, супроводжує номер послідовності (sequence number). Якщо сегменти приходять у невірному порядку, TCP на підставі цих номерів може переставити їх перед тим як передати повідомлення в застосування.
- *Керування потоком даних* (flow control). Протокол TCP повідомляє віддаленому застосуванню, який обсяг даних можливо прийняти від нього у будь-який момент часу.

Це значення називають *оголошеним вікном* (advertised window), воно дорівнює обсягу вільного простору у буфері, призначеному для отримання даних. Вікно динамічно змінюється: під час читання застосуванням даних із буфера збільшується, у разі надходження даних мережею -зменшується. Це гарантує, що буфер не може переповнитися. Якщо буфер заповнений повністю, розмір вікна зменшують до нуля. Після цього TCP, пересилаючи дані, очікуватиме, поки у буфері не вивільниться місце.

- TCP-з'єднання є *повнодуплексними* (full-duplex). Це означає, що з'єднання у будь-який момент часу можна використати для пересилання даних в обидва боки. TCP відстежує номери послідовностей і розміри вікон для кожного напрямку передавання даних.

Для встановлення зв'язку між двома процесами на транспортному рівні (за допомогою TCP або UDP) недостатньо наявності IP-адрес (які ідентифікують мережні інтерфейси хостів, а не процеси, що на цих хостах виконуються). Щоб розрізнити процеси, які виконуються на одному хості, використовують концепцію ***портів*** (ports).

Порти ідентифікують цілочисловими значеннями розміром 2 байти (від 0 до 65 535). Кожний порт унікально ідентифікує процес, запущений на хості: для того щоб TCP-сегмент був доставлений цьому процесові, у його заголовку зазначається цей порт. Процес-сервер звичайно використовує заздалегідь визначений порт, на який можуть вказувати клієнти для зв'язку із цим сервером. Для клієнтів порти зазвичай резервують динамічно (оскільки вони потрібні тільки за наявності з'єднання, щоб сервер міг передавати дані клієнтові).

Для деяких сервісів за замовчуванням зарезервовано конкретні номери портів у діапазоні від 0 до 1023 (відомі порти, well-known ports); наприклад, для протоколу HTTP (веб-серверів) це порт 80, а для протоколу SMTP – 25. В UNIX-системах відомі порти є привілейованими – їх можуть резервувати тільки застосування із підвищеними правами. Відомі порти розподіляються централізовано, подібно до IP-адрес.

12.2. Реалізація стека протоколів TCP/IP

Мережні протоколи стека TCP/IP можна використовувати для зв'язку між рівноправними сторонами, але найчастіше такий зв'язок відбувається за принципом «клієнт-сервер», коли одна сторона (сервер) очікує появи дейтаграм або встановлення з'єднання, а інша (клієнт) відсилає дейтаграми або створює з'єднання.

Розглянемо ***основні етапи процесу обміну даними між клієнтом і сервером*** із використанням протоколу прикладного рівня, що функціонує в рамках стека протоколів TCP/IP. Як приклад такого протоколу візьмемо HTTP, при цьому сторонами, що взаємодіють, будуть веб-браузер (клієнт) і веб-сервер. Припустимо, що локальний комп'ютер зв'язаний з Інтернетом за допомогою мережного пристрою ***Ethernet***. Для простоти вважатимемо, що під час передавання повідомлення не піддають фрагментації. На рис. 12.2 номери етапів позначені цифрами у дужках.

1. Застосування-клієнт (веб-браузер) у режимі користувача формує HTTP-запит до веб-сервера. Формат запиту визначений протоколом прикладного рівня (HTTP), зокрема у ньому зберігають шлях до потрібного документа на сервері. Після цього браузер виконує ряд системних викликів. При цьому у ядро ОС передають вміст HTTP-запиту, IP-адресу комп'ютера, на якому запущено веб-сервер, і номер порту, що відповідає цьому серверу.

   Далі перетворення даних пакета відбувається в ядрі.

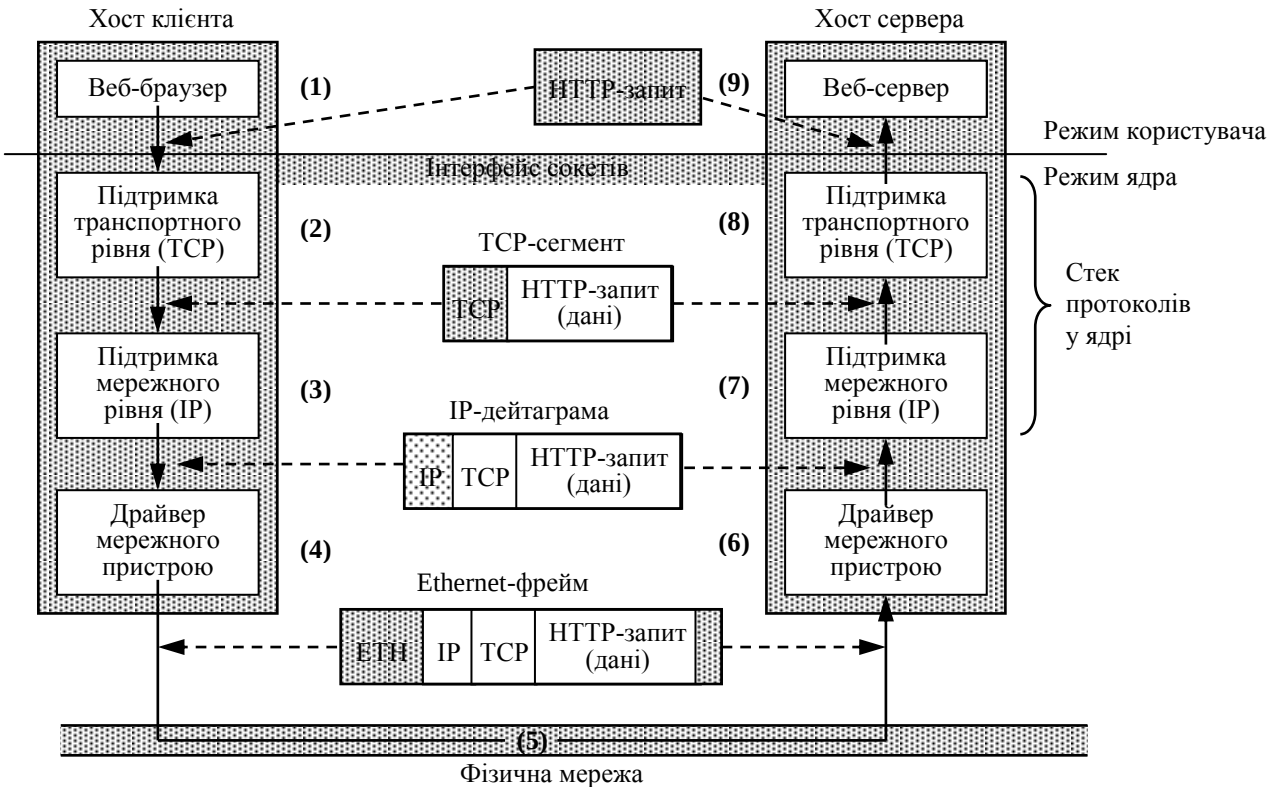

Рис. 12.2. Переміщення пакета у стеку протоколів TCP/IP

2. Спочатку повідомлення обробляють засобами підтримки протоколу транспортного рівня (TCP). У результаті його доповнюють TCP-заголовком, що містить номер порту веб-сервера та інформацію, необхідну для надійного пересилання даних (номер послідовності тощо). HTTP-запит інкапсулюється у TCP-сегмент та стає корисним навантаженням (payload) – даними, які пересилають для обробки в режимі користувача.
3. TCP-сегмент обробляють засобами підтримки протоколу мережного рівня (IP). При цьому він інкапсулюється в IP-дейтаграму (його доповнюють IP-заголовком, що містить IP-адресу віддаленого комп'ютера та іншу інформацію, необхідну для передавання мережею).
4. IP-дейтаграма надходить на рівень драйвера мережного пристрою (Ethernet), який додає до неї інформацію (заголовок і трейлер), необхідну для передавання за допомогою Ethernet-пристрою. Пакет з Ethernet-інформацією називають Ethernet-фрейліолі. Фрейм передають мережному пристрою, який відсилає його мережею. Фрейм містить адресу призначення у Ethernet, що є адресою мережної карти комп'ютера в тій самий локальній мережі (або іншого пристрою, який може переадресувати пакет далі в напрямку до місця призначення). Апаратне забезпечення Ethernet забезпечує реалізацію передавання даних фізичною мережею у вигляді потоку бітів.
   Дотепер пакет переходив від засобів підтримки протоколів вищого рівня до протоколів нижчого. Кажуть, що пакет опускався у стеку протоколів.
5. Тепер пакет переміщатиметься мережею. При цьому можуть здійснюватися різні його перетворення. Наприклад, коли Ethernet-фрейм доходить до адресата в мережі Ethernet, відповідне програмне або апаратне забезпечення виділяє IP-дейтаграму із фрейму, за IP-заголовком визначає, яким каналом відправляти її далі, інкапсулює дейтаграму відповідно до характеристик цього каналу (наприклад, знову в Ethernet-фрейм) і відсилає її в наступний пункт призначення. На шляху повідомлення може перейти в мережі, зв'язані модемами, і тоді формат зовнішньої оболонки буде змінено (наприклад, у формат протоколів SLIP або PPP), але вміст (IP-дейтаграма) залишиться тим самим.

Зрештою, пакет доходить до адресата. Його формат залежить від мережного апаратного забезпечення, встановленого на сервері. Якщо сервер теж підключений до мережі за допомогою мережного адаптера Ethernet, він отримає Ethernet-фрейм, подібний до відісланого клієнтом. Далі відбувається декілька етапів демультиплексування пакетів. Кажуть, що пакет піднімається у стеку протоколів.

6. Драйвер мережного пристрою Ethernet виділяє IP-дейтаграму із фрейму і передає її засобам підтримки протоколу IP.
7. Засоби підтримки IP перевіряють IP-адресу в заголовку, і, якщо вона збігається з локальною IP-адресою (тобто IP-дейтаграма дійшла за призначенням), виділяють TCP-сегмент із дейтаграми і передають його засобам підтримки TCP.
8. Засоби підтримки TCP визначають застосування-адресат за номером порту, заданим у TCP-заголовку (це веб-сервер, що очікує запитів від клієнтів). Після цього виділяють HTTP-запит із TCP-сегмента і передають його цьому застосуванню для обробки в режимі користувача.
9. Сервер обробляє HTTP-запит (наприклад, відшукує на локальному диску відповідний документ).

12.3. Система імен DNS

Розглянемо найважливішу службу прикладного рівня, без якої мережна взаємодія в рамках Інтернету була б фактично неможливою.

Доменна система імен (Domain Name System, DNS) – це розподілена база даних, яку застосування використовують для організації відображення символьних імен хостів (доменних імен) на IP-адреси. За допомогою DNS завжди можна знайти IP-адресу, що відповідає заданому доменному імені. Розподіленість DNS полягає в тому, що немає жодного хосту в Інтернеті, який би мав усю інформацію про це відображення. Кожна група хостів (наприклад, та, що пов'язує всі комп'ютери університету) підтримує свою власну базу даних імен, відкриту для запитів зовнішніх клієнтів та інших серверів. Підтримку бази даних імен здійснюють за допомогою застосування, яке називають DNS-сервером або сервером імен (name server).

Доступ до DNS з прикладної програми здійснюють за допомогою розпізнавача (resolver) – клієнта, який звертається до DNS-серверів для перетворення доменних імен в IP-адреси (цей процес називають *розв'язанням доменних імен* – domain name resolution). Звичайно розпізнавач реалізований як бібліотека, компонована із застосуваннями. Він використовує конфігураційний файл (в UNIX-системах це – /etc/resolv.conf), у якому зазначені IP-адреси локальних серверів імен. Якщо застосування потребує розв'язання доменного імені, код розпізнавача відсилає запит на локальний сервер імен, отримує звідти інформацію про відповідну IP-адресу і повертає її у застосування.

Зазначимо, що і розпізнавач, і сервер імен зазвичай виконуються в режимі користувача (щодо серверів імен це не завжди справедливо: так, у Windows-системах частина реалізації такого сервера виконується в режимі ядра). Стек TCP/IP у ядрі інформацією про DNS не володіє.

Простір імен DNS є ієрархічним (рис. 12.3). Кожний вузол супроводжує символьна позначка. Коренем дерева є вузол із позначкою нульової довжини. Доменне ім'я будь-якого вузла дерева – це список позначок, починаючи із цього вузла (зліва направо) і до кореня, розділених символом «крапка». Наприклад, доменне ім'я виділеного на рис. 12.3 вузла буде «`www.kpi.kharkov.ua.`». Доменні імена мають бути унікальними.

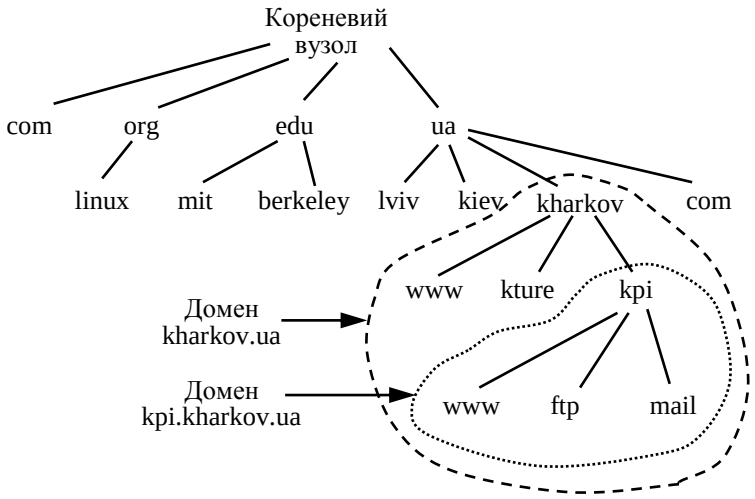

Рис. 12.3. Простір імен DNS

Доменом (domain) називають піддерево ієрархічного простору імен. Для позначення домену (яке ще називають суфіксом домену) використовують доменне ім'я кореня цього піддерева: так, хост `www.kpi.kharkov.ua.` належить домену із суфіксом `kpi.kharkov.ua.`, той, у свою чергу, – домену із суфіксом `kharkov.ua.` і т. д.

Доменне ім'я, що завершується крапкою, називають *повним доменним іменем* (Fully Qualified Domain Name, FQDN). Якщо крапка наприкінці імені відсутня, вважають, що це ім'я може бути доповнене (до нього може бути доданий суфікс відповідного домену). Такі імена можуть використовуватись у рамках домену. Наприклад, ім'я `mail` можна використати для позначення хоста всередині домену `kpi.kharkov.ua.`, повне доменне ім'я для цього хоста буде `mail.kpi.kharkov.ua.`. У застосуваннях крапку наприкінці доменних імен хостів звичайно не ставлять (посилаються на `www.kpi.kharkov.ua` замість `www.kpi.kharkov.ua.`), ми теж далі цього не робитимемо.

Серед доменів верхнього рівня (суфікс для яких не містить крапок, окрім кінцевої) виділяють усім відомі com, edu, org тощо, а також домени для країн (ua для України). Є спеціальний домен `arpa`, який використовують для зворотного перетворення IP-адрес у DNS-імена.

Розподіл відповідальності за зони DNS-дерева – найважливіша характеристика доменної системи імен. Немає жодної організації або компанії, яка б керувала відображенням для всіх позначок дерева. Є спеціальна організація (Network Information Center, NIC), що керує доменом верхнього рівня і делегує відповідальність іншим організаціям за інші зони. Зоною називають частину DNS-дерева, що адмініструється окремо. Прикладом зони є домен другого рівня (наприклад, kharkov.ua). Багато організацій розділяють свої зони на менші відповідно до доменів наступного рівня (наприклад, `kpi.kharkov.ua`, `kture.kharkov.ua` тощо), аналогічним чином делегуючи відповідальність за них. У цьому разі зоною верхнього рівня вважають частину домена, що не включає виділені в ній зони.

Після делегування відповідальності за зону для неї необхідно встановити кілька серверів імен (як мінімум два – основний і резервний). Під час розміщення в мережі нового хоста інформація про нього повинна заноситься у базу даних основного сервера відповідної зони. Після цього інформацію автоматично синхронізують між основним і резервним серверами.

Якщо сервер імен не має необхідної інформації, він її шукає на інших серверах. Процес отримання такої інформації називають ітеративним запитом (iterative query).

Розглянемо ітеративний запит отримання IP-адреси для імені `www.kpi.kharkov.ua`. Спочатку локальний сервер зв'язується із кореневим сервером імен (`root name server`), відповідальним за домен верхнього рівня (`.`). Станом на 2004 рік в Інтернеті було 13 таких серверів, кожен із них мав бути відомий усім іншим серверам імен. Кореневий сервер імен зберігає інформацію про сервери першого рівня. Отримавши запит на відображення імені, він визначає, що це ім'я належить не до його зони відповідальності, а до домену `.ua`, і повертає локальному серверу інформацію про адреси та імена всіх серверів відповідної зони. Далі локальний сервер звертається до одного із цих серверів з аналогічним запитом. Той сервер містить інформацію про те, що для зони `kharkov.ua` є свій сервер імен, у результаті локальний сервер отримує адресу цього сервера. Процес повторюють доти, поки запит не надійде на сервер, відповідальний за домен `kpi.kharkov.ua`, що може повернути коректну IP-адресу.

Кешування IP-адрес дозволяє значно зменшити навантаження на мережу. Коли сервер імен отримує інформацію про відображення (наприклад, IP-адресу, що відповідає доменному імені), він зберігає цю інформацію локально (у спеціальному кеші). Наступний аналогічний запит отримає у відповідь дані з кеша без звертання до інших серверів. Інформацію зберігають у кеші обмежений час. Зазначимо, що, коли для сервера не задана зона, за яку він відповідає, кешування є його єдиним завданням. Це поширений *сервер кешування* (caching-only server).

Елемент інформації у базі даних DNS називають ***ресурсним записом*** (Resource Record, RR). Кожний такий запис має клас і тип. Для записів про відображення IP-адрес класом завжди є IN.

Розглянемо деякі типи DNS-ресурсів. Найважливішим із них є *A -запис,* що пов'язує повне доменне ім'я з IP-адресою. Саме на основі таких записів сервери імен повертають інформацію про відображення. У конфігураційному файлі сервера імен bind для домену kpi.kharkov.ua A-запис для www.kpi.kharkov.ua задають так:

```
www    IN      A     144.91.1.21
```

Ще одним типом запису є *CNAME-запис,* що задає *аліас доменного імені.* Одній і тій самий IP-адресі (A-запису) може відповідати кілька таких аліасів. Аліаси `ftp.kpi.kharkov.ua` і `mail.kpi.kharkov.ua` задають так:

```
ftp     IN     CNAME      www
mail    IN     CNAME      www
```

12.4. Програмний інтерфейс сокетів Берклі

Програмний інтерфейс сокетів Берклі – це засіб зв'язку між прикладним рівнем мережної архітектури TCP/IP і транспортним рівнем. Причина такого розташування полягає в тому, що це засіб зв'язку між кодом застосування (який виконують на прикладному рівні) і реалізацією стека протоколу ОС (найвищим рівнем якої є транспортний). Фактично цей інтерфейс є набором системних викликів ОС.

Назва цього API пов'язана з тим, що він уперше був реалізований у 80-х роках XX століття у версії UNIX, розробленій Каліфорнійським університетом у Берклі (BSD UNIX).

Більшість системних викликів роботи із сокетами приймають як параметри покажчики на спеціальні структури даних, що відображають адреси сокетів. У разі використання TCP/IP зі звичайним IP-протоколом (IPv4) адресу задають структурою `sockaddr_in`. Вона визначена у заголовному файлі `<netinet/in.h>` і містить, зокрема, такі поля:

- sin_family – для TCP/IP має дорівнювати константі AF_INET;
- sin_port – цілочисловий номер порту для TCP або UDP;
- sin_addr – відображення IP-адреси (структура in_addr з одним полем s_addr).

Реалізуючі системні виклики роботи із сокетами, необхідно враховувати можливість використання різних протоколів, а отже, передавання в них різних структур, що відображають адреси. Під час розробки API сокетів Берклі було ухвалено рішення використати як універсальний тип відображення адрес покажчик на спеціальну структуру sockaddr, визначену у <sys/socket.h>.

Наприклад, прототип системного виклику bind(), що пов'язує сокет з адресою, має такий вигляд:

```
int bind(int, struct sockaddr *. socklen_t);
```

На практиці адреса задається так: визначають структуру типу sockaddr_i n і під час виклику покажчик на неї перетворюють до sockaddr *:

```
struct sockaddr_in my_addr;
// ... заповнення структури my_addr, див. далі
bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr));
```

Безпосередньо структуру sockaddr у застосуваннях не використовують. Якщо виклик повернув покажчик на неї, перед використанням у застосуванні його потрібно привести до покажчика на структуру для конкретного протоколу (наприклад, на sockaddr_in).

Історично так склалося, що відображення у пам'яті даних, для яких виділяють більш як один байт (наприклад, двобайтового цілого, що відповідає у C типу short), відрізняється для різних комп'ютерних архітектур. У деяких із них старший байт розташовують у пам'яті перед молодшим (такий порядок називають зворотним – big-endian byte order), а в інших – молодший перед старшим (це прямий порядок – little-endian byte order). Наприклад, архітектура IA-32 використовує прямий порядок байтів, а комп'ютери Sun Sparcstation та Power PC – зворотний.

Проблема полягає в тому, що деякі елементи пакетів (керуючі дані), якими обмінюються комп'ютери для організації передавання інформації, займають у пам'яті більш як один байт. Так, відображення порту займає 2 байти, а IP-адреси – 4 байти. Якщо такий блок інформації буде передано комп'ютеру з іншим порядком байтів, він не зможе коректно його інтерпретувати.

Для вирішення цієї проблеми реалізації стека протоколів на клієнті та на сервері потрібно під час передавання даних мережею узгоджено використовувати єдиний порядок байтів. Такий порядок, який називають мережним порядком байтів (network byte order), визначає мережний протокол. Фактично, це зворотний порядок. Відповідно, порядок байтів, прийнятий для локального комп'ютера, називають порядком байтів хоста (host byte order). Він може бути як прямим, так і зворотним.

Програміст має перетворювати керуючі дані в мережний порядок байтів перед передаванням інформації із мережі та робити зворотне перетворення після її отримання. Це зокрема стосується IP-адрес і номерів портів. Для реалізації перетворення використовують функції htonl() і htons() (перетворення до мережного порядку), ntohl() і ntohs() (перетворення до порядку хоста):

```
#include <netinet/in.h>
uint16_t htons(uint16_t hostval); //htonl() – для 32-бітних даних
uint16_t ntohs(uint16_t netval); //ntohl() – для 32-бітних даних
```

Далі наведемо приклади використання цих функцій.

У протоколі IPv4 IP-адреса є цілим значенням розміром 4 байти. Для того щоб перетворити точково-десяткове подання такої адреси (n. n. n. n) у ціле число, слід використати функцію `inet_aton()`:

```
#include <arpa/inet.h>
int inet_aton(const char *c_ip, struct in_addr *s_ip);
```

де: `c_ip` – рядкове відображення IP-адреси: "192.168.10.28";

`s_ip` – структура і `n_addr`, що міститиме цілочислове відображення адреси, розташоване в пам'яті відповідно до мережного порядку байтів (поле `s_addr`); цю структуру заповнюють усередині виклику.

У разі успішного перетворення ця функція повертає ненульове значення, у разі помилки – нуль.

Розглянемо приклад заповнення структури, що задає адресу сокета:

```
struct sockaddr_in my_addr={0}; //структура має бути обнулена
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(7500); // задавання порту
inet_aton("192.168.10.28". &(my_addr.sin_addr)); //задавання адреси
```

Для обнулення структур часто використовують функцію `bzero()`:

```
#include <strings.h>
bzero(&my_addr, sizeof(my_addr));
// заповнення my_addr
```

Зворотне перетворення IP-адреси (із цілого числа у точково-десяткове подання) здійснюють функцією `inet_ntoa()`:

```
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr s_ip);
```

Ця функція розміщає у статичному буфері рядок із точково-десятковим поданням IP-адреси, заданої структурою `s_ip`, і повертає покажчик на цей буфер:

```
printf("IP-адреса: %s\n", inet_ntoa(my_addr.s_addr));
```

Зазначимо, що наступні спроби виклику функції стиратимуть статичний буфер, тому для подальшого використання його вміст потрібно копіювати в окремий буфер пам'яті.

Для створення сокетів перший етап, який необхідно виконати на клієнті та сервері, – створити дескриптор сокета за допомогою системного виклику `socket()`:

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

де: `family` – комунікаційний домен (AF_INET – домен Інтернету на основі IPv4, AF_UNIX – домен UNIX);

`type` – тип сокета (SOCK_STREAM – потоковий сокет, SOCK_DGRAM – дейтаграмний сокет, SOCK_RAW – *сокет прямого доступу* (raw socket), призначений для роботи із протоколом мережного рівня, наприклад IP);

`protocol` – тип протоколу (звичайно 0; це означає, що протокол вибирають автоматично, виходячи з домену і типу сокета; наприклад, потокові сокети домену Інтернет використовують TCP, а дейтаграмні – UDP).

Ось приклад використання `socket()`:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Виклик `socket()` повертає цілочисловий дескриптор сокета або -1, якщо сталася помилка (при цьому, як і для інших викликів, змінна `errno` міститиме код помилки). Цей сокет можна використовувати для різних цілей: організації очікування з'єднань сервером, задавання з'єднання клієнтом, отримання інформації про мережну

конфігурацію хоста (в останньому випадку він може бути навіть не пов'язаний з адресою).

Для системних викликів інтерфейсу сокетів Берклі далі за замовчуванням передбачатиметься використання заголовного файла <sys/socket.h>.

Розглянемо головні системні виклики, які використовують під час розробки серверів та клієнтів з використанням потокових сокетів на основі протоколу TCP (рис. 12.4). Це – основний вид сокетів, що найчастіше трапляється в реальних застосуваннях.

1. Перший системний виклик – ***створення дескриптора сокета*** за допомогою системного виклику `socket()`. Цей дескриптор не пов'язаний із конкретною адресою і недоступний для клієнтів.

2. ***Для пов'язування сокета з адресою*** необхідно використати системний виклик `bind()`:

```
int bind(int sockfd, const struct sockaddr *pmy_addr, socklen_t alen);
```

де: `sockfd` – дескриптор сокета, створений за допомогою `socket()`;

`pmy_addr` – покажчик на заздалегідь заповнену структуру, що задає адресу сокета (наприклад, `sockadd_in`);

`alen` – розмір структури, на яку вказує `pmy_addr`.

Цей виклик повертає -1, якщо виникла помилка.

Під час виконання `bind()` застосуванням-сервером необхідно задати наперед відомий порт, через який клієнти зв'язуватимуться з ним. Можна також вказати конкретну IP-адресу (при цьому допускатимуться лише з'єднання, для яких вона зазначена як адреса призначення), але найчастіше достатньо взяти довільну адресу локального хоста, скориставшись константою INAODR_ANY:

```
struct sockaddr_in my_addr = { 0 };
int listenfd = socket(...);
// ... задача my_addr.sin_family і my_addr.sin_port
my_addr.sin_addr.s_addr = INADDR_ANY;
bind(listenfd, (struct sockaddr *)&my_addr, sizeof(my_addr));
```

Найпоширенішою помилкою під час виклику `bind()` є помилка з кодом EADDRINUSE, яка свідчить про те, що цю комбінацію «IP-адреса – номер порту» вже використовує інший процес. Часто це означає повторну спробу запуску того самого сервера.

```
if (bind(listenfd, ...) == -1 && errno == EADDRINUSE)
  {printf("помилка, адресу вже використовують\n"); exit(-1); }
```

Щоб досягти гнучкості у вирішенні цієї проблеми, рекомендують дозволяти користувачам налаштовувати номер порту (задавати його у конфігураційному файлі, командному рядку тощо).

3. За замовчуванням сокет, створений викликом `socket()`, є клієнтським активним сокетом, за допомогою якого передбачають з'єднання із сервером. Щоб ***перетворити*** такий ***сокет у серверний прослуховувальний*** (listening), призначений для приймання запитів на з'єднання від клієнтів, необхідно використати системний виклик `listen()`

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

де: `sockfd` – дескриптор сокета, пов'язаний з адресою за допомогою `bind()`;

`backlog` – задає довжину черги запитів на з'єднання, створеної для цього сокета.

Фактично внаслідок виклику `listen()` створяться дві черги запитів: перша з них містить запити, для яких процес з'єднання ще не завершений, друга – запити, для яких з'єднання встановлене. Параметр `backlog` визначає сумарну довжину цих двох черг;

для серверів, розрахованих на значне навантаження, рекомендують задавати достатньо велике значення цього аргументу:

```
listen(listenfd, 1024);
```

Внаслідок виклику `listen()` сервер стає цілковито готовий до приймання запитів на з'єднання.

4. Спроба клієнта встановити з'єднання із сервером після виклику `listen()` має завершуватися успішно. Після створення нове з'єднання потрапляє у чергу встановлених з'єднань. Проте, для сервера воно залишається недоступним. Щоб отримати ***доступ до з'єднання***, на сервері його потрібно прийняти за допомогою системного виклику `accept()`:

```
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

де: `sockfd` – дескриптор прослуховувального сокета; `cliaddr` – покажчик на структуру, у яку буде занесена адреса клієнта, що запросив з'єднання; `addrlen` – покажчик на змінну, котра містить розмір структури `cliaddr`.

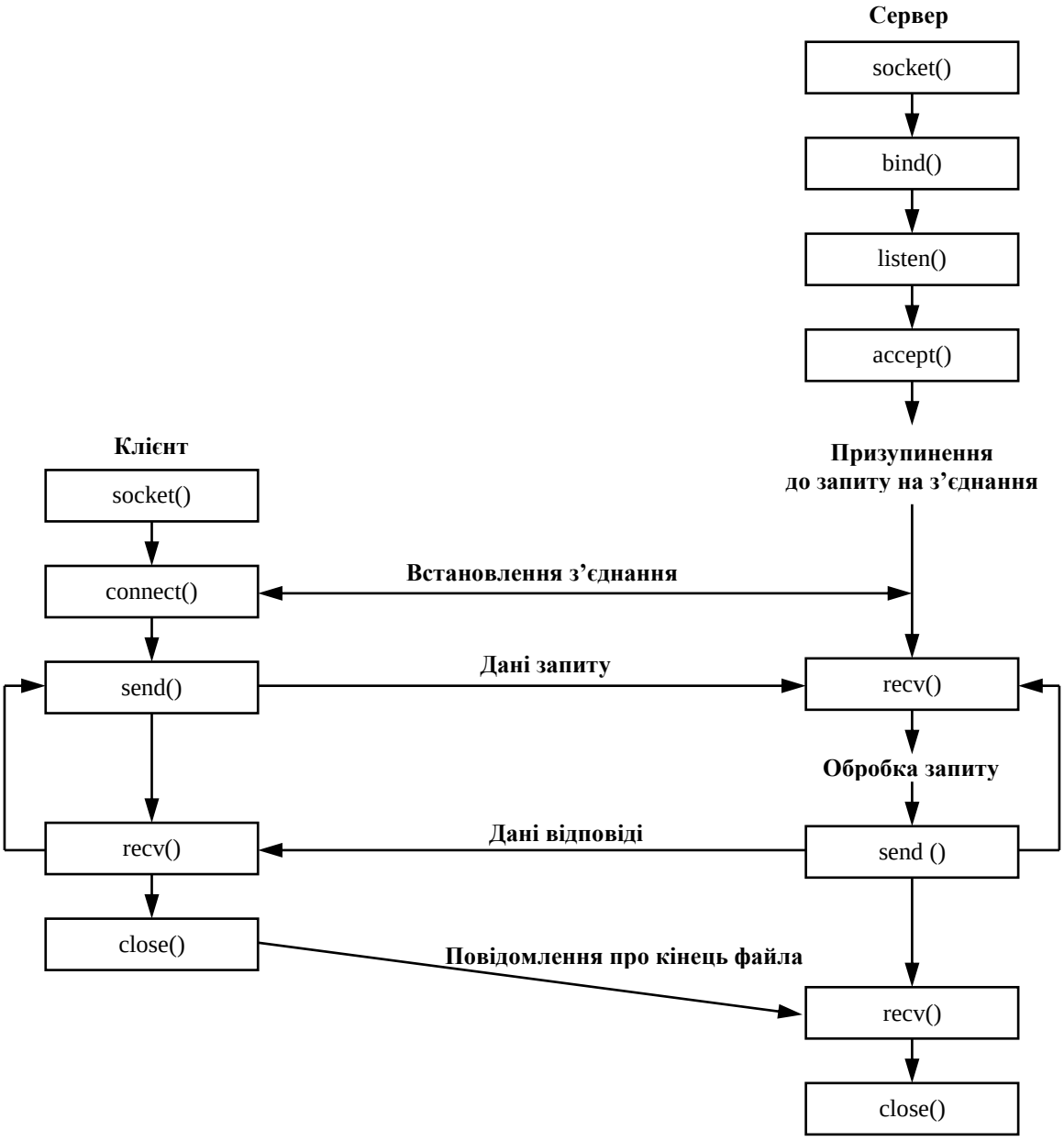

Рис. 12.4. Обмін даними на основі потокових сокетів

Головною особливістю цього виклику є повернене значення – дескриптор нового

сокета з'єднання (connection socket), який створений внаслідок виконання цього виклику і призначений для обміну даними із клієнтом. Прослуховувальний сокет після виконання `accept()` готовий приймати запити на нові з'єднання.

```
struct sockaddr_in their_addr = { 0 };
int listenfd, connfd, sin_size = sizeof(struct sockaddr_in);
//    listenfd    =    sockett...),    binddistenfd,    ...),
      listen(listenfd, ...)
connfd=accept(listenfd, (struct sockaddr *)&their_addr, &sin_size);
// використання connfd для обміну даними із клієнтом
```

Виклик `accept()` приймає з'єднання, що стоїть першим у черзі встановлених з'єднань. Якщо вона порожня, процес переходить у стан очікування, де перебуватиме до появи нового запиту. Саме на цьому системному виклику очікують ітеративні та багатопотокові сервери, що використовують сокети.

Дотепер ми розглядали виклики, які мають бути використані на сервері. ***На клієнті ситуація виглядає інакше***. Після створення сокета на клієнті за допомогою `socket()` необхідно встановити для нього з'єднання за допомогою системного виклику `connect()`, після чого можна відразу обмінюватися даними між клієнтом і сервером.

```
int connect(int sockfd, const struct sockaddr *saddr, socklen_t alen);
```

де: `sockfd` – сокет, створений за допомогою `socket()`; `saddr` – покажчик на структуру, що задає адресу сервера (IP-адресу віддаленого хоста, на якому запущено сервер і порт, який прослуховує сервер); `alen` – розмір структури `saddr`.

Ось приклад створення з'єднання на клієнті:

```
struct sockaddr_in their_addr = { 0 };
sockfd = socket(...);
// заповнення their_addr.sin_family, their_addr.sin_port
inet_aton("IP-адреса-сервера", &(their_addr.sin_addr));
connect(sockfd, (struct sockaddr *)&their_addr, sizeof(their_addr));
// обмін даними через sockfd
```

У разі виклику `connect()` починають створювати з'єднання. Повернення відбувається, якщо з'єднання встановлене або сталася помилка (при цьому поверненим значенням буде -1). Зазначимо, що, якщо `connect()` повернув помилку, користуватися цим сокетом далі не можна, його необхідно закрити. Коли потрібно повторити спробу, створюють новий сокет за допомогою `socket()`.

Клієнтові зазвичай немає потреби викликати `bind()` перед викликом `connect()` – ядро системи автоматично виділяє тимчасовий порт для клієнтського сокета.

Після використання ***всі з'єднання необхідно закривати***. Для цього застосовують стандартний системний виклик `close()`:

```
close(sockfd);
```

Після того, як з'єднання було встановлене і відповідний сокет став доступний серверу (клієнт викликав `connect()`, а сервер – `accept()`), можна обмінюватися даними між клієнтом і сервером. Для цього використовують стандартні системні виклики `read()` і `write()`, а також спеціалізовані виклики `recv()` і `send()`:

```
// прийняти mbytes або менше байтів із sockfd і зберегти їх у buf
ssize_t recv(int sockfd, void *buf, size_t bytes_read, int flags);
// відіслати nbytes або менше байтів із buf через sockfd
ssize_t send(int sockfd, const void *buf, size_t bytes_sent, int flags);
```

Виклики `recv()` і `send()` відрізняються від `read()` і `write()` параметром

flags, що задає додаткові характеристики передавання даних. Тут задаватимемо цей параметр, що дорівнює нулю:

```
int bytes_received = recv(sockfd, buf, sizeof(buf), 0);
int bytes_sent = send(sockfd, buf, sizeof(buf), 0);
```

Важливою особливістю обміну даними між клієнтом і сервером є те, що send() і recv() можуть отримати або передати меншу кількість байтів, ніж було запитано за допомогою параметра nbytes, при цьому така ситуація не є помилкою (особливо часто це трапляється для recv()). Для відсилання або отримання всіх даних у цьому разі необхідно використати відповідний системний виклик повторно:

```
// отримання даних обсягом sizeof(buf)
for (pos =0; pos < sizeof(buf); pos += net_read)
  net_read = recv(sockfd, &buf[pos], sizeof(buf)-pos, 0);
printf("від сервера: %s", buf);
```

У разі помилки ці виклики повертають -1. Серед кодів помилок важливими є ECONNRESET (віддалений процес завершився негайно, не закривши з'єднання) і EPIPE (для send() це означає, що віддалений процес завершився за допомогою close(), не прочитавши всіх даних із сокета; у цьому випадку також буде отримано сигнал SIGPIPE).

Виклик recv(), крім того, може повернути нуль. Це означає, що з'єднання було коректно закрите на іншій стороні (за допомогою виклику close()). Так у коді сервера можна відстежувати закриття з'єднань клієнтами:

```
net_read = recv(sockfd, ...):
if (net_read == 0) { printf("З'єднання закрите\n"); }
```

Розглянемо ***приклад розробки найпростішого луна-сервера***, що негайно повертає клієнтові всі отримані від нього дані (в UNIX-системах така служба доступна за стандартним портом із номером 7).

Опишемо основний цикл сервера (інший код є стандартним – підготовка структур даних, виклик socket(), bind() і listen()).

У головному циклі спочатку необхідно прийняти з'єднання. Після цього в циклі зчитують дані із сокета з'єднання і відсилають назад клієнтові. Цей внутрішній цикл триває доти, поки клієнт не закриє з'єднання або не буде повернено помилку. У кінці ітерації головного циклу сокет з'єднання закривають:

```
for(; ;) {
  sockfd = accept(listenfd,
    (struct sockaddr *)&their_addr, &sin_size);
  printf("сервер: з'єднання з адресою %s\n",
    inet_ntoa(their_addr.sin_addr));
do {
    bytes_read = recv(sockfd, buf, sizeof(buf), 0);
    if (bytes_read > 0) send(sockfd, buf, bytes_read, 0);
  } while (bytes_read > 0); // поки сокет не закритий
  close(sockfd);
} close(listenfd);
```

Зазначимо, що в даному прикладі сервер можна перервати тільки за допомогою сигналу (Ctrl+C, kill() тощо), при цьому за замовчуванням прослуховувальний сокет закрито не буде. Коректне закриття сокета може бути зроблене в оброблювачі сигналу або у функції завершення.

Недолік такого сервера очевидний – поки обробляються дані, нові клієнти не

можуть створити з'єднання (не виконується `accept()`).

Тому розглянемо, як можна уникнути цієї проблеми. Приклад луна-клієнта, що відсилає серверу дані, введені із клавіатури, і відображає все отримане у відповідь.

Сокет і виклик `connect()` створюються стандартним способом. Обмін даними відбувається в нескінченному циклі (для виходу потрібно ввести рядок "вихід", після чого з'єднання буде коректно закрите). Зазначимо, що для простоти весь код повного отримання даних наведено тільки для `recv()`:

```
// sockfd = sockett...}, connect(sockfd. ...)
for (; ;) {
  fgets(buf, sizeof(buf), stdin):
  if (strcmp(buf, "вихід\n") == 0) break;
  stdin_read = strlen(buf);
  send(sockfd, buf, stdin_read, 0);
  for (pos = 0; pos < stdin_read; pos += net_read)
    net_read = recv(sockfd, &buf[pos], stdin_read = pos, 0);
  printf("від сервера отримано: %s", buf);
}
close(sockfd);
```

Зупинимося на ***розробці найпростішого багатопотокового сервера***. Відмінності для багатопотокової реалізації фактично торкнуться тільки основного циклу сервера. У ньому необхідно приймати з'єднання і створювати для його обробки окремий потік у неприєднаному стані (замість очікування завершення його виконання, потрібно очікувати нових з'єднань). Дескриптор сокета передають як параметр у функцію потоку.

```
for(; ;) {
  connfd = accept(listenfd, ...);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_create(&tid, &attr, &process_request, (void *) connfd);
}
```

Функція потоку перетворює параметр до цілочислового типу, виконує обмін даними із клієнтом і закриває сокет з'єднання:

```
void *process_request(void *data) {
   int connfd = (int) data;
   // ... обмін даними із клієнтом через connfd
   close(connfd); }
```

Наведемо ***приклад використання `select()` для повідомлення про стан сокетів*** у реалізації вже відомого луна-сервера.

Насамперед зауважимо, що `select()` в UNIX працює з усіма файловими дескрипторами, доступними для процесу (серед них можуть бути дескриптори сокетів). Ці дескриптори – цілі числа із послідовною нумерацією від нуля. Таке відображення визначає низку особливостей використання виклику `select()`.

Опишемо синтаксис `select()`:

```
#include <sys/select.h>
#include <sys/time.h>
int select(int fd_count, fd_set *readset, fd_set *writeset,
     fd_set *exceptset, const struct timeval *timeout);
```

де: `fd_count` – кількість дескрипторів, зміну стану яких потрібно відстежувати

(оскільки дескриптори нумеруються від нуля, це число дорівнює максимальному номеру дескриптора плюс один);

`readset, writeset` – набори дескрипторів, для яких відповідно потрібно відстежувати готовність до читання і записування даних (роботу із такими наборами буде розглянуто окремо);

`timeout` – покажчик на структуру, що задає максимальний час очікування під час виконання цього виклику (NULL – нескінченне очікування).

Цей виклик повертає кількість дескрипторів, які змінили свій стан, або значення -1 у разі помилки.

Перед викликом `select()` необхідно задати набори дескрипторів для відстеження готовності до дій, які становлять інтерес. Якщо не потрібно відстежувати готовність до певної дії (читання або записування), замість відповідного набору, слід передати нульовий покажчик.

Тип `fd_set` є непрозорим для користувача (хоча прийнято відображати набір дескрипторів у вигляді бітового масиву, де позиції окремих бітів відповідають номерам дескрипторів, і говорити про задавання і очищення біта для дескриптора). Доступ до наборів дескрипторів здійснюють за допомогою макросів:

```
// очищення всіх бітів fdset
void FD_ZERO(fd_set *fdset);
// задавання біта в наборі fdset для дескриптора fd
void FD_SET(int fd, fd_set *fdset);
// очищення біта в наборі fdset для дескриптора fd
void FD_CLR(int fd. fd_set *fdset);
// перевірка, чи увімкнуто біт для fd у наборі fdset
int FD_ISSET(int fd, fd_set *fdset);
```

Наприклад, для задавання набору і увімкнення біта для дескриптора потрібно виконати такий код:

```
fd_set fdarr;
FD_ZERO(&fdarr);
fd = socket(...);  // або open()
FD_SET(fd, &fdarr);
```

Крім того, змінні типу `fd_set` можна присвоювати одна одній.

Коли біт для дескриптора увімкнуто перед викликом `select()`, це означає, що застосування збирається відстежувати готовність до відповідної дії для цього дескриптора. Під час виконання `select()` усі біти для дескрипторів, які не готові до виконання дії, очищаються. У результаті, коли біт виявляється увімкнутим після виклику `select()`, це означає, що такий дескриптор готовий до виконання дії:

```
// очікування готовності до читання дескрипторів із набору fdarr
select(20, &fdarr, NULL, NULL, NULL);
// тут можна досліджувати fdarr
if (ISSET(fd, &fdarr)) {
// дескриптор fd готовий до читання }
```

Розглянемо особливості визначення готовності до дій для дескрипторів на прикладі сокетів. Готовність до читання визначають як наявність даних, які можна прочитати із сокета з'єднання, наявність запиту на нове з'єднання для прослу-ховувального сокета або закриття з'єднання під час читання (коли `recv()` повертає нуль). Готовність до записування визначають як можливість записати дані в сокет з'єднання або закриття

з'єднання під час записування (коли `send()` генерує SIGPIPE).

Опишемо ***особливості розробки луна-сервера на основі системного виклику `select()`.*** Спочатку потрібно визначити основний набір дескрипторів сокетів та його копію:

```
fd_set fdarr, fdarr_copy;
```

Максимальний номер дескриптора, готовність якого потрібно відстежувати, у Linux задають як результат виконання системного виклику `getdtablesize()`, що повертає розмір таблиці дескрипторів процесу:

```
int listenfd, connfd, fd;    // дескриптори сокетів
int maxfd = getdtablesize();
```

Тепер потрібно створити прослуховувальний сокет `listenfd` і викликати для нього функції `bind()`, `connect()` і `listen()` (ці дії є стандартними).

Після виконання всіх дій з `listenfd` його необхідно помістити в набір дескрипторів. Далі, під час виконання функції `select()`, поява запитів на з'єднання на цьому сокеті свідчитиме про готовність до читання даних.

```
// listenfd = socket(...),
    bind(listenfd ...), listen(listenfd ...)
FD_ZERO(&fdarr);
FD_SET(listenfd, &fdarr);
```

Розглянемо, навіщо потрібна копія `fdarr`. Справа в тому, що набір дескрипторів під час виконання `select()` змінюється так, що стає непридатним для передавання в цей виклик. Задані біти в наборі тепер відповідають тим дескрипторам, які готові до виконання дій, а не тим, для яких необхідно відстежувати цю готовність. Доцільно організувати код так, щоб увімкнуті біти `fdarr` завжди відповідали дескрипторам, для яких потрібно відстежувати готовність. Для цього `fdarr` формують під час роботи сервера, але ніколи не передають у `select()`. Перед кожним викликом `select()` створюють копію `fdarr`, і саме її змінюють усередині `select()` і мають за основу для перевірки готовності дескрипторів.

Наведемо структуру основного циклу сервера. Його тіло складається із трьох основних частин:

- створення копії набору дескрипторів і виклику `select()`;
- обробки запитів на нові з'єднання (вона полягає у перевірці біта для прослуховувального сокета і прийманні з'єднання, якщо біт увімкнуто);
- обходу набору дескрипторів поточних з'єднань для перевірки готовності до читання і виконання власне читання.

```
for (;  ;) {
  fdarr_copy = fdarr;  // створення копії набору дескрипторів
  select(maxfd, &fdarr_copy, NULL, NULL, NULL);
  // обробка запиту на нове з'єднання
  // перевірка сокетів, готових до читання, на підставі fdarr_copy
}
```

Нове з'єднання з'являється, коли прослуховувальний сокет опиняється у стані готовності до читання. У цьому разі необхідно прийняти нове з'єднання (викликати `accept()`) і додати сокет з'єднання у `fdarr` (увімкнувши у ньому відповідний біт). Тепер готовність до читання буде перевірено і для цього з'єднання. Після завершення цієї дії необхідно перейти до наступної ітерації циклу (тобто до виклику `select()`):

```
if (FD_ISSET(listenfd, &fdarr_copy)) {
  connfd = accept(listenfd,
    (struct sockaddr *)&their_addr, &sin_size);
  FD_SET(connfd, &fdarr); //додати дескриптор до основного набору
continue;   // повернутися до select()
}
```

Перевірку сокетів з'єднання виконують обходом усіх наявних дескрипторів (крім прослуховувального сокета) і перевіркою кожного з них на готовність до читання. Якщо дескриптор сокета до читання готовий, дані із нього читають за допомогою `recv()` і відсилають назад викликом `send()`. Якщо `recv()` повернув нуль, то це означає, що клієнт закрив з'єднання. У такому разі необхідно закрити відповідний дескриптор сокета і вимкнути відповідний біт у наборі. Далі цей сокет не перевірятимуть.

```
for (fd = 0; fd < maxfd; fd++) {
if (fd != ifd && FD_ISSET(fd, &fdarr_copy)) {
   bytes_read = recv(fd, buf, sizeof(buf), 0);
   if (bytes_read <= 0) {
     close(fd);
     FD_CLR(fd, &fdarr);
   }
else send(fd, buf, bytes_read, 0);
} }
```

Отже, у наборі `fdarr` підтримують у ввімкнутому стані біти для прослуховувального сокета і всіх активних сокетів з'єднання.

Системні виклики інтерфейсу сокетів використовують IP-адреси. Щоб отримати ***доступ до віддаленого хоста на основі доменного імені***, потрібно це ім'я попередньо розв'язати, відіславши запит на локальний DNS-сервер за допомогою розпізнавача.

Найбільш використовуваними системними викликами інтерфейсу розпізнавача є `gethostbyname()` і `gethostbyaddr()`. Перший з них застосовують для перетворення доменних імен в IP-адреси (IPv4), другий – для перетворення IP-адрес у доменні імена. Ці виклики потребують підключення заголовного файла `<netdb.h>`.

Потокові та дейтаграмні сокети – це інтерфейс між застосуванням і протоколом транспортного рівня (TCP або UDP). Звідси очевидно, що реалізацію протоколу прикладного рівня здійснює саме застосування на базі інтерфейсу сокетів.

Прикладом такого протоколу може бути протокол зв'язку програми-чату, яка використовує потокові сокети. Кожне повідомлення від користувача має складатися із двох частин: імені користувача і тексту повідомлення. Сервер відсилає отримані повідомлення всім іншим користувачам. Вихідний потік даних від сервера може мати такий вигляд:

`ІванПривітМиколаДо побачення`

Зауважимо, що повідомлення мають змінну довжину. Завдання полягає в тому, щоб дати можливість клієнтам у загальному потоці байтів, отриманих від сервера, виділяти окремі повідомлення і відокремлювати в них ім'я користувача від тексту повідомлення.

Перший підхід до розв'язання цього завдання полягає в тому, щоб зробити всі повідомлення однієї довжини (із доповненням до неї, наприклад, нульовими символами). Це спричиняє пересилання мережею значного обсягу непотрібної інформації навіть у разі не дуже великих повідомлень.

Коректнішим підходом у даному разі є інкапсуляція даних через розробку *формату пакета* із виділенням у ньому:

- заголовка, що містить довжину пакета;
- поля `name` (ім'я користувача фіксованої довжини);
- поля `chatdata` (текст повідомлення змінної довжини).

Фактично угода про формат пакета визначає мережний протокол.

Сервер формує пакет на підставі повідомлення користувача (таку підготовку до пересилання мережею називають також ***маршалізацією*** – marshaling), клієнт виділяє повідомлення з пакета (робить ***демаршалізацією*** – demarshaling).

Приймемо за правило, що довжина поля `name` становить 8 байт (коротші імена користувачів доповнюють до цієї довжини нульовими символами). Далі вважатимемо, що максимальна довжина тексту повідомлення (поля `chatdata`) становить 240 байт. Звідси випливає, що максимальна довжина пакета (248 байт) може бути відображена заголовком завдовжки 1 байт.

Розглянемо, який вигляд матимуть пакети для наведених раніше даних. Перший пакет (повідомлення від Івана):

```
0E                  B2 E2 EO ED 00 00 00 00   CF FO E8 E2 B3 F2
довжина - 14 байт)   І  в  а  н  (доповнення)  П  р  и  в  і  т
```

Другий пакет

```
13                  CD E8 EA EE EB E0 00 00   C4 EE 20 EF EE E1 ...
довжина - 19 байт)   М  и  к  о  л  а         Д  о     п  о  б ...
```

Зазначимо, що загалом довжина може бути подана кількома байтами, у цьому разі слід звертати увагу на те, що для пересилання мережею вони повинні бути перетворені до мережного порядку байтів.

Розглянемо отримання пакетів клієнтом. Клієнтові доступна довжина поточного пакета (із неї починаються дані пакета), крім того, відома максимальна довжина (249 байт, включаючи заголовок). У застосуванні необхідно виділити буфер, достатній для розміщення двох пакетів максимальної довжини:

```
char buf[498];
```

У цьому буфері відбуватиметься реконструкція пакетів у міру їхнього надходження. Після кожного отримання даних за допомогою `recv()` їх поміщають у буфер і перевіряють, чи весь пакет отримано.

```
if (buffer_len > 1 && buffer_len >= (buffer[0] + 1)
// пакет отримано повністю
```

Тут `buffer_len` – поточний обсяг даних у буфері, `buffer[0]` – перший байт буфера, що містить довжину пакета (крім байта заголовка). Перевірку на те, чи отримано більш як один байт даних, роблять тому, що тільки в цьому разі можна визначити довжину пакета. Коли пакет отримано повністю, його можна використовувати у застосуванні, після чого поверх нього у буфер почнуть записувати наступний пакет.

Особливою ситуацією є отримання за `recv()` блоку даних, що містить інформацію із двох пакетів (останні байти одного і перші байти іншого). У результаті буфер міститиме повний пакет і неповну частину ще одного пакета (саме тому у буфері зарезервоване місце для двох пакетів). Оскільки довжина першого пакета відома, можна визначити, скільки байтів буфера належить другому пакету:

```
second_bytes = buffer_len - (buffer[0] + 1);
```

Далі можна обробити перший пакет, перемістити дані другого пакета на початок буфера і відкоригувати `buffer_len`:

```
strncpy(buf, buf + buffer_len, second_bytes);
buffer_len = second_bytes;
```

Тепер буфер знову готовий прийняти дані, отримані за допомогою `recv()`.

Наведені принципи роботи із протоколами прикладного рівня справедливі й для

складніших протоколів, які використовують на практиці. Деякі ***особливості реальних протоколів*** розглянемо на прикладі HTTP.

Блок заголовків HTTP-пакета, на відміну від описаного простого протоколу, має змінну довжину. Завершення цього блоку визначають за наявністю у потоці даних комбінації символів "\r\n\r\n". Окремі елементи блоку заголовків (його рядки) розділяються символами "\r\n".

Є два типи HTTP-пакетів зі спільними принципами побудови: запит (request) і відповідь (response). Перший рядок запиту містить команду для сервера (метод), адресу документа на сервері та версію протоколу. У цьому разі метод GET визначає запит на отримання документа із сервера на основі його адреси:

```
GET /test.html HTTP/1.1
```

Перший рядок відповіді містить версію протоколу і код відповіді (наприклад, код 200 означає коректне повернення даних):

```
HTTP/1.1 200 OK
```

Інші рядки блоку заголовка у специфікації HTTP називаються просто заголовками і мають формат `ім'я_заголовка: значення_заголовка`. Наприклад, версія протоколу HTTP/1.1 задає для запиту обов'язковий заголовок `Host:`, який визначає доменне ім'я комп'ютера-сервера, задане клієнтом:

```
Host: server.com
```

За блоком заголовків може слідувати тіло пакета. Є два способи визначити його довжину. Перший спосіб універсальний і рекомендований до використання. Для цього у блоці заголовків задають заголовок `Content-Length:`, значенням якого є довжина тіла пакета. Програмі-клієнту потрібно виділити довжину із цього заголовка:

```
// header_buf - буфер із даними блока заголовків
clen_str = strstr( header_buf, "Content-Length: ");
if (clen_str != NULL)
  sscanf(clen_str, "Content-Length: %d\r\n", &content_length);
```

Подальші дії аналогічні до розглянутих раніше для простого протоколу. Другий підхід зводиться до того, що сервер після пересилання HTTP-пакета закриває з'єднання. Для цього в запиті необхідно задати заголовок

```
Connection: close
```

Клієнту потрібно відстежити кінець з'єднання (коли `recv()` поверне нуль). Цей підхід простіший у реалізації, але прийнятний не для всіх випадків, оскільки часто клієнт (браузер) запитує сторінки не по одній, а групами (наприклад, HTML-сторінку і всі графічні файли, на які вона посилається), і пересилання всієї групи сторінок за допомогою одного з'єднання дає змогу заощаджувати ресурси і час.

Тіло пакета може бути відсутнім (наприклад, його немає в запиті методом GET).

Розглянемо приклад формування HTTP-запиту:

```
char request[] = "GET /index.html HTTP/1.1\r\n"
                 "Host:server.com\r\n\r\n";
// ...з'єднання через стандартний порт HTTP (порт 80)
send(sockfd, request, sizeof(request), 0);
// ...обробка відповіді
close(sockfd);
```

12.5. Архітектура мережної підтримки Windows XP

Розглянемо основні компоненти мережної підтримки Windows XP (рис. 12.5).

Мережні API надають засоби для доступу до мережі із прикладних програм, незалежні від протоколів. Звичайно такі API частково реалізовані в режимі користувача (компоненти їхньої підтримки в режимі ядра – це драйвери, які називають драйверами мережних API). Серед мережних API найширше використовують інтерфейси поіменованих каналів, Windows Sockets (розглянемо далі) та віддаленого виклику процедур (RPC).

Драйвери транспортних протоколів приймають IRP-пакети від драйверів мережних API і обробляють запити, що містяться там, як вимагають реалізовані в них протоколи (TCP, IP тощо). Під час цієї обробки може знадобитися обмін даними через мережу, додавання або вилучення заголовків пакетів, взаємодія із драйверами мережних апаратних пристроїв. Драйвери протоколів відповідають за підтримку мережної взаємодії організацією повторного збирання пакетів, встановлення їхньої послідовності, відсилання повторних пакетів і підтверджень. Усі драйвери транспортних протоколів надають драйверам мережних API універсальний *інтерфейс транспортного драйвера* (Transport Driver Interface, TDI).

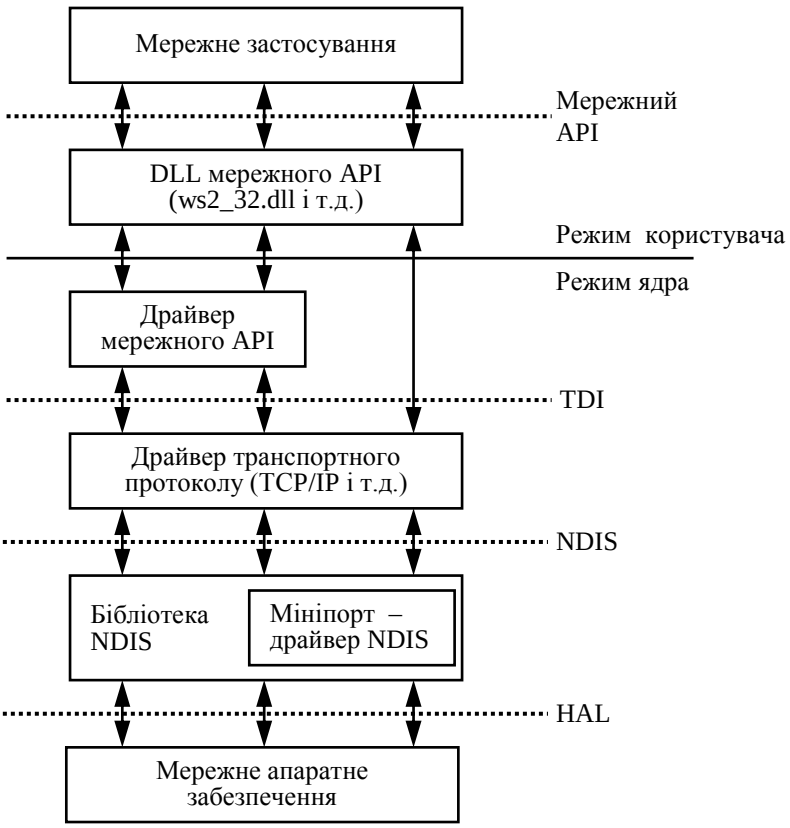

Рис. 12.5. Мережна архітектура Windows XP

Прикладом драйвера транспортного протоколу є драйвер підтримки TCP/IP (tcpip.sys). *Мініпорт-драйвери NDIS* відповідають за організацію взаємодії драйверів транспортних протоколів і мережного апаратного забезпечення. Вони взаємодіють з іншими компонентами режиму ядра винятково за допомогою функцій спеціальної бібліотеки NDIS (ndis.sys), що реалізують універсальну *специфікацію інтерфейсу мережного драйвера* (Network Driver Interface Specification, NDIS). Деякі з цих функцій призначені для доступу із драйверів протоколів до засобів мініпорт-драйверів, інші мініпорт-драйвери викликають для безпосереднього доступу до мережних апаратних пристроїв через HAL. Прикладами мініпорт-драйверів NDIS є драйвери конкретних мережних адаптерів.

Основою підтримки ***спільного використання файлів і принтерів*** є спеціальний протокол *спільної файлової системи Інтернету* (Common Internet File System, CIFS). Він визначає правила взаємодії з файловими клієнтами і серверами. Реалізація CIFS для

Windows XP складається із двох частин – клієнтської (редиректор) і серверної (сервер), реалізованих як драйвери файлових систем.

Драйвер редиректора перехоплює запити на виконання файлового введення-виведення (наприклад, виклик функцій `ReadFile()`, `WriteFile()` або звертання до драйвера принтера), перетворює їх у CIFS-повідомлення і пересилає на віддалений хост, використовуючи драйвер транспортного протоколу. Драйвер сервера приймає повідомлення від драйвера протоколу, перетворює їх назад у запити введення-виведення і передає драйверу локальної файлової системи (наприклад, NTFS) або драйверу принтера. Повернення даних відбувається у зворотному порядку.

Інтерфейс керування драйверами CIFS реалізовано у вигляді фонових процесів `Workstation` (для редиректора) і `Server` (для сервера). Ці процеси перехоплюють запити на створення і зміну *мережних ресурсів* (network shares), які адміністратор системи відкрив для спільного використання (цими ресурсами можуть бути розділи диска, каталоги, принтери тощо). Для керування такими ресурсами можна використовувати утиліту net.exe, що входить у поставку Windows XP. Для створення мережного ресурсу її виклик матиме вигляд `net share ім'я_ресурсу = шлях`, де `шлях` задає шлях до локального каталогу, який потрібно зробити розділюваним:

```
c:>net share myshare=c:\mydir
```

Як зазначалося раніше час вивчення поіменованих каналів, універсальним способом іменування мережних ресурсів є спеціальні *UNC-імена:* `\\ім'я_машини\ім'я_ресурсу`. У разі використання такого імені активізується спеціальний драйвер (Multiple UNC Provider, MUP), що послідовно відсилає IRP-пакети кожному зареєстрованому у системі редиректору доти, поки один із них не розпізнає ресурс і не надішле відповіді. Після цього MUP-драйвер кешує інформацію, і далі всі запити, що використовують це UNC-ім'я, негайно відправляються цьому редиректору.

```
HANDLE fh = CreateFile("\\\\myserver\\myshare\\myfile.txt",
   GENERIC_READ, ... );
```

Крім UNC-імен, для доступу до ресурсів можна використати стандартні імена томів або портів принтера. Для керування відображенням мережних ресурсів на такі імена теж використовують утиліту `net.exe`. Для задавання зв'язку ресурсу з іменем тому її виклик матиме вигляд `net use ім'я_пристрою ім'я_ресурсу`, де `ім'я_пристрою` задають як символьне позначення тому (C:, D: тощо) або символьне ім'я принтера (наприклад, LPT1:), а ім'я_ресурсу – UNC-ім'я цього мережного ресурсу.

```
c:>net use N: \\myserver\myshare
```

Фактично CIFS можна розглядати як основу реалізації розподіленої файлової системи, оскільки прикладне програмне забезпечення може використати надані нею ресурси як локальні. Виконання виклику `ReadFile()` для файла `N:\myfile.txt` у прикладній програмі не залежить від того, чим є `N:` – розділом локального диска або мережним ресурсом для доступу до спільно використовуваного каталогу віддаленого комп'ютера.

Важливою характеристикою протоколу CIFS є те, що він опублікований як стандарт і може бути реалізований у продуктах сторонніх розробників. Таким продуктом зокрема є Samba, що реалізує підтримку CIFS для UNIX-систем. Samba-клієнти можуть використовувати ресурси, надані серверами лінії Windows XP, Samba-сервери – надавати свої ресурси для використання Windows-клієнтами.

12.6. Програмний інтерфейс Windows Sockets

Розглянемо мережний API, який найбільше використовують у системах лінії

Windows XP. Цей API був побудований на зразок API сокетів Берклі і дістав назву Windows Sockets (скорочено Winsock).

У режимі користувача Winsock API підтримують кількома системними DLL. Найважливіша з них – `ws2_32.dll`, де реалізовано основні функції, що використовуються у застосуваннях. На рівні ядра підтримку Windows Sockets здійснюють драйвером файлової системи `afd.sys`, що реалізує операції із сокетами і звертається для пересилання даних до відповідного драйвера транспортного протоколу (звичайно це драйвер підтримки TCP/IP).

Інтерфейс Winsock API посідає таке саме місце у багаторівневій мережній архітектурі, що й інтерфейс сокетів Берклі. Він розташований між рівнем застосувань і транспортним рівнем. У ньому відсутні засоби безпосередньої підтримки протоколів прикладного рівня, вона може бути реалізована на основі інтерфейсу Winsock.

У програмному інтерфейсі Windows Sockets реалізовані всі основні моделі взаємодії між клієнтом і сервером, у тому числі синхронне та асинхронне введення-виведення даних. Реалізація синхронного введення-виведення ґрунтується на сокетах із блокуванням, інтерфейс яких практично не відрізняється від інтерфейсу сокетів Берклі. їхнє використання можливе, коли не потрібні висока продуктивність і масштабованість застосування. На основі сокетів із блокуванням можна реалізовувати багатопотокові сервери (наприклад, використовуючи технологію «потік для запиту»). Підтримують і реалізацію серверів із повідомленням на основі `select()`, хоча цей підхід також не є ефективним.

Більш ефективною і складнішою технологією, яка також реалізована у Windows Sockets, є *асинхронні сокети.* Є два підходи до їхнього використання. Перший із них використовує повідомлення на основі повідомлень і ґрунтується на функції `WSAAsynchSelect()`. При цьому інформацію про події, що відбулися із сокетом, передають у вигляді повідомлень Windows у віконну процедуру, де вона може бути оброблена. За другого підходу – повідомлення на основі подій (що використовує функцію `WSAEventSelect()`) – аналогічну інформацію передають сигналізацією об'єктів подій. Цей підхід з'явився у версії Winsock 2.

Найефективнішим підходом, реалізованим у Winsock, є сокети із перекриттям, що застосовують асинхронне введення-виведення. Сокети також можна використовувати у поєднанні з портами завершення введення-виведення.

Як зазначалося, інтерфейс сокетів із блокуванням майже збігається з інтерфейсом Berkeley Sockets. Проте є деякі відмінності.

- Використання сокетів Берклі вимагає підключення набору різних заголовних файлів. Windows Sockets звичайно потребує лише заголовних файлів <winsock.h> або <winsock2.h>. Під час компонування потрібно задавати бібліотеку підтримки Winsock (ws2_32.lib для версії 2).
- Перед використанням будь-яких засобів Windows Sockets необхідно ініціалізувати бібліотеку, після використання – вивільнити зайняті ресурси. Для цього є функції `WSAStartup()` і `WSACleanup()`:

```
#include <winsock.h>
WSADATA wsadata;
WSAStartup(MAKEWORDCl.l), &wsadata);
// ... робота із засобами Winsock
WSACleanup();
```

Перший параметр `WSAStartup()` визначає версію бібліотеки, потрібну застосуванню. Звичайно для його формування використовують макрос MAKEWORD, якому

передають основний і додатковий номери необхідної версії. Наприклад, аби вказати, що застосування потребує використання Winsock версії 2.0, перший параметр WSAStartup потрібно задати як MAKEWORD(2,0)). Структуру WSADATA зазвичай ніде, крім виклику `WSAStartup()`, не використовують.

- Важливою відмінністю Winsock є підхід до розуміння дескрипторів сокетів. Як відомо, інтерфейс сокетів Берклі визначає, що такі дескриптори є звичайними файловими дескрипторами (задані цілими числами, які виділяють процесу послідовно – від менших номерів до більших). Для Winsock це не так: дескриптори сокетів не розглядають як файлові дескриптори, і на їхні значення покладатися не можна. Такі дескриптори належать до спеціального типу SOCKET (а не до типу `int`, як для сокетів Берклі).
- Оскільки дескриптори сокетів у Winsock відрізняються від файлових дескрипторів, для їхнього закриття не можна використати `close()`. Замість цього необхідно скористатися спеціальною функцією `closesocket()`.
- У разі помилки виклики Winsock повертають спеціальне значення INVALID_SOCKET і не задають змінної `errno` (для визначення коду помилки необхідно використати функцію `WSAGetLastError()`). Обробка помилок у Winsock-застосуваннях має такий вигляд:

```
Isock = socket(...);
if (Isock == INVALID_SOCKET) {
printf("Помилка із кодом %d\n", WSAGetLastError());
exit(); }
```

- Windows Sockets не визначає функцію і `net_aton()`, хоча реалізація і `net_ntoa()` у цій бібліотеці є. Для переведення крапково-десяткового відображення IP-адреси в цілочислове потрібно скористатися функцією `inet_addr()`:

```
unsigned long addr = inet_addr("IP-адреса");
 my_addr.sin_addr = *((struct in_addr *)&addr);
```

Ця функція є також і в інтерфейсі сокетів Берклі, але там її використовувати не рекомендують.

Розглянемо ***особливості інтерфейсу асинхронних сокетів***, що використовують повідомлення через події. Для кожного сокета, стан якого необхідно відстежувати, необхідно створити об'єкт-подію, що належить до типу WSAEVENT.

```
WSAEVENT event = WSACreateEvent();
```

Після створення події необхідно пов'язати її із сокетом, тобто вказати, які зміни стану цього сокета призводитимуть до сигналізації події. Для встановлення такого зв'язку використовують функцію `WSAEventSelect()`:

```
int WSAEventSelect(SOCKET sock, WSAEVENT event, long net_events);
```

де `net_events` – бітова маска мережних подій. Кожній події відповідає певний біт цієї маски. Застосування очікуватиме отримання повідомлень про події, біти яких задані в масці. Серед прапорців, що визначають біти для подій, можна виділити:

- FD_READ – готовність до читання даних із сокета;
- FD_WRITE – готовність до записування даних у сокет;
- FD_ACCEPT – наявність нового з'єднання для сокета;
- FD_CLOSE – закриття з'єднання.

```
//сигналізація event у разі спроби читання або записування через sock
WSAEventSelect(sock, event, FD_READ | FD_WRITE);
```

Після виклику WSAEventSelect() застосування може перейти в режим очікування зміни стану об'єктів-подій. Для цього потрібно викликати функцію

```
DWORD WSAWaitForMultipleEvents(DWORD ecount, const WSAEVENT
      *events, BOOL waitall, DWORD timeout, BOOL alertable);
```

де: ecount – кількість подій, зміну стану яких відстежують (як і для WaitForMultipleObjects(), вона не може бути більшою за 64);

events – масив об'єктів-подій;

waitall – якщо TRUE, вихід із функції відбувається у разі сигналізації всіх подій масиву, якщо FALSE – будь-якої з них.

Коли сигналізовано одну подію із масиву, WSAWaitForMultipleEvents() повертає значення, що визначає її індекс у масиві events. Щоб отримати цей індекс, потрібно використати такий код:

```
res = WSAWaitForMultipleEventst...);
cur_event = res - WSA_WAIT_EVENT_0;
```

Обмеження кількості очікуваних подій вимагає виконання додаткових дій у тому випадку, коли очікують більшу кількість з'єднань. Рекомендовано створювати кілька потоків, у кожному з яких відбувається очікування зміни стану підмножини подій.

Після сигналізації події необхідно визначити, які дії із сокетом її викликали. Для цього використовують функцію

```
int WSAEnumNetworkEvents(SOCKET sock, WSAEVENT event,
      LPWSANETWORKEVENTS net_events);
```

де: sock – сокет, зміна стану якого викликала сигналізацію події;

event – подія, яку було сигналізовано (внаслідок виклику функції стан цієї події буде скинуто);

net_events – покажчик на структуру WSANETWORKEVENTS, яку заповнюють під час виклику функції і яка міститиме інформацію про всі дії із сокетом.

Структура WSANETWORKEVENTS містить такі поля:

- lNetworkEvents – маска мережних подій;
- iErrorCode – масив повідомлень про помилки.

Зазначимо, що в масці мережних подій може бути увімкнуто кілька прапорців. Це означає, що сигналізація була викликана кількома одночасними подіями для сокета.

Після виконання WSAEnumNetworkEvents() можна перейти до обробки запитів введення-виведення на підставі того, які біти увімкнуто в масці мережних подій (наприклад, якщо задано FD_READ, можна перейти до читання даних з відповідного сокета).

Розглянемо, як можна розробити луна-сервер на базі повідомлення про події.

Базова структура такого сервера подібна до розглянутого раніше сервера на основі select(). Необхідно динамічно підтримувати масиви подій і сокетів. Перші елементи цих масивів відповідатимуть прослуховувальному сокету, інші динамічно формуватимуться з появою запитів на з'єднання: після виклику accept() відповідні сокет і подія додаватимуться в масив, у разі розриву з'єднання - вилучатимуться з масиву.

Такі масиви змінюються динамічно, для цього добре підходять структури даних стандартної бібліотеки C++ (такі, як list). У цьому прикладі обмежимося роботою зі звичайними масивами. Нові дескриптори і події додаватимуться в кінець масиву, у разі вилучення з масиву такі елементи зміщуватимуться до його початку.

```
#include <winsock2.h>
const int MAX_CLIENTS = 64;
```

```
WSAEVENT events[MAX_CLIENTS]: // довжина масивів обмежена
SOCKET socks[MAX CLIENTS];
// ...
WSAStartup(MAKEWORD(2.0). &wsadata);  // Winsock 2
// ... lsock = socket(...), bind(lsock, ...), listen(sock, ...)
// додавання в масиви прослуховувального сокета і події для нього
socks[0] = lsock;
events[0] = WSACreateEvent();
```

Після додавання в масиви інформації про прослуховувальний сокет і відповідну подію необхідно зазначити, що сигналізацію цієї події спричинятиме поява нових з'єднань на сокеті.

```
WSAEventSelect(lsock, events[0], FD_ACCEPT);
```

Очікування сигналізації подій необхідно виконувати в циклі.

```
WSANETWORKEVENTS net_events;
for (; ;) {
  res = WSAWaitForMultipleEvents(
    num_socks, events, FALSE, WSAJNFINITE, TRUE);
  cur_event = res - WSA_WAIT_EVENT_0;
  WSAEnumNetworkEvents(
    socks[cur_event], events[cur_event], &net_events);
  //... обробка запитів введення-виведення на основі net_events
}
```

Проаналізуємо обробку запитів введення-виведення. Очевидно, що маємо інформацію про сокет, який викликав подію (`socks[cur_event]`), і про характер цієї події (`net_events.lNetworkEvents`). У разі появи нового з'єднання на прослуховувальному сокеті інформацію про сокет з'єднання і відповідну подію додають у масиви, після чого сокет пов'язують із цією подією для очікування читання або закриття з'єднання:

```
if (net_events.lNetworkEvents & FD_ACCEPT) {
  csock = accept(socks[cur_event], ...);
  // у кінець масивів додають сокет і нову подію
  socks[num_socks] = csock;
  events[num_socks] = WSACreateEvent();
  // очікування read() або close()
  WSAEventSelect(socks[num_socks], events[num_socks],
       FD_READ | FD_CLOSE);
num_socks++;
}
```

У разі появи запиту на читання (коли на клієнті були введені дані) виконують дії із сокетом з'єднання, що спричинив цей запит.

```
if (net_events.lNetworkEvents & FD_READ) {
  recv(socks[cur_event], ...);
  send(socks[cur_event], ...);
}
```

Нарешті, у разі закриття з'єднання на клієнті необхідно закрити подію та сокет і вилучити інформацію з відповідних масивів.

```
if (net_events.lNetworkEvents & FD_CLOSE) {
WSACloseEvent(events[cur_event]);
closesocket(socks[cur_event]);
```

```
for (i = cur_event+1; i < num_socks; i++) {
  events[i-1] - events[i];
  socks[i-1] - socks[i];
  }
  num_socks-i;
}
```

Перевага цього підходу порівняно із використанням `select()` полягає насамперед в тому, що програміст має повну інформацію про те, для якого сокета відбулася подія і який її характер, що спрощує програмування і дає змогу досягти більшої ефективності (немає необхідності обходити весь масив дескрипторів).

Висновки

- Під час розгляду сучасних мережних архітектур використовують багаторівневий підхід. Найрозповсюдженішою моделлю мережної архітектури є TCP/IP, у якій виділяють чотири рівні: канальний (фізичний), мережний, транспортний і прикладний. Засоби підтримки перших трьох рівнів звичайно реалізують у ядрі ОС.
- Реалізацією мережної архітектури TCP/IP є набір протоколів Інтернету (стек протоколів TCP/IP), який містить, зокрема, протокол IP на мережному рівні та протокол TCP на транспортному рівні. Цей набір протоколів підтримує більшість сучасних ОС.
- Під час передавання пакета його спочатку опускають униз у стеку протоколів операційної системи хоста-відправника (при цьому відбувається його інкапсуляція в пакети протоколів нижнього рівня – спочатку транспортного, потім мережного, потім канального). Потім пакет канального рівня передають фізичною мережею, а після прибуття на хост-одержувач – піднімають у стеку протоколів ОС цього хоста (при цьому відбувається його послідовне демультиплексування із пакета канального рівня, мережного і транспортного). У результаті застосування-адресат отримує пакет протоколу прикладного рівня.
- Ядро ОС може працювати лише з IP-адресами. Для того щоб можна було задавати символьні імена хостів, використовують розподілену систему імен DNS.
- Основою для реалізації доступу із режиму користувача до засобів підтримки стека протоколу TCP/IP є програмний інтерфейс сокетів Берклі. Він розташований між прикладним і транспортним рівнями мережної архітектури TCP/IP. Його аналогом для Windows-систем є інтерфейс Windows Sockets.

Контрольні запитання та завдання

1.У мережі, що складається з Linux- і Windows-хостів, використовують спільний стек протоколів (з реалізацією всіх його рівнів). Перелічіть всі рівні стека протоколів для хостів обох типів, код реалізації яких потребуватиме корекції, якщо розробники ядра Linux змінять інтерфейс сервісу для транспортного рівня. Що потрібно буде коригувати після аналогічної зміни інтерфейсу протоколу для мережного рівня?

2.Які рівні мережної архітектури повинні бути реалізовані у програмному забезпеченні маршрутизатора?

3.Внаслідок обміну даними між хостами А і В зафіксовано, що хост А відправив Л/ТР-дейтаграм, хост В одержав ./V IP-дейтаграм, хост В відправив *М* IP-дейтаграм, хост А одержав 4хМ IP-дейтаграм. Поясніть, як це могло статися.

4.Якщо пакет не дійшов за призначенням упродовж певного проміжку часу, протокол TCP забезпечує його повторний запит. Може статися так, що «загублений» пакет надалі «віднайдеться» (наприклад, у випадку, якщо він був затриманий маршрутизатором або проходив мережею *з* низькою пропускною здатністю) і прийде за призначенням разом із пакетом, отриманим внаслідок повторного запиту. Які засоби протоколу TCP дають змогу вирішити цю проблему?

5.Якщо реалізація протоколу TCP відправляє перші дані після встановлення з'єднання, номер послідовності для них визначають на підставі показів системного годинника. Чому не можна завжди починати пересилання даних через з'єднання з того самого номера послідовності (наприклад, з нуля)?

6.Кожна IP-дейтаграма з інкапсульованим TCP-сегментом містить такі 5 полів: інформацію про транспортний протокол (protocol), IP-адресу і порт джерела (from_host, from_port), IP-адресу і порт призначення (to_host, to__port). Які системні виклики відповідають за задавання і зміну кожного з цих полів у процесі встановлення з'єднання з використанням потокових сокетів?

7.Який із транспортних протоколів (TCP або UDP) краще використовувати як базовий протокол для передачі мультимедіа-даних через Інтернет у реальному режимі (Real Audio і т. д.)?

8.Чи може користувач одночасно працювати з веб-браузером і клієнтом електронної пошти, використовуючи одне модемне з'єднання? Якщо така робота можлива, як розрізняти дані, призначені для кожного з цих застосувань?

9.Як використання кешування може підвищити надійність системи іменування (наприклад, DNS)?

10. Запропонуйте серверні архітектурні вирішення для реалізації таких сервісів:

а) інтерактивний потоковий, який повідомляє всіх підключених клієнтів про інформацію, одержану від кожного з них. З'єднання можна підтримувати протягом декількох годин, але дані передають упродовж коротких проміжків часу (довжиною кілька мілісекунд). За секунду може бути встановлено до 10 з'єднань. Приклад – інтерактивний чат у локальній мережі;

б) дейтаграмний (відправлення пакета невеликої довжини у відповідь на кожен запит) з частотою з'єднань до 1-2 за секунду. Приклад - сервіс часу;

в) дейтаграмний з частотою з'єднань до кількох тисяч за секунду. Приклад - локальна база даних;

г) потоковий, що приймає дані від різних клієнтів великими фрагментами. Час обробки запиту – до 10 с. Частота з'єднань – до 10 за секунду від різних клієнтів.

Приклад – сервер друкування.

11. Розробіть веб-клієнт і веб-сервер з використанням сокетів. Клієнт повинен приймати від користувача (наприклад, у командному рядку) доменне ім'я веб-сервера, номер порту та ім'я запитуваної сторінки, формувати HTTP-запит методом GET і відправляти його на сервер. Сервер повинен приймати запит, знаходити відповідний файл сторінки і відправляти його клієнту в складі HTTP-відповіді. Заголовки запиту обробляти не потрібно. Після одержання HTTP-відповіді від сервера клієнт повинен відображати її на екрані. Можливі такі типи сервера:

а) ітеративний;

б) який обробляє кожен запит окремим процесом-нащадком;

в) на основі моделі «потік для запиту» (можна скористатися розв'язком завдання 4 з розділу 15);

г) який реалізує пул потоків (для Windows XP можна використати порт завершення введення-виведення на основі розв'язку завдання 7 з розділу 15). Клієнт може використовувати таймер очікування (див. розв'язок завдання 8 з розділу 15).

12. Розробіть клієнт-серверну систему, що реалізує віддалене визначення максимального введеного числа («онлайновий аукціон»). Під час запуску клієнта необхідно вказати доменне ім'я і порт сервера. Далі цей інтерфейс клієнта повинен дозволяти користувачу ввести з клавіатури довільну кількість цілих чисел, супроводжуючи кожне з них іменем (наприклад, "ivanov 100", імена можуть бути різними). Після одержання кожного числа від клієнта сервер повинен обчислювати максимум із чисел, введених дотепер, і, якщо він змінився, сповіщати про це всіх користувачів (за допомогою повідомлення у форматі "новий максимум дорівнює 100. введений користувачем ivanov (адреса *n.n.n.n.* з'єднання 4)"). Введення числа 0 спричиняє розрив з'єднання для цього клієнта (сервер також повинен повідомляти про це користувачів відправленням повідомлення у форматі "клієнт з адресою *n.n.n.n* розірвав з'єднання 1"). Під час розробки коду серверного застосування використовуйте введення-виведення з повідомленням (UNIX) або асинхронні сокети (Windows XP).